Tools for Improving Performance Portability in Heterogeneous Environments

Jorge Fernández Fabeiro

PhD Thesis

2017

PhD Advisors:
Diego Andrade Canosa
Basilio B. Fraguela Rodríguez

PhD Program in Information Technology Research

UNIVERSIDADE DA CORUÑA

Tools for Improving Performance Portability in Heterogeneous Environments

---

Jorge Fernández Fabeiro

PhD Thesis

2017

PhD Advisors:

Diego Andrade Canosa
Basilio B. Fraguela Rodríguez

PhD Program in Information Technology Research

UNIVERSIDADE DA CORUÑA

Dr. Diego Andrade Canosa
Profesor Contratado Doctor
Dpto. de Ingeniería de Computadores
Universidade da Coruña

Dr. Basilio Bernardo Fraguela Rodríguez
Profesor Titular de Universidad
Dpto. de Ingeniería de Computadores
Universidade da Coruña

CERTIFICAN

Que la memoria titulada "*Tools for Improving Performance Portability in Heterogeneous Environments*" constituye un trabajo original de investigación realizado por D. Jorge Fernández Fabeiro bajo nuestra dirección en el marco del Programa de Doctorado en Investigación en Tecnologías de la Información de la Universidade da Coruña, y concluye la Tesis Doctoral que presenta para optar al grado de Doctor con la Mención de Doctor Internacional.

En A Coruña, a 27 de marzo de 2017

Fdo.: Diego Andrade Canosa
Director de la Tesis Doctoral

Fdo.: Basilio Bernardo Fraguela Rodríguez
Director de la Tesis Doctoral

Fdo.: Jorge Fernández Fabeiro
Autor de la Tesis Doctoral

Agradecimientos

El trabajo de investigación que recoge esta tesis no habría sido posible sin el trabajo y la dedicación de mis directores Diego Andrade y Basilio B. Fraguela. A Diego tengo que agradecerle su apoyo constante en el trabajo diario, ofreciendo soluciones a cuantos problemas fueran surgiendo y sabiendo dar el consejo oportuno en el momento oportuno. A Basilio, su denodado esfuerzo para encontrar la forma de que el trabajo realizado fuese siempre un poco mejor. Me gustaría agradecer también la colaboración del resto de miembros del Grupo de Arquitectura de Computadores, especialmente a Ramón Doallo, que ha participado activamente en el desarrollo de alguno de los trabajos que componen esta tesis.

Tampoco puedo olvidar a todos los proyectandos, doctorandos y contratados con los que he tenido la suerte de compartir tantos buenos ratos en cafés, comidas, cenas y demás momentos de descanso. Citaros a todos sería harto complicado, así que simplemente todo aquel que sienta que con una broma, una sugerencia o unas palabras de ánimo haya podido ayudarme a lo largo de estos años, muchas gracias.

I must thank also the staff of the 7th floor of the EEMCS of TU Delft, specially Ana Lucia, Henk, Jie, Elric and Ana. Thanks to you all for making my stay in The Netherlands a really *gezellig* time.

The following organisms must be acknowledged for funding and supporting this work: the Computer Architecture Group, the Department of Computer Engineering and the University of A Coruña; the Galician Government under the projects Consolidation Program of Competitive Reference Groups with references 2010/06 and GRC2013/055; the Ministry of Economy and Competitiveness of Spain under the projects with references TIN2010-16735 and TIN2013-42148-P; the HiPEAC3 (ICT-287759) and HiPEAC4 (ICT-687698) european Networks of Excellence, and

NESUS (COST Action, ref. IC1305), and the Spanish networks CAPAP-H.

Por último, y en absoluto por ello menos importante, nunca terminaré de estar agradecido a mis padres, por su cariño incondicional y por transmitirme los valores que me han venido guiado a lo largo de mi vida personal y académica.

Jorge

Resumo

Actualmente a computación paralela atópase dominada parcialmente polos múltiples dispositivos heteroxéneos dispoñibles. Estes dispositivos difiren entre si en características tales como o conxunto de instruccións que executan, o número e tipo de unidades de computación que inclúen ou a estrutura dos seus sistemas de memoria. Nos últimos anos apareceron linguaxes, bibliotecas e extensións que permiten escribir unha soa vez a versión paralela dun código e executala nun amplio abano de dispositivos, sendo de entre todos eles OpenCL a solución máis extendida. Porén, a portabilidade funcional non implica portabilidade de rendemento. Deste xeito, uns dos grandes problemas que segue aberto neste campo é a automatización da portabilidade de rendemento, isto é, a capacidade de adaptar automaticamente un código dado para a súa execución en calquera dispositivo e obter un bo rendemento. Esta tese aborda este problema propondo tres solucións diferentes. As tres están baseadas na aplicación de optimizacións de código a código usadas habitualmente en dispositivos heteroxéneos. Tanto o conxunto de optimizacións a aplicar como a forma de aplicalas dependen de varios parámetros de optimización para os que é preciso fixar determinados valores en función do dispositivo concreto.

A primeira solución proposta é OCLoptimizer, un optimizador de código a código que partindo de *kernels* OpenCL anotados e ficheiros de configuración de apoio, obtén versións optimizadas dos devanditos *kernels* para un dispositivo concreto. Amais, cando o *kernel* a optimizar é único, tamén automatiza a xeración dun código de *host* funcional para ese *kernel*.

As outras dúas solucións foron implementadas utilizando *Heterogeneous Programming Library* (HPL), unha biblioteca C++ que permite programar sistemas heteroxéneos de xeito fácil e portable. A primeira destas solucións explota as capa-

cidades de xeración de código en tempo de execución de HPL para xerar versións dun produto de matrices que se adaptan automaticamente ás características dun dispositivo concreto. A última solución consiste no desenvolvemento e incorporación a HPL dun optimizador capaz de obter en tiempo de execución versións optimizadas dun código HPL para un dispositivo dado. Mentres as dúas primeiras solucións usan procesos de procura para atopar os mellores valores para os parámetros de optimización, esta última alternativa baséase para iso en heurísticas definidas a partir de recomendacións xerais de optimización.

Abstract

Parallel computing is currently partially dominated by the availability of heterogeneous devices. These devices differ from each other in aspects such as the instruction set they execute, the number and the type of computing devices that they offer or the structure of their memory systems. In the last years, languages, libraries and extensions have appeared to allow to write a parallel code once and run it in a wide variety of devices, OpenCL being the most widespread solution of this kind. However, functional portability does not imply performance portability. This way, one of the problems that is still open in this field is to achieve automatic performance portability. That is, the ability to automatically tune a given code for any device where it will be executed so that it ill obtain a good performance. This thesis develops three different solutions to tackle this problem. The three of them are based on typical source-to-source optimizations for heterogeneous devices. Both the set of optimizations to apply and the way they are applied depend on different optimization parameters, whose values have to be tuned for each specific device.

The first solution is OCLoptimizer, a source-to-source optimizer that can optimize annotated OpenCL kernels with the help of configuration files that guide the optimization process. The tool optimizes kernels for a specific device, and it is also able to automate the generation of functional host codes when only a single kernel is optimized.

The two remaining solutions are built on top of the Heterogeneous Programming Library (HPL), a C++ framework that provides an easy and portable way to exploit heterogeneous computing systems. The first of these solutions uses the run-time code generation capabilities of HPL to generate a self-optimizing version of a matrix multiplication that can optimize itself at run-time for an specific device. The last

solution is the development of a built-in just-in-time optimizer for HPL, that can optimize, at run-time, a HPL code for an specific device. While the first two solutions use search processes to find the best values for the optimization parameters, this last alternative relies on heuristics based on general optimization strategies.

Resumen

Actualmente la computación paralela se encuentra dominada parcialmente por los múltiples dispositivos heterogéneos disponibles. Estos dispositivos difieren entre sí en características tales como el conjunto de instrucciones que ejecutan, el número y tipo de unidades de computación que incluyen o la estructura de sus sistemas de memoria. Durante los últimos años han aparecido lenguajes, librerías y extensiones que permiten escribir una única vez la versión paralela de un código y ejecutarla en un amplio abanico de dispositivos, siendo de entre todos ellos OpenCL la solución más extendida. Sin embargo, la portabilidad funcional no implica portabilidad de rendimiento. Así, uno de los grandes problemas que sigue abierto en este campo es la automatización de la portabilidad de rendimiento, es decir, la capacidad de adaptar automáticamente un código dado para su ejecución en cualquier dispositivo y obtener un buen rendimiento. Esta tesis aborda este problema planteando tres soluciones diferentes al mismo. Las tres se basan en la aplicación de optimizaciones de código a código usadas habitualmente en dispositivos heterogéneos. Tanto el conjunto de optimizaciones a aplicar como la forma de aplicarlas dependen de varios parámetros de optimización, cuyos valores han de ser ajustados para cada dispositivo concreto.

La primera solución planteada es OCLoptimizer, un optimizador de código a código que a partir de kernels OpenCL anotados y ficheros de configuración como apoyo, obtiene versiones optimizada de dichos kernels para un dispositivo concreto. Además, cuando el *kernel* a optimizar es único, automatiza la generación de un código de *host* funcional para ese *kernel*.

Las otras dos soluciones han sido implementadas utilizando *Heterogeneous Programming Library* (HPL), una librería C++ que permite programar sistemas he-

terogéneos de forma fácil y portable. La primera de estas soluciones explota las capacidades de generación de código en tiempo de ejecución de HPL para generar versiones de un producto de matrices que se adaptan automáticamente en tiempo de ejecución a las características de un dispositivo concreto. La última solución consiste en el desarrollo e incorporación a HPL de un optimizador al vuelo, de forma que se puedan obtener en tiempo de ejecución versiones optimizadas de un código HPL para un dispositivo dado. Mientras las dos primeras soluciones usan procesos de búsqueda para encontrar los mejores valores para los parámetros de optimización, esta última alternativa se basa para ello en heurísticas definidas a partir de recomendaciones generales de optimización.

Prólogo

La evolución del hardware a lo largo de las últimas décadas ha derivado en que actualmente múltiples tipos de dispositivos desempeñen papeles críticos en el ámbito de la computación de altas prestaciones. Estos dispositivos diferen entre sí en detalles como el juego de instrucciones que ejecutan, el número y las capacidades de los elementos de procesamiento que los componen, o la estructura de sus respectivas jerarquías de memoria. Tales diferencias supusieron que la inmensa mayoría de los mecanismos de programación disponibles para cada uno de ellos fueron diseñados originalmente teniendo en cuenta las características propias de cada tipo de dispositivo. Por este motivo, varios lenguajes, librerías y extensiones han sido propuestas con el objetivo de la portabilidad de código en mente, esto es, con la idea de permitir que los programadores puedan escribir un determinado código paralelo una vez y ejecutarlo en una amplia variedad de dispositivos. Las soluciones que han ido surgiendo han abordado esta cuestión desde diferentes perspectivas que van desde librerías y lenguajes de propósito específico a estándares abiertos como OpenCL.

OpenCL propone un modelo virtualmente capaz de representar cualquier tipo de dispositivo, de manera que los códigos programados utilizando este estándar pueden ser ejecutados en cualquier dispositivo que lo soporte. Sin embargo, esta capacidad de abstracción viene acompañada de un cierto aumento de la complejidad de programación, sobre todo si se compara con otras soluciones más cercanas a las características de cada dispositivo. Así, las aplicaciones OpenCL se componen de un código de *kernel* que implementa la computación propiamente dicha, y un código de *host* que gestiona el entorno de ejecución necesario para lanzar dicho *kernel* en un dispositivo determinado. Tomando como base el estándar OpenCL han ido apareciendo diferentes desarrollos especialmente enfocados en mejorar la programabilidad de los sistemas heterogéneos. Un ejemplo exitoso de estas propuestas es la

librería HPL (*Heterogeneous Programming Library*), una solución de alto nivel que ofrece una interfaz portable y fácil de usar para explotar las capacidades de los sistemas heterogéneos actuales. Los usuarios escriben sus *kernels* usando un lenguaje propio de HPL construido sobre C++, siendo estos *kernels* convertidos en tiempo de ejecución en código OpenCL. Así, si un dispositivo es compatible con OpenCL, podrá ser programado usando HPL. Efectivamente, tanto HPL como otras propuestas similares ofrecen portabilidad funcional en tanto en cuanto están construidas sobre OpenCL. Sin embargo, este es solamente el primer escollo a superar. Aunque un mismo código OpenCL se pueda ejecutar en múltiples tipos de dispositivos, las diferentes capacidades de éstos hacen que para obtener rendimientos aceptables en cada caso sea necesario escribir diferentes versiones optimizadas manualmente para cada uno de ellos. De esta limitación surge el interés de ofrecer algún tipo de mecanismo que permita a los programadores escribir un código una vez y que éste se adapte automáticamente para un determinado dispositivo. La consecución de herramientas de programación dotadas de esta característica, conocida como *portabilidad de rendimiento*, es uno de los principales problemas abiertos hoy en día en el ámbito de la computacion heterogénea.

Así, el objetivo de la presente tesis es desarrollar y evaluar un conjunto de soluciones de alto nivel capaces de ofrecer portabilidad de rendimiento en sistemas heterogéneos. Puesto que por su diseño, OpenCL es una alternativa válida para ofrecer portabilidad de código, muchas de las propuestas construidas sobre dicho estándar van un paso más allá e intentan ofrecer también en la medida de lo posible portabilidad de rendimiento. Después de un estudio previo de algunas de estas propuestas, se han extraído una serie de características que pueden ser consideradas como un buen punto de partida para la implementación de herramientas que aspiren a ofrecer portabilidad de rendimiento:

Transformaciones de código a código Para ser considerada como capaz de realizar transformaciones *de código a código*, una herramienta debe recibir como entrada programas escritos en algún lenguaje de alto nivel y devolver versiones modificadas de los mismos, ya sea en el mismo o en otro lenguaje de similares características. En esta tesis se exploran diversos mecanismos que transforman códigos escritos por los usuarios en lenguajes de alto nivel en versiones en OpenCL optimizadas para múltiples dispositivos.

Optimizaciones parametrizadas Las optimizaciones aplicadas a un código suelen depender de un conjunto de parámetros de configuración que determinan cómo va a ser transformado y, por tanto, el rendimiento que éste ofrecerá al ser ejecutado. Por ejemplo, para un dispositivo concreto puede que resulte interesante desenrollar un bucle y, de ser así, habría que fijar un valor adecuado para el factor de desenrollamiento a aplicar. Dicho factor de desenrollamiento funcionaría entonces como un *parámetro de optimización*, lo que conduce a hablar de *optimizaciones parametrizadas*.

Búsqueda de valores para los parámetros de optimización La portabilidad de rendimiento puede articularse a través de un mecanismo capaz de elegir un conjunto de optimizaciones adecuado en función de las características del código de entrada y de las capacidades de un determinado dispositivo. Si dichas optimizaciones están parametrizadas, una apropiada selección de los valores de estos parámetros será fundamental para maximizar el rendimiento del código generado. En esta tesis se exploran diferentes opciones para llevar a cabo esta selección, implementando en concreto algoritmos de búsqueda tanto exhaustiva como informada, así como heurísticas basadas en estrategias generales de optimzación de código para sistemas heterogéneos.

Interfaces de usuario de alto nivel La interfaz que ofrece OpenCL para ejecutar códigos en los dispositivos soportados descansa en un conjunto de operaciones de bajo nivel que pueden resultar dificultosas para los usuarios más inexpertos. En esta tesis se exploran diferentes alternativas para liberar a los usuarios de dichas interacciones o para que, al menos, éstas queden reducidas a la mínima expresión posible.

A continuación se ofrece una descripción de las principales características de cada una de las herramientas desarrolladas, pivotando dichas explicaciones en torno a las propiedades que se acaban de comentar.

OCLoptimizer Esta primera propuesta es una herramienta de optimización iterativa código a código. Recibe como entradas un kernel OpenCL anotado por el usuario y un fichero de configuración que establece el entorno y ciertas condiciones del proceso de optimización. Los usuarios deben anotar los kernels con

directivas que indiquen qué secciones del código deben ser optimizadas y con qué técnicas, las cuales dependen de un conjunto de parámetros cuyos rangos también han de ser fijados por los usuarios. Esto permite que los procesos iterativos de búsqueda, tanto exhaustiva como informada, implementados en la herramienta exploren múltiples combinaciones de valores para dichos parámetros y generen las versiones de código correspondientes. Para poder probar estas versiones y elegir finalmente la más rápida, la herramienta también es capaz de generar el código de *host* que permite lanzar kernels OpenCL en un dispositivo dado. De esta forma, OCLoptimizer pretende liberar al programador de dos tareas tediosas y propensas a errores como son la optimización manual de sus kernels OpenCL para diferentes dispositivos y la necesidad de escribir un código de *host* para cada uno de ellos. Otra característica destacable de esta herramienta es su soporte de optimización de códigos OpenCL compuestos de varios kernels, algo que se complica notablemente si además existen dependencias de datos entre los mismos, siendo esta una situación que la herramienta es capaz de gestionar.

Kernels HPL autoadaptativos Las características del mecanismo por el cual HPL convierte en tiempo de ejecución los *kernels* escritos en su lenguaje en código OpenCL abren la posibilidad de generar diferentes versiones OpenCL a partir de un mismo código de usuario. Así, estas capacidades de generación de código en tiempo de ejecución han sido explotadas para conformar un conjunto de técnicas de optimización que, a su vez, pueden ser utilizadas para construir kernels capaces de adaptar su código y, por tanto, su rendimiento, para un determinado dispositivo. La implementación de estas técnicas de optimización está parametrizada, de forma que el código resultante de su aplicación depende de los valores dados a dichos parámetros. Como caso de estudio para explicar y evaluar el funcionamiento de este enfoque se ha implementado una versión autoadaptativa de una multiplicación de matrices. En dicha versión es posible ajustar parámetros como el factor de desenrollamiento y la planificación de las instrucciones de los bucles, el reparto de trabajo entre los diferentes elementos de procesado de un dispositivo, qué estructuras se guardan en memoria local (una abstracción de OpenCL que representa escalones intermedios en la jerarquía de memoria de los dispositivos, habitualmente alguno de los niveles de memoria caché) o la vectorización de diferentes operaciones tanto

de computación como de acceso a memoria. Los valores de estos parámetros de optimización se ajustan mediante una búsqueda iterativa basada en un algoritmo genético. En líneas generales, las optimizaciones incluidas en esta versión autoadaptativa del producto de matrices se inspiran en implementaciones existentes para dispositivos de diferentes tipos y fabricantes. Merecen una mención especial a este respecto las librerías ViennaCL y clBLAS, ya que están implementadas en OpenCL y también proporcionan mecanismos para ofrecer portabilidad de rendimiento entre dispositivos. Precisamente por estos motivos estas librerías también han sido tomadas como referencia a la hora de evaluar el rendimiento de este producto de matrices autoadaptativo.

Optimizador *just-in-time* para HPL Con la implementación y posterior evaluación del kernel autoadaptativo del producto de matrices afloraron algunos inconvenientes que dieron pie a seguir buscando mecanismos más refinados mediante los cuales HPL pueda ofrecer portabilidad de rendimiento. Por una parte, la aplicación de las técnicas parametrizadas obliga a los usuarios a reescribir sus códigos. Por otra, el algoritmo genético implementado necesita generar la versión correspondiente a cada combinación de parámetros explorada y ejecutarla a continuación para evaluar su rendimiento. Con el objeto de mitigar estos inconvenientes se ha incorporado en HPL un optimizador al vuelo (*just-in-time*) capaz de adaptar el código automáticamente para un determinado dispositivo sin retrasar considerablemente su ejecución. Este proceso recibe como entrada un *kernel* HPL que el usuario debe escribir de forma elemental, esto es, describiendo el cálculo de un punto de su problema y sin aplicar manualmente ninguna técnica de optimización. Dicho código es analizado y transformado en un árbol sintáctico o AST (*Abstract Syntax Tree*). Siguiendo un orden determinado experimentalmente, se aplican una serie de optimizaciones extraídas de un conjunto de recomendaciones generales para la optimización de códigos en entornos heterogéneos. Estas optimizaciones están implementadas en forma de transformaciones parametrizadas de dicho árbol sintáctico, siendo los valores asignados a esos parámetros los que determinan qué optimizaciones concretas se aplican y bajo qué condiciones. En este caso los valores de los parámetros no se fijan mediante algoritmos de búsqueda implementados a tal efecto, sino mediante un conjunto de heurísticas que intentan trasladar las recomendaciones antes comentadas a dichos parámetros.

Para validar su funcionamiento, estas tres herramientas han sido utilizadas para optimizar implementaciones OpenCL y HPL de códigos habituales en dominios tales como el álgebra lineal densa (multiplicación de matrices y otras operaciones relacionadas), el procesado de señales e imágenes (convoluciones y filtros) o la física de partículas (interacciones electromagnéticas y gravitatorias), entre otros. Todos estos procesos de optimización han sido ejecutados para diferentes tamaños de estos problemas sobre dispositivos de múltiples tipos (procesadores multinúcleo de propósito general, unidades de procesamiento gráfico y otras aceleradores *many-core*) de diversos fabricantes (Intel, AMD, Nvidia).

Metodología de trabajo

El trabajo de investigación recogido en esta Tesis Doctoral se descompone en cinco grandes bloques dedicados a la obtención de una visión de conjunto del estado actual de la computación heterogénea, la implementación de cada una de las tres herramientas anteriormente descritas y, finalmente, la conclusión de trabajo de investigación con la elaboración del presente documento. A su vez, cada una de dichas herramientas ha sido desarollada de acuerdo con una metodología iterativo-incremental de modo que, tras una fase de contextualización previa, sucesivas iteraciones han ido incrementando las funcionalidades implementadas en cada una de las herramientas.

Bloque 1: Contextualización

Objetivo: Análisis del estado del arte de la computación heterogénea

Tareas:

- Estudio de dispositivos heterogéneos y mecanismos de programación
- Estudio de los principales desafíos existentes en este ámbito: portabilidad de código y portabilidad de rendimiento
- Estudio del alcance de diferentes soluciones que abordan dichos desafíos, prestando especial atención a OpenCL y HPL

- Planteamiento de propuestas de aproximación a la portabilidad de rendimiento: optimización parametrizada

Bloque 2: OCLoptimizer

Contextualización

Objetivo: Adquisición de habilidades y conocimientos técnicos necesarios

Tareas:

- Estudio del estándar OpenCL y familiarización con su modelo de programación: kernels y hosts
- Estudio de librerías de análisis y transformación de código: familiarización con Clang/LLVM y su interfaz de programación
- Estudio de diferentes algoritmos de búsqueda exhaustiva e informada

Iteración 2.1: Optimización individual de kernels OpenCL

Objetivo: Herramienta capaz de recibir kernels OpenCL de usuario y generar versiones optimizadas para diferentes dispositivos

Tareas:

- Interfaz de usuario: directivas parametrizadas de optimización y fichero de configuración
- Implementación de técnicas de optimización: "unroll" y "unroll-and-jam"
- Implementación de un mecanismo de análisis y transformación de código mediante Clang/LLVM
- Procesos de búsqueda de valores optimizados para los parámetros: algoritmos en anchura y genético
- Generación automática de códigos de host OpenCL

Evaluación: Selección de kernels de uso habitual en diferentes aplicaciones y optimización de los mismos utiizando la herramienta implementada

Iteración 2.2: Optimización del espacio de trabajo de OpenCL

Objetivo: Funcionalidad adicional: optimización del espacio de trabajo de OpenCL

Tareas:

- Interfaz de usuario: macros complementarias a las directivas y extensión de las opciones del fichero de configuración
- Extensión del proceso de generación automática de códigos de host
- Procesos de búsqueda de valores optimizados para la configuración del espacio de trabajo: algoritmos exhaustivo y genético

Evaluación: Optimización combinada del espacio de trabajo y del código de kernel de las aplicaciones seleccionadas

Iteración 2.3: Optimización de aplicaciones multi-kernel

Objetivo: Funcionalidad adicional: optimización de aplicaciones multi-kernel

Tareas:

- Replicación del proceso combinado de optimización para cada kernel
- Kernels interdependientes: identificación de las dependencias y replicación del proceso de optimización condicionada a las mismas

Evaluación: Optimización del problema "Integer Sort" de los NAS Parallel Benchmarks, compuesto de kernels de ambos tipos.

Bloque 3: Kernels HPL auto-adaptativos

Contextualización

Objetivo: Adquisición de habilidades y conocimientos técnicos sobre HPL

Tareas:

- Estudio de la arquitectura de HPL
- Familiarización con su modelo de programación: kernels e interfaz de usuario de alto nivel

Iteración 3.1: Definición de un conjunto de optimizaciones y aplicación a un caso de uso

Objetivo: Implementación de kernels HPL capaces de generar automáticamente diferentes versiones de código adaptadas a diferentes dispositivos

Tareas:

- Estudio de las capacidades de generación de código en tiempo de ejecución (RTCG) de HPL
- Selección de un conjunto de optimizaciones aplicadas habitualmente en códigos para dispositivos heterogéneos
- Implementación parametrizada del conjunto de optimizaciones aplicando RTCG.
- Proceso de búsqueda de valores optimizados para los parámetros: algoritmo genético

Evaluación: Implementación de un caso de uso de kernel autoadaptativo (producto de matrices) y generación automática de versiones optimizadas para diferentes tipos de dispositivo

Iteración 3.2: Incorporación de optimizaciones adicionales al caso de uso

Objetivo: Incorporación y refinado de optimizaciones adicionales aplicadas en implementaciones ya existentes del caso de uso seleccionado

Tareas:

- Estudio de implementaciones ya existentes del caso de uso e identificación de posibles optimizaciones adicionales
- Implementación RTCG de las optimizaciones identificadas: vectorización, reordenamiento y planificación de instrucciones
- Extensión del proceso de búsqueda para soportar las nuevas optimizaciones implementadas

Evaluación: Extensión de la implementación del caso de uso del producto de matrices, generación automática de versiones optimizadas para diferentes tipos de dispositivo y comparación de resultados con los de las soluciones estudiadas

Bloque 4: Optimizador *just-in-time* para HPL

Contextualización

Objetivo: Adquisición de conocimientos sobre el funcionamiento interno de HPL

Tareas:

- Análisis de la implementación de los tipos de datos propios de HPL
- Análisis del mecanismo de evaluación/ejecución de kernels HPL
- Análisis del mecanismo de traducción a código OpenCL

Iteración 4.1: Extensión del mecanismo de generación de código de HPL

Objetivo: Incorporación de una etapa intermedia de representación de kernels en forma de árbol

Tareas:

- Definición de una representación sintática intermedia: árbol de sintaxis abstracta (AST), e implementación de la misma
- Sustitución del mecanismo actual de traducción de kernels a código OpenCL por otro de construcción de su representación intermedia
- Implementación de un mecanismo de generación de código OpenCL a partir de un árbol sintático previo

Evaluación: Ejecución de los ejemplos de prueba incluidos en la distribución de HPL, comparando la corrección de la representación intermedia implementada y su posterior traducción a código OpenCL

Iteración 4.2: Implementación del proceso de optimización just-in-time

Objetivo: Incorporación de un proceso completo de optimización just-in-time para kernels HPL

Tareas:

- Selección de un conjunto de optimizaciones aplicadas habitualmente en códigos para dispositivos heterogéneos

- Implementación de dichas optimizaciones en forma de transformaciones parametrizadas sobre el AST de un kernel
- Definición e implementación de una heurística de optimización: orden de aplicación y determinación de los valores de los parámetros
- Interfaz de usuario: kernels HPL simplificados que indiquen la forma general de cálculo de un punto del problema

Evaluación: Selección de kernels de uso habitual en diferentes aplicaciones, implementación simplificada de los mismos en HPL y prueba de optimización al vuelo de los mismos

Bloque 5: Conclusión del trabajo de investigación

Objetivo: Elaboración de la memoria final de la Tesis Doctoral

Tareas:

- Recopilación de los trabajos de investigación desarollados
- Estructuración, organización y ampliación del contenido
- Recopilación y exposición de conclusiones
- Análisis de posibles líneas de trabajo futuro
- Redacción de la memoria final de la Tesis Doctoral

Medios

Para la elaboración de la tesis se emplearon los medios detallados a continuación:

- Soporte económico proporcionado por el Grupo de Arquitectura de Computadores de la Universidade da Coruña, la propia Universidade da Coruña y el Banco Santander (beca para estudios de máster 2011-2012)

- Redes de investigación en las que se integra esta tesis:

    - High-Performance Embedded Architectures and Compilers Network of Excellence, HiPEAC3 NoE (ref. ICT-287759).

- High-Performance Embedded Architectures and Compilers Network of Excellence, HiPEAC4 NoE (ref. ICT-687698).
- Network for Sustainable Ultrascale Computing (NESUS). ICT COST Action IC1305.
- Open European Network for High Performance Computing on Complex Environments (ComplexHPC). ICT COST Action IC0805
- Red de Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogéneas (CAPAP-H4) (ref. TIN2011-15734-E).
- Red de Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogéneas (CAPAP-H5) (ref. TIN2014-53522-REDT).
- Red de Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogéneas (CAPAP-H6) (ref. TIN2016-81840-REDT).

- Proyectos de investigación que financiaron esta tesis:
  - Architectures, Systems and Tools for High Performance Computing (Ministerio de Economía y Competitividad, TIN2010-16735).
  - Consolidación y Estructuración de Unidades de Investigación Competitivas: Grupo de Arquitectura de Computadores de la Universidad de A Coruña (Xunta de Galicia, ref. 2010/6)
  - Consolidación y Estructuración de Unidades de Investigación Competitivas: Grupo de Arquitectura de Computadores de la Universidad de A Coruña (Xunta de Galicia, GRC2013-055).
  - Nuevos desafíos en la computación de altas prestaciones: Desde arquitecturas hasta aplicaciones. (Ministerio de Economía y Competitividad, TIN2013-42148-P).

- Estancias de investigación realizadas a lo largo del desarrollo de esta tesis:
  - Estancia de septiembre a diciembre de 2015 (3 meses) en el grupo de investigación *Parallel and Distributed Systems* de la *Delft University of Technology*, bajo la supervisión de los profesores Henk Sips y Ana Lucia Varbanescu. Financiada mediante una beca INDITEX-UDC para estancias de investigación en el extranjero.

- Recursos de computación empleados en la realización de esta tesis:
  - Clúster *pluton* del Grupo de Arquitectura de Computadores de la Universidade da Coruña, compuesto de:
    - 8 nodos con CPU 2 × Intel Xeon E5-2660 de 8 cores y 64 GB de RAM. Cada nodo cuenta con una GPU NVIDIA K20m con 5 GB de RAM. La red de interconexión es Infiniband FDR.
    - 4 nodos con CPU Intel Xeon X5650 de 6 cores y 12 GB de RAM. Cada nodo cuenta con dos GPUs NVIDIA M2050 con 3 GB de RAM cada una. La red de interconexión es Infiniband QDR.
    - 1 nodo con CPU 2 × Intel Xeon E5-2660 de 8 cores y 64 GB de RAM. Cuenta con un acelerador Intel Xeon Phi 5110P de 60 cores y 8 GB de RAM.
    - 1 nodo con CPU 2 × Intel Xeon E5-2650v2 de 8 cores y 64 GB de RAM. Cuenta con una GPU AMD FirePro S9150 con 16 GB de RAM. La red de interconexión es Infiniband FDR.
  - Máquina *Mercurio* del Grupo de Arquitectura de Computadores de la Universidade da Coruña. 1 Nodo con CPU Intel Core 2 con 2 GB de RAM. Cuenta con una GPU AMD HD6970 con 2GB de RAM.

Conclusiones

Las plataformas heterogéneas son predominantes actualmente en el ámbito de la computación de altas prestaciones. Este tipo de plataformas reúnen dispositivos paralelos de múltiples tipos, compuestos a su vez de elementos de procesado caracterizados por capacidades de procesamiento o una jerarquía de memoria de características muy diversas. Algunas de estas arquitecturas, como los procesadores multinúcleo x86, ofrecen capacidades paralelas sin descuidar la retrocompatibilidad con diseños previos. Otras, como por ejemplo las unidades de procesamiento gráfico (GPU), evolucionaron hasta convertirse en dispositivos masivamente paralelos de computación de propósito general. También hay propuestas que tratan de aunar lo mejor de ambos mundos: diseños como las aceleradoras Xeon Phi de Intel o las APU

(Accelerated Processing Units) de AMD son capaces de ejecutar códigos inicialmente ideados para CPUs tradicionales sobre arquitecturas masivamente paralelas como las de las GPUs. Tal variedad de diseños creados por múltiples fabricantes dieron lugar a infinidad de soluciones de programación más o menos adaptadas a cada caso concreto. Algunas de estas soluciones se dirigen un tipo de dispositivo y fabricante concretos, como CUDA, creada por Nvidia para extraer paralelismo de sus GPUs. Otras soluciones fueron diseñadas planteando aproximaciones más amplias en forma de entornos de programación paralela, caso de los estándares OpenMP, basado en directivas de compilación, o MPI, que sigue un paradigma de paso de mensajes.

Dentro de este contexto surgieron evoluciones de estándares anteriores, como es el caso de la versión 4.0 de OpenMP, o estándares nuevos como OpenACC (basado en directivas como OpenMP) u OpenCL, que fueron un paso más allá al proponer diferentes mecanismos portables en código para programar dispositivos heterogéneos. En el caso de OpenCL, el modelo de programación ofrecido se compone de *kernels* programados en una extensión de C y un código de *host* cuyo objeto es preparar el entorno de ejecución para lanzar dichos *kernels*. Los códigos de *host* deben escribirse utilizando una interfaz que expone ciertas funcionalidades de bajo nivel y que pueden resultar complicada para los usuarios más inexpertos. Este inconveniente fue uno de los detonantes para la aparición de múltiples propuestas para la programación de sistemas heterogéneos que, basadas en OpenCL, intentar ocultar este tipo de detalles a los usuarios. Un ejemplo exitoso de ello es *Heterogeneous Programming Library* (HPL), una librería en la que se basan algunas de las herramientas desarrolladas en esta tesis. Siendo la portabilidad de código una de las principales fortalezas de OpenCL, la diversidad de dispositivos que éste soporta hace que un mismo *kernel* no tenga por qué ofrecer directamente un rendimiento óptimo en cada uno de ello. Sin embargo, existe un conjunto de estrategias genéricas de optimización que pueden ser aplicadas a estos códigos para adaptarlos a diferentes dispositivos. La aplicación de dichas estrategias puede realizarse mediante una serie de transformaciones de código dependientes de una serie de parámetros, de forma que valores particulares de esos parámetros para un dispositivo concreto den lugar a una versión de código optimizada para el mismo. Esta aproximación es la seguida en todas las herramientas desarolladas en esta tesis, las cuales son capaces de generar código OpenCL optimizado para múltiples dispositivos aplicando este tipo de técnicas parametrizadas y encontrando valores adecuados para estos parámetros a través de

diferentes procedimientos de búsqueda.

La primera solución presentada es OCLoptimizer, un optimizador de código a código para *kernels* OpenCL. Las entradas de esta herramienta son un fichero de configuración y un *kernel* anotado por el usuario con indicaciones acerca de las optimizaciones a probar. Las salidas generadas son, por una parte, una versión del *kernel* optimizada para un dispositivo concreto y, por otra, un código de *host* adaptado al kernel generado. La herramienta también es capaz de optimizar aplicaciones compuestas de varios códigos, variando el tratamiento que hace de los mismos según las dependencias que existan entre ellos. El proceso de optimización implementado en la herramienta se compone de dos partes. En primer lugar, se busca una configuración adecuada del espacio de trabajo de OpenCL para el kernel de entrada, pudiendo realizar dicha búsqueda de forma exhaustiva probando todas las combinaciones legales posibles, o mediante un proceso informado guiado por un algoritmo genético. Respecto de los *kernels*, los usuarios pueden anotar los bucles de sus códigos para que la herramienta pruebe diferentes factores de desenrollamiento sobre los mismos. En función del número de bucles anotados y el rango de prueba fijado para cada uno, es posible desencadenar una explosión combinatoria al generar el espacio de búsqueda del proceso de optimización del *kernel*. Por ello, para esta segunda parte no se ofrece un proceso puramente exhaustivo como el anterior, sino que se opta por una aproximación en anchura o *breadth-first* (BFS) que va procesando una a una las directivas encontradas a medida que se va recorriendo el código anotado. También se ha implementado en este caso una búsqueda genética que prueba de cada vez diferentes combinaciones de factores de desenrollamiento para todas las directivas del *kernel*. Las operaciones de análisis y transformación de código necesarias para optimizar los *kernels* están implementadas utilizando Clang, el *front-end* para C de la infraestructura de compilación LLVM. El funcionamiento de la herramienta ha sido validado en una CPU, una GPU y una aceleradora Intel Xeon Phi, obteniendo resultados satisfactorios para la optimización de códigos OpenCL compuestos tanto de un único kernel como de varios. En cuanto a las aplicaciones de un solo kernel, la aceleración media alcanzada fue de 2.22 utilizando algoritmos genéticos (GA) y de 2.86 combinando los algoritmos exhaustivo y en anchura (ES+BFS), si bien el proceso de búsqueda de la primera aproximación resultó ser unas 10 veces más rápido que el de la segunda. Respecto de la combinación ES+BFS, las aceleraciones medias obtenidas fueron respectivamente de 1.59, 2.54 y 4.46 para CPU, GPU y Xeon Phi,

de modo que el uso de la herramienta resultó beneficioso en las tres plataformas. El soporte de aplicaciones compuestas de varios códigos fue validado mediante la optimización para CPU y GPU del problema IS de los *benchmarks* SNU NPB, alcanzado aceleraciones de 2.45 y 1.19 respectivamente.

En cuanto a las soluciones basadas en la librería HPL, los *kernels* auto-adaptativos fueron la primera aproximación implementada en el contexto de la presente tesis. Específicamente, estos códigos explotan, a través de mecanismos de programación genérica, las posibilidades de generación de código en tiempo de ejecución ofrecidas por HPL. Resultado destacable de este trabajo es la implementación de un conjunto de técnicas parametrizadas de optimización que pueden ser utilizadas para generar versiones portables en rendimiento de *kernels* HPL, estando la generación de dichas versiones gobernada por los valores dados a los parámetros correspondientes. De esta forma, a partir de un mismo *kernel* HPL es posible generar versiones con repartos de trabajo de granularidades diferentes, desenrollar o aplicar *tiling* sobre los bucles del código con diferentes factores o probar diferentes órdenes de planificación de las instrucciones que dichos bucles ejecutan si están anidados. Así mismo, secciones completas de código pueden ser generadas o no de acuerdo a una determinada condición. Esto se aplica especialmente en función de si se desea o no explotar la memoria local de un dispositivo. Estas y otras técnicas fueron utilizadas para implementar un kernel auto-adaptativo de una multiplicación de matrices configurable mediante una docena de parámetros de optimizacion. La búsqueda de los valores más adecuados para estos parámetros fue guiada mediante un algoritmo genético diseñado e implementado de forma similar al incorporado en OCLoptimizer. El funcionamiento de este caso de uso fue evaluado en GPUs Nvidia y AMD, en una CPU multinúcleo Intel y en una aceleradora Intel Xeon Phi, comparando su rendimiento con el de las implementaciones auto-adaptativas del producto de matrices proporcionadas por clBLAS y ViennaCL, dos librerías populares de álgebra lineal escritas en OpenCL. La aceleración media obtenida por los códigos generados a partir del kernel HPL auto-adaptativo fue de 1.74 respecto de clBLAS y 1.44 respecto de ViennaCL. En cuanto a la duración del proceso de búsqueda de dichas versiones, el algoritmo genético implementado fue de media 1.18 veces más rápido que el *profiler* incorporado en clBLAS, así como unas 160 veces más rápido que la búsqueda exhaustiva realizada por ViennaCL.

La primera aproximación a la implementación de kernels autoadaptativos basada en las capacidades de generación de código en tiempo de ejecución de HPL puede resultar demasiado compleja para aquellos usuarios con habilidades de programación más elementales. Es a esta clase de usuarios a la que principalmente se dirige el optimizador al vuelo o *just-in-time* para HPL, segunda herramienta basada en esta librería desarrollada en el contexto de esta tesis. Para hacer uso de este optimizador, los usuarios deben implementar un kernel HPL en el que solamente especifiquen cómo se calcula un punto del espacio de soluciones de su problema. Este código será analizado por el optimizador, el cual se encuentra empotrado en el flujo de trabajo de la librería, de modo que en tiempo de ejecución se generará una versión optimizada que será finalmente lanzada en el dispositivo solicitado por el usuario. Esta aproximación simplifica notablemente la tarea de programar un kernel HPL autoadaptativo, puesto que ahora a partir de una implementación elemental es posible generar múltiples versiones diferentes. Las tareas de análisis y transformación del *kernel* no se realizan directamente sobre el código de entrada, sino que éste es previamente cargado en un árbol de sintaxis abstracta (AST), lo que a su vez obligó a modificar la manera en que HPL traducía originalmente sus *kernels* a código OpenCL. De esta forma fue posible construir un conjunto de técnicas de optimización implementadas como transformaciones sobre un AST dado y que intentan plasmar una serie de estrategias comúnmente aplicadas en la optimización de códigos para dispositivos heterogéneos. En concreto, el optimizador es capaz de desenrollar y aplicar *tiling* sobre los bucles de computación del *kernel* de entrada, transformar el código para que ciertas estructuras de datos sean previamente copiadas en la memoria local disponible en algunos tipos de dispositivos, ajustar la granularidad del reparto de iteraciones del *kernel*, y explotar la región de memoria privada para realizar determinados cálculos y así reducir la contención en los accesos a la memoria global. Todas estas técnicas de optimización son, en mayor o menor medida, interdependientes, por lo que se aplican siguiendo un orden previamente fijado de forma experimental. Así mismo, todas ellas dependen de un conjunto de parámetros, de modo que en función de los valores asignados a éstos se aplican unas u otras técnicas y el código generado por cada una de ellas varía en consecuencia. Estos valores determinan la configuración del espacio de trabajo de HPL, los tamaños de *tile* y los factores de desenrollamiento aplicados a los bucles, la explotación o no de la memoria local, así como los tamaños de bloque de memoria privada

y si el espacio para este bloque debe ser declarado como un array o un conjunto de variables privadas. Si se elige esta última opción, la computación realizada sobre dicho espacio de memoria privada será además completamente desenrollada. Como en las herramientas anteriores, valores diferentes de estos parámetros implican la generación de diferentes versiones de código para un mismo *kernel* de entrada, de manera que si estos valores se escogen en función de las capacidades de un dispositivo concreto, la versión generada estará optimizada para el mismo. Por el momento, los valores se fijan heurísticamente intentando trasladar las recomendaciones generales antes mencionadas a las características de cada dispositivo. Para validar esta herramienta se han optimizado implementaciones elementales de ocho problemas diferentes en tres plataformas distintas: una GPU Nvidia, una GPU AMD y una CPU multinúcleo Intel. Las versiones generadas por el optimizador han alcanzado aceleraciones de entre 1.83 y 57.19 en la GPU Nvidia, de entre 1.21 y 82.98 en la GPU AMD, y de hasta 27.32 en la CPU de Intel. Así mismo, el proceso de optimización resulta bastante ligero, necesitando entre 1 y 59 milisegundos para generar las versiones de código OpenCL. Este coste además se va amortizando a lo largo de las ejecuciones de un kernel, puesto que HPL almacena los kernels ya generados en una caché interna. Así, una vez se obtiene una versión optimizada de un kernel en una aplicación, ésta puede ser reutilizada sin tener que generarla de nuevo. Estos resultados muestran que, mediante un conjunto de transformaciones parametrizadas dirigidas mediante heurísticas, el optimizador incorporado es capaz de tomar como entrada kernels HPL elementales y generar automáticamente y al vuelo versiones de los mismos optimizadas para diferentes plataformas.

Principales contribuciones

- Estudio y análisis de múltiples soluciones para obtener portabilidad de rendimiento en sistemas de computación heterogénea.
- Estudio y prueba de diferentes arquitecturas y entornos.
- Diseño, implementación y prueba de varias soluciones para facilitar la portabilidad de rendimiento en sistemas heterogéneos, combinando:
    - diversos mecanismos de programación: lenguajes y librerías;

    - diversos procesos de análisis y transformación de código;
    - diversos métodos de búsqueda: exhaustiva, informada y heurística.

- Estudio de rendimiento de estas soluciones comparándolos con los de otras alternativas relevantes existentes.

Publications from the thesis

- Fabeiro, J.F.; Andrade, D.; Fraguela, B.B. OCLoptimizer: an Iterative Optimization Tool for OpenCL. Proceedings of the International Conference on Computational Science (ICCS 2013), 2013.
- Fabeiro, J.F.; Andrade, D.; Fraguela, B.B.; Doallo, R. Writing Self-Adaptive Codes for Heterogeneous Systems. Proceedings of 20th Euro-Par International Conference on Parallel Processing (Euro-Par 2014), Lecture Notes on Computer Science 8632, pp. 800-811, 2014.
- Fabeiro, J.F.; Andrade, D.; Fraguela, B.B.; Doallo, R. Automatic Generation of Optimized OpenCL Codes using OCLoptimizer. The Computer Journal 58(11), pp. 3057-3073, 2015.
- Fabeiro, J.F.; Andrade, D.; Fraguela, B.B. Writing a Performance-Portable Matrix Multiplication". Parallel Computing (2016) 52, pp. 65-77, 2016.
- Fabeiro, J.F.; Andrade, D.; Fraguela, B.B.; Doallo, R. How to Write Performance Portable Codes using the Heterogeneous Programming Library, Proceedings of the 19th Workshop on Compilers for Parallel Computing (CPC'16), 2016.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The evolution of hardware during the last decades has made available multiple kinds of devices that play a critical role in the current parallel computing landscape. These devices differ in architectural details such as the instruction sets they execute, the number and capabilities of the computing elements they include, or the structure of their memory hierarchies. Due to these differences, the vast majority of the mechanisms originally available to program them were too focused on the capabilities of each kind of device, which made the programming of these architectures more and more difficult along the years.

For this reason, several languages, libraries and extensions have been proposed pursuing code portability, that is, to allow programmers to write a parallelized code once and run it in a wide variety of devices. Over time, different approaches have appeared to tackle this issue, ranging from domain-specific libraries or languages to far-reaching open standards like OpenCL [55], which is the most widespread solution of this kind.

OpenCL proposes a comprehensive abstraction model virtually able to represent every kind of computing device. However, offering a programming interface that fully supports such a model also means a significant increase on its complexity when compared to other solutions based on lower levels of abstraction. Lately, several proposals to improve the programmability of heterogeneous devices have been built on top of OpenCL. A successful example of them is the Heterogeneous Programming Library (HPL), a C++ framework that provides an easy and portable way to exploit

heterogeneous computing systems [114]. The HPL back-end generates OpenCL code, thus, HPL can be used to program the same devices as OpenCL.

OpenCL and the multiple proposals built on top of it allowed heterogeneous computing to conquer effective functional portability, but this was just the first pitfall to overcome. Different types of devices are built following different architectures, each with its own particularities and capabilities. For example, computing elements inside current multicore processors expect larger and more complex workloads than those included in Graphic Processing Units (GPUs). A single OpenCL program can be executed in both kinds of devices, but the workload distribution set in that single code is very unlikely to fit the capabilities of computing units of both devices at the same time. As a consequence, different hand-tuned OpenCL programs must be written in order to obtain good performance in each device. However, it would be preferable to develop a mechanism that allows users to write a single code and then, in an automatic or at least automatable way, tune the code for different devices. This capability is known as performance portability, and it is one of the most important open problems in heterogeneous computing.

The purpose of this PhD thesis is to develop and evaluate three different solutions to provide performance portability on heterogeneous devices. All of them are based on applying typical source-to-source optimizations to codes parallelized for heterogeneous devices. As part of this optimization process, the value of several optimization parameters has to be tuned for each specific device. This tuning is in fact what really optimizes the code for each device.

The first solution is OCLoptimizer, a source-to-source optimizer which can optimize annotated OpenCL kernels with the help of configuration files that guide the optimization process. The tool not only optimizes kernels for a given device, but it is also able to automate the generation of functional host codes when only a single kernel is optimized. As part of the optimization process, OCLoptimizer has to tune the value of several optimization parameters. The search space of possible values for these parameters can be explored using either a genetic algorithm or a breadth-first search guided by the execution time of each version of the code.

The two remaining solutions are built on top of HPL. The first of these solutions uses the run-time code generation capabilities of HPL to generate a self-optimizing

version of a code that can optimize itself at run-time for an specific device. In this solution, exemplified using the matrix multiplication, the exploration of the search space of possible values of the optimization parameters is alway done using a genetic search. The last solution is the development of a built-in just-in-time optimizer for HPL that can optimize, at run-time, a HPL code for a specific device. The values of the optimization parameters associated to this optimizer are set by means of heuristics based on general optimization strategies.

The rest of this chapter is organized as follows. In Section 1.1 the reader can go through the recent history of several computing device types, and how this evolution led to the eruption of heterogeneous computing. Section 1.2 explains why unified ways of programming these device types are needed and introduces some proposals, focusing on OpenCL and the solutions built on top of it. Section 1.3 introduces the performance portability problem and explores different approaches available to achieve it. Finally, the proposals and contributions of this thesis to automate performance portability are presented in Section 1.4.

1.1. How and why heterogeneity arose

This section contains a brief review of the historical evolution of the different kinds of heterogeneous devices. First, we focus on the recent history of conventional CPUs and GPUs, then we review the recent developments on modern accelerators and the usage of field programmable gate arrays, and finally we introduce the idea of modern heterogeneous clusters.

The general-purpose processor

Traditional general-purpose processors have played the leading role in computing history for many years. In the past, the application of concepts like pipelining, execution speculation, branch prediction, or the use of math co-processors for floating-point operations made possible to exploit instruction-level parallelism. These efforts made on hardware engineering had also a response in software development, where multitasking support was enabled in operating systems.

The sustained increase in silicon-based circuitry integration levels led to power dissipation and consumption issues. These problems forced manufacturers to branch from building more complex processors with higher clock frequencies to think of designs gathering several computing units inside a same silicon die. A result of this evolution were the multicore processors, which integrate several cores that have a partially shared memory hierarchy. Frequently, upper cache levels are private, whereas the lower ones and the main memory are shared. Multithreaded software can take advantage of multicore processors by running different threads simultaneously in different cores and using common memory regions for data sharing. Thread management operations are OS-dependent, and standards like OpenMP offer higher level interfaces to write multithreaded codes. There are OpenMP implementations available for FORTRAN [83] and C/C++ [84]. These implementations provide programmers with a set of compiler directives with which they can tag sections of their programs to be run in multiple execution threads.

The Graphics Processing Unit (GPU)

In their early days, video cards were just devoted to do the essential work to get texts and simple shapes drawn in computer displays, the CPU being in charge of the rest of graphic processing. As times went on, software became more and more complex, and so did the requirements of its graphical user interfaces. These increasing requirements were gradually satisfied by adding specific circuitry units devoted to accelerate particular stages of graphic pipelines to the display adapters. Earlier implementations of these pipelines consisted in fixed units dedicated to their corresponding processing stage, and no custom programming or reconfiguration was allowed on them. These implementations became more and more sophisticated, which led to CPUs being freed from executing graphic tasks. Programming interfaces like OpenGL [95] or Direct3D [77] allowed complex graphic processes like 3D rendering to be directly implemented and executed in video cards. When graphic pipelines became so sophisticated that they needed their own programming mechanisms to squeeze the capabilities of their different units, they became fully operational computing devices and they started to be called Graphics Processing Units (GPUs).

Many tasks run by graphic pipelines units usually consist in parallel processings of huge data collections, such as vector operations. The use of these massive

parallel capabilities of GPUs to compute data streams from non-graphic problems led to the idea of performing General-Purpose Computing on Graphics Processing Units (GPGPUs). At the dawn of this new paradigm, graphic programming interfaces depended on their own programming languages, like OpenGL Shading Language [54], High Level Shader Language [24] or Nvidia Cg [66]. These languages were thought to implement operations on a visualization matrix, so they relied on abstractions like textures, geometries or projections. Such concepts are essential in graphics processing, but their direct application to general purpose computing was, at least, cumbersome, and sometimes even impossible. Despite these difficulties, the potential shown by GPGPU computing encouraged its sustained development. As a result, graphic pipelines with units fully capable of performing general purpose computing tasks appeared, and with them, new programming frameworks like Close To Metal [87], BrookGPU [12], Brook+ [4] and, above all, CUDA [80], which even being a proprietary initiative only supported in NVIDIA devices, is nowadays the largest player in the GPGPU computing market.

Modern accelerators: manycores and SoCs

Any computational problem must be characterized in order to choose the device whose capabilities best match its requirements. For example, a GPU is expected to deal with problems that are intensive in vector operations much better than a CPU. However, branch prediction techniques usually implemented in CPUs make them better to deal with algorithms with complex control flows. Nowadays, programmers are quite used to face the implementation of pipelined problems consisting on several steps run iteratively, each step being often composed in turn of multiple operations from which it is usually possible to extract parallelism in a massive way. This way, there was room in the hardware market for a device that provided the kind of massively parallel capabilities of GPUs, but which were X86-compatible, keeping many of the interesting characteristics of traditional CPUs. Manycore accelerators like they Intel Xeon Phi filled this gap and they offered a solution to compute the aforementioned kind of problems in an integrated way [97]. This accelerator is the evolution of a prior architecture designed by Intel called Larrabee, that was conceived as an attempt to build a GPU-like device by combining several x86 multicore processors. That dual conception allows programmers to exploit parallelism

in Xeon Phi by means of both multithreading (e.g., OpenMP) and multiprocessor (MPI) approaches.

Just as advances in integration levels allowed complete CPUs to be gathered into a same silicon die, manufacturers have also been working on the integration of units originally devoted to different functions in the same die, which lead to the creation of the so-called Systems-on-a-Chip (SoCs). For example, during the last years AMD has worked on the development of their Accelerated Processing Units (APUs), which integrate a multicore CPU and a GPU in a single chip. There has been other noteworthy developments in this field, albeit they are rather oriented to increase the computing power of mobile devices. Examples are the ARM Cortex, NVIDIA Tegra, or Qualcomm Snapdragon families. Interestingly, efforts to exploit the capabilities of these new architectures in high performance computing environments are being made too. For example, AMD, ARM and Qualcomm created the HSA Foundation as a shelter for their Heterogeneous System Architecture initiative [47]. This new architecture generalizes the idea of SoC as a combination of "latency" (i.e., CPUs) and "throughput" (i.e., GPUs) computing units, and provides hardware with a high-level programming infrastructure that relies on a full stack of multiple compilers, intermediate representations and low-level programming languages.

Field Programmable Gate Arrays

There are problems in fields like signal processing, medical imaging, or cryptography, that have very specific computing needs. While CPUs, GPUs or accelerators are able to provide efficient solutions for them, these problems must be also solved in real-time or embedded systems that cannot devote these resources to them. Because of that, these systems resort to specific-purpose integrated circuits to deal with them. However, these algorithms are so specific that such integrated circuits cannot be used to execute other applications, which implies an increased cost in the hardware. Field Programmable Gate Arrays (FPGAs) provide a solution to this issue. They are integrated circuits composed of a number of interconnected logic blocks in which any combinational boolean function can be implemented. Programmers can customize the behavior of these devices by providing an extensive description of both the boolean functions to be implemented by the logical blocks and how they are connected. Manufacturers offer the so-called hardware description languages

(HDLs) in order to define such behavior, VHDL [9] and Verilog [16] being the most common.

Putting things together: heterogeneous clusters

Cluster computing, which is based on interconnecting several computers with multicore processors, is quite popular nowadays. In such systems it is very common to use standards both at the intranode and the inter-node levels. For example, OpenMP is widely used to extract fine-grain parallelism from multicores, while the Message Passing Interface (MPI) [70] coordinates the collaborating nodes. Moreover, during last years adding to these nodes devices like GPUs or other accelerators became a trend, giving place to the so-called heterogeneous clusters. For example, two of top three entries of the November 2016 release [112] of the TOP500 list followed that approach.

1.2. Towards a unified programming approach

Every kind of computing device has its own defining characteristics, and in many cases these characteristics set strict conditions on the mechanisms available to exploit their capabilities. For example, a programmer writing explicit parallel code will have to resort sooner or later to facilities to coordinate parallel tasks. Similarly, physical circuit reconfigurations will be required when working with FPGAs. It has been also already explained that the specific capabilities of each device type must be matched to the characteristics of the problems to be addressed by means of such devices, so that the user will eventually need to deeply dive into the specific properties or architectural details of a particular device in order to efficiently program it. Nevertheless, programmers want to write algorithms in the most simple and intuitive way. For this reason, improving the programmability is a big issue in modern heterogeneous systems.

Some proposals in this field leave compilers in charge of nearly all the details about generating device-specific code, so that users only have to indicate which parts of their programs are going to be parallelized. Such proposals are usually based on directives and they are derived from the OpenMP standard. One example is Ope-

nACC, a directive-based standard [82] generalized for multiple types of massively parallel processors such as GPUs, multicores or manycore accelerators. OpenACC directives can be used in C/C++ or Fortran programs in order to extract parallelism, underlying compilers being responsible of finding particular optimizations compatible for both the language and the architecture. This feature is its main advantage and, at the same time, its main drawback, since programmers must trust blindly in the optimization capabilities of the compiler, and sometimes they are more based in theoretical platitudes than in real device-specific properties. OmpSs [13] is another directive-based proposal to program heterogeneous systems created as an effort to extend OpenMP by supporting complex dependency patterns, heterogeneity [14] and data movement for task parallelism. Concurrently with the development of these new directive-based approaches to heterogeneous systems programming, such capabilities were steadily added to the OpenMP standard itself. Thus, version 4.0 introduced a whole set of directives that allowed users to distribute loop iterations among device threads, to pack those threads in groups called *teams* or to manipulate device-owned data structures. Released in 2015, OpenMP 4.5 is the latest version of the standard and it improves those device memory management capabilities, by complementing directives with explicit routines to allocate, deallocate and map structures to device memory as well as to perform data transfers [85].

Other solutions try to solve the problem by defining an abstraction where all the available devices can fit, together with a common programming model based on it. The OpenCL standard, introduced in [55], proposes a heterogeneous computing architecture that follows that approach. On the hardware side, a computing platform is modeled as a combination of one or more devices. Each device is structured in compute units composed of several processing elements, those elements being in charge of eventually running the computation of a stake of the addressed problem. On the source code side, OpenCL offers a programming model in which each processing element executes several instances of a code snippet called *kernel* that implements the computation to be executed by the device. The way the capabilities of any device or combination of them are exploited depends on how have users written those kernels in order to distribute the solution space of their problems among the processing elements. Moreover, users must also write a *host* code that is in charge of operations such as device lookup and selection, management of input and output memory buffers, or queuing of kernel execution requests. A more detailed

explanation about the OpenCL architecture is provided in Section 2.1.

The flexibility of the OpenCL standard has allowed the manufacturers of different devices to provide their own implementations. Thus, Intel offers drivers and full software development kits [49] for their CPUs (both with and without integrated GPUs), Xeon Phi manycores, and even the FPGAs [50] they inherited from Altera after their acquisition in 2015. Altera was the pioneer company on giving OpenCL support for FPGAs [3]. In a similar way, AMD offers both drivers and the Accelerated Parallel Processing platform for its CPUs, GPUs and APUs [7]. OpenCL support is also included in NVIDIA GPU drivers [79]. ARM does not give support for it in Cortex SoCs yet, although an SDK for Mali GPUs is available [8]. OpenCL code can also be run on Qualcomm Adreno GPUs [90], one of the components of their Snapdragon SoCs. Thanks to such a variety of supported devices, OpenCL has a substantial codebase [45] that covers problems from different academic and industrial fields. There are OpenCL implementations for complex computational physics problems [41], financial simulations [72], mathematical libraries [19], deep learning applications [88], image and signal processing [91], and much more.

OpenCL offers a set of models and programming mechanisms that are generic enough to allow programmers to write any code once, and then run it on any kind of device with OpenCL support [22], although it has been shown that this is not always so simple [102]. Thus, a deeper analysis of the OpenCL codebase shows that in many cases these programs are just OpenCL translations, and they keep too many device-specific details from the original implementation. This is mainly related to how programmers are made responsible for the control of the interactions between the host system and the compute devices available. This control must be implemented in the host code by means of a low-level API that relies on programmers having an exhaustive knowledge about an important number of issues such as devices, kernels, memory objects, or command queues. This leads to quite verbose and error-prone host codes [76]. In other words, the adaptation that host codes need depending on the devices available on each computing platform eventually hinders the expected OpenCL code portability.

Some tools built on top of OpenCL aim to improve its performance portability and, at the same time, its programmability. Some of them are libraries that use OpenCL to implement specific algorithms and operations. For example, Vien-

naCL [94] implements basic linear algebra operations (BLAS). Another example is the mathematical package clMath [6] from AMD, that besides the BLAS operations offers implementations of other operations like sparse algebra routines, fast Fourier transforms or random number generation.

Other tools define skeletons of common computing patterns. The back-end of these approaches generates the corresponding OpenCL code. In these approaches, programmers have to write their codes using these skeletons, which are available through an embedded language or an API. Examples of this are SkelCL [105], SkePU [25] or Marrow [67].

Other group of solutions are domain-specific. For example, PARTANS [64] is based on skeletons to express stencil computations, which are the main building blocks of thermodynamic simulations. HALIDE [91] is a domain-specific language (DSL) for expressing image processing computations.

Other solutions are more focused on making the usage of the OpenCL API easier. For example, PyOpenCL [57] implements a Python version of the OpenCL API, which is simpler than the original ones for C and C++. Other tools [119, 63] go a step further and try to hide the API by automating its management.

There are also approaches based on compiler directives built on top of OpenCL, like accULL [93], an implementation of the OpenACC standard [82] able to extract the parallelization demanded with such directives by means of OpenCL code generated at runtime.

Finally, the Heterogeneous Programming Library (HPL) [114] is a C++ framework that provides an easy and portable way to exploit heterogeneous computing systems on top of the OpenCL standard. In HPL, kernels can be written either in OpenCL [117] or in an embedded language built on top of C++ that follows the same programming model as OpenCL. In the latter case, the effective OpenCL implementation of the kernels is generated at run-time as soon as users invoke their execution using the C++ API provided. Moreover, the HPL API considerably simplifies many of the management operations that make OpenCL host code quite verbose and error-prone. It is worth highlighting the efforts made in order to automate memory management, so that users hardly have to worry about declaring the structures needed for data input and output. Thus, the validity of HPL as a mechanism

to improve OpenCL programmability has been largely proved both in multi-device environments in a single node [116] as well as in heterogeneous clusters [115].

1.3. Performance portability

A truly portable programming approach for heterogeneous systems needs: (1) a unified programming mechanism that allows to implement any kind of problem in any kind of device, and (2) some way to ensure that the code written using such programming mechanism efficiently exploits the capabilities of the available devices.

Several tools introduced in Section 1.2, mainly OpenCL and all the solutions built on top of it, effectively address the first issue, or, in other words, they are functionally portable. The second issue, however, is more difficult to tackle. The only way users have to ensure that their codes are fully optimized for a particular device is to tune them by hand, which requires both a thorough knowledge of hardware architectures and a tough programming effort. Thus, users would considerably benefit from some kind of framework able to tune their codes to some extent depending on the capabilities of the underlying devices. A framework compliant with such features may be termed as performance-portable.

A motivating example on top of OpenCL

OpenCL is said to provide functional portability as, with some limitations, the same kernel code can be run on several devices if they support OpenCL. Unfortunately, OpenCL does not provide automatic performance portability. This way, there is no guarantee that a given OpenCL code will achieve good performance no matter the kind of device, or in devices of the same kind but from different vendors, or even from the same vendor but with different architectural designs [22]. Moreover, a thorough knowledge about both device capabilities and problem characteristics is fundamental to maximize the performance of codes.

Thus, the natural point of departure to tackle this issue is to take the initial naive OpenCL implementation of a problem and try to apply general optimizations that are expected to be beneficial for a particular kind of device. This approach

Kernel version (Listing)	Device type	
	GPU (AMD FirePro S9150)	CPU (Intel Xeon E5-2650v2)
Naive baseline (Listing 1.1)	100%	100%
CPU-friendly (Listing 1.2)	4%	123%
GPU-friendly (Listing 1.3)	701%	73%

Table 1.1: Impact of optimization techniques on naive kernel performance

should improve the performance of the code in that device, but if we try to run this optimized version on another device the performance may not be so good. Furthermore, it is usual that optimizations which are beneficial on some devices, are quite detrimental on others [99, 36].

A running example based on a naive matrix multiplication OpenCL kernel will illustrate this situation. Three versions of the code are used in this test: a naive one, and another two optimized to some extent for an Intel Xeon E5-2650v2 CPU and an AMD FirePro S9150 GPU, respectively. Table 1.1 compares the relative performance of the three implementations on both devices. The naive version is the baseline of this comparison and its performance in both platforms is 100%. The performance of the other two versions is expressed as a relative percentage to this baseline, so that values greater than 100 imply better performance, while values below the baseline indicate slowdowns.

In the naive version, shown in Listing 1.1, the loop in lines 9-10 performs a dot product to compute a single point of the resulting matrix, and work is distributed among as many threads as positions the resulting matrix has. In OpenCL terminology, the lightweight threads that perform the work implemented in a kernel are called *work-items*. Regarding the optimizations, CPU cores are expected to take advantage of coarser-grain work distributions [60, 99]. Since the CPU used in this example is from Intel, its OpenCL optimization guide [51] is a good source for advice about how to achieve such work distribution. A first guideline given in that document is to let the OpenCL runtime infer a suitable amount of *work-items* to be packed together in a *work-group*, which is the term for such packs in OpenCL. The runtime should maximize the work that each core performs by assigning to it work-groups with as many work-items as possible. A fine-grained kernel will maxi-

```
1  __kernel void matmul(__global float* c,
2                       __global float* a, global float* b,
3                       int M, int N, int K)
4  {
5    size_t idx = get_global_id(0);
6    size_t idy = get_global_id(1);
7    int k;
8    (c[idy*N+idx] = 0.0f);
9    for(k=0; k<K; k++)
10     c[idy*N+idx] += a[idy*K+k]*b[k*N+idx];
11 }
```

Listing 1.1: Matrix multiplication OpenCL kernel: naive version

mize the number of work-items available for the OpenCL runtime to pack, and this should favor such a work distribution. The naive version was run in the CPU following this approach. Keeping this idea in mind, the guide also encourages users to try different values for that work-group size, since sometimes configurations outperforming the runtime-selected one can be found by hand. This way, the performance of OpenCL kernels in CPUs can be often improved by manually setting such coarse-grained distributions and applying additional common CPU-friendly optimization techniques.

The code in Listing 1.2 shows a version of the kernel that has been optimized following this latter approach. Thus, the loops in lines 22 and 24 are distributing blocks of 128×64 positions among the threads, which increases the work performed by each work-item. Tiling is another popular technique applicable to this code in CPU because it improves the performance of the cache hierarchy, as information is accessed per tiles. Thus, the dot product is computed in tiles of 8 elements. The loop in line 20 traverses the tiles, and each tile is computed by the loop in line 25. In addition, the naive version stores the result in the original global array `c`, but this is very inefficient because each position of `c` has to be written several times. In this new version, the computation is written on a private array `pc`, which is initialized in lines 14-18, and the results are copied to array `c` at the end of the kernel, in lines 34-38. Table 1.1 shows that this CPU-optimized version runs 23% faster than the naive one on the CPU, whereas it runs 25 times slower on the GPU.

In turn, GPUs prefer the work to be divided among a larger number of threads, each having a lower workload. Moreover, such devices have available an special type

of faster memory, called *local memory* in OpenCL terminology, which is shared among each group of processing elements. Listing 1.3 shows a matrix multiplication optimized for a GPU. The main changes in this version with respect to the CPU-friendly one are that it uses more threads with a finer grain (blocks of 4×4 positions), that the tile size is tuned for the GPU, and that it exploits the aforementioned local memory to optimize accesses to both `a` and `b`. This way, all the threads of the same group collaborate to copy slices of `a` and `b` to their local memory counterparts `la` (lines 27-31) and `lb` (lines 32-36). Line 37 contains a barrier to synchronize all the threads of the same group that collaborate in the copy. The main computation in line 43 uses now these local arrays `la` and `lb` instead of the global ones `a` and `b`.

```
1  __kernel void matmul(__global float* c,
2                       __global float* a, global float* b,
3                       int M, int N, int K)
4  {
5    size_t idx = get_global_id(0);
6    size_t idy = get_global_id(1);
7    size_t szx = get_global_size(0);
8    size_t szy = get_global_size(1);
9
10   __private float pc[128][64];
11
12   int py,px,kk,k,y,x;
13
14   for(py=0; py<128; py++) {
15     for(px=0; px<64; px++) {
16       pc[py][px] = 0.0f);
17     }
18   }
19
20   for(kk=0; kk<K; kk+=8) {
21     py = 0;
22     for(y=idy*128;y<((idy*128)+128);y++) {
23       px = 0;
24       for(x=idx*64;y<((idx*64)+64);x++) {
25         for(k=kk; k<kk+8; k++) {
26           pc[py][px] += a[y*K+k]*b[k*N+x];
27         }
28       }
29       px++;
30     }
31     py++;
32   }
33
34   for(py=0; py<128; py++) {
35     for(px=0; px<64; px++) {
36       c[((idy*128)+py)*N+((idx*64)+px)] = pc[py][px];
37     }
38   }
39 }
```

Listing 1.2: Matrix multiplication OpenCL kernel: CPU-friendly optimizations

```
__kernel void matmul(__global float* c,
                     __global float* a, global float* b,
                     int M, int N, int K)
{
  size_t idx = get_global_id(0);
  size_t idy = get_global_id(1);
  size_t lidx = get_local_id(0);
  size_t lidy = get_local_id(1);
  size_t szx = get_global_size(0);
  size_t szy = get_global_size(1);
  size_t lszx = get_local_size(0);
  size_t lszy = get_local_size(1);

  __local float la[64][32];
  __local float lb[32][64];
  __private float pc[4][4];

  int py,px,kk,k,lr,lc,y,x;

  for(py=0; py<4; py++) {
    for(px=0; px<4; px++) {
      pc[py][px] = 0.0f);
    }
  }

  for(kk=0; kk<K; kk+=32) {
    for(lr=lidy;lr<64;lr+=lszy) {
      for((lc=lidx);lc<32;lc+=lszx)) {
        la[lr][lc] = a[((idy/lszy)*64)+lr][kk+lc];
      }
    }
    for(lr=lidy;lr<32;lr+=lszy) {
      for((lc=lidx);lc<64;lc+=lszx)) }
        lb[lr][lc] = b[kk+lr][((idx/lszx)*64)+lc];
      }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    py = 0;
    for(y=idy*4;y<((idy*4)+4);y++) {
      px = 0;
      for(x=idx*4;y<((idx*4)+4);x++) {
        for(k=0; k<32; k++) {
          pc[py][px] += la[(lidx*4)+py][k]*lb[k][(lidx*4)+px];
        }
      }
      px++;
    }
    py++;
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  for(py=0; py<4; py++) {
    for(px=0; px<4; px++) {
      c[((idy*4)+py)*N+((idx*4)+px)] = pc[py][px];
    }
  }
}
```

Listing 1.3: Matrix multiplication OpenCL kernel: GPU-friendly optimizations

Line 49 contains a barrier to synchronize the threads of the same group before the computation of a new tile of the result starts. Table 1.1 shows that this GPU-friendly version is about 7 times faster than the naive one. However, it suffers from a 27% slowdown in the CPU, probably due to the overriding of the default cache behaviour of the device by the local memory exploitation strategy implemented.

In summary, both CPU and GPU-friendly kernels have been generated by applying suitable optimization techniques to a naive implementation but, when the optimizations applied were not aligned with the capabilities of each device, the performance of the versions obtained decreased notably. This shows how the selection of both the optimization techniques for each device and the configurations to apply them is crucial to achieve good performance across different devices, or, in other words, to achieve performance portability.

A multifaceted domain

As we have just concluded from the previous example, achieving performance portability in heterogeneous environments by means of OpenCL involves finding different versions of kernels that are tuned for the different devices available. Thus, many proposals are built on top of OpenCL aiming not only to enable performance portability but, in some cases, also to automate it or, at least, to provide mechanisms to facilitate such automation.

Multiple aspects characterize these proposals. Some of them are quite domain-specific, whereas others present themselves as extensively general-purpose. Either specific or generic, at some point along their workflows these proposals must implement code generation procedures. Such procedures are needed to transform user inputs to OpenCL equivalents able to run in different devices. Once the code generation mechanism is set, challenges about code optimization arise. The motivating example showed how optimizations can be designed as code transformations depending on parameters, and how by choosing the appropriate optimizations and tuning the values given to such parameters it is possible to generate multiple versions optimized to some extent for different devices. Some common optimization strategies were introduced in the motivating example. In turn, finding proper values for the optimization parameters is a tricky issue that depends noticeably on code proper-

ties and device capabilities, so that proposals usually include more or less complex mechanisms to select tuned values for such parameters. Most of these algorithms perform some kind of evaluation of the versions they generate, real code executions and performance analysis being the most common.

Let us start our review of the related bibliography on performance portability by describing some solutions based on domain-specific languages (DSLs). For instance, HALIDE [91] is a DSL for image processing built on top of C++. It generates optimized code at run-time for multiple kinds of devices on both single and multi-device environments, OpenCL being one of the back-end options for GPUs. Its compiler is driven by an iterative auto-tuner that performs a genetic search to find an optimized schedule for a program. The configuration of each schedule defines the parameter values for the optimizations, which include work distribution, vectorization or loop unrolling. To be tested, schedules are lowered to LLVM intermediate representation, and then the back-end code is generated from it. Research about less time-consuming alternatives for that search process, based on code analysis and user guidance have been also conducted [73].

Other domain-specific proposals expose their capabilities by means of libraries. Such approach is very common in the linear algebra domain, ViennaCL [94] and clBLAS [19] being two representative examples built on top of OpenCL. Research on parametrized optimization of this kind of kernels is extensive, specially for the matrix multiplication routines [23, 59, 68, 69, 92, 107]. The parameters of this code are related to aspects like work distribution, which matrices must be cached in slices to local memory and the size of those slices, vectorization of different stages of the kernel, or loop tiling and unrolling. Thus, exhaustive search processes were run over parametrized implementations of such routine in ViennaCL [110]. Results of this work were processed and bundled into a heuristic auto-tuner distributed with the library. In turn, clBLAS offers default well-performing versions based on general optimizations, although it also includes a profiler that can be run to characterize the available devices and adapt its kernel generation to them.

A third approach for building domain-specific solutions consists in implementing specific optimization strategies for recurring computing patterns from a domain, and then asking users to write their programs in terms of such patterns. For instance, PARTANS [64] is an auto-tuning framework for stencil computations on multi-GPU

systems. Users must write their stencil operators in OpenCL kernels, which can be composed in algorithms and run by means the C++ API provided by the framework. The code of these kernels has to be written in terms of some macros provided by the framework. These macros represent domain-specific concepts like the origin point of the stencil operator or the different offsets from that origin in each dimension. Similarly to what happens in HALIDE, the auto-tuning process does not search directly for optimized values for these macros. Instead, it tunes both the task partitioning and the parameters for other domain-specific optimizations, the values for the macros being inferred from the optimized configuration found. The auto-tuning process in PARTANS has both offline and online stages. One of the steps of the first stage optimizes the task partitioning, the decision depending on exhaustively generating and running all the possible stencils. The online stage, in turn, optimizes some domain-specific aspects, and can be performed by means of exhaustive, hill climbing, or dichotomous search algorithms.

The scope of this approach can be broadened by looking for more recurring patterns. This is the idea behind SkelCL [105], a portable skeleton library for single and multi-GPU [106] environments. It offers data-parallel algorithmic skeletons for map, zip, reduce, scan and allpairs [104] operations, among others, in form of highly optimized OpenCL kernels. Users have to think their algorithms in terms of such operations and write their program using the C++ API provided. By design, this library does not support the parametrized tuning of each individual kernel, but any performance adaptation relies on hard-coded rules to distribute tasks among the available GPUs.

Regarding other high-level solutions, many approaches are able to manage to some extent programs originally written in or on top of common high-level languages like C, C++ or FORTRAN, while others are auto-tuners that expect OpenCL kernels as input. An example of the former is a multi-objective auto-tuning framework developed by Jordan et al. [52] on top of the Insieme [86] compiler infrastructure. This framework can receive as inputs programs written in C, C++, OpenMP, MPI, and OpenCL, which are loaded into an intermediate representation defined by the Insieme compiler. The code is analyzed and decomposed in regions susceptible to be optimized. The optimizations applied to each region depend on parameters like unroll factors or work granularity. An iterative algorithm based on evolutionary

methods and pruning mechanisms looks for tuned configurations for these parameters. Versions are evaluated by running them on the target device. All the search process is performed online as a part of the program compilation, resulting in a number of configurations that are selected and translated to C, OpenCL or MPI code by the compiler back-end. The decision about which version is picked as optimal remains application-specific and could be forwarded to the user.

Fang et al. propose Sesame [36], a framework that bundles knowledge obtained after a systematic study on the optimization space for many-core devices. This study evaluates the impact that the vector capabilities of processors [33] or the usage of local memory chips [32] usually included in heterogeneous devices have on the performance of OpenCL kernels. Regarding local memory exploitation, they build some tools that operate on OpenCL kernels either to enable it [34], or to disable or make it more general [31]. The code analysis and transformation operations performed by these tools are implemented by means of the LLVM infrastructure, whereas the information for auto-tuning comes from micro-benchmarking. The framework expects OpenCL kernels as inputs as long as the included tool does it also, but the extensible design allows to support codes written in other high-level languages.

CLTune [78] is a tool particularly devoted to auto-tune OpenCL kernels in a parametrized way. Users must define which parameters they consider that may affect the performance of their codes, and then refactor their kernels in terms of such parameters. The tool also provides a high level interface that hides some details of the OpenCL host API. Ranges of valid values for each parameter have to be specified through this interface. The strategies implemented to tune the parameter values for a device are a randomized search, a simulated annealing technique and a particle swarm evolutionary algorithm. The three strategies use the execution time of the versions generated as evaluation criteria.

OrCL [17] is an auto-tuner for OpenCL kernels built on top of Orio [44], an extensible optimization and auto-tuning framework. As an extension of Orio, users must annotate their kernels with a thorough description that includes, among other details, the optimization parameters, how these parameters are mapped to the different optimizations supported by the tool, and the performance counters used to evaluate the versions generated. All the code transformation procedures are also inherited from Orio, and they are based on lightweight independent Python modules

rather than on a typical full compiler infrastructure. The optimization parameters can be tuned by means of exhaustive, randomized, simplex, and simulated annealing search algorithms. The versions generated are evaluated through the TAU performance measurement system [100] taking into account their execution times, although users can choose other counters overriding this behavior in their code annotations.

Other high-level solutions, in turn, ask users to rewrite their codes in their own description languages. For instance, the Many-Core Levels (MCL) framework developed by Hijma et al [46], is composed of the Many-Core Programming Language (MCPL), an imperative and C-like embedded language to write kernels for heterogeneous devices, and a compiler able to optimize them with a collection of code transformations ranging from common general optimizations to device-specific tweaks. The framework is also the test bed for the *stepwise-refinement for performance* optimization methodology, which is based in an iterative process in which

	Scope		**Interface**			IR			**Optimization search method**				**Input code support**		
Proposals	Domain-specific	General purpose	Libraries	Languages	Code processing tools	Code-based	Compiler-like	λ-expressions	Exhaustive search	Informed algorithms	Profiling/Heuristics	User-driven process	Full support	Code annotations	Full rewriting
HALIDE	✓			✓			✓			✓					✓
ViennaCL	✓		✓				✓		✓		✓				✓
clBLAS	✓		✓				✓				✓				✓
PARTANS	✓		✓			✓			✓	✓					✓
SkelCL		✓	✓			✓					✓	✓			✓
Jordan et al.		✓			✓		✓		✓	✓		✓	✓		
Sesame		✓			✓		✓				✓		✓		
CLTune		✓			✓	✓			✓	✓					✓
OrCL		✓			✓		✓		✓	✓				✓	
MCL/MCPL		✓		✓			✓					✓			✓
Steuwer et al.		✓		✓				✓	✓		✓				✓

Table 1.2: Summary of performance-portable proposals described

the compiler proposes from more general to more specific optimizations and provides feedback about the potential performance achievable. Once the user picks an optimization, the compiler applies it, and depending on the optimization picked and on the device properties, a new set of more specific optimizations is presented. At the end of the process the kernel is translated into OpenCL or C++ source code.

Steuwer et al. propose in [103] a functional high-level notation to describe problems in a simple way, and a whole set of rewrite rules to transform such a simple description in a dense λ-calculus expression. Each primitive in the expression is mapped to parametrized routines that generate OpenCL code snippets. Such parameters represent, for instance, local workspace sizes or vector lengths. Thus, a working OpenCL code results from the evaluation of the input expression. In order to obtain optimized versions, three consecutive search processes must be performed. First, the search space of some general optimization rules is heuristically pruned. Then, another heuristic is applied to prune the options to implement such rules in OpenCL. Finally, parameter values are selected by pruning again the search space and exhaustively generating and executing all the remaining versions. A performance-portable matrix multiplication has been generated as a test case for this approach both on CPUs and on several desktop [92] and mobile GPUs [107].

1.4. Thesis approaches and contributions

As Table 1.2 summarises, the discussion on the handful of solutions for performance portability in the previous section revealed that there are multiple approaches in this field. The tools presented in this PhD Thesis are built combining some of these approaches, namely:

Source-to-source transformations To be termed as *source-to-source*, any tool of this kind must receive high-level language programs and returns them modified in some way, but still written in either the same or a different high level language. In this Thesis we explore several mechanisms to transform user kernels written in high-level languages into OpenCL versions optimized for different devices.

Parametrized optimizations In general, optimizations depend on some configuration parameters that define whether the input code is going to be transformed or not, and if so, also the way it will perform when executed. Let us remind the motivating example from Section 1.3. There, the transformation applied to adjust work granularity in a kernel was the same no matter the device, but the particular block size was set differently for each device. In this case, the block size worked as a *parameter*. Thus, optimizations of this kind become *parametrized optimizations*.

Search methods to find suitable optimizations Achieving performance portability involves selecting a proper set of optimizations depending on code properties and device capabilities. Since the transformations performed to optimize codes are parametrized, methods to find suitable values for such parameters become fundamental. In this Thesis we explore different options to perform this task, namely exhaustive and informed search algorithms, as well as heuristics based on general strategies to optimize codes for heterogeneous devices.

High-level user interfaces The host API offered by OpenCL relies on many low-level operations, and inexperienced users may struggle to deal with it. This PhD Thesis explores different alternatives to discharge users from these tedious and error-prone tasks or, at least, to reduce them to a minimum extent.

Thus, three different tools intending to improve performance portability on heterogeneous environments were developed. The first one, OCLoptimizer, is built on top of OpenCL and the LLVM-Clang compiler infrastructure [62]. It receives user-annotated OpenCL kernels and a configuration file, and it generates tuned OpenCL kernels. The other two proposals exploit different features of the Heterogeneous Programming Library (HPL). First, we developed self-adaptive HPL kernels. The runtime code generation capabilities of the library allow users to bundle parametrized optimizations in their kernels, and depending on the values set for these parameters, the versions generated can be tuned for different devices. Second, a just-in-time optimizer was embedded in the HPL workflow. Namely, the optimizer modifies the way the library generates OpenCL kernels. Thus, users are expected to write naive HPL kernels that, before being translated into OpenCL code, can be tuned by means of parametrized transformations performed by the optimizer. Parameter values are set

by means of heuristics. Now, the main features of each tool and how it combines the aforementioned approaches to performance portability are discussed.

1.4.1. OCLoptimizer

This tool is a source-to-source optimizer that both receives and produces OpenCL kernels. Its users are expected to annotate their OpenCL kernels with directives that tag the sections to be optimized and specify the techniques to apply. OCLoptimizer also requires configuration files with information about the kernels and the environment. The code analysis and transformation operations were built on top of the LLVM-Clang compiler infrastructure. In order to optimize a kernel, the tool first loads it by means of Clang into an abstract syntax tree (AST). The nodes of this tree include default methods to rewrite it again as OpenCL code. For each optimization supported, the tool implements a version of such methods. This way, in order to generate the optimized version of a kernel, the tool asks the AST of the kernel to rewrite itself applying these methods. When nodes representing annotated sections are visited during the rewriting process, overridden methods are called and sections are rewritten in their optimized form. These optimizations are parametrized, so that codes can be tuned for a particular device by choosing appropriate values for these parameters. The exploration of the search space of the parameter values can be performed using either exhaustive or genetic algorithms. Moreover, since versions are evaluated according to their execution time, the tool is able to automate the generation of working OpenCL host codes. This makes users free from dealing with the tricky OpenCL host API details.

In a first iteration [26], the unroll and unroll-and-jam optimization techniques were included in this tool. This allows to unroll the loops tagged with the corresponding annotation, the unroll factor being the parameter that drives the code rewriting process. The tool supports two search processes to select the unroll factors to apply to each annotated loop, namely either a breadth-first search or a genetic algorithm. In the first case, the optimization space is visited in a loop-by-loop basis: when the first annotated loop is found, it is unrolled and different versions are generated and tested, the rest of the loops remaining untouched. Then, the fastest version, or a number of them that can be chosen by the user in the annotation, is

selected to advance to the next iteration of the optimization process, which starts by unrolling the next annotated loop found in the generated version. The process stops when all the annotated loops have been unrolled. Then, the version with the shortest execution time is selected as optimal. When the search is performed by means of the genetic algorithm, the optimization process is simpler, since the algorithm tries different combinations of factors, applies them to unroll all the annotated loops, and runs the generated version to evaluate it.

After a second iteration [29], two new major features were added. First, the parametrized optimization approach was extended in order to find also optimized configurations for the OpenCL workspace. This optimization is performed prior to the application of loop unrolling, and requires additional information in both the user kernels and the configuration file. Regarding the kernels, users must write them using some special macros. The tools needs these macros to modify the iteration distribution for the target device and the workspace configurations chosen. Both an exhaustive search and a genetic algorithm can be run to find an optimized workspace configuration. The limits of this search process are specified by the user in the tool configuration file. Once an optimized workspace configuration is selected, the kernel with the corresponding iteration distribution and the user annotations is taken as input for the optimization process already implemented in [26]. As a second new feature, the tool was extended with support for optimizing programs composed of several kernels. For this, the workflow of the tool varies depending on whether the input kernels are independent or inter-dependent. When the kernels are independent, separate annotated codes and configuration files must be provided for each kernel. At the end of the process both an optimized workspace configuration and an optimized kernel are identified for each input kernel. Nevertheless, when kernels are inter-dependent, the optimization process must be divided into two steps. First, each kernel is optimized separately using the annotated code and a configuration file as inputs. At the end of this step, the optimized workspace configurations obtained for each kernel are kept, and the optimized versions of the kernels are used as inputs of the second step, in which the annotated kernels are effectively optimized. The complexity of the optimization process for several kernels makes the tool unable to generate a single working host code for the whole application.

1.4.2. Self-adaptive HPL kernels

The Heterogeneous Programming Library (HPL) provides an easy and portable way to exploit heterogeneous computing systems. Kernels can be written in an embedded language built on top of C++, and they follow a programming model based on work-items like that of OpenCL. The effective OpenCL implementation of HPL kernels is generated at run-time when users request their execution by means of the high-level C++ API provided.

Users can exploit the run-time code generation (RTCG) capabilities of the library by combining properly in their kernels sentences written in the embedded language and in regular C++. Thus, sentences involving data types and functions from the embedded language are translated into OpenCL code, whereas those written in regular C++ can be used to control some aspects of the OpenCL code generation. This allows users, for example, to select one among a number of different HPL code snippets to be eventually translated. Notice that decisions about code generation taken in these C++ sentences can be somehow parametrized, for example, depending on the kind of device. This feature opened the door for parametrizing optimizations inside HPL kernels.

Thus, in a first iteration [28], a first set of parametrized optimizations were designed and used to implement a matrix multiplication kernel in HPL. In this first experience with HPL, depending on the values assigned to the corresponding parameters, work distribution can be adjusted, compute loops can be both tiled and unrolled, and none, one or both input matrices can be copied in chunks to the local memory. Experiments performed with the OCLoptimizer tool showed that exhaustive search methods could be extremely time-consuming. For this reason, in this case the values for a total of ten optimization parameters are found only by means of a genetic algorithm. Thus, each time the algorithm needs to test a combination of values, it launches the HPL kernel to execution. The library generates the corresponding OpenCL version according to the values set for the parameters, and then runs it in the target device. As for the genetic algorithm, as usual, the faster the generated version is, the better. Thus, by finding proper values for the parameters, the kernel is able to adapt by itself to the capabilities of different target devices.

In a second iteration [27, 30], new parametrized optimizations were designed and included in the self-adaptive matrix multiplication kernel. Here, loop interchange and instruction scheduling optimizations were also applied to the compute loops of the kernel. Moreover, both local memory accesses and compute loops could be vectorized with independent vector lengths. The genetic search algorithm was updated to take into account the parameters of these new optimizations, now increased up to a total of fourteen. Also, some restrictions to avoid illegal parameter combinations and to prune the search space are added. In order to assess the validity of this solution, the performance of both the adaptation process and the generated kernels was compared to those of ViennaCL 1.5.1 and clBLAS 2.4.

1.4.3. HPL-embedded just-in-time optimizer

In the original implementation of HPL the user kernels written with its embedded language are translated at run-time into OpenCL code by means of the Portable Expression Template Engine (PETE) [43]. In a few words, as the engine parses each expression in the input kernel, it generates its OpenCL string equivalent. On top of this process, a class in HPL gathers the equivalent strings of the expressions into a working OpenCL kernel. Our just-in-time optimizer modifies this behavior, using PETE to load the input HPL kernel into an abstract syntax tree (AST). This tree is implemented following a typical composition design pattern [39]. Each node class from the AST hierarchy implements a method that recursively asks its children to emit their OpenCL equivalents. When a node receives the strings from its children, it composes its own string and returns it up in the tree. Thus, invoking this method for the root node eventually generates a full OpenCL kernel.

With this modification, HPL kernels are loaded into an AST before their translation to OpenCL code, which gives room to implement parametrized optimizations as transformations on the tree. The optimizer includes transformations to tile and unroll compute loops, to cache some structures in local memory, to adjust the work granularity and to exploit the private memory of the devices. These transformations are driven by both the global and local workspace configurations, the dimensions of the blocks of iterations assigned to each work-item, tile widths, and unroll factors. These optimizations have been selected considering well-known basic strategies

recommended for improving codes for heterogeneous devices. The values for the optimization parameters are set by means of heuristics based on general optimization strategies for heterogeneous devices. Additional data about the naive kernel, such as the problem sizes, or the lengths of the loops inside the compute section, are also needed to optimize the code.

To get their kernels processed by the just-in-time optimizer, HPL programmers have to write them naively, just encoding the calculation of one point of the solution and with no optimization features. They also have to enclose the code that computes that single point inside a *compute* section, leaving variable declarations and other parts of the kernel outside of it. The optimizer takes advantage of this hint, which allows to simplify the process. Before the optimization process starts, the AST has to be populated with information about the access patterns that appear in the code. Also, notice that the optimizations implemented are inter-dependent, and because of that, they are applied in an experimentally fixed order. Moreover, both the way the optimization process is implemented as well as the replacement of time-consuming search algorithms with heuristics to select the optimizations parameters, make this optimizer very lightweight so that its execution does not overshadow the performance improvements achieved by the tuned kernels it generates.

Just as in the case of the two former tools, this optimizer keeps the source-to-source approach by expecting HPL naive kernels and generating OpenCL optimized versions. The optimizations are applied by means of transformations performed on the kernel AST that are driven by multiple parameters, so that the parametrized optimizations approach is followed too. Regarding the strategy to choose values for those parameters, the usage of heuristics removes all the drawbacks associated to time-consuming search algorithms. Finally, users are asked to write naive single-point kernels, which is consistent with a proven high-level approach for programming heterogeneous devices like HPL. In summary, this proposal is built following a combination of approaches that provides performance portability, and thus embedding it in the HPL workflow turns this tool into a performance-portable framework.

Chapter 2

OCLoptimizer

At the outset of the heterogeneous computing, it was quite common that along with each different new device or architecture, manufacturers released also specific programming frameworks for them. For instance, the Cell multicore processor was launched along with its own set of C/C++ language extensions [48]. In a similar vein, the different vendors of GPUs launched their own solutions, such as CUDA [81] from NVIDIA or Close To Metal [87] from ATI. Some initial efforts were made too in order to enable functional portability for different devices. Thus, the BrookGPU [12] compiler for the Brook programming language intended to provide a unified GPGPU programming framework based on the OpenGL, DirectX and Close To Metal interfaces.

However, the first initiative that really offered a unified solution to that issue was the OpenCL standard [55]. The design of this framework provides functional portability, i.e., it allows programmers to write a single code once and run it on multiple kinds of devices. Moreover, OpenCL was launched by The Khronos Group, an industrial consortium gathering multiple hardware and software manufacturers. Such an industrial endorsement and the aforementioned code portability made OpenCL one of the most flexible options to program heterogeneous devices. Nevertheless, research on this field continued to evolve, and nowadays it is directed to build higher-level frameworks or, at least, to widen the scope of some popular solutions already available. For instance, AMD is working on the Radeon Open Computing Platform (ROCm) [5]. This platform gathers, among other components, the Het-

erogeneous Compute Compiler (HCC), which is a single source C++ compiler for both CPUs and GPUs, and the Heterogeneous-Compute Interface for Portability (HIP), a high-level programming framework that will enable the development of virtually GPU-universal applications by using either the aforementioned HCC or the NVIDIA CUDA compiler as back-ends. Despite this trend, the flexibility of OpenCL still makes it an interesting mechanism to program heterogeneous systems, to the extent of, for example, being considered a feasible candidate to supersede the typical programming approaches for FPGAs.

Unfortunately, OpenCL applications that perform adequately on a given device often require major changes even just to perform reasonably well in others [58, 60, 99]. As a result, codes that are functionally portable by design must be hand-tuned by their programmers for different target devices. In other words, the OpenCL standard enables performance portability on its top, but such a feature is not accomplished automatically.

This chapter presents OCLoptimizer, the first solution developed in this thesis to achieve performance portability. This proposal is a source-to-source iterative optimization tool. Its inputs are an annotated OpenCL kernel, and a configuration file that will guide the optimization process. OCLoptimizer uses this information to generate an optimized version of the input kernel for a selected device, as well as a fully working host code for it. This host code comes with a default data initialization routine which can be overridden by the user in order to adapt the code for her needs. The tool intends to release the programmer from two difficult and error-prone tasks: (1) hand-optimizing the kernel for a given device, and (2) writing its associated host code. The annotations introduced in the kernel take the form of compiler directives used to specify parametrized code transformations. Internally, when an specific device is targeted, OCLoptimizer follows an iterative search process that searches an optimized combination of transformations and values for their associated parameters. Another interesting feature of OCLoptimizer is that it supports the optimization of OpenCL codes composed of several kernels, both independent and inter-dependent. In this latter case, dependencies among kernels are properly taken into account.

The rest of this chapter is organized as follows. Sections 2.1 and 2.2 present the OpenCL standard and the Clang front-end for the LLVM compiler framework,

respectively, which are the two technologies OCLoptimizer is built on. Section 2.3 describes the OCLoptimizer tool, including both its inputs and its workflow. Section 2.4 describes how the tool also supports OpenCL codes composed of multiple kernels. Section 2.5 presents the experimental results. Then, in Section 2.6 we expose our conclusions about the development of this tool, followed by a discussion on related work in Section 2.7.

2.1. The OpenCL standard

The Open Computing Language (OpenCL [55]) is an standard that defines a framework for programming heterogeneous systems. It was created by The Khronos Group, an industrial consortium devoted to the creation of free open standards for, among other fields, parallel computing on multiple platforms and devices. Thus, OpenCL was the first industry standard directly addressing the heterogeneous computing challenges when its first version, OpenCL 1.0, was released in December 2008. The standard kept evolving since then, and nowadays the community works on a definitive release of the OpenCL 2.2 specification, which was initially published in March 2016.

OpenCL allows users to exploit the capabilities of multiple devices present in quite different (i.e., *heterogeneous*) systems, from GPUs included in smartphones and desktop CPUs to many-core accelerators included in the nodes of many modern supercomputers. The main advantage of OpenCL is its functional portability, so that a single OpenCL program can be run in such a wide range of devices. This is possible thanks to its two-layered design. On the one hand, the standard defines a software layer controlled by the programmers, and on the other hand, manufacturers are in charge of the hardware layer. Manufacturers are responsible of implementing the standard properly, although they can provide additional device- or vendor-specific features. All the hardware implementation details, such as drivers and runtime, are transparent for programmers.

The OpenCL specification is defined in four parts called models: the *platform* model, the *execution* model, the *memory* model and the *programming* model. The rest of this section is devoted to describe the main properties of these models.

2.1.1. The Platform Model

The OpenCL platform model defines a high-level abstraction that gathers one or more OpenCL-compatible devices. Figure 2.1 shows the main components of this model, and how they are related to each other. First, there is a *host*, which is a computer with a CPU. This host controls the interaction of the whole platform with the external environment, being in charge of operations such the I/O management. One or more OpenCL-supporting *devices* can be connected to the host. Each device consists of one or more *compute units*, which are further divided into one or several *processing elements*. Such elements are characterized by executing SIMD (Single Instruction, Multiple Data) instructions, so that only one instruction is run simultaneously in several processing elements.

2.1.2. The Execution Model

The OpenCL execution model defines the parts that compose any OpenCL application: the host program, and one or several kernels. Kernels are functions written in a C99-based language provided by OpenCL called OpenCL C. These functions are run in parallel by the *processing elements* of a device in order to, in a few words, process some input memory objects to generate their corresponding outputs. The host program is the main program of the application. As it can be inferred from its name, it runs on the host CPU and it is in charge of tasks such as defining contexts for the OpenCL devices and commanding those devices to execute kernels.

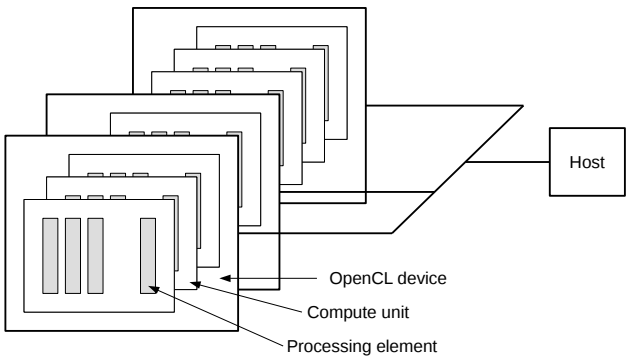


Figure 2.1: OpenCL platform model

Kernels

When the host submits a kernel for execution on an OpenCL device, the runtime defines an n-dimensional integer workspace. This workspace can have up to three dimensions. An instance of the submitted kernel, called *work-item*, is created for each point in this space. Each work-item can be identified by its coordinates in the workspace, such coordinates being known as the *global ID*. These work-items can be organized to form *work-groups*, each group having its own local index space. Thus, for each work-item in a group there is also a *local ID* referring to its coordinates in the local index space. Work-groups provide a coarser decomposition of the workspace, and they can be also identified as a whole with a *work-group ID*. Moreover, work-items gathered in a same work-group share several noteworthy properties. First, the execution of the work-items gathered in a same group can not be split among different compute units. Second, the work-items in a group share some processor resources on the device, namely a local *on-chip* memory. This memory allows the work-items to access common data very fast. Finally, only the work-items sharing a work-group can be synchronized. Namely, programmers can set barriers in their kernels to force that synchronization.

Figure 2.2 depicts an example of a two-dimensional global workspace composed of 16×16 work-items. This space is divided in 16 work-groups of 4×4 work-items. The group containing the highlighted work-item has a work-group ID $(w_x, w_y) = (3, 2)$, and inside that group, that work-item has a local ID $(l_x, l_y) = (1, 2)$. Regarding the global workspace, it can be also identified by the global ID $(g_x, g_y) = (13, 10)$. Let us notice that the identifiers of the example does not follow the (y, x) order expected for a matrix representation, but the inverse (x, y) form, as the OpenCL standard specification follows this latter one.

Host Program

The host program is in charge of tasks that are fundamental to run any OpenCL kernel. It establishes the environment within which the kernels are defined and executed. This environment is called *context*, and it is created and manipulated by means of some functions of the OpenCL API. The contexts are defined in terms of these resources:

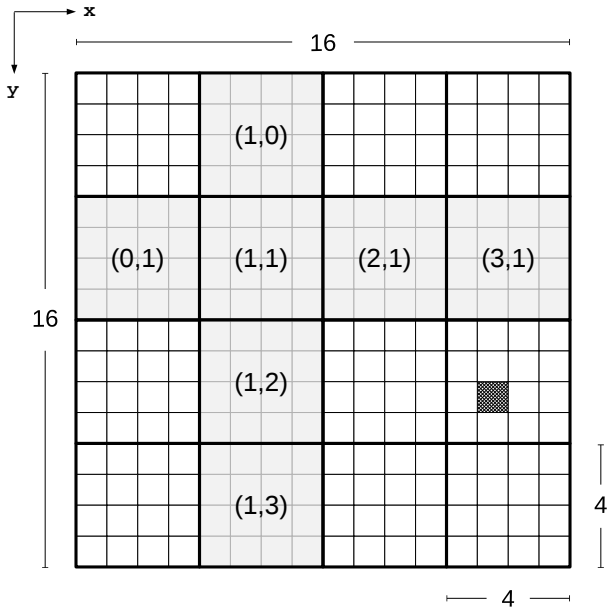


Figure 2.2: OpenCL workspace example

- **Devices**: A collection of OpenCL devices on which the host can launch the execution of a kernel.
- **Kernels**: Functions written in the C-based OpenCL programming language and which will be run on devices.
- **Program objects**: Source code and binaries implementing a kernel or a collection of them.
- **Memory objects**: A collection of memory buffers that can be operated either by the host program, or the devices by means of the kernels, or both.
- **Command queues**: Objects through which the interaction of the host and the devices occur. They are in charge of submitting commands to devices.

Once a context is created, the user can create command queues to control the execution of kernels on the OpenCL devices registered in that context. These queues accept three different kinds of commands:

- **Kernel execution commands**, which request the execution of a kernel on a device.
- **Memory commands**, which are responsible of data transfers between the memory visible by the host program and the memory of the devices.

- **Synchronization commands**, which allow users to manipulate the order in which any other commands are executed.

These commands can also affect the execution of the host program depending on whether they are enqueued as blocking or non-blocking commands. In the first case, the execution of the host program will be blocked until the completion of the command. Otherwise, the host program simply continues its execution just after the command is enqueued.

2.1.3. The Memory Model

The memory model of OpenCL covers issues like how the data objects manipulated by both the host program and the kernels are defined, the scope up to which these objects are visible, or the rules to use them safely. All these interactions, which are summarized in the scheme of Figure 2.3, pivot around an scheme of five distinct memory regions: *Host*, *Global*, *Constant*, *Local* and *Private* memory. These regions are described now in turn.

Host Memory

A relevant stake of OpenCL devices are accelerators that usually operate memory systems separated from that of the host CPU. Moreover, OpenCL concurrency relies on a relaxed consistency model based on events that notify about the completion of the enqueued commands and on barrier synchronizations performed in the kernels. In order to support these features, the memory objects manipulated by an OpenCL host program have to be defined in a separated space from the host CPU main memory.

Global Memory

Any work-item can read and write random memory positions in this region, which is the main memory space of the device and can span up to several GB. It also works as a gateway through which the devices can send to or receive data from

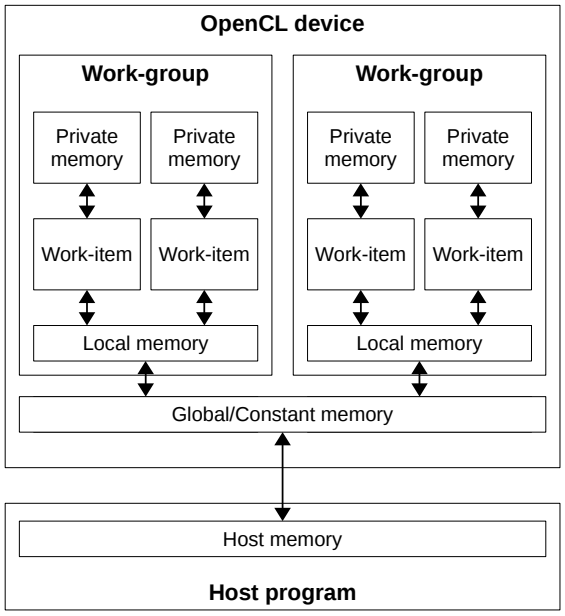


Figure 2.3: OpenCL memory model

the host program. In modern accelerators and processors, this memory space may be a memory hierarchy with several levels of cache. Such a memory layout intends to mitigate the time penalties that might arise when a kernel performs global memory accesses lacking of either locality, or coalescence, or both. The term coalescence is related to the ability of many devices to pack several memory accesses into a reduced number of memory transactions. In order to be coalesced, global memory accesses must follow a pattern meeting two essential conditions: consecutive work-items must access consecutive positions in global memory, and the region they are accessing must be properly aligned. Let us add that work-items follow a row-major order in both global and local workspaces, and that a memory region is aligned when its size is a multiple of the data type size.

Constant Memory

As its name implies, this memory is constant for the kernels, all the changes on it being performed by the host application. It is very useful to store small sets of constant values whose allocation in the global memory could hamper the exploitation of cache hierarchies or hide potential coalesced accesses.

Local Memory

Only the work-items packed into the same work-group can interact with this region, which is usually implemented as a very fast *on-chip* memory. All the work-items in a group share the same view of this memory space, so it is usually exploited as a scratchpad memory. Coherency issues arising because of the concurrent modification of this memory by the work-items that share it can be solved by means of the aforementioned work-group level synchronization barriers.

Private Memory

In general, data is stored in private memory when it is not stored in any of the other memory regions. In principle, this data would be placed in processor registers. However, both space and capabilities of this kind of storage are limited, which can give place to a *register spilling* problem. When registers are overused, or an array is declared as private and the device is not able to dynamically index its registers, private data will be pushed to global memory. This situation may cause important performance penalties.

2.1.4. Programming models supported

OpenCL was designed having in mind both data- and task-parallel programming models. Data-parallel algorithms are usually defined in terms of concurrent computations that perform the same operation(s) on each element of a set of data structures. Such problems naturally fit the execution model of OpenCL, since users can define workspaces matching the layout of their data structures, and then create work-items so that each one of them runs an instance of the kernel on an element, or group of elements, of these structures. Although data parallelism is the main target of the OpenCL execution model, task-parallel algorithms can be also supported. Thus, users can define kernels that execute a single work-item but extract parallelism by other means, such as vector operations. A kernel defined in such terms can be enqueued to run on a device as a task. Also tasks representing different computations can be executed in parallel in an OpenCL application. This way, task parallelism can be extracted by combining and synchronizing the execution of multiple kernels.

A data-parallel example: vector addition

The code shown in Listing 2.1 defines a vector addition kernel in OpenCL and implements a host program to run it in a GPU available in the system. The kernel of this example gets two *float* vectors as inputs, `a` and `b`, and computes `a + b` to generate another `float` vector `c` as an output. Some omissions and simplifications are done for the sake of clarity, such as not including error checks or assuming that there is a single platform and that it provides access only to a GPU.

Actually, the vector addition kernel implementation corresponds only to the string defined in lines 6-10. This kernel is a function that uses two input vectors `a` and `b` defined in global memory, and another vector `c`, also defined in global memory, which will hold the result. Notice that each argument, expressed by a pointer to the beginning of the vector, is modified with the keyword `__global` in order to specify the memory region to which the associated data structure belongs. The code of this kernel specifies that each work-item performs the addition of a single element of the array `a` and another one in the same position of array `b`, and then stores the result in the associated position of the array `c`. Function `get_global_id(int dim)` returns the index of a work-item in the global workspace in the dimension `dim`. Since this problem has a single dimension, and dimensions begin to count from 0, here `dim`=0.

The `main()` function implements the host program. The first step in that function, performed in line 15, consists in choosing a platform from those found in the system. Once a platform is selected, we can inspect it and pick a device to run our kernels. This is done in line 17. Namely, we are getting the identifier of the first GPU available in the platform. In line 18, the selected device is included in a context, which is used in the next line to define a command queue. As a result of these operations, the minimum environment needed to run our kernel is ready.

The kernel source code, contained in the string defined in lines 6-10, must be transformed into an executable binary. This is done by the functions called in line 21, which creates a program object from the string, and in line 22, which compiles the program for the chosen device. In line 23, a kernel object, which will be used for associating arguments to the kernel and later requesting its execution, is obtained from the program. In this example, for simplicity reasons, the kernel function is hard-coded into the aforementioned string declared in the host code. However, it

```
1  #include <CL/cl.h>
2  #include <stdio.h>
3
4  #define VECLEN 1024
5
6  const char = *kernel_code =
7  "__kernel void addvec(__global float *a,__global float *b,__global float *c) { \n \
8    int i = get_global_size(0); \n \
9    c[i] = a[i] + b[i]; \n \
10 }";
11
12 int main(int argc, char** argv)
13 {
14   cl_platform_id platform;
15   clGetPlatformIDs(1,&platform,NULL);
16   cl_device_id device;
17   clGetDeviceIDs(platform,CL_DEVICE_TYPE_GPU,1,&device, NULL);
18   cl_context context = clCreateContext(NULL,1,&device,0,NULL);
19   cl_command_queue queue = clCreateCommandQueue(context,device,0,NULL);
20
21   cl_program program = clCreateProgramWithSource(context,1,&kernel_code,NULL,NULL);
22   clBuildProgram(program,1,&device,NULL,NULL,NULL);
23   cl_kernel kernel = clCreateKernel(program,"vecadd",NULL);
24
25   int size = VECLEN * sizeof(int);
26   cl_mem a_buffer = clCreateBuffer(context,CL_MEM_READ_ONLY,size,NULL,NULL);
27   cl_mem b_buffer = clCreateBuffer(context,CL_MEM_READ_ONLY,size,NULL,NULL);
28   cl_mem c_buffer = clCreateBuffer(context,CL_MEM_WRITE_ONLY,size,NULL,NULL);
29
30   int *a_host = (int*)malloc(size);
31   int *b_host = (int*)malloc(size);
32   int *c_host = (int*)malloc(size);
33
34   { ... } // a_host and b_host initializations
35
36   clEnqueueWriteBuffer(queue,a_buffer,CL_TRUE,0,size,a_host,0,NULL,NULL);
37   clEnqueueWriteBuffer(queue,b_buffer,CL_TRUE,0,size,b_host,0,NULL,NULL);
38
39   size_t global_work_size = VECLEN;
40   clSetKernelArg(kernel,0,sizeof(cl_mem),&a_buffer);
41   clSetKernelArg(kernel,1,sizeof(cl_mem),&b_buffer);
42   clSetKernelArg(kernel,2,sizeof(cl_mem),&c_buffer);
43   clEnqueueNDRangeKernel(queue,kernel,1,NULL,&global_work_size,NULL,0,NULL,NULL);
44
45   clEnqueueReadBuffer(queue,c_buffer,CL_TRUE,size,c_host,0,NULL,NULL);
46
47   { ... } // c_host usage
48
49   return 0;
50 }
```

Listing 2.1: OpenCL example: vector addition

can be also read from a separate text file. This latter approach is common in real applications, which are composed of several kernels and/or kernels more complex than the one in this simple example.

As commented in Section 2.1.3, the host program manipulates memory objects that are defined in the memory space of the device, which is typically separate from the host CPU. Lines 26-28 define the memory buffers required for the execution of our kernel. The function for buffer creation includes parameters such as the context the buffer belongs to, or its access type in the kernel. The most common access types are read-only, write-only and read-write, respectively identified by the constants `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY`, and `CL_MEM_READ_WRITE`.

In lines 36-37 the content of these buffers is initialized in the device memory by copying the input values that originally reside in the host memory. This is achieved by means of the `clEnqueueWriteBuffer` function. In line 36, this command is enqueued in the `queue` passed as argument to copy the data pointed by the host memory pointer `a_host` in the device buffer `a_buffer`. In line 37, the copy of data pointed by `b_host` to the buffer `b_buffer` is enqueued.

The execution parameters of the kernel are set in lines 39-43. First, the size of a unidimensional workspace with `VECLEN` work-items is defined in line 39. Then, lines 40-42 specify the arguments passed to the kernel, this is, the input buffers `a_buffer` and `a_buffer`, and the ouput buffer `c_buffer`. Finally, in line 43, a command to execute the program stored on the `kernel` object is enqueued. This is done by calling the `clEnqueueNDRangeKernel` function, whose arguments specify, among other relevant parameters, the `queue` which will run the kernel and the global workspace definition. In this case there is no local workspace defined (the value for the sixth argument is `NULL`). In such a case, the OpenCL runtime is left in charge of choosing a local domain suitable for the properties of the target device.

Commands to launch kernels are non-blocking operations. Because of that, beyond this point the kernel runs in the GPU while the rest of the execution of host code continues on the host CPU. Any further code using the result stored in `c_buffer` (represented in the example with an elision in line 47) would need to retrieve it from the device first. This transfer of the data of `c_buffer` to the host memory structure pointed by `c_host` is done by calling the function `clEnqueueReadBuffer` in line 45, which enqueues a command to read this data back. The synchronization behavior of commands like `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` can be specified by means of a block setting flag passed in their respective third arguments. Thus, when this flag is `CL_TRUE`, the host application will wait for the

completion of the command. Otherwise, the control is returned to the host immediately after the command is enqueued. In this example, the host will be waiting for the `enqueueReadBuffer` command to finish, i.e., it will wait until the data transfer is completed.

From this example it can be concluded that writing a working host code for a kernel is a task composed of several well-defined steps: context definition, program compilation, memory buffers definition, transfer of input buffers to the device, kernel execution enqueuing, and transfer of the output buffers to the host. As we can see, all these steps are performed by means of functions offered by the OpenCL host API, and many of them rely on details that non-experienced users may easily omit or misunderstand. Thus, one of the aims of the OCLoptimizer tool presented in this chapter is to help users to get rid of these repetitive and error-prone tasks by automating the development of the host code.

2.2. LLVM and the CLANG front-end

In this section a brief description of the LLVM compiler infrastructure and of Clang, its front-end for C/C++ source code, is made. The code analysis and transformation operations performed in OCLoptimizer are implemented on top of this toolchain. Thus, also a quick introduction about how source-to-source transformations can be performed using Clang to rewrite source code is given. Finally, some issues related to the version of the Clang distribution included in OCLoptimizer are discussed.

2.2.1. The LLVM compiler framework

The Low-Level Virtual Machine (LLVM) is a compiler framework designed to support transparent program analysis and transformation operations for arbitrary software [62]. It is based on a *lifelong* approach that intends to maximize the chances for optimizing a program along all the phases of its life cycle, from compile- and link-time to run-time, and even at idle-time between runs. LLVM achieves that objective through two components. First, a compiler designed in such a way that it can go back

and forth along the program timeline as needed to perform any kind of optimization. Second, an internal code representation (called LLVM internal representation, or LLVM IR) which is abstract enough to support such lifelong optimizations. This code representation is built from an abstract RISC-like instruction set enriched with higher-level information about data types or both data and control flow of programs. It does not represent high-level language constructs, which makes it source-language indepedent, and at the same time it is able to capture the key operations of processors, but without being subject to machine-specific constraints.

The most popular design for a traditional static compiler is a three-phase approach whose major components are a front-end, an optimizer and a back-end. This design can be generalized both to support codes written in multiple source languages and to generate code for multiple target machines. Figure 2.4 shows how the LLVM compiler framework is designed in such vein [61]. Let us focus on the front-ends for the different source code languages, which must be provided externally. These front-ends are responsible for parsing, validating and diagnosing errors in the input code, and then translating it into the LLVM IR. Usually, these front-ends build first an abstract syntax tree (AST), on which they can perform some compile-time language-specific optimizations. Next, the tree is converted to LLVM representation. By means of different front-ends, LLVM supports compiling Ada, C/C++, D, Delphi, Fortran, Haskell, Objective-C and Swift, among other languages. Some of them were derived from those implemented in the GCC Compiler Collection (GCC). Others, in turn, are original developments, such as the Clang front-end for C/C++ and some extensions of those languages.

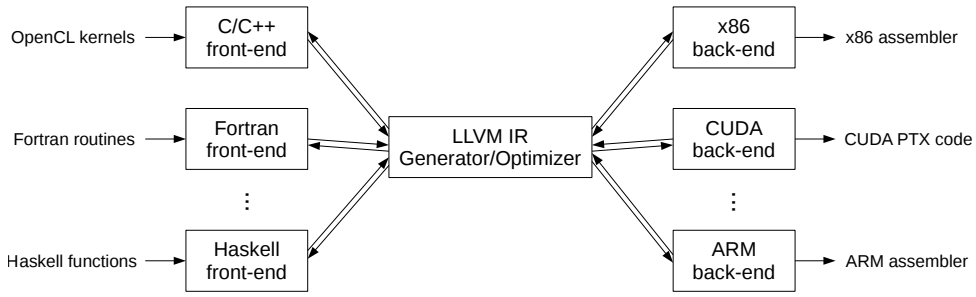


Figure 2.4: LLVM compiler framework general design

2.2.2. The Clang front-end

Clang was born as an LLVM front-end for the C, C++ and Objective-C/C++ programming languages. It has also been enriched with support for some C-based parallel programming frameworks, namely OpenMP, OpenCL and CUDA. Clang also includes a code static analyzer, and some programmer tools built on top of it. The Clang design was thought by a team of Apple developers that needed a compiler front-end more tailored to their software projects than GCC. In particular, Clang is designed to collect more information at compile-time and also to keep the form of the original code as long as possible along all its life cycle.

The front-end capabilities of Clang rely on a basic module that implements common functionalities such as source file management or error diagnostics. This module is complemented with another one that provides support for all the LLVM-related features. On top of both, the typical sequential workflow of a compiler front-end is built. Thus, in a first stage, a lexical analysis of the string of code is performed. Then, the identified tokens are syntactically analyzed by a parser and, finally, an abstract syntax tree (AST) representing the code is generated. The code is loaded into the tree in terms of both variables and types declarations, multiple kinds of types (built-in, functions, arrays, pointers), and statements, which are further divided into the usual code constructs representing alternative and repetitive programming structures, and expressions such as literals, function calls, array references, or n-ary operators. Figure 2.5 depicts this high-level description of the architecture of Clang.

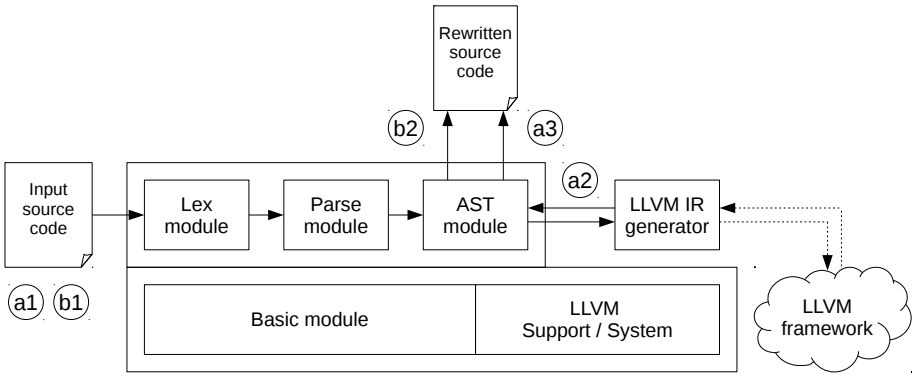


Figure 2.5: Clang high-level architecture and workflow

Source-to-source transformations

Both the main design goals and the high-level architecture described make Clang particularly suitable to support code refactoring tasks following a source-to-source approach. Clang offers multiple options to perform such tasks, all of them starting by providing an input code file, as the steps *a1* and *b1* of the workflow of Figure 2.5 shows.

One possibility is to use Clang as a black box and directly transform the C code into its LLVM IR equivalent, make some transformations on top of that representation, retrace our steps to obtain a new AST from the modified LLVM IR, and finally rewrite that tree into a transformed version of the C code. This way of operation is represented by steps *a2* and *a3* in the workflow of Figure 2.5. Let us remind that the LLVM IR abstraction is partially achieved by not storing source-language constructs, so that the resulting C code is not likely to resemble the original input and, hence, it might be hardly understood by its programmer. Another possibility is to exploit the front-end internals and, as the step *b2* of Figure 2.5 shows, use the functionalities provided by the AST module to perform some transformations on the code. As long as it has not been processed outside Clang, this tree will keep all the source-language-related information, and thanks to this, the rewritten source code will resemble the original input.

OCLoptimizer was intended as a source-to-source optimization tool for OpenCL kernels. Therefore, a mechanism able to take an input code, load it into a manipulable intermediate form, and transform somehow such representation to eventually produce a human-readable optimized version seems to be a valid solution to implement it. The first two stages of such optimization process could be performed equally by any of the two source-to-source transformation approaches described. However, we are specially interested in producing code as much human-readable as possible, which makes the second approach the most suitable for that purpose.

Once the code is loaded into its tree-like representation, the AST module of Clang offers different possibilities to transform it. One can choose to transform the tree directly by replacing the subtree enclosing an objective code section with its optimized counterpart. This solution requires a thorough knowledge of the Clang AST class hierarchy, since errors on the composition of the transformed tree could

lead to incorrect or even non-working kernels. Another option consists in keeping the tree unaltered and postponing the code optimization transformations until the final rewriting step. By keeping the AST, we can still identify the subtrees that represent the code sections to optimize, and then we can extend the default rewriting procedure of Clang to capture such cases and produce our own optimized versions of the original code sections. This implies a direct manipulation of code strings in order to generate optimized versions, which can be also quite error-prone. However, OpenCL programming knowledge is enough to implement it. So, this latter alternative was the chosen one.

Rewriting code with Clang

The AST module of Clang provides the class `ASTConsumer`, which implements a basic processor for any tree generated from an input code. Any user-written class intended to manipulate an AST must extend `ASTConsumer`, since it gives access to the method that launches the recursive traversal of the tree. Once the mechanisms to access and traverse a tree are set, the next step consists in extending the default behavior of the consumer when it visits the nodes during the traversal of the AST. Clang also provides the `Visitor` helper classes, which have specialized implementations to visit different kinds of nodes. The user-written consumer can extend some of these classes and override their `Visit` methods, which will be invoked when the corresponding nodes are visited. Thus, by overriding these methods it is possible to enrich the tree traversal with operations that manipulate the nodes. In this case, we are interested in rewriting the code. To do that, such overridden versions of the `Visit` methods must obtain the string equivalents by calling the `ConvertToString()` method of the corresponding node. These strings are just copies, so that any modification on them will not give place to side effects anywhere else, the original AST remaining untouched. The effective rewriting operation has to be performed through the class `Rewriter`. This class provides a `ReplaceText()` method, which registers a new equivalent for a given node. Once all the tree has been traversed and the refined visitors have obtained and registered their new strings, the transformed version of the input code can be generated. When this final version is ready, it is stored in a `RewriteBuffer`, from which the string containing the whole code can be written to a file.

Version-related issues

The preceding explanations about the source-to-source transformation options and the code rewriting procedures are based on LLVM 3.0 and Clang 3.0, which were the releases available when the development of the first iteration of OCLoptimizer started. Let us notice that both the LLVM compiler framework and the Clang front-end are very successful and dynamic open-source community-driven projects. As a result, their capabilities and the structure of the internal classes implementing them are continuously evolving, and thus, such explanations and, specially, the references to particular classes or methods, are likely to be outdated. Nevertheless, we consider them useful to illustrate how the source-to-source transformations performed by OCLoptimizer are implemented.

Furthermore, although the support for OpenCL kernels provided by Clang 3.0 seemed enough to start, it was not full yet. Thus, some limitations arose when parsing the OCLoptimizer annotations and occurrences of the OpenCL vector types. These limitations were averted by means of workarounds which are introduced along the description of the tool workflow. During the developent of OCLoptimizer, several new versions of Clang with full support for OpenCL were released. However, those versions also included major changes on both the class hierarchy and the programming interface. Due to this, and since our workarounds allowed to effectively parse the OpenCL kernels, we kept them rather than rebuilding the whole tool to adapt it to the new internal structure of Clang.

2.3. The OCLoptimizer tool

OCLoptimizer is a source-to-source iterative optimization tool for OpenCL. As Figure 2.6 depicts, this tool performs three main steps based on an annotated source code of a kernel and a configuration file, both required as inputs. First, it generates a suitable host code for the kernel. Then, it performs two searches driven by the execution time in order to optimize the OpenCL code for the platform where the tool is executed. The first search constitutes the second step of the tool. Its aim is to select an optimized combination of the index spaces of the kernel, both the global one, which determines the number of work-items or threads that run it in

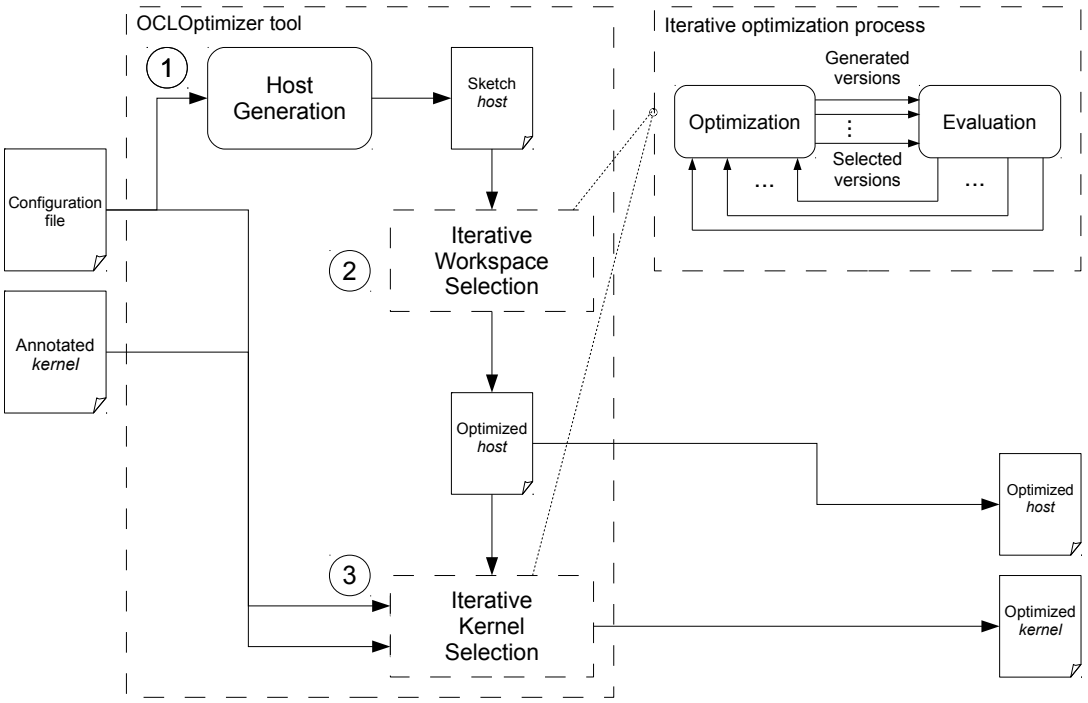


Figure 2.6: General workflow of OCLoptimizer

parallel, and the local one, which controls the number of threads per work-group. The combination of these two index spaces, which have between one and three dimensions, will be called in what follows the workspace of the kernel. Finally, in the third step, OCLoptimizer runs an iterative compilation process driven by the annotations of the user in the source code, generating an optimized version of the kernel as result.

The host generation stage only requires the specifications in the configuration file, described in Section 2.3.3. The generated host code is a stand-alone program with all the stages required to run an OpenCL kernel, which were illustrated and briefly described in Section 2.1.4. The initialization of the kernel inputs whose value is not specified in the configuration file may be random or performed by means of a code provided by the user. The host code receives as arguments the parameters that define the workspace configuration of the kernel, namely, the global and the local sizes for each dimension of the workspace. This facilitates the search of an optimized workspace configuration, as it eliminates the need to recompile the host code. We now explain in detail the two search processes performed by the tool.

```
1 __kernel void addvec(const unsigned int n, __global float *a,
2          __global float *b, __global float *c)
3 {
4   int i = get_global_size(0);
5   c[i] = a[i] + b[i];
6 }
```

Listing 2.2: OCLoptimizer vector addition example: base kernel

The vector addition code ($C = A + B$) shown in Listing 2.2 is used as a running example through this chapter. Line 4 of shows that this is a naive kernel linked to an element-wise work distribution, where each instance of the kernel calculates one position of the solution. The input configuration file for this kernel is used also in Section 2.3.3 to describe the main components of such files. Let us advance that this file indicates that the sizes of the arrays are 1024, the code must be optimized for a GPU, the input arrays must be initialized randomly, the local and global workspaces have one dimension, and the kernel receives four parameters: a scalar `n`, which takes the value of the arrays size, and the three arrays involved in the computation, `A`, `B` and `C`.

The inputs of the kernel are initialized at the beginning of the generated host code. This is followed by all the steps of a usual OpenCL host code required to locate the device where the computation will be done: platforms discovery, context creation and devices discovery. Once a device of the type specified in the configuration file is selected, the kernel is loaded and compiled. Then, the array inputs specified in the configuration file are transferred to the device through a command queue. The generated host code and the kernel are written for a generic workspace configuration which is passed as a parameter to the host code. This workspace configuration is used to enqueue the kernel. Finally, the host code enqueues the commands to read the results generated by the kernel.

2.3.1. Workspace optimization

The configuration file indicates the number of dimensions of the kernel workspace as well as the minimum and the maximum values that the tool has to explore for each dimension of the global and the local index spaces. OCLoptimizer looks for

an optimized combination of sizes in each dimension of both index spaces within these limits by means of a search process. This process is guided by the execution time of each combination in the target device specified in the configuration file. For this, the parameterized host code generated by the tool must be run using each workspace configuration to test. It must be noticed that at this stage of OCLoptimizer, the host code can only launch the base kernel provided by the user, since the kernel optimization process is performed in a further stage of the tool workflow. However, the loops in the kernel must adjust their iteration space in order to match each different workspace that can be requested during the search. Thus, for the vector addition example, the base kernel cannot we written as shown in Listing 2.2, as such code lacks this adaption capability. Rather, it has to be rewritten in a way that supports a generic workspace configuration. This is achieved using a set of three macros that OCLoptimizer provides with this purpose. These macros, called `GENINIT`, `GENLIMIT`, and `GENSTEP`, adapt the initialization, limit, and step of a loop, respectively, and are declared in a header file that is automatically included in the kernel by the tool. The programmer can use these macros in the loop(s) selected to distribute their iterations among the work-items of the workspace, as in multidimensional workspaces, a given dimension of the workspace is associated to a different loop. The three macros receive three parameters: (1) `<n>`, the number of iterations of the original loop, (2) `<s>`, an indication about whether the iterations are distributed among a number of points of the global (`g`) or the local (`l`) index space, and (3) `<d>`, the workspace dimension associated to this loop (0, 1 or 2).

Listing 2.3 shows the syntax that a user must follow to include these macros in the vector addition kernel of our running example, although letting the macro arguments as generic values to be filled in. The kernel has been extended with a single loop because the vector addition problem as a single dimension, and thus one loop is enough to manage it. As we can see, the `GENSTEP` macro calculates an

```
1 __kernel void addvec(const unsigned int n, __global float *a,
2          __global float *b, __global float *c)
3 {
4   int i;
5   for(i=GENINIT(<n>,<s>,<d>);i<GENLIMIT(<n>,<s>,<d>);i=i+GENSTEP(<n>,<s>,<d>))
6     c[i] = a[i] + b[i];
7 }
```

Listing 2.3: OCLoptimizer vector addition example: generic base kernel

adequate step for the loop whereas the `GENINIT` and `GENLIMIT` macros calculate the lower and the upper bound of the loop, respectively. These three macros are intended to generate a well-fitting distribution for the target platform. For example, in CPUs or accelerators such as the Xeon Phi, the macros give place to a consecutive distribution, which is expected to favor cache locality and auto-vectorization. In turn, when a GPU is detected, the macros give place to a cyclic distribution of the iterations among the threads. This way, the access pattern followed by the references generates coalesced accesses, which are expected to improve the performance in this kind of platforms. In both cases, if the input kernel is programmed in a naive way in the style of our vector addition example, the work-items are going to access global memory positions through their global identifiers. Due to this, the value for the `<s>` parameter of the macros should be `g`. However, OCLoptimizer is also able to process kernels that are already optimized to some extent, but which are still improvable by means of other optimizations offered by the tool. For instance, if a kernel exploits the local memory and, thus, the work-items iterate in some loop on local memory positions through their local identifiers, then the value for the `<s>` parameter for that loop should be `l`.

Once the described adaptation is performed, the user must choose between two search strategies for this first optimization process. Such strategies are an exhaustive search (ES) and a genetic algorithm (GA) [42], which are discussed in turn.

Exhaustive search

A search algorithm is defined as exhaustive when it explores the whole solution space of a problem. While such behavior is its main advantage, it also introduces some major drawbacks. Since an exhaustive algorithm visits the whole search space, it will eventually find the optimal solution inside it. However, depending on the problem, the cardinality of such space can be so high that the search process might take a long time to finish. Moreover, usually there are solutions that are known beforehand to be clearly suboptimal.

Figure 2.7 summarizes the steps followed by the exhaustive search of an optimized workspace configuration. First, the tool generates the set of workspace configurations included in the ranges specified in the configuration file. In order to

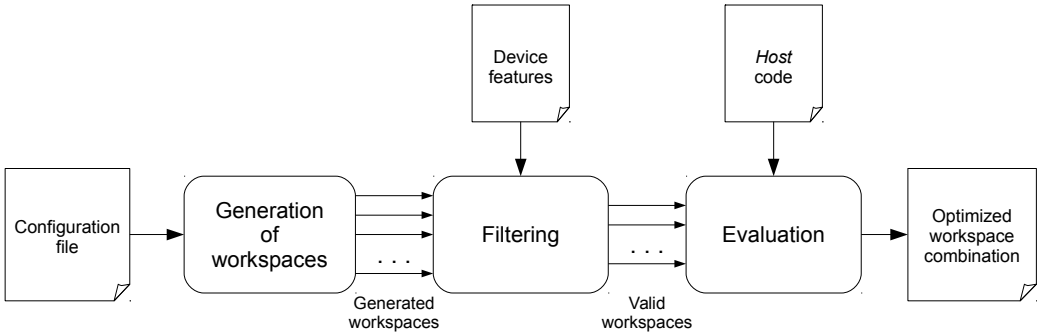


Figure 2.7: Iterative process to select an optimized workspace configuration in OCLoptimizer tool using an exhaustive search.

reduce the number of combinations, the points considered for each dimension i of a given index space are m_i, $2m_i$, $4m_i$, $8m_i$, ... M_i, where m_i and M_i are the minimum and the maximum values specified for dimension i of that index space in the configuration file, respectively. The execution model of OpenCL introduces constraints that, for example, force the local workspace to be a divisor of the global one in every dimension. Also, some OpenCL implementations limit the size of each dimension of a work-group, and also set a maximum value for the number of work-items grouped in it. OpenCL runtimes keep information about the capabilities and restrictions of the devices they support, and they make them available to programmers by means of their API. Namely, the values involved in the restrictions just mentioned can be obtained by calling the `clGetDeviceInfo()` method of the OpenCL API and reading the fields `CL_DEVICE_MAX_WORK_GROUP_SIZE` and `CL_DEVICE_MAX_WORK_ITEM_SIZES`, respectively. These restrictions make some configurations impossible and hence they are discarded in advance. No matter they are deliberate or imposed, these filters are useful to reduce the size of the set of workspace configurations to explore before launching the search process. Once these filters have been applied, all the surviving configurations have to be evaluated. Let us remind that a workspace-parametrized host code is generated on the first stage of the workflow, so it can be used at this point to run each workspace configuration and get its execution time. Many of these configurations are quite suboptimal, and the full execution of all of them may make too long. To mitigate this issue, the tool keeps the minimum execution time measured for a single configuration along the search process, and kills every test that lasts more than such a limit. Notice that, eventually, the configuration linked to such minimum time will be the optimal one.

Genetic algorithm search

Genetic algorithms are a particular case of a more general kind of search methods called *evolutionary* algorithms, which are inspired, as such name suggests, by biological evolution phenomena. Thus, in genetic algorithms the search space of a problem is modeled as a *population* of individuals. Each *individual* is identified with a *chromosome*, such chromosomes being composed of multiple genes. Each *gene* represents some feature of the individual, which can take values called *alleles* within a range. In the same vein as in nature, where only the fittest survive, the aim of these algorithms is to maximize a function called *fitness function*. This function, as it can be inferred from its name, measures the quality of the individuals in a population.

As an evolutionary search method, another defining characteristic of genetic algorithms is that their populations must evolve along time, which leads to the concept of *generation*. The number of individuals in the generations is usually determined by experimentation. Thus, the algorithm implements *reproduction* mechanisms through which pairs of individuals are selected to make them breed a new generation. The selection criteria for such pairs are usually biased to mate quite fit individuals. Each mating pair undergoes a *crossover* operation on which their chromosomes are cut in gene strips by one or several points and, by mixing and pasting back such strips, new individuals are formed. Moreover, *mutations* on the genes of these newborn individuals are randomized. Pairs from the current generation will be mated to reproduce until the newborn offspring fills a new one. However, the population is not expected to evolve indefinitely. Rather, the algorithm iterates on the creation of new generations until a termination condition is satisfied. Such condition usually depends on both a number of the latter generations and the fitness of their individuals.

This kind of algorithms is able to find high-quality solutions for a problem without visiting exhaustively all the search space, this being its main advantage. However, it is very sensitive to inaccurate configurations of the operation parameters. On the one hand, there are configurations that can hamper considerably the variety of individuals. This increases the odds that the algorithm ends prematurely and, as a consequence, returns a local optimum or even a meaningless solution. On the other hand, an algorithm implementing a too randomized evolution is very likely to lurch across the solution space and never reach its termination condition.

Let us now explain how an optimized workspace configuration can be found by means of a genetic algorithm. In this problem, each feasible workspace configuration is treated as an individual, and the genes in its chromosome represent each dimension of both the global and the local workspace sizes. Regarding the quality of a configuration, the shorter the execution time of a workspace configuration is, the better. Since the algorithm looks for a maximum, in this case the fitness function is defined as the inverse of the execution time of the workspace configuration corresponding to a given individual. As the flowchart in Figure 2.8 shows, the data input of the algorithm is the *initial population*. For this search process, the number of initial individuals was previously determined by experimentation, each individual being generated by randomizing values for global and local workspaces sizes. Such values are constrained by the same limitations described for the exhaustive search algorithm. Sometimes, these restrictions might be so tight that there will be

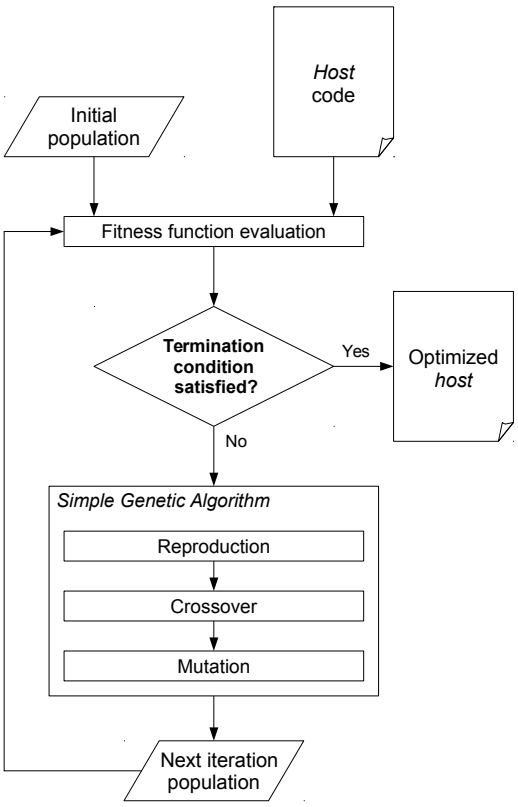


Figure 2.8: Iterative process to find an optimized workspace configuration using a genetic algorithm

fewer feasible individuals than the size set for the initial population. If this happens, random clones of existing individuals are created until the desired population size is reached. Once the initial population is generated, the fitness function is evaluated for each individual. Evaluations are performed by passing the corresponding workspace configuration to the host code generated by the tool, running it three times using the given configuration, and getting the average execution time. The termination condition set for this search is that the individual with the maximum fitness, i.e., the fastest workspace configuration, is the same in three consecutive generations. This implies that after the evaluation of the first generation, the condition cannot be satisfied yet. Thus, a new generation is bred. As the feedback loop in the flowchart shows, this new generation will be the input for the next iteration of the algorithm. This process is repeated until at least three new generations are created, and then, when along the three latter generations the fastest workspace configuration found remains the same. When this happens, the algorithm returns an optimized workspace configuration to be used during the kernel optimization process.

2.3.2. Optimized kernel code generation

Once both the global and the local workspace configurations have been chosen, the tool launches an iterative optimization process in which a series of code transformations are applied to the kernel code. These optimizations are suggested by the user by means of compiler directives inserted in the kernel code. This stage has been built on top of version 3.0 of the Clang front-end for LLVM [62]. All the code manipulation tasks are performed on the Abstract Syntax Tree (AST) representation of the input kernel, rather than on the LLVM intermediate representation (IR). The reason is that this enables us to generate an output optimized kernel that is much more human-readable, similar to the input kernel, and easier to maintain than the one obtained by other approaches such as the usage of the LLVM IR.

In order to optimize it, the input kernel must be annotated with special OCLoptimizer directives. These annotations precede the piece of code affected by the transformation, the general form of an annotation being

```
#pragma oclopts <name> <params> [tolerance t] [number n]
```

where `<name>` is the name of the optimization technique to apply and `<params>` stands for its parameters, which usually vary among techniques. These parameters are mostly used to define a range of values to test when the corresponding technique is applied. The optional field `tolerance` restricts the versions that proceed to the next level of the iterative process to those whose execution times are below a tolerance `t` ($0 \leq$ `t` ≤ 100) percentage above the time of the fastest one found among them. Finally, the argument `number` can be used to limit the number of versions that proceed to the next iteration of the process.

The current version of OCLoptimizer only applies the unroll and the unroll-and-jam techniques [2] and it selects an optimized unroll factor for each annotated loop. The general form of the pragma associated to these techniques is

```
#pragma oclopts unroll <init> <end> <step> [tolerance t] [number n]
```

where `unroll` is the name of the technique and the parameters `<init>`, `<end>`, and `<step>` are the first, the last and the step values used to build the search space of unroll factors that the tool has to explore, respectively.

One of the workarounds implemented to avert the limitations found in Clang 3.0 is related to the processing of the optimization directives, which were not properly parsed. To avert this problem, these annotations are converted in a previous preprocessing stage to calls to a set of empty functions created for that purpose. These functions are defined in a header file delivered with the tool and which is automatically included by it in the input kernels. Each type of optimization directive is mapped to a single function, while the parameters of the annotation are the arguments of the corresponding function. Thus, when the code consumer implemented with Clang visits any call to such functions, both the optimization type and the arguments are captured and then applied to the annotated section. The other workaround intends to avert the lack of compatibility with the OpenCL vector types, which causes errors when parsing their data types. The solution to this issue consisted on automatically adding to the input kernel an explicit declaration

```
typedef <vtype> <type><n> __attribute__((ext_vector_type(<n>)));
```

on which `<vtype>` is the vector type to define (e.g., `float4`), `<type>` is the corresponding base type (`float`) and `<n>` is the vector length (`4`).

Listing 2.4 shows the annotated version of the base kernel described in Listing 2.2. We can see that it has the same parameters as in Listing 2.2, and that is has been modified to be able to adapt to different workspace configurations by adding the loop already illustrated in Listing 2.3. Notice how the macros `GENINIT`, `GENLIMIT` and `GENSTEP` are used to distribute the computation of the `n` positions of the vectors to process among the work-items available in the dimension `0` of the global (`g`) workspace. Moreover, the loop is annotated with an `unroll` pragma that commands OCLoptimizer to test the range of unroll factors between 2 and 8 with step 2, i.e., the search space to explore is $\{2, 4, 6, 8\}$. Listing 2.5 shows the kernel version that the tool generates for the input when an unroll factor of 2 is selected and the target device is a GPU. Notice how the code is not only unrolled with the selected factor but it follows an interleaved distribution of the iteration space. If the target were a CPU, then the macros would assign blocks of consecutive iterations to work-items in order to favor locality and auto-vectorization. Both the unrolled loop boundary and the step are calculated to match the workspace configuration selected in the previous stage.

```
1  __kernel void addvec(const unsigned int n, __global float *a,
2           __global float *b, __global float *c)
3  {
4    int i;
5    #pragma oclopts unroll 2 8 2
6    for(i=GENINIT(n,g,0), i<GENLIMIT(n,g,0);i=i+GENSTEP(n,g,0))
7      c[i] = a[i] + b[i];
8  }
```

Listing 2.4: OCLoptimizer vector addition example: annotated kernel

```
1  __kernel void addvec(const unsigned int n, __global float *a,
2           __global float *b, __global float *c)
3  {
4    int idx = get_global_id(0);
5    int szx = get_global_size(0);
6    int i;
7    for(i=idx; i<n; i+=(szx*2)) {
8        a[i+(szx*0)]=b[i+(idx*0)]+c[i+(szx*0)];
9        a[i+(szx*1)]=b[i+(idx*1)]+c[i+(szx*1)];
10   }
11 }
```

Listing 2.5: OCLoptimizer vector addition example: generated kernel

In our running example, there is only one loop annotated with a pragma that generates 4 versions, one for each unroll factor to test. Finding an optimized version among them is as easy as running the four versions and picking the best-performing one. Let us consider another kernel with 2 loops, each annotated with a similar pragma. In this case, 2 pragmas generating 4 versions each would give place to $4 \times 4 = 16$ versions, and so on. This reasoning can be expanded to a kernel having n loops, each loop i annotated with a pragma p_i that generates v_i versions, with $i = 1, 2, \ldots, n$. Thus, the number of versions generated would be $v_1 \times v_2 \times \ldots \times v_n$, which can lead to a combinatorial explosion of the search space of feasible versions. To avoid such situations, providing the tool with alternatives to a pure exhaustive exploration becomes advisable. Because of that, two iterative search processes, namely a breadth-first search (BFS) and, again, a genetic algorithm (GA), are implemented. Let us discuss them in turn.

Breadth-first search

A breadth-first search (BFS) is an algorithm for traversing graph- or tree-like data structures. It starts at some node of the data structure to visit and explores the immediate neighbor nodes first, before moving to those on the next level. Here the "breadth" qualifier is used in contrast to the orthogonal "depth" algorithms, which traverse data structures along each branch as far as possible before backtracking. Figure 2.9 shows this opposition by numbering the nodes of a tree depending on how it is traversed. Notice also how in the breadth-first example nodes are indeed visited level by level: root ($\{1\}$), first level ($\{2, 3, 4\}$), second level ($\{5, 6, 7, 8\}$), and third level ($\{9, 10, 11, 12\}$).

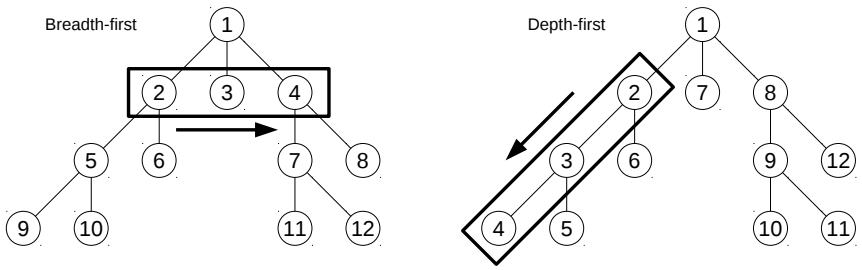


Figure 2.9: Traversal orders of breadth- and depth-first search algorithms

In a pure exhaustive algorithm, the search space of a problem can be modeled as an unordered set, since all states are going to be visited in order to find a solution. However, since a BFS algorithm expects graph- or tree-like structures as input, so some kind of neighborhood relationships (for graphs), and also a hierarchy (for trees), must exist among the possible solutions. For our kernel optimization problem, such a hierarchy relationship is established among annotations, so that they are processed one by one in the order they appear in the source code, the application of each pragma giving place to a new level of versions. Figure 2.10 compares the pure exhaustive and BFS approaches for the case of a kernel with two annotated loops generating four versions each.

Eventually, all the versions explored by the pure exhaustive search are going to be also explored by the BFS algorithm, although in its specific breadth-first order. However, the level-by-level approach followed by the latter method introduces the generation of some intermediate versions. In the example of Figure 2.10, such versions are $\{v_1, v_2, v_3, v_4\}$ and they appear when the first pragma has been just applied but the second one has not been processed yet. This intermediate versions are going to be quite useful to control the branching of our BFS strategy. This way,

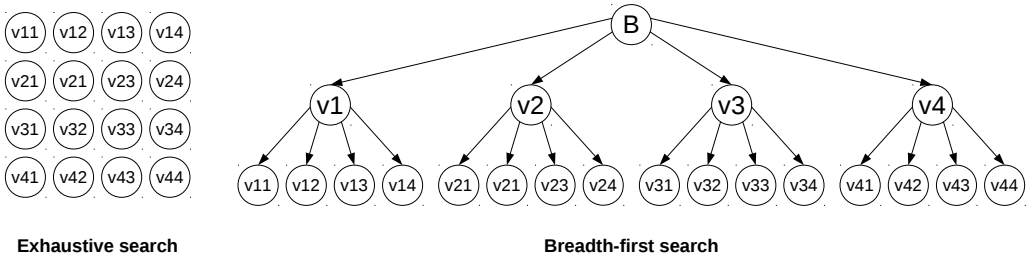


Figure 2.10: ES vs. BFS when exploring the search space for two pragmas with four different values each

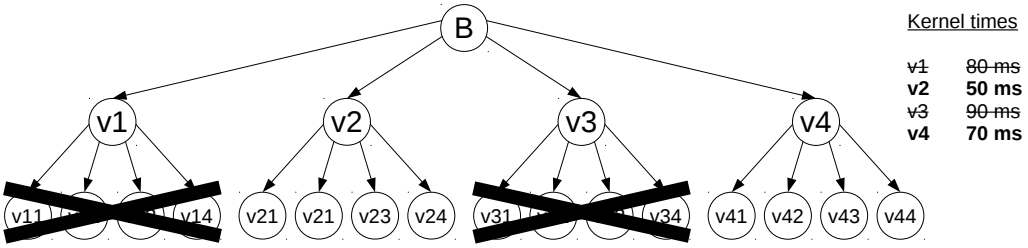


Figure 2.11: Pruning technique example for a 16-version BFS search

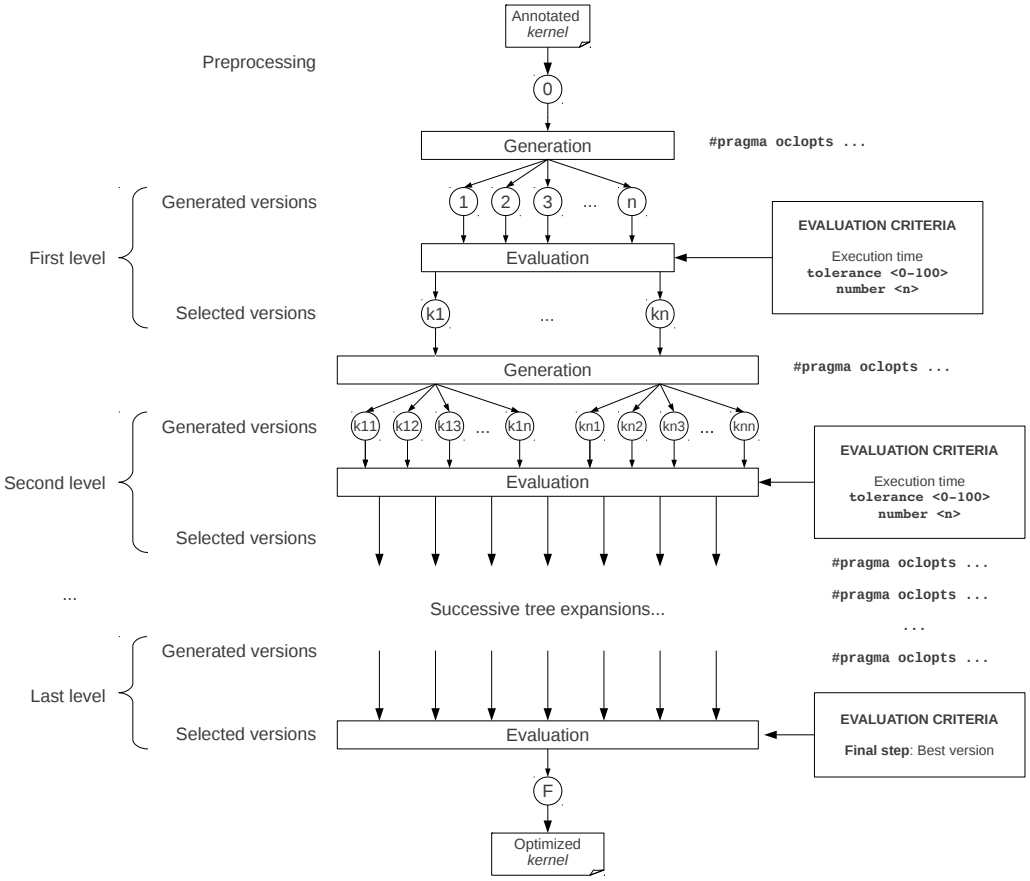


Figure 2.12: Iterative process to select the optimized kernel in OCLoptimizer tool using a breadth-first search

we can run it and, for example, keep only the two fastest ones and discard the rest, as Figure 2.11 shows. As a consequence, the versions $\{v_{11}, v_{12}, v_{13}, v_{14}, v_{31}, v_{32}, v_{33}, v_{34}\}$ are not going to be generated. On the one hand, this is an advantage, as we are pruning the search tree and hence reducing the number of possible solutions. On the other hand, $\{v_1, v_3\}$ being worse than $\{v_2, v_4\}$ does not imply that all the versions pruned are slower that the best of those effectively generated.

Figure 2.12 depicts how this strategy has been generalized to implement the kernel optimization search process following a BFS algorithm. As said before, the pragmas in the annotated kernel are processed one by one, the application of each one giving place to a new level of versions. Two consecutive operations are performed to process each single directive: *generation* and *evaluation*. In the generation stage the pragma is applied, which gives place to a number of versions of the kernel. For

example, if an `unroll` pragma is found, new versions of the kernel are generated by unrolling the annotated loop with the range of unroll factors configured in the pragma. In the evaluation stage, these kernels are run and their execution times are collected. If the user has specified the `tolerance` and/or `number` modifiers in the pragma, they are applied to keep only the versions that satisfy the criteria indicated. Notice how this works as a pruning technique to control the branching of the BFS strategy. Thus, the surviving versions are used as base kernels for the next iteration of the optimization process, on which the next pragma found will be applied. This process is repeated until all the pragmas are processed. In such a moment, the fastest version of the final step is returned as the optimized version.

Genetic algorithm search

The workflow of the genetic search implemented to get an optimized kernel version is quite similar to that used to obtain an optimized workspace configuration, as Figure 2.13 shows. Nevertheless, they solve different problems and explore different search spaces, which introduces several particularities that are worth mentioning.

In this problem, each directive found in the input kernel is modelled as a gene, all directives being gathered into a chromosome. Now individuals are single combinations of the values that the parameters of each directive can take. Thus, for a kernel just having `unroll` pragmas, a chromosome will contain a set of unroll factors, each factor being used to unroll the corresponding annotated loop. The `tolerance` and `number` modifiers are dismissed when the directives are processed by the genetic algorithm, since they are specific parameters for pruning the BFS strategy. Regarding the quality of a version, the shorter the kernel execution time is, the better. As the algorithm looks for a maximum, the fitness function is defined again as the inverse of the execution time associated to a given individual.

In relation to the initial population, the number of individuals is also previously determined by experimentation. These individuals are randomly created too, but now the alleles of their genes are values taken from the parameter ranges of the respective annotations. If the parameter ranges set in the directives are not wide enough to fill the initial population with random individuals, clones of those already existing will be created.

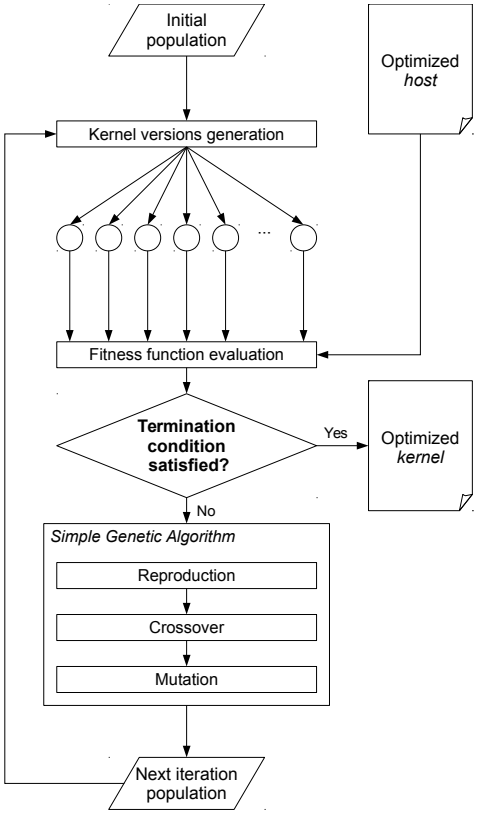


Figure 2.13: Iterative process to select the optimized kernel in OCLoptimizer using a genetic algorithm

In this search process, the evaluation of the fitness function for an individual requires a previous *kernel generation* task. In this step, the value stored in each gene is used to apply the optimization defined in the corresponding directive. As a result, a version of the kernel with those optimizations applied is generated. To evaluate the fitness function for this version, the generated kernel is run three times using the host code and the optimal workspace configuration obtained in the previous stages, and getting the average kernel execution time.

The termination condition is quite similar too, as the search process stops when the fastest version found does not vary along three generations. In that case, this version is returned. Otherwise, the algorithm makes the population to evolve by the already described operations of *reproduction*, *crossover* and *mutation*, giving place to a new generation of individuals and repeating the process.

Finally, it deserves to be mentioned that both workspace configuration and kernel optimization genetic algorithms were implemented using the GAlib package [118], written in C++ by Matthew Wall at the Massachusetts Institute of Technology.

2.3.3. Configuration file

The OCLoptimizer configuration file defines several variables that drive the generation of the host code and the search of the workspace configuration. This file has five sections that are now described in turn using the example file in Listing 2.6.

The *common parameters* section initializes variables that will be used through the rest of the file and it configures some general settings of the OpenCL host code to be generated. In the example this section initializes the variable `N` to 1024. Then, it establishes that the host code has to use a GPU to perform the computation. Alternatively, the `device` variable could take the values `CPU`, which is used to select the main processor, or `ACC`, which is used to select a Xeon Phi if available. If there are two devices of the same type, the current implementation of OCLoptimizer selects the first one. The `initialization` variable specifies how the kernel inputs whose value are not specified in the configuration file should be initialized in the host code. In the example, the `random` value indicates that it should be initialized with random values. In some cases, a random initialization would not be valid as the contents of the input data should fulfill certain conditions. In that case, the `initialization` variable should be set to `code` and the path of the file containing the initialization code should be provided as the third parameter of the tool.

The *compiler parameters* section configures the compilation process. It provides the location of the library and headers files of the OpenCL implementation to use, and the compilation mode, which selects the way the intermediate versions of the host code to optimize are compiled by the tool. Currently OCLoptimizer only supports the `system` compilation mode, which performs the compilation using a call to the system default compiler.

The *workspace definition* section sets the parameters related to the workspace configuration. These are the number of dimensions of the workspace (`ndims`), and for each dimension, the global and the local size. As the optimized global and local

```
# common parameters
N = 1024
device = gpu
initialization = random
# compiler parameters
mode = system
ocllibpath = /usr/local/lib
oclincludepath = /usr/local/include
# workspace
ndims =1
[ dim0 ]
globalsize = 1,N
localsize = 1,32
# workspace restrictions
localsize < globalsize
# kernel parameters
nparam = 3
[ param0 ]
name = n
size = 1
type = uint
mode = r
value = N
[ param1 ]
name = A
size = N
type = float*
mode = w
[ param2 ]
name = B
size = N
type = float*
mode = r
[ param3 ]
name = C
size = N
type = float*
mode = r
```

Listing 2.6: OCLoptimizer vector addition example: configuration file content

size will be found iteratively, the user has to specify, separated by commas, the minimum and the maximum value to test for the sizes of all the local and global dimensions. The values associated to dimension `X` are preceded by a `[dimX]` clause. In this example the workspace only has one dimension composed of between 1 and N (`globalsize=1,N`) work-items and each work-group is composed of between 1 and

32 work-items (`localsize=1,32`).

The *workspace restrictions* section specifies conditions that must be satisfied by the workspace definition. Workspace configurations that do not fulfill these conditions must be discarded. In the example, the local workspace size must be smaller than the global workspace size.

Finally, the *kernel parameters* section defines the number, the size and the type of each parameter of the kernel. It must also indicate for each parameter whether it is a read-only value (r) or a read/write value (w). Unspecified parameters take a default value. The information associated to the `X`-th parameter is preceded by a `[paramX]` clause. In the example, the kernel receives four parameters called `n`, `A`, `B` and `0 C`, respectively. The first parameter is a read-only (`mode=r`) scalar (`size=1`) called `n` of type `uint`. Its default value is `value=N`. Arrays `A`, `B` and `C` (`name=A`, `name=B`, `name=C`) have `size=N` elements of type float (`type=float*`) and the first one can be modified inside the kernel (`mode=w`) while the two others are read-only (`mode=r`).

2.4. Support for codes with several kernels

The tool can also optimize OpenCL applications composed of multiple kernels. In this case, the tool loses its ability to generate a working host code for the whole application. Keeping this ability would require a much more complex configuration file. Namely, additional information such as the relations and the data flow between the different kernels should be provided. Nevertheless, the tool obtains an optimized workspace configuration and generates an optimized version for each input kernel. Thus, users can eventually integrate these outputs in their own applications.

The process followed by OCLoptimizer to optimize codes with multiple kernels, which consists of several fully automated steps, is now described. Some of the kernels that compose the application may have to use the same workspace configuration, while others can use one of their own. These two types of kernels are processed differently by the tool. The kernels that can use their own workspace configuration are optimized one by one in a separated optimization process, as Figure 2.14 depicts. In these cases, the user has to provide a separate configuration file (CF_i) and an

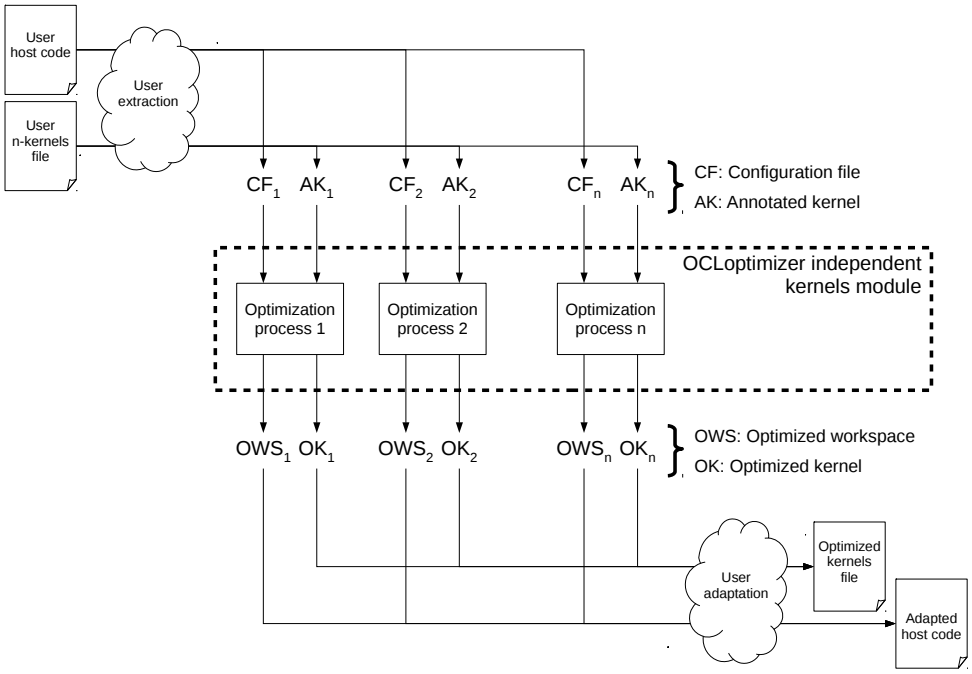


Figure 2.14: Workflow of OCLoptimizer for several independent kernels

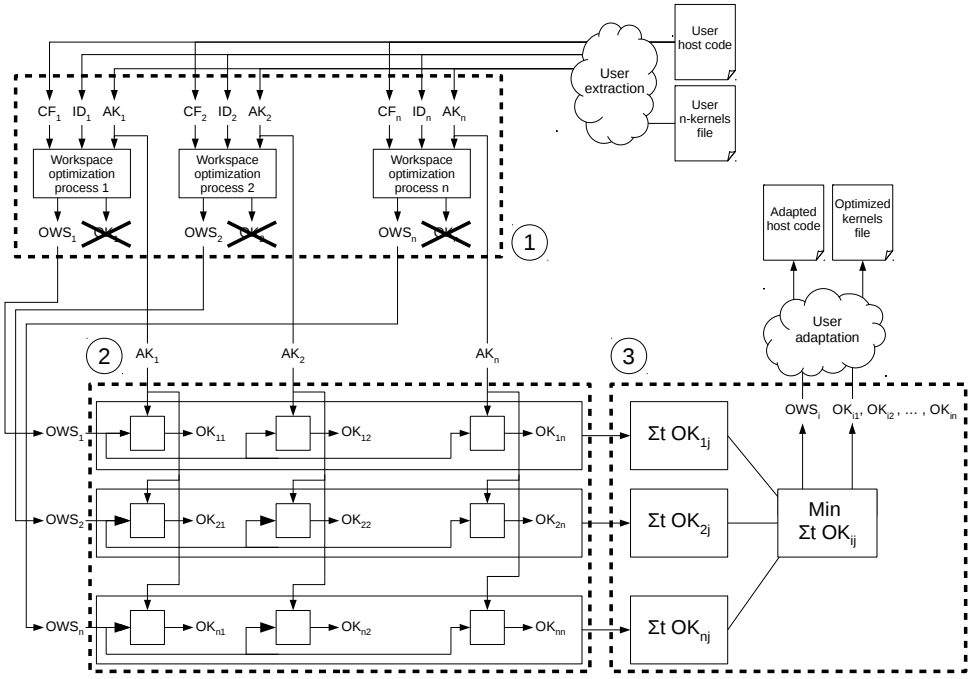


Figure 2.15: Workflow of OCLoptimizer for several inter-dependent kernels

annotated code for each kernel (AK_i). An optimized workspace configuration and an optimized kernel result from each separate process. These outputs can be included in the user application to optimize it.

In the case of groups of kernels that have to use the same workspace configuration, OCLoptimizer follows a three step workflow shown in Figure 2.15. In the first step, as many optimization processes as kernels to optimize are launched. Each process requires a configuration file (CF_i), an annotated kernel (AK_i), and if necessary, an initialization code. In the case of inter-dependent kernels this last optional parameter is very important. Sometimes, the inputs of these inter-dependent kernels are intermediate results of the algorithm they implement and they have to comply with certain characteristics, otherwise such kernels will not run properly. This way, the inputs of these kernels must be generated using a code provided by the user. Such inputs are represented as ID_i in the Figure 2.15. In addition, the workspace configuration of this kind of kernels usually has to match certain conditions. These conditions can be specified in the workspace restrictions section of the configuration file associated to each kernel (see Section 2.3.3). The output of this step is a set of optimized workspace configurations, whereas the optimized kernels generated are discarded.

In the second step, the tool tries one by one all the workspace configurations obtained in the first step. Thus, each workspace configuration is used to optimize all the annotated kernels of the group. Notice that in this step only the kernel optimization process described in Section 2.3.2 is performed, since the candidate workspace configurations are already given as inputs. The result of this step is a set of possible optimizations of the user application, each composed of a candidate workspace configuration and its corresponding group of optimized kernels.

Finally, in a third step, the tool evaluates each possible optimization of the user application, selecting the one that gives place to the shortest execution time for the group of optimized kernels. The workspace configuration and the group of kernels selected in this step are returned to the user, who can use them to optimize the application.

2.5. Experimental results

As we have seen, OCLoptimizer has the ability to optimize OpenCL codes composed of either one single kernel or multiple kernels. Both features are validated now in turn. This way, Section 2.5.1 shows the validation performed with several OpenCL codes composed of one kernel, while Section 2.5.2 shows the experiments for an OpenCL application composed of five different kernels.

2.5.1. Codes with a single OpenCL kernel

This part of the validation is based on five computationally intensive single-kernel codes: an N-body simulation [1] (NBODY), a matrix multiplication (MATMUL), a discretization of the Laplacian operator with a nine-point stencil [101] (STENCIL), a Sobel Edge Detector [53] (SOBEL) and a Direct Coulomb Summation [108] (DCS). The NBODY kernel has two unrollable loops and the MATMUL kernel has three unrollable loops, whereas the STENCIL, SOBEL and the DCS kernels have four unrollable loops each. Regarding the worskpace configuration, NBODY has one-dimensional (global and local) workspaces, MATMUL, STENCIL and SOBEL have two-dimensional workspaces, and finally the workspaces of DCS have three dimensions. Hand-tuned versions of these kernels, which use local memory and have been vectorized wherever possible, have been used as inputs for this part of the validation process. Also, calculations common to several threads are performed collaboratively in order to improve the performance. The experiments were run on three different platforms:

- The **CPU**: A dual-socket system with two Intel Xeon E5-2660 Sandy Bridge with eight 2.2Ghz cores and Hyper-Threading (8×2 threads per processor, for a total of 32) and 64 GB of RAM. Intel OpenCL driver version 1.2-3.2.1.16712. Single-precision theoretical peak performance of 563 GFLOPS.
- The **GPU**: An NVIDIA Tesla Kepler K20m 5 GB GDDR5. OpenCL runtime: NVIDIA CUDA Toolkit 5.0.35 with OpenCL driver version 325.15. Single-precision theoretical peak performance of 3524 GFLOPS.
- The **Accelerator**: An Intel Xeon Phi 5110P with sixty 1.053GHz cores with 8

GB of RAM. OpenCL runtime: Intel OpenCL 1.2-3.2.1.16712. Single-precision theoretical peak performance of 2022 GFLOPS.

The exhaustive search of the workspace in the three platforms tested all the legal combinations of powers of two up to the problem size in the dimensions of the global index space. Those of the local space were tested up to the maximum size allowed by the device. The GA search of the workspace configuration used populations of 5 chromosomes in all the systems. Regarding the kernel optimization process, the directives used no `tolerance` or `number` modifiers and they were setup to consider all the possible unrolls. The GA search used in this case populations with 5% of the total number of possible chromosomes (combinations of unroll factors for the loops).

Tables 2.1 to 2.6 summarize the performance results obtained in the three platforms using two combinations of search processes: the longest one, which is exhaustive search (ES) for the workspaces and BFS for the kernels (ES+BFS), and the shortest one, which uses GA search for both optimization processes.

These six tables have the same structure. The first column contains the name of the code and the second one is the problem size. Three different sizes were taken into account for each code. Next, columns 3-5 contain the speedup achieved in the workspace optimization process and the global and the local workspace sizes (WSs) selected by OCLoptimizer for each dimension of the problem separated by commas, respectively. The speedup in column 3 has been calculated respect to the corresponding input baseline hand-tuned kernel. The size of each dimension of the global workspace is set to the size of the loop whose iterations are being distributed among the work-items and the local worksizes are left to be selected automatically by OpenCL. Columns 6-8 refer to the selection of unroll factors. In particular, column 6 shows the speedup achieved, which is calculated with respect to the optimized code resulting from the workspace optimization process. As a result, the total speedup provided by OCLoptimizer is the product of the speedups in columns 3 and 6 and will be discussed in Table 2.7. Finally, columns 7 and 8 contain, separated by commas, the unroll factors (UFs) selected by the tool and the maximum ones taken into account, respectively. Notice that the maximum unrolling of the loops that iterate on elements to process depends on the workspace selection performed in the previous stage. The reason is that the bigger the workspace is in some dimension, the fewer elements the loop of each thread has to process.

Code	Size	Workspaces optimization			Unroll optimization		
		Speedup	Global WS	Local WS	Speedup	Optimized UFs	Maximum UFs
NBODY	16384	3.50	8192	16	1.07	1,9,4	16,16,16
	32768	1.89	32768	256	1.03	1,8,8	16,16,16
	65536	1.02	8192	128	1.03	8,3,8	16,16,16
MATMUL	1024	1.00	1024,1024	AUTO	1.09	1,1,4	1024,1024,256
	2048	1.01	1024,1024	256,1	1.14	1,2,7	2048,2048,256
	4096	1.86	8,32	8,1	1.20	6,6,8	4096,4096,256
STENCIL	1024	1.63	16,256	1,32	1.00	1,1,1	1024,1024,3
	2048	1.51	16,2048	8,2	1.12	1,21,3	2048,2048,3
	4096	1.54	2,2048	2,64	1.06	1,5,3	4096,4096,3
SOBEL	1024	1.50	4,256	1,4	1.09	2,3,114,3	3,1024,1024,3
	2048	1.28	8,2048	2,4	1.15	3,1,19,3	3,2048,2048,3
	4096	1.30	16,4096	4,4	1.14	3,1,17,3	3,4096,4096,3
DCS	64	1.05	32,64,64	4,32,8	1.00	1,1,1,1	64,64,64,64
	128	1.00	128,128,128	4,4,4	1.00	1,1,1,2	128,128,128,128
	256	1.00	256,256,256	4,16,8	1.00	2,1,1,2	256,256,256,256

Table 2.1: Speedups and configurations selected using ES+BFS in the CPU

Code	Size	Workspaces optimization			Unroll optimization		
		Speedup	Global WS	Local WS	Speedup	Optimized UFs	Maximum UFs
NBODY	16384	3.52	256	2	1.05	2,10,1	16,16,16
	32768	1.90	512	16	1.03	1,13,8	16,16,16
	65536	1.00	65536	AUTO	1.06	1,5,5	16,16,16
MATMUL	1024	1.00	1024,1024	AUTO	1.02	1,1,29	1024,1024,256
	2048	1.00	2048,2048	AUTO	1.05	1,1,13	2048,2048,256
	4096	1.00	4096,4096	AUTO	1.27	1,1,93	4096,4096,256
STENCIL	1024	1.11	512,32	8,2	1.19	16,2,1	1024,1024,3
	2048	1.31	1,256	1,2	1.04	1,248,3	2048,2048,3
	4096	1.00	4096,4096	AUTO	1.46	1,1,3	4096,4096,3
SOBEL	1024	1.15	128,1024	8,16	1.13	3,1,2,1	3,1024,1024,3
	2048	1.16	16,512	8,8	1.07	2,1,91,3	3,2048,2048,3
	4096	1.25	256,4096	8,4	1.10	1,1,15,3	3,4096,4096,3
DCS	64	1.05	64,32,16	32,2,16	1.00	1,1,1,1	64,64,64,64
	128	1.00	8,128,64	4,1,2	1.00	1,1,16,2	128,128,128,128
	256	1.00	256,256,256	AUTO	1.00	1,1,15	256,256,256,256

Table 2.2: Speedups and configurations selected using GA in the CPU

Code	Size	Workspaces optimization			Unroll optimization		
		Speedup	Global WS	Local WS	Speedup	Optimized UFs	Maximum UFs
NBODY	16384	1.50	8192	256	1.07	1,4,16	16,16,16
	32768	1.69	32768	128	1.07	1,10,16	16,16,16
	65536	1.70	65536	128	1.29	1,2,14	16,16,16
MATMUL	1024	1.00	1024,1024	AUTO	1.35	1,1,256	1024,1024,256
	2048	1.08	256,2048	64,4	1.11	1,8,32	2048,2048,256
	4096	1.06	512,4096	64,2	1.11	1,8,16	4096,4096,256
STENCIL	1024	2.36	64,1024	32,16	1.16	1,1,3	1024,1024,3
	2048	2.71	64,2048	32,16	1.16	1,1,3	2048,2048,3
	4096	2.87	64,2048	64,16	1.14	2,1,3	4096,4096,3
SOBEL	1024	3.85	64,256	32,4	1.15	1,4,16,3	3,1024,1024,3
	2048	4.29	256,512	128,1	1.21	3,4,1,1	3,2048,2048,3
	4096	4.53	256,512	128,1	1.25	2,8,1,1	3,4096,4096,3
DCS	64	1.15	32,64,64	32,1,4	1.44	1,1,2,6	64,64,64,64
	128	1.00	128,128,128	AUTO	1.54	1,1,1,11	128,128,128,128
	256	1.03	128,128,128	16,8,8	1.16	1,2,2,256	256,256,256,256

Table 2.3: Speedups and configurations selected using ES+BFS in the GPU

Code	Size	Workspaces optimization			Unroll optimization		
		Speedup	Global WS	Local WS	Speedup	Optimized UFs	Maximum UFs
NBODY	16384	1.47	8192	64	1.06	1,4,8	16,16,16
	32768	1.67	32768	64	1.06	1,2,15	16,16,16
	65536	1.22	8192	64	1.07	4,1,16	16,16,16
MATMUL	1024	1.00	1024,1024	AUTO	1.02	1,1,29	1024,1024,256
	2048	1.00	2048,2048	AUTO	1.05	1,1,13	2048,2048,256
	4096	1.00	4096,4096	AUTO	1.17	1,1,34	4096,4096,256
STENCIL	1024	2.10	128,1024	16,16	1.03	1,4,3	1024,1024,3
	2048	2.08	512,64	512,1	1.24	26,3,3	2048,2048,3
	4096	2.72	256,2048	64,4	1.08	1,8,3	4096,4096,3
SOBEL	1024	3.66	512,64	128,1	1.16	1,15,2,3	3,1024,1024,3
	2048	4.02	512,256	16,32	1.11	3,2,2,3	3,2048,2048,3
	4096	2.13	512,16	32,8	1.13	2,2,1,3	3,4096,4096,3
DCS	64	1.00	64,64,64	AUTO	1.06	1,1,1,8	64,64,64,64
	128	1.00	128,128,128	AUTO	1.52	1,1,1,6	128,128,128,128
	256	1.00	256,256,256	AUTO	1.14	1,1,1,10	256,256,256,256

Table 2.4: Speedups and configurations selected using GA in the GPU

Code	Size	Workspaces optimization			Unroll optimization		
		Speedup	Global WS	Local WS	Speedup	Optimized UFs	Maximum UFs
NBODY	16384	14.58	16384	16	1.60	1,1,16	16,16,16
	32768	8.14	32768	16	1.60	1,1,16	16,16,16
	65536	4.26	65536	32	1.38	1,16,6	16,16,16
MATMUL	1024	1.00	1024,1024	AUTO	1.36	1,1,4	1024,1024,256
	2048	1.00	2048,2048	AUTO	1.00	1,1,1	2048,2048,256
	4096	1.00	4096,4096	AUTO	1.00	1,1,1	4096,4096,256
STENCIL	1024	1.46	32,512	1,4	1.35	2,8,3	1024,1024,3
	2048	1.61	8,2048	1,1	1.69	1,8,3	2048,2048,3
	4096	1.69	32,1024	1,1	1.86	2,8,3	4096,4096,3
SOBEL	1024	1.46	16,512	1,1	1.65	2,2,7,3	3,1024,1024,3
	2048	1.52	16,512	1,1	2.40	2,1,4,3	3,2048,2048,3
	4096	1.51	32,4096	2,2	2.77	3,1,16,3	3,4096,4096,3
DCS	64	1.00	64,64,64	AUTO	1.12	1,1,1,8	64,64,64,64
	128	1.00	128,128,128	AUTO	1.07	1,1,1,8	128,128,128,128
	256	1.01	256,256,256	4,4,4	1.06	1,1,1,8	256,256,256,256

Table 2.5: Speedups and configurations selected using ES+BFS in the Accelerator

Code	Size	Workspaces optimization			Unroll optimization		
		Speedup	Global WS	Local WS	Speedup	Optimized UFs	Maximum UFs
NBODY	16384	6.33	8192	128	1.70	2,2,6	16,16,16
	32768	7.33	16384	16	1.51	1,2,16	16,16,16
	65536	3.58	8192	4	1.68	4,7,4	16,16,16
MATMUL	1024	1.00	1024,1024	AUTO	1.00	1,1,1	1024,1024,256
	2048	1.00	2048,2048	AUTO	1.00	1,1,1	2048,2048,256
	4096	1.16	256,4096	2,32	1.01	1,2,1	4096,4096,256
STENCIL	1024	1.00	1024,1024	AUTO	1.46	1,1,3	1024,1024,3
	2048	1.25	512,1024	256,1	2.00	2,4,3	2048,2048,3
	4096	1.60	32,4096	4,4	1.93	1,16,3	4096,4096,3
SOBEL	1024	1.00	1024,1024	AUTO	1.76	1,1,1,3	3,1024,1024,3
	2048	1.32	512,1024	4,8	2.27	1,2,4,3	3,2048,2048,3
	4096	1.00	4096,4096	AUTO	2.26	1,1,1,3	3,4096,4096,3
DCS	64	1.00	64,64,64	AUTO	1.06	1,1,1,8	64,64,64,64
	128	1.00	32,32,64	1,4,1	1.04	2,4,1,8	128,128,128,128
	256	1.01	128,128,256	4,4,1	1.03	1,2,1,12	256,256,256,256

Table 2.6: Speedups and configurations selected using GA in the Accelerator

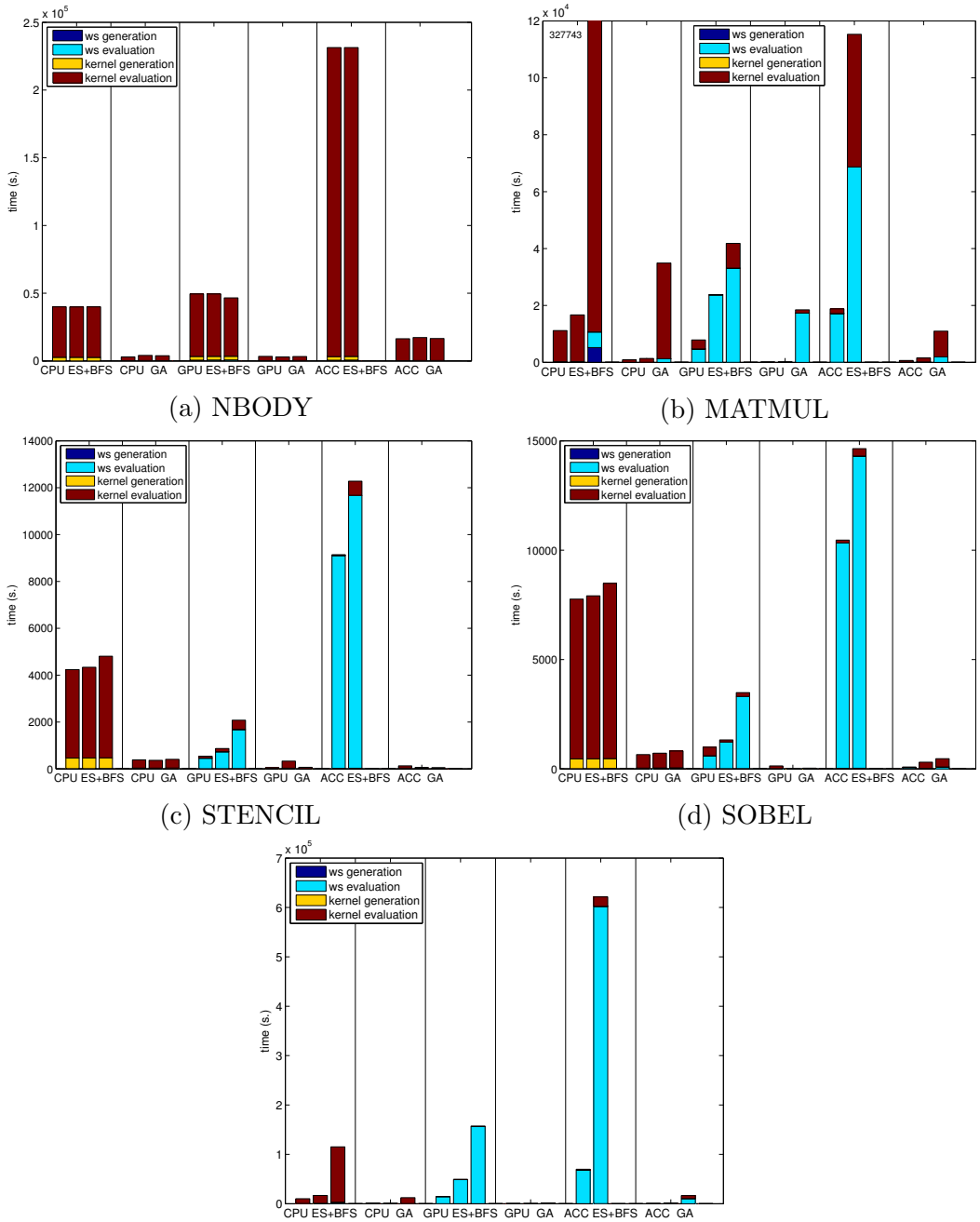
Code	Size	CPU	GPU	Accelerator
		ms (Speedup) (GFLOP/s)	ms (Speedup) (GFLOP/s)	ms (Speedup) (GFLOP/s)
NBODY	16384	178.99 (3.75) (25.50)	29.74 (1.61) (153.46)	143.25 (23.33) (31.86)
	32768	704.48 (1.95) (25.91)	87.44 (1.81) (208.76)	510.76 (13.02) (35.74)
	65536	2818.35 (1.05) (25.91)	287.46 (2.19) (254.00)	2266.06 (5.88) (32.22)
MATMUL	1024	38.29 (1.09) (56.08)	6.61 (1.35) (324.88)	146.64 (1.36) (14.64)
	2048	282.73 (1.15) (60.76)	57.91 (1.20) (296.66)	1537.07 (1.00) (11.18)
	4096	15200.90 (2.23) (9.04)	459.04 (1.18) (299.41)	16137.20 (1.00) (8.52)
STENCIL	1024	1.25 (1.63) (15.88)	0.28 (2.73) (71.30)	3.18 (1.97) (6.27)
	2048	2.85 (1.70) (27.95)	0.94 (3.16) (84.78)	7.13 (2.73) (11.18)
	4096	10.49 (1.63) (30.37)	3.59 (3.27) (88.80)	22.83 (3.14) (13.97)
SOBEL	1024	1.63 (1.63) (26.38)	0.43 (4.41) (100.31)	2.98 (2.41) (14.44)
	2048	4.81 (1.47) (35.78)	1.43 (5.21) (120.26)	6.16 (3.66) (27.93)
	4096	17.55 (1.49) (39.19)	5.24 (5.64) (131.18)	19.31 (4.17) (35.62)
DCS	64	11.08 (1.05) (15.14)	1.45 (1.65) (115.75)	19.17 (1.12) (8.75)
	128	175.92 (1.01) (15.26)	20.81 (1.54) (129.02)	274.23 (1.07) (9.79)
	256	2802.33 (1.00) (15.33)	459.93 (1.19) (93.38)	4293.58 (1.07) (10.00)

Table 2.7: Global speedups using ES+BFS

The average global speedup achieved for codes with a single OpenCL kernel using ES+BFS is 2.86, compared to the 2.22 achieved by GA. The tool obtains the largest speedups in the Xeon Phi (4.46 on average using ES+BFS) and, in this case, most of the speedup comes from the workspace optimization and, more precisely, from the NBODY test case. Since the tool usually selects large workspace sizes in all the kinds of devices, the margin left to the unrolling optimization is narrower in terms of search space, which can restrict the speedups obtained from the selection of the unroll factors (UFs). Nevertheless, the simplicity of their cores and their management of branches allow GPUs to remarkably benefit from unrolling. For example, DCS with size 128 achieves 54% more performance thanks to the kernel code tuning.

Table 2.7 summarizes the execution time, the global speedup and the performance measured in GFLOP/s achieved in our experiments for each code, problem size, and platform using the ES+BFS search. The speedup achieved by the optimized single kernel codes generated using the configurations selected by ES+BFS search is on average a 29% better than those generated using the configurations selected by GA. On exchange, the execution time of the tool is much longer when using the ES+BFS search, as we will see now.

Figures 2.16a to 2.16e show the time required by the search processes discussed for NBODY, MATMUL, STENCIL, SOBEL and DCS, respectively. Each figure is divided into six sections, one for each combination of a device (CPU, GPU and



(a) NBODY

(b) MATMUL

(c) STENCIL

(d) SOBEL

(e) DCS

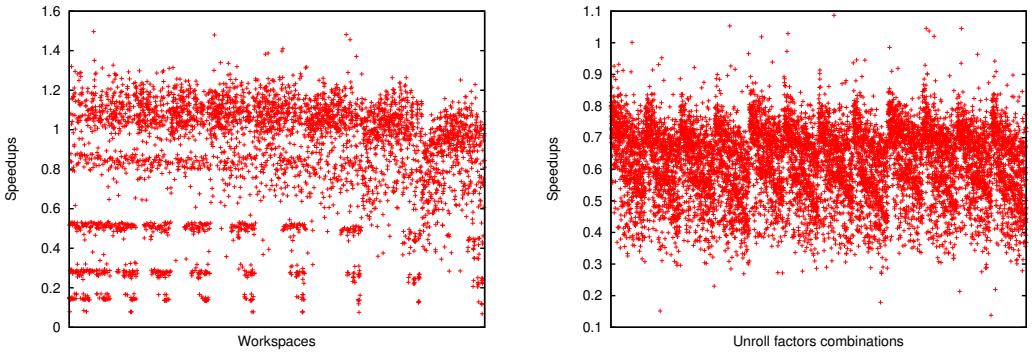
Figure 2.16: Search time distribution for OCLoptimizer test cases

Accelerator) and search process (ES+BFS or GA). Each section shows the search time for each tested problem size, from the smallest one to the largest one with bars divided into four stages: the workspace generation and evaluation times, which characterize the first optimization process, and the kernel optimization generation and evaluation times of the second search process.

The results show that the execution time of the tool is usually large because it generates a large number of versions of the code to be optimized. Unsurprisingly, ES+BFS requires longer search times than GA, as it generates more versions. On average, the search time using ES+BFS is ten times longer than using GA.

Most of the execution time is consumed by the evaluation process, which is conducted by executing the different versions generated. In the future, we want to evaluate the possibility of reducing the evaluation time by avoiding some or all the executions by means of the application of analytical models or heuristics. On the other hand, the generation time is negligible.

The time required by the unrolling optimization is usually longer that the one required by the workspace optimization because this second iterative process generates a larger number of versions. In some of the GPU tests the workspace optimization takes more time as the workspace range to be explored is wider and it generates a larger number of versions. Moreover, as it was said previously, the worksizes selected are usually large, which leaves a narrower margin for the unrolling technique.



(a) Workspaces (b) Unroll factors

Figure 2.17: Speedups for different workspaces and unroll factors for SOBEL in CPU

Figures 2.17a and 2.17b represent the speedups achieved using different workspace configurations and unroll factors, respectively. In both cases, the different workspaces and unroll factors have been generated by OCLoptimizer using ES+BFS for the SOBEL filter and size 1024×1024 on the CPU. The order of the workspace configurations and the unroll factors in the x-axis is the one in which they are generated by the tool. On the one hand, the results show that, in this example, the search of the workspace configuration explores a huge range of combinations for both global and local work sizes, and how this exhaustive search is done following a tree-like structure. On the other hand, the amorphous distributions of the speedups denote that the iterative optimization is adequate to guide these optimizations.

2.5.2. Codes with several kernels

The tool has been tested on the Integer Sort (Benchmark) of the NAS Parallel Benchmarks (NPB). The baseline of these experiments is the (Seoul National University) SNU OpenCL NPB [96] implementation of this benchmark, which has 5 different kernels. There are two versions of this benchmark, one suitable for CPU and another one suitable for GPU. In both implementations, several kernels use as inputs intermediate results which have to comply with certain characteristics, thus, special initialization codes had to be provided to the tool. In the GPU implementation, three of these kernels are inter-dependent and they must use the same workspace configuration, while in the CPU implementation the number of inter-dependent kernels is four. The experiments have been run on the same CPU and GPU used in the experiments of Section 2.5.1. The SNU NPB CPU version is used as the baseline for the CPU experiments, and its GPU version for the GPU. The Accelerator platform has not been used in these experiments as the SNU NPB suite does not have an implementation optimized for the Xeon Phi.

Tables 2.8 and 2.9 contain the speedups achieved by OCLoptimizer using the ES+BFS and the GA search processes respectively. The experiments were performed for three problem sizes: S, W and A. The tables show the speedups obtained from the workspace optimization and the unroll optimization, both calculated following the same approach as in Section 2.5.1. The last column contains the execution time of the best version of the benchmark generated and the speedup with respect to the

Device	Size	Workspace optimization	Unroll optimization	Overall results
		Speedup	Speedup	Time in ms (Speedup)
CPU	class S	3.69	1.08	8.92 (3.99)
	class W	1.99	1.04	20.13 (2.07)
	class A	1.12	1.15	85.93 (1.29)
GPU	class S	1.03	1.04	1.46 (1.07)
	class W	1.12	1.10	5.03 (1.22)
	class A	1.10	1.02	49.67 (1.12)

Table 2.8: Speedups and execution times for the IS benchmark using ES+BFS

Device	Size	Workspace optimization	Unroll optimization	Overall results
		Speedup	Speedup	Time in ms (Speedup)
CPU	class S	2.51	1.24	11.72 (3.11)
	class W	1.82	0.97	24.84 (1.77)
	class A	1.04	1.00	102.65 (1.04)
GPU	class S	1.03	1.02	1.50 (1.06)
	class W	1.08	1.03	5.17 (1.11)
	class A	1.09	1.02	50.11 (1.12)

Table 2.9: Speedups and execution times for the IS benchmark using GA

baseline. The workspace configurations and unroll factors chosen are not reported because of the large amount of data they imply given the existence of up to 5 kernels in the codes. As expected, the ES+BFS search obtains better results than the GA. The speedups in the CPU (3.03 on average for ES+BFS) are larger than in the GPU (1.13 on average for ES+BFS), and most of the speedup comes from the workspace optimization. These observations are similar to those made for the single kernel codes. The main conclusion of this experiment is that OCLoptimizer not only supports codes with strong inter-dependencies between their kernels, but it can also achieve respectable speedups despite working on hand-tuned state of the art implementations such as these two IS SNU NPB codes.

2.6. Conclusions

Two of the main weaknesses of OpenCL are the low level of its host API, which makes the development of its host codes tedious and error-prone, and, more importantly, the lack of performance portability. In this chapter we present OCLoptimizer, a tool that addresses both issues with a reduced programming effort. Given a configuration file and a kernel annotated with indications on the optimizations to try, OCLoptimizer is able to generate a working host code, find an optimized

workspace configuration and tune the kernel for the platform where the tool is executed. Moreover, thanks to the iteration distribution macros introduced to support generic configurations for both global and local workspaces, the input kernels can be not only naive implementations mapped to global memory positions but also more optimized versions that exploit the local memory to some extent. Furthermore, the tool also supports the automated optimization of both groups of independent kernels and applications with inter-dependent kernels. As far as we know, this latter feature is unique, although in this case the generation of the host code is not automated.

Our tool finds an optimized workspace and an optimized kernel code through search processes based on measurements of the execution time. While the workspace search can be exhaustive or guided by a genetic algorithm (GA), the kernel optimization can be performed following a breadth first search (BFS) that considers each optimization directive individually or a GA that considers all of them at once.

An evaluation performed using a CPU, a GPU and the new Intel Xeon Phi processor shows that OCLoptimizer successfully tunes OpenCL codes for the different platforms. This validation targets codes with both a single and multiple OpenCL kernels.

In codes with a single OpenCL kernel, the achieved speedup is 2.22 when using the GA in the workspace and kernel code search processes and 2.86 when using ES+BFS. In these experiments, the maximum speedup using the GA is 11.07, while using ES+BFS it is 23.33. Notice that although the speedups of GA are more modest than those of ES+BFS, the searches guided by the GA are, on average, ten times faster than those that rely on ES+BFS, which makes it more attractive in some scenarios. Focusing on ES+BFS, the average speedups it achieves are 1.59, 2.54 and 4.46 for the CPU, the GPU and the Intel Xeon Phi, respectively. These speedups show that all the platforms benefit from the usage of our tool, the effect being stronger in the accelerators. This is not surprising, as accelerators are known to be more sensitive than CPUs to code and workspace changes. Both kinds of optimizations are very important, as in every device considered we have found situations in which one of them gave place to the biggest performance improvement.

The IS benchmark of the SNU NPB has been used to validate the support of the tool for codes composed of several OpenCL kernels. In this case, the experiments

were run only on the CPU and the GPU, achieving an average global speedup of 1.79 (2.45 for CPU and 1.19 for GPU). These speedups are more modest than those observed in the single kernel benchmarks but the baseline used for these experiments is a hand-tuned state-of-the-art implementation of the benchmark. These experiments confirm that the ES+BFS approach is more effective than the GA search and that most of the speedup comes from optimizing the workspace configuration. However, in this benchmark the largest speedups are achieved in the CPU.

2.7. Related work

Iterative search techniques based on actual runtime measurements [56, 89] or analytical models [37, 38] have been widely used to automatically tune codes for different architectures. On the other hand, while performance portability in the context of parallel languages has been studied for a long time [75], it has lately regained interest due to the heterogeneity of the available accelerators.

For example, the elastic computing framework [120] separates functionality from implementation details using specialized functions. For each of these elastic functions, the framework explores a collection of alternative implementations and then selects the optimal one depending on the computing resources available and some run-time parameters. This work is limited by the fact that the code has to be expressed using the available specialized functions. This important limitation is not so strong in OCLoptimizer, which processes native OpenCL code written using special macros and annotated with pragmas.

Iterative compilation is used in [23] to select the optimal parameters for GPU codes in a given platform according to a set of pre-defined parametrized templates. This work is specifically focused on obtaining a portable linear algebra library by selecting optimal parameters specific for such operations. OCLoptimizer targets any OpenCL platform, and it selects optimized workspace configurations and unroll factors for any input code.

From the point of view of providing an adaptive scheduling, StarPU [10] automates the efficient mapping of tasks in heterogeneous environments, although it cannot tune the performance of each individual task. However, OCLoptimizer does

tune the performance of individual tasks.

VForce [71] provides performance portability in a transparent way across different kinds of accelerators to programs written in the VSIPL++ (Vector Signal Image Processing Library extension), a domain-specific language focused on image and signal processing. The auto-tuner presented in [21] works on top of the SkePU [25] skeleton programming framework. It performs a previous machine learning process to predict the best execution plan for applications running in multi-GPU systems. The PARTANS framework [64], which is specifically designed to express stencil computations in such systems, includes auto-tuning mechanisms to optimize the task partitioning of computations. By design, this optimization tunes indirectly some domain-specific aspects of the kernels. All these approaches share their domain-specific nature, such a limitation not being present in OCLoptimizer as it targets OpenCL kernels no matter the particular domain of the problem they solve.

Orio [44] is an extensible framework for the generation and empiric evaluation of optimized codes for multiple targets. Like OCLoptimizer, the auto-tuning tools built on top of Orio rely on annotated kernels. However, these annotations must provide a thorough description about the target environment, the optimization conditions and the computation performed. Thus, they include details such as some program building options or the input sizes for the problem, the optimization parameters, and how these parameters are mapped to the different optimizations supported by the tool. Exhaustive, randomized, simplex, and simulated annealing search strategies can be used for tuning the values for the parameters. By default, all these algorithms are driven by the execution time of the generated codes. Moreover, in the annotations requesting the optimization of a particular code section, a copy of that section must be also included, i.e., the framework does not parse the original code but that one replicated inside the annotation. OCLoptimizer proposes a lighter format for the annotations, which redounds on cleaner annotated kernel files. On the one hand, all the configuration parameters that are not directly related to the kernel transformations themselves are specified in a separated file, and, on the other hand, it does not require the user to replicate the code inside the annotation but parses it directly by means of Clang.

An example of an auto-tuner built on top of Orio is orCUDA [65], which generates complete optimized CUDA code from an annotated C loop. These annotations

drive an iterative optimization process to select the size of the grid of threads, the size of the the thread blocks and certain parameters of different optimization techniques, including loop unrolling. This work is focused on CUDA, so it cannot be used to tune codes for CPUs or other accelerators, including non-Nvidia GPUs, and, although they present examples with more complex codes, the validation only uses small kernels, with a single loop, which are used in the the resolution of partial differential equations. OCLoptimizer also targets non-Nvidia GPUs, CPUs and other accelerators as it is based on OpenCL, and the codes included in our experimental results are more complex than those in this work.

Focusing on OpenCL, OrCL [17] is an auto-tuner also built on top of Orio, and hence it is based on parametrized kernel annotations. These parameters allow to tune the iteration distribution among work-items in both global and local workspaces, the unroll factors for inner compute loops, and whether work-items in a group should copy input data chunks located in global memory into local memory before operating on it. OCLoptimizer does not automate this usage of local memory as a cache, although it does support input kernels already tuned in such a way. Moreover, OrCL can be also commanded to provide the compiler with some hints about the sizes for the global and local workspaces and the vector lengths that the OpenCL auto-vectorizer can try. Such hints are translated into kernel attributes added to the code. Unlike OCLoptimizer, unrolling is performed by adding `#pragma unroll` annotations on kernels instead of effectively transforming the loops. Furthermore, the unroll of the loops derived from the iteration distribution adjustment is not supported. The tool is validated by optimizing five simple linear algebra kernels for several NVIDIA and AMD GPUs, and for an Intel Xeon Phi accelerator. The codes used to validate OCLoptimizer were more complex, some being part of an application with inter-dependent kernels, and also an Intel multicore CPU was used as target device. Both OrCUDA and OrCL are also able to use the information provided in the annotations to built CUDA host function calls and OpenCL host codes respectively.

The uCLbench microbenchmarking suite [109] offers a tool to characterize the properties of the devices available in a platform and the OpenCL implementations installed on them. The results of this profiling process are translated into guidelines that programmers can follow to tune their codes manually. The main changes re-

quired to port the performance of OpenCL codes that have been tuned for GPUs to CPUs are discussed in [60] and [99]. A common point in both papers is the importance of adapting the granularity of a kernel depending on the kind of the target device, which is one of the transformations applied and tuned by OCLoptimizer.

GLOpenCL [20] is a unified development framework that supports OpenCL on different types of multicores. This framework consists of a compiler and a runtime library. The compiler is based on LLVM and it performs a set of source-to-source transformations such as serialization of logical threads, elimination of synchronization operations and variable privatization. Its effectiveness is validated by testing five different kernels on different multicore platforms. The results show that the performance achieved using GLOpenCL is close to that obtained by vendor-provided implementations. Unlike OCLoptimizer, that tool does not select an optimized workspace configuration.

Finally, Dolbeau et al [22] discuss the performance differences observed when the same OpenCL code is run on different platforms. They use the CAPS compiler to generate autotuned OpenCL code. This compiler can optimize the group size but not the global workspace. Nevertheless, OCLoptimizer obtains important performance gains from the selection of both the global and the local workspace sizes.

Chapter 3

Self-adaptive HPL kernels

The Heterogeneous Programming Library (HPL) [114] is a C++ framework that improves the programmability of heterogeneous systems. To achieve this purpose, it provides an embedded kernel language to express parallelized computations, and an higher-level API that makes the execution of these kernels considerably easier than through a conventional OpenCL host code. HPL uses OpenCL kernels as its back-end, so that it inherits its functional portability, and hence can be termed as a unified programming mechanism for heterogeneous systems. Moreover, the framework provides programmers with the necessary tools to make their kernels also performance-portable. Namely, a proper combined usage of the embedded kernel language and plain C++ code constructs enables the run-time code generation (RTCG) capabilities of the framework, which can be exploited to write self-adaptive generic kernels. While other tools enable RTCG using similar mechanisms [11, 18], they only target regular CPUs.

This chapter introduces a set of techniques implemented following the aforementioned approach, and which can be used as building blocks to develop self-optimizing kernels in HPL. We explain these techniques focusing on the matrix multiplication algorithm as a case study. The adaptability of the resulting implementation relies on several configuration parameters of each kernel that drive certain aspects about how its code is generated and optimized. The values for these parameters are adjusted along an iterative search process based in a genetic search algorithm, which generates and runs a kernel for each combination of values to test. The parametrized

implementation of these optimization techniques allows to select tuned unroll factors for some loops, a tuned granularity for the work performed by each instance of the kernel, which variant of an algorithm is going to be used, which data structures are stored in local memory, the best loop ordering and the best vector size. Our self-optimizing matrix multiplication is based on existing implementations for NVIDIA GPUs [59], AMD GPUs [68], and any kind of devices supporting OpenCL [110], this latter one being a solid foundation for a performance-portable approach. Our implementation uses not only techniques similar to those introduced in these previous works, but also new ones. The performance of our kernels is compared to two state-of-the-art adaptive implementations, clBLAS [19] and ViennaCL [110]. These two implementations were chosen because (a) they use OpenCL, and thus, they target the same range of platforms as HPL, and (b) they provide adaptive mechanisms to enable performance portability. Our study also covers the OpenCL-based clMAGMA library [15], as it relies on clBLAS for its OpenCL BLAS routines.

The rest of the chapter is organized as follows. Section 3.1 introduces the HPL framework, describing both its main features and its architecture, and then focusing on the fundamentals that users must know to program their own kernels. Section 3.2 explains first how the RTCG capabilities of HPL can be enabled and used to write parametrized generic kernels, then a set of optimization techniques implemented in that way are described, and finally a search process to find suitable values for these parameters is outlined. Section 3.3 focuses on the case study, presenting the matrix multiplication algorithm implemented as an HPL self-adaptive kernel, its tunable parameters, and how optimized values for them are found by means of a genetic search. The experimental results obtained in several platforms are discussed in Section 3.4. Our conclusions about the development of this tool are exposed in Section 3.5, followed by a review in Section 3.6 of some related pieces of work about other approaches for both the generation of self-adaptive code and the optimization of linear algebra routines in heterogeneous devices.

3.1. The Heterogeneous Programming Library

The Heterogeneous Programming Library (HPL), which is publicly available at `http://hpl.des.udc.es`, intends to improve the programmability of heterogeneous

systems while providing portability through an approach where parallelism is extracted by means of computational kernels written in an embedded language built on top of C++. These kernels are translated into an intermediate representation (IR) which is currently OpenCL code. By targeting the same range of devices supported by OpenCL, the library provides functional portability to the same extent as OpenCL. Moreover, the way the kernel language is built on top of C++ enables the use of this latter language as metaprogramming mechanism to generate different kernel codes on run-time. These generic programming and run-time code generation (RTCG) features can be exploited to provide performance portability, and also to automate it to some extent. Thus, the self-adaptive kernels approach we present in this chapter relies on such powerful capabilities to achieve both code and performance portability.

3.1.1. Framework architecture

The architecture of the HPL library is clearly inspired on that of the OpenCL standard, being also organised in several models that define its parts. Regarding the hardware model, it is composed by a host equipped with a standard CPU and memory, with a number of computing devices attached. The host runs the sequential parts of the code, while the devices run the parallel parts. Each device has a number of processors that execute SPMD parallel code on data present in the memory of their device. The memory model distinguishes the same kinds of memory as OpenCL, this is, global, local, constant and private memory regions, each with the same properties.

While all the processors in a device must run the same code in SPMD mode, processors in different devices can run different codes. This way, the library supports both data and task-parallelism. Likewise OpenCL, the host memory space is separated from the global memory space of each device. The kernels can work only with data available in the devices, hence an automated mechanism to transfer input data from the host to the devices, and output data backwards, is provided.

Parallel computations are expressed as kernels written in the embedded language. Several instances of each kernel, or work-items using OpenCL terminology, can be executed in parallel, each instance being univocally identified. The number

of instances of the kernels and their identifiers are defined by a global domain of non-negative integers with up to 3 dimensions. This way, instances are identified inside this domain with tuples of global identifiers. In turn, these instances can be associated in groups. With this purpose, we can define local domains as equal portions of the global domain. Instances are identified inside its group using tuples of local identifiers. As in OpenCL, threads in such groups can be synchronized through barriers in order to share a small scratchpad memory.

In the OpenCL running example developed in Section 2.1.4 we described the process that a user must follow to write an OpenCL host code able to run a kernel on the minimum environment needed. We also exposed the difficulties that inexperienced users may experience when dealing with concepts like execution contexts, compilation of kernel objects or command queue management through the host API provided by OpenCL for such purpose. HPL also contributes to mitigate this problem by offering a higher level interface that hides or even automates many details related to host programming.

In addition, generic programming is enabled by supporting the use of templates both in kernels and data types, and plain C++ code can be included in kernels to exploit the aforementioned run-time code generation capabilities. Such features considerably simplify the procedures of run-time generation and selection of multiple kernel versions. This Chapter thoroughly explanins how such capabilities can be exploited.

3.1.2. Programming front-end

The library provides users with a programming front-end which is composed of the following three main components:

- A **template class** `Array` to define both the variables to be transferred between the host and the devices, and the variables that are local to the kernels.
- The **kernels**, which are functions that express computations in the embedded language provided by the framework.
- A **host API** that will be used by the code to inspect the devices available in a platform and to order the execution of the kernels.

The code in Listing 3.1 implements in HPL the SAXPY (Single-precision real Alpha X Plus Y) vector BLAS routine, which computes $Y = a \times X + Y$. This code will be used as a running example to support the description and some indications about the usage of these components. Let us start pointing out line 1 of the example, which must be included in any code using the library. Line 2 relies on the C++ `using namespace` construction to be able to use the functions and data types imported from the HPL library without having to precede them by their namespace in each use.

The `Array` data type

All the kernel variables must have the type `Array<type, n [, memFlag]>`, which represents an n-dimensional array of elements of a C++ `type`, or a scalar for `n`=0. The flag `memFlag` is used to indicate the kind of memory in which the variable is stored, the four types of OpenCL memory available being identified by the values `Global`, `Local`, `Constant` and `Private`. The elements that compose an array may be any of the usual C++ arithmetic types or a struct.

As we have just said, when `n` is 0 the variable is a scalar indeed. The library provides also some convenience types (`Int`, `UInt`, `Float`, `Size_t`, ...) that simplfy the definition of scalars of their respective C++ types. Such types can be used both in the host code (line 9) and in the kernel function (line 4, third argument of the function). As the native OpenCL code, HPL offers also vector types such as `Int2`, `Float4`, etc. These vectors can be indexed to access their components

```
1  #include "HPL.h"
2  using namespace HPL;
3
4  void saxpy(Array<float,1> y, Array<float,1> x, Float a) {
5      y[idx] = a * x[idx] + y[idx];
6  }
7
8  int main(int argc, char *argv) {
9    Float a;
10   Array<float, 1> x(1000), y(1000);
11   //x, y and a are filled in with data (not shown)
12   eval(saxpy)(y, x, a);
13 }
```

Listing 3.1: HPL SAXPY running example

and manipulated with several functions and operators, including the standard ones. Both vectors and scalars can be mixed too to perform computations.

The arrays passed as input/output arguments to the kernels must be declared in the host memory space. These variables are initially stored in the host memory, but the library detects their usage as kernel arguments and in such case, if needed, it automatically allocates a buffer for each one in the required device and performs the appropriate data transfers. When a host array or kernel argument declaration has no specification for the `memoryFlag`, it is asummed as `Global`. The arrays `x` and `y` declared in line 10 are examples of this. In turn, variables defined inside a kernel cannot be tagged as `Global` or `Constant`, but they are `Private` by default. Nevertheless, they can be also defined as `Local`, such arrays being shared by all the threads in a group.

Embedded kernel language

Along with the usage of the data types described, the HPL kernels also require their control flow structures to be written using both special keywords and a formatting slightly different from those of C++. Namely, the constructs are the same as in C++ but their name finishes with an underscore (`if_`, `for_`, ...). Moreover, the arguments to `for_` loops are separated by commas instead of semicolons.

The library provides an API based on predefined variables to obtain the global, local and group identifiers as well as the sizes of the domains and numbers of groups. For example, `idx` provides the first dimension of the global identifier of a work-item, while `szx` provides the global work size for that dimension. If we add the `l` prefix

Meaning	Dimension number		
	First (`x`)	Second (`y`)	Third (`z`)
Global identifier	`idx`	`idy`	`idz`
Local identifier	`lidx`	`lidy`	`lidz`
Global domain size	`szx`	`szy`	`szz`
Local domain size	`lszx`	`lszy`	`lszz`
Group id	`gidx`	`gidy`	`gidz`
Number of groups	`ngroupsx`	`ngroupsy`	`ngroupsz`

Table 3.1: HPL predefined variables

to these keywords we obtain their local counterparts, and if we replace the letter `x` with `y` or `z`, we obtain the same values for the second and the third dimensions respectively. Table 3.1 gathers all the predefined variables provided by HPL.

The HPL kernels are written as regular C++ functions that use the aforementioned elements and whose parameters are passed by value if they are scalars, and by reference otherwise. In our SAXPY running example, the function defined in lines 4-6 from Listing 3.1 implements a kernel for which each instance `idx` computes a different position of the result `y[idx]`.

However, with the current implementation, our running example suffers from the drawback that it is specialized for computing just `float` vectors. HPL offers a very useful feature to overcome this issue, since its kernel functions can also be instantiations of C++ function templates. The function shown in Listing 3.2 implements a templated version of the SAXPY kernel[1]. Notice also how the templated type definition requires the scalar argument `a` to be typed as `Array<T,0>`.

```
1 template <typename T>
2 void saxpy(Array<T,1> y, Array<T,1> x, Array<T,0> a) {
3     y[idx] = a * x[idx] + y[idx];
4 }
```

Listing 3.2: Generic HPL SAXPY function example

Moreover, HPL provides several functions that are very useful when developing more complex kernels. For example, some kind of synchronization mechanism is needed to control the access to shared portions of data when threads have to read data that other threads have written. This is achieved by means of the HPL `barrier()` function, which performs a barrier synchronization among all the threads in a group. It expects an argument indicating the memory scope, either local (`LOCAL`) or global (`GLOBAL`) or both (`LOCAL|GLOBAL`), for which a coherent view must be kept for all the threads after the barrier. The library also supports the definition of internal HPL functions to be invoked within an HPL kernel. Such invocations must be done by means of `call()`. For instance, `call(f)(a,b)` works as expected, calling the function `f` with the arguments `a` and `b`. The first time a function is called by means of `call()`, HPL internally generates the code for the routine and compiles

[1]We are aware that the name saxpy no longer makes sense when the data used are no longersimple-precision elements. We use the same name for the sake of simplifying the example.

it, subsequent `call()` occurrences for the same function just invoking it. In turn, when a plain C++ invocation is done, the code of the routine is generated and then inlined inside the code of the calling function. This latter way of invocation is only valid for functions that do not include an HPL `return_` statement.

Such dual behavior of HPL depending on how functions are invoked raises the issue about how its kernels are translated into a runnable binary for a given device. This process is transparent for the user and it consists of two separate steps. First, the kernel is *instantiated* by running it as a regular C++ code compiled along with the rest of the host application. The fact that kernels are written in its own embedded language allows the library to capture aspects of the code such as the data definitions and manipulations, or the control flow structures, and then use all the information gathered to build a suitable intermediate representation (IR). In the second step of the aforementioned process this representation is compiled into a binary for the target device. The implementation of HPL on top of which the self-adaptive kernels explained in this chapter are built relies on OpenCL C as IR, the generated code being thus functionally portable across OpenCL-supported devices. There are not, however, any restrictions precluding the usage of other IRs as a back-end. In fact, the just-in-time optimizer described in Chapter 4 translates HPL kernels into a tree-like IR instead of generating OpenCL C equivalents straightaway.

Since the instantiation of an HPL kernel starts with it being run as a regular C++ routine, both variables of C++ standard types and code constructs like control flow structures can appear in the kernel. However, such C++ code fragments do not appear directly in the kernel IR. Rahter, they work as metaprogramming instructions that drive the translation process. Self-adaptive kernels are built on top these run-time code generation capabilities (RTCG) of HPL, which allow to apply different optimizations and then produce different code versions depending on the device targeted. The details about how such features can be exploited will be given when introducing such kernels.

Host interface

The most important component of the host interface is the function `eval`, which requests the execution of the kernel `f` with the syntax `eval(f)(...)`, the argument

```
1  void saxpy(Array<float,1> y, Array<float,1> x, Float a) {
2    y[idx] = a * x[idx] + y[idx];
3  }
4
5  int main(int argc, char *argv) {
6    float my_y[1000];
7    Float a;
8    Array<float,1> x(1000), y(1000,my_y);
9    // myy, myx and a are filled in with data (not shown)
10   eval(saxpy).device(GPU).global(1000).local(10)(y, x, a);
11 }
```

Listing 3.3: Array usage and workspace configuration on SAXPY running example

`f` being the kernel. The execution of the kernel can be parametrized by inserting specifications, in the form of methods, between the `eval(f)` invocation and the argument list. Line 10 in Listing 3.3 shows some examples of such methods and how they are used in the SAXPY HPL code.

One of the properties that can be specified after calling `eval()` is the configuration of the kernel execution domains introduced in Section 3.1.1. Thus, in our example, by invoking `global(1000)` a unidimensional global domain of 1000 elements is set, the further call of `local(10)` dividing that global domain into local domains of 10 elements each. When not specified, by default the global size is equal to the size of the first argument, whereas the local size is automatically selected by the library. Another property configurable by means of such chained calls is the execution target device. In our example, `device(GPU)` orders HPL to run the kernel on the first GPU available. By default, the library picks the first device in the system which is not a standard CPU. If no alternative is found, the kernel is run on the CPU. Moreover, some device management operations are also provided to allow the user to choose any of the computing devices available.

The SAXPY example from Listing 3.3 also shows in Line 8 how the host arrays to be passed as kernel arguments can be created. Here, array `x` is created from scratch, making the library responsible for allocation and deallocation of its host memory space. Nevertheless, array `y` is defined providing a pointer `my_y` to a host buffer already allocated in Line 6. Thus in this case the library will not allocate any memory for this array in the host, using instead the one provided. In the devices, however, the management will be analogous to that of any other array.

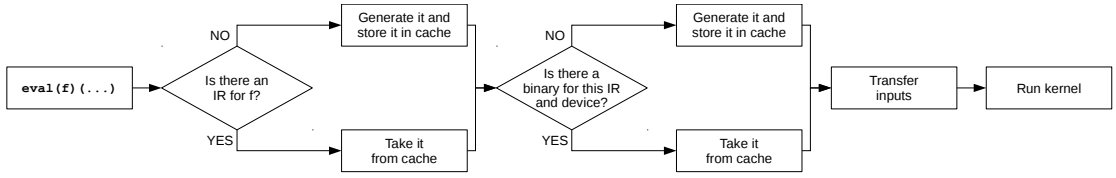


Figure 3.1: HPL kernel invocation algorithm

The flowchart in Figure 3.1 depicts the sequence of steps performed by the library when a kernel is invoked for execution. The library keeps an internal cache with the IRs previously generated, so that this cache is sought first to check whether a translation for the invoked kernel is already available. If this is not the case, the kernel is instantiated and the IR is stored in the cache. Once the IR is ready, a second internal cache is queried to check whether such IR was already compiled into a binary for the target device. In a similar vein, if there is no binary available, the IR is compiled and the resulting binary is cached. At this point, HPL transfers to the device just the data needed for the execution and, finally, the kernel is launched for execution.

By keeping both IR and binary caches, the instantiations and compilations of the kernels are minimized, since each kernel is translated into its IR just the first time it is used, and such IR is compiled into a binary only if it does not exist for the target device. However, in some situations it could be interesting to regenerate a kernel, for instance when C++ code constructs are included to exploit the run-time code generation capabilities of the library. In such cases, users may vary the behavior of that C++ code in order to generate different versions of a same HPL kernel. This feature, which is one of the foundations of our self-adaptive kernels approach, is provided by the `reeval()` function. It follows the same syntax as `eval()`, but t forces the instantiation of a kernel no matter previous versions are available in the HPL caches or not.

Support for native OpenCL C kernels

The HPL embedded language has both identical semantics and an analogous syntax to those of C, which reduces the programming effort needed to transform any OpenCL kernel written in C into an HPL kernel. However, users may prefer to keep

their original codes instead of translating them, and hence the library provides way to include native kernels in HPL applications. This mechanism requires users to define a kernel handle that takes the form of a regular C++ function, and then to associate it to the native kernel code. This is achieved by means of the `nativeHandle()` function, which takes as arguments a pointer to the handle function, a string with the name of the native kernel function to link and the string containing the OpenCL source code. Once the link is set, the native code can be invoked for execution by invoking `eval()` using the handle as the argument. The `saxpy_simple` SAXPY OpenCL native kernel is implemented in lines 2-7 of Listing 3.4, the code being stored in the `kernel_code` string. The handle is defined as the `saxpy_ocl` C++ empty function in line 10. Then, they are linked in line 17 and, finally, the handle is invoked with `eval(saxpy_ocl)` in line 18, which eventually leads to the native kernel execution.

```
1  const char * const kernel_code = TOSTRING(
2    __kernel void saxpy_simple(__global float *y,
3                         const __global float *x, const float a)
4    {
5      size_t idx = get_global_id(0);
6      y[idx] = a * x[idx] + y[idx)];
7    }
8  );
9
10 void saxpy_ocl(Array<float,1> y, In<Array<float,1>> x, Float a) { }
11
12 int main(int argc, char *argv) {
13   float my_y[1000];
14   Float a;
15   Array<float,1> x(1000), y(1000, my_y);
16   // myy, myx and a are filled in with data (not shown)
17   nativeHandle(saxpy_ocl, "saxpy_simple", kernel_code);
18   eval(saxpy_ocl).device(GPU).global(1000).local(10)(y, x, a);
19 }
```

Listing 3.4: SAXPY running example using a native kernel

3.2. Towards performance-portable kernels

By now we have introduced the HPL architecture and its programming basics, proving also that, since it translates the user-written kernels into OpenCL code, it provides functional portability on heterogeneous environments. There are some advanced features of the library, namely its run-time code generation capabilities (RTCG), that can be exploited to make a single HPL base kernel be able to give place to different OpenCL versions. Thus, performance portability can be achieved on top of such HPL kernels if the conditions driving the code generation process depend to some extent on the properties of the given target device.

The reminder of this Section is organized as follows. First, in Section 3.2.1 we develop another running example to introduce the aforementioned run-time code generation capabilities, which are used in turn in Section 3.2.2 to program parametrized HPL kernels. These kernels produce different OpenCL versions depending on the values set at run-time for a number of parameters. Thus, in Section 3.2.3 we propose such a parametrized HPL implementation for a set of well-known optimization techniques. Finally, in Section 3.2.4, we recall the search strategies implemented in the kernel optimization process performed by OCLoptimizer and outline an adaptation of the genetic algorithm to find optimized combinations of values for the parameters required by the set of techniques presented.

The combination of HPL kernels implemented using such parametrized optimization techniques and a search process able to find suitable values for those parameters depending on the target device gives place to the concept of performance-portable self-adaptive kernels.

3.2.1. HPL run-time code generation capabilities

When an HPL kernel is instantiated, it is first run as a regular C++ routine, which allows to introduce C++ variables and code constructs in it. Regarding the variables, they will not appear in the resulting IR. Rather, they will be respectively replaced by constants with their values at the points of the kernel in which they interact with the HPL embedded language elements. In relation to code constructs such as computations or control flow structures, they will simply be executed during

```
1  int M = ...;
2  int N = ...;
3
4  void mxv(Array<float,2> a, Array<float,1> x,
5        Array<float,1> y, Int n)
6  {
7    Int k;
8    for_(k=0, k<n, k++)
9      y[idx] += (a[idx][k] * x[k]);
10 }
11
12 int main(...) {
13   ...
14   Array<float,2> a(M,N);
15   Array<float,1> x(N), y(M);
16   ...
17   eval(mxv).global(M)(a, x, y, N);
18 }
```

Listing 3.5: MxV code: base version with kernel as a function

the instantiation of the kernel. In this way, they can be used to compute at runtime values that can become constants in the kernel, to choose among different HPL code versions to include, or to simplify the generation of repetitive codes.

Let us illustrate these uses by means of another running example, this one implementing a matrix-vector product in HPL. Lines 4-10 of Listing 3.5 contain the base kernel of this example. Here, each element `idx` of a global domain of size `M` computes the dot product of the `idx`-th row of the matrix `a` and the input vector `v`, accumulating the result in the `idx`-th component of the output vector `y`. Any programmer might want to optimize this base kernel by, for instance, unrolling the dot product loop of lines 8-9. Several programming issues that can be addressed by means of the RTCG capabilities of HPL can be considered at this point. First, there is no doubt that unrolling a loop implies some code rewriting operations. Roughly speaking, the instruction that updates the loop counter must be adapted, the loop body must be replicated as many times as the unroll factor indicates, and the occurrences of the loop counter in each replicated instruction have to be adapted too.

The kernel in Listing 3.6 shows how regular C++ code can be used to enrich the base HPL kernel and considerably simplify this optimization process. In this new version, the original dot product loop (lines 8-9 from Listing 3.5) has been modified

```
1  int M = ...;
2  int N = ...;
3  int UF = ...;
4  int SMALL_N = ...;
5
6  void mxv(Array<float,2> a, Array<float,1> x, Array<float,1> y)
7  {
8    Int k;
9    if(N >= SMALL_N) {
10     for_(k=0, k<N, k+=UF) {
11       for(int uf=0;uf<UF;uf++) {
12         y[idx] += (a[idx][k+uf] * x[k+uf]);
13       }
14     }
15   }
16   else {
17     for_(k=0, k<N, k++) {
18       y[idx] += (a[idx][k] * x[k]);
19     }
20   }
21 }
```

Listing 3.6: MxV code: kernel with C++ constructs

(lines 10-14) to make it able to unroll itself with an unroll factor `UF`. Notice how the body of the HPL `for_` loop now contains a C++ `for` (lines 11-13), instead of the single instruction of the original loop. When an HPL kernel is instantiated, the C++ loops found are just executed. In this example, `UF` copies of an adapted version of the single instruction from the original loop body will be generated. The adaptation consists in adding the `uf` counter of the regular loop to the `k` counter of the HPL `for_` loop. Thus, the occurrences of `uf` in each copy will be replaced by the corresponding value in each unrolled version, from `k+0` to `k+(UF-1)`. This happens because the `uf` counter itself is a C++ variable, and thus its value is captured in each iteration in the associated copy generated. Furthermore, the instruction updating the counter of the `for_` of line 10 must be also changed to ensure that the HPL loop advances in steps of the same length as the unroll factor. As `UF` is a global C++ variable, the value set by the user for this variable will be captured or frozen in the IR code generated by HPL.

Let us assume now that, if the input matrix `a` has less columns than a given value `SMALL_N`, the programmer prefers to keep the original version of the kernel rather

than to unroll the dot product loop. C++ `if` blocks can be used to manage such situations. Likewise the regular `for` loops, `if` blocks are simply run when a kernel is instantiated and, therefore, only the code contained into the branch that is executed will be generated. In our example, the `if` whose condition is defined in line 9 plays this role. When the matrix `a` has enough columns, the code in lines 10-14 is instantiated, otherwise, the original implementation kept in lines 17-19 is processed. Other details relevant when unrolling a loop, namely the treatment of the remaining iterations when the total number is not divisible by the factor applied, have been omitted in this introductory example for the sake of clarity. This issue is covered in the loop unrolling explanation given in Section 3.2.3.

3.2.2. Programming parametrized HPL kernels

We have just seen how, depending on the values taken by some global variables, different versions can be generated for the same base HPL kernel. However, even though those global variables are an effective mechanism for parametrizing the generation of multiple versions, the way these variables are defined introduces some inconveniences that hamper the natural flexibility of this programming approach. By now, all the kernel functions defined were global, which forced the C++ variables included in the kernels to be defined as global too. This is untidy, as the relation between the variables and the kernel they parametrize is not obvious. In addition, as the number of kernels in our application grows, so does the number of different variables in the global space of the application needed to parametrize them, several of them having possibly the same meaning, but for different kernels. This clutters the global space and increases the possibilities of programming errors, besides going against basic principles of software engineering such as encapsulation. Fortunately, the object oriented properties of C++ coupled with the large variety of kinds of kernels supported by HPL provides an easy solution to this problem.

Any class that defines a `operator()` method is a functor. This operator allows the objects of such classes to be treated as functions and, furthermore, since they are also plain C++ classes, state information can be stored in their attributes. This way, any HPL kernel can be implemented as the `operator()` method of a class, and the variables that drive its code generation process can be stored in the class

properties rather than in the global space as independent variables. Listing 3.7 shows the implementation of the `MxV` functor class, equivalent to the example kernel from Listing 3.6. Notice how now the global variables `N`, `UF` and `SMALL_N` are defined as properties of the class (line 4) and the kernel is implemented as the method `operator()` of the class (lines 10-25). Moreover, the class should also provide some getters and setters for its properties, although they have been elided for clarity (line 8).

```
1  class MxV
2  {
3    private:
4      int N, UF, SMALL_N;
5
6    public:
7      ...
8      // Getters/setters for properties elided
9      ...
10     void operator()(Array<float,2> a, Array<float,1> x, Array<float,1> y)
11     {
12       Int k;
13       if(N >= SMALL_N) {
14         for_(k=0, k<N, k+=UF) {
15           for(int uf=0;uf<UF;uf++) {
16             y[idx] += (a[idx][k+uf] * x[k+uf]);
17           }
18         }
19       }
20       else {
21         for_(k=0, k<N, k++) {
22           y[idx] += (a[idx][k] * x[k]);
23         }
24       }
25     }
26 };
27
28 int main(...) {
29   ...
30   int m, n, iuf, small_n;
31   ...
32   Array<float,2> a(m,n);
33   Array<float,1> x(m), y(n);
34   ...
35   MxV mxv;
36   mxv.set_N(n);
37   mxv.set_UF(iuf);
38   mxv.set_small_n(small_n);
39   eval(mxv).global(m)(a, x, y);
40   ...
41   mxv.set_UF(<new unroll factor>);
42   reeval(mxv).global(m)(a, x, y);
43   ...
44 }
```

Listing 3.7: MxV code: functor implementation and usage

Lines 28 to 44 of Listing 3.7 contain the sketch of an HPL application using the functor class defined above. First, an object `mxv` of the `MxV` functor class is instantiated (line 35). Then, the setter methods of the class are invoked (lines 36-38) to store the values contained in the user-defined C++ variables from line 30. Finally, the HPL kernel implemented inside the functor is launched for execution by calling `eval()` in line 39. When the kernel invocation process was explained in Section 3.1.2, we said that function `eval()` expects a function implementing an HPL kernel as argument. As an instance of a functor class, the object `mxv` can be treated as a function, which makes it a valid argument for `eval()`. Thus, the library will instantiate the HPL kernel contained in the `operator()` method, reading the values from the object properties `N`, `UF` and `SMALL_N` when needed. In order to try a version of the same base kernel applying a different unroll factor, first a new value must be set for the attribute `UF` (line 41), and then the kernel must be executed with a call to `reeval()`, which will give place to a reinstantiation using the new unroll factor (line 42). This programming mechanism is so flexible that it allows users, for instance, to write an HPL application able to iterate on a list of unroll factors, generate a version for each one, and evaluate the performance of the resulting code.

3.2.3. Parametrized optimization techniques

Some parametrized optimization techniques that exploit the RTCG capabilities of HPL are now proposed following the explanations in Sections 3.2.1 and 3.2.2. These techniques can be used to build self-optimizing kernels able to: (1) unroll one or several loops using a given unroll factor, (2) apply the tiling technique to one or several loops using a given tile size, (3) select the best granularity of the computation performed by each instance of the kernel, (4) select the most suitable variant of an algorithm depending on the target device, (5) decide which data structures are stored in local memory, (6) select an optimized loop order, and (7) choose an optimized vector size in the vectorized portions of code. These techniques will be illustrated using a functor implementation of the base kernel of our matrix-vector running example. Such implementation is recalled in Listing 3.8.

```
1  class MxV {
2    void operator()(Array<float,2> a, Array<float,1> x,
3                    Array<float,1> y)
4    {
5      Int k;
6      for_(k=0, k<N, k++)
7        y[idx] += (a[idx][k] * x[k]);
8    }
9  };
10 int main(...) {
11   ...
12   MxV mxv;
13   eval(mxv).global(M)(av, xv, yv);
14 }
```

Listing 3.8: MxV code: base version as a functor

Loop unrolling

Loop unrolling is a well-known optimization technique whose main benefits are that it unveils instruction level parallelism, minimizes branch penalty and reduces the number of control instructions executed. Loop unrolling using arbitrary unroll factors can be introduced in HPL kernels using RTCG. C++ code will be used in conjunction with the HPL embedded language to generate the unrolled loops. Let us see an example starting from the matrix-vector product (MxV) code shown in Listing 3.8. This code defines the HPL kernel in lines 2-8. Each instance of the kernel processes one row from the input matrix, thus a single loop is required to multiply each element of the row by the corresponding element of the input vector.

Listing 3.9 shows an unrolled version of the kernel. The loop in lines 6-9 is an unrolled version of the original loop, thus, its stride is now the unroll factor (`uf`). The body of the loop is replicated `uf` times by a native C++ loop (lines 7-8). As the number of iterations of the loop `N` may not be a multiple of `uf`, the loop limit is set to `N-uf` in order to prevent out of range array accesses. If there are iterations left after that loop, they are processed without unrolling by the code in lines 10-11. The value for the unroll factor is provided to the kernel from the main procedure by setting the appropriate attribute of the class that defines the kernel (line 17).

```
1  class MxV { //Other portions of the class have been elided
2    void operator()(Array<float,2> a, Array<float,1> x,
3                    Array<float,1> y)
4    {
5      Int k;
6      for_(k=0, k <= (N - uf), k += uf) {
7        for(aux=0; aux<uf; aux++)
8          y[idx] += (a[idx][k+aux] * x[k+aux]);
9      }
10     for_(k,k<N,k++)
11       y[idx] += (a[idx][k] * x[k]);
12   }
13 }
14 int main(...) {
15   ...
16   MxV mxv;
17   mxv.set_uf(unrolling_factor);
18   eval(mxv).global(M)(av, xv, yv);
19 }
```

Listing 3.9: MxV code: unrolled version

Loop tiling

An adequate exploitation of the memory hierarchy available is a determining factor when trying to make a code perform well on multiple heterogeneous devices. Loop tiling is a generic programming technique extensively applied to improve such exploitation. Instead of keeping the loops iterating on the full dimensions of the structures, this transformation breaks the iteration spaces of such loops into smaller tiles. As a result, these structures will be accessed using smaller blocks which are more likely to fit into upper memory hierarchy levels, thus increasing data reuses and lowering data miss rates. Finding an optimized tile size for each situation is the main issue in the application of this technique. Moreover, an adequate selection of this size could lead into a very propitious scenario for applying either unroll or vectorization techniques or both of them in a later step. These combinations of code transformations may maximize the performance of a kernel not only by improving the use of memory hierarchy, but also by increasing the number of independent instructions available to be executed and vectorizing them. The code in Listing 3.10 applies this transformation for a generic `tile_size` to the matrix-vector multiplication example. The original single loop iterating on the whole dimension `N` with step

```
1  class MxV { // Other portions of the class have been elided
2    void operator()(Array<float,2> a, Array<float,1> x,
3                    Array<float,1> y)
4    {
5      Int kk,k,klim;
6      for_(kk=0,kk<N,kk+=tile_size){
7        klim = kk + tile_size;
8        if_(klim >= N) klim = N;
9        for_(k=kk,k<klim,k++){
10         y[idx] += (a[idx][k] * x[k]);
11       }
12     }
13   }
14 };
15 int main(...) {
16   ...
17   MxV matvec;
18   matvec.set_tW(tile_size);
19   eval(matvec).global(M)(av, xv, yv);
20 }
```

Listing 3.10: MxV code: tiled version

1 (see lines 6-7 in Listing 3.8) is replaced with two nested loops: the outermost one (lines 6-12) iterates on `N` in chunks of size `tile_size`, whereas the innermost one (lines 9-11) does it on the corresponding tile in an element-wise way. Notice how the end limit of this latter loop is defined in order to avoid exceeding the bounds of the original loop when its size `N` is not divisible by the selected tile size (lines 7-8).

Granularity adjustment

HPL creates one instance (or thread in HPL terminology) of the kernel for each point of the global domain. The amount of work performed by each thread must be tuned for each platform in order to maximize performance. For example, CPUs tend to be more effective using threads with larger workloads than GPUs. It is interesting to be able to tune that granularity at run-time depending on the type of device we are using. We can do that in HPL by changing the number of points in the global domain. For example, in our MxV code the number of threads created is equal to the number of rows of the input matrix, thus, each thread processes one row of this matrix. If we reduce the number of threads, each thread should process

```
1  class MxV { //Other portions of the class have been elided
2    void operator()(Array<float,2> a, Array<float,1> x,
3                    Array<float,1> y)
4    {
5      Int ii, i, ilim, k;
6      for_(ii = idx*bszx, ii < M, ii += szx*bszx)
7        for_(i = ii,i < min(xx+bszx, M), i++)
8          for_(k = 0, k < N, k++)
9            y[i] += a[i][k] * x[k];
10   }
11 }
12 int main(...) {
13   ...
14   int szx = <# threads of the global domain>;
15   int bszx = <block size>;
16   ...
17   eval(mxv).device(dev).global(sz_x)(av, xv, yv);
18 }
```

Listing 3.11: MxV code: auto-adjustable granularity version

several rows from the input matrix. In this technique, the adaptability of the code stems from the fact that the code is written in a generic way, based on the value of a set of optimization parameters. Thus, the code has to be rewritten for a generic grain size, the grain size being in this case the number of rows of the input matrix processed by each thread. In our proposal, rows are distributed using a block-cyclic policy, thus, grains of `bszx` rows are assigned cyclically to the available threads. An optimized value of `bszx` should be later found for each device. In the MxV code, this block size will not have a big influence in the performance, but in other problems some values of `bsz` may benefit locality or coalescing, and as a result, they will have a big impact in the performance.

In order to implement this distribution of the rows, the MxV kernel code must be changed to add two outer loops that process the blocks of `bszx` rows assigned to each thread. The loop headers in lines 6-7 of Listing 3.11 select the appropriate set of rows to be processed by each thread following a block-cyclic policy. The resulting kernel is written in a generic way, so that if different values are provided for the size of the global domain and the block size, the granularity of the work performed by each thread is automatically adjusted at run-time.

Algorithm variant selection

Programmers may need to implement different versions of an algorithm or, at least, adapt the implementation of some of its steps depending on the features of the target device. For instance, a version that exploits local memory is good for GPUs, but it may introduce unnecessary synchronization points in CPUs. Similarly, the best strategy to divide work among the participating threads also varies depending on the type of device. These two examples show how properties such as the type of device and the multiple architectural details usually drive these decisions.

By means of the RTCG capabilities of HPL, any native C++ alternative structure will provide the support for this feature. First, the algorithm variants must be distributed among the branches of the alternative structure, and then, the conditions driving the selection must be included where the native control structure expects them. Moreover, thanks to the nature of HPL, the conditions that choose among the code blocks can evaluable either at compile time or dynamically at runtime

Listing 3.12 shows the skeleton of a MxV vector kernel where a different variant of the algorithm would be selected depending on the type of device, which is a property that can be queried at runtime. Notice the native C++ `if-else` structure in lines 5-9 is supporting here the selection mechanism, which will generate one or another implementation depending on the evaluation of the condition set in line 5.

```
1  class MxV { //Other portions of the class have been elided
2    void operator()(Array<float,2> a, Array<float,1> x,
3                    Array<float,1> y)
4    {
5      if (device==GPU) {
6        // Version better suited to GPUs
7      } else {
8        // Default version for CPUs and other devices
9      }
10   }
11 }
```

Listing 3.12: MxV code: algorithm version selection

Memory region selection

By default, any memory structure manipulated by an HPL kernel must be available in the global memory of the target device. Nevertheless, there are performance issues that may make advisable to copy a structure, either sliced or as a whole, to other memories of the device. In GPUs, for instance, the threads in a group share a local memory on which the global memory structures are often cached. Moreover, sometimes it is interesting to exploit the registers of a processor by copying blocks of data to the private memory they belong to, and then performing the computations directly on this memory. This latter optimization is usually applied on CPUs, but it can be also exploited in GPUs, which leads to the combination of both. Thus, such programming variations motivate the implementation of a mechanism to dynamically select the region in which a data structure must be stored.

This selection mechanism, which is a use case of the algorithm variant selection just introduced, is supported by two main code transformations. First, code to copy the structure to a buffer allocated in the desired memory region must be generated. In the MxV example, the kernel is written in such a way that we can choose between storing vector `x` in local memory or keeping in global memory. A

```
1  class MxV { //Other portions of the class have been elided
2    void operator()(Array<float,2> a,
3                    Array<float,1> x, Array<float,1> y,
4                    Array<float,1,Local> lx)
5    {
6      Int k;
7      if(copyX) {
8        for_(k=lidx, k<N, k+=lszx)
9          lx[k] = x[k];
10       barrier(LOCAL);
11     }
12     for_(k=0, k<N, k++)
13       y[idx]+=a[idx][k]*(copyX ? (Float)lx[k] : (Float)x[k]);
14   }
15 }
16 int main(...) {
17   ...
18   eval(mxv).device(dev).global(M).local(lsz_x)(av,xv,yv,lxv);
19 }
```

Listing 3.13: MxV code: local memory usage

boolean parameter `copyX` will be set in the host to indicate whether we want to copy that array in local memory. Listing 3.13 contains the MxV kernel modified to implement this behavior. The kernel uses the run-time code generation capabilities of HPL to generate code to copy `x` to local memory only if `copyX` is activated (lines 7-11). Second, if we have chosen to copy the structure, then all the references to it must be redirected to its copy in the local memory. In the example, either the global array `x` or its local copy will be used depending on the value of the `copyX` parameter in line 13. Notice also how the compact in-line notation for a native C++ `?:` operator has been used to implement this selection.

Loop interchange and instruction scheduling

Loop interchange, when legal, can have a big impact on the performance of a kernel. For example, it can change the order in which n-dimensional structures are traversed. Some traversal orders can reduce the number of required simultaneous registers or favor locality or automatic vectorization detection. Traditionally, the best loop order is either selected by the programmer or optimized at compile-time. In HPL, RTCG capabilities can be used to change the loop order at run-time.

The code in Listing 3.14 shows an example of how this technique can be applied to our matrix-vector product HPL kernel. In the version presented in Listing 3.11 the granularity of the kernel can be adjusted, so that each thread processes the multiplication of `M/szx` consecutive rows of matrix `a` by vector `x`. The product within each thread can be done using the traditional order, where matrix `a` is accessed by rows, or it can be done by traversing per columns the chunk of `M/szx` rows of `a` processed by each thread. This order can be changed by swapping the two loops in the kernel. In HPL, this code transformation can be done at run-time using a new technique based on indirections. Arrays `init`, `end` and `step` have one position per loop (2 in the example) containing the initialization, limit and step of the counters of each one of the actual loops that we want to reorder. This way, we call actual loop `j` the one whose data is stored in the `j`-th position of these vectors. The loops with indices `c[0]` and `c[1]` are just container loops where the real loops are placed. The loop order can be changed modifying the contents of the arrays `ord` and `ptr`. This way, the number of the actual loop `j` to be implemented by the container loop, `i`, with index `c[i]` is stored in `ord[i]`. Also, the references inside the loops

```
1  class MxV { // Other portions of the class have been elided
2    int init[2]={0,0}; int end[2]={M/szx,N}; int step[2]={1,1};
3    int ord[2], ptr[2]; // initialized by set_order
4    void operator()(Array<float, 2> a, Array<float, 1> x,
5                    Array<float, 1> y)
6    {
7      ...
8      Array<int, 1, Private> c(2);
9      for_(c[0]=init[ord[0]],c[0]<end[ord[0]],c[0]+=step[ord[0]]) {
10       for_(c[1]=init[ord[1]],c[1]<end[ord[1]],c[1]+=step[ord[1]]) {
11         y[idx*(M/szx)+c[ptr[0]]] +=
12           a[idx*(M/szx)+c[ptr[0]]][c[ptr[1]]] * x[c[ptr[1]]];
13       }
14     }
15   }
16 };
17
18 int main(...) {
19   ...
20   MxV mxv;
21   mxv.set_order(0,1); // sets ord[0]=1 and ptr[ord[0]]=ptr[1]=0
22   mxv.set_order(1,0); // sets ord[1]=0 and ptr[ord[1]]=ptr[0]=1
23   eval(mxv).global(sz_x)(av, xv, yv);
24 }
```

Listing 3.14: MxV code: version with interchangeable loops

have indexing functions that depend on the indices of the container loops, `c[i]`. Each `ptr[j]` contains the index of vector `c` that implements the actual loop `j`, that is, whenever `ord[i]=j`, then `ptr[j]=i`. This way, any reference to the indexing variable of the actual loop `j` in the original code can be systematically replaced by `c[ptr[j]]`, ensuring that the appropriate loop index will be used no matter which is the loop ordering chosen.

Recalling the example from Listing 3.14, the instruction in line 21 requests that the container loop 0 (`c[0]`) implements the actual loop 1 (`ord[0]=1`). Similarly, the instruction in line 22 configures the container loop 1 (`c[1]`) so that it implements the actual loop 0, (`ord[1]=0`). Regarding the `ptr` array, `ptr[ord[0]]`, which is `ptr[1]` in this order, points to the index of container `c[0]`, and `ptr[ord[1]]`, which is `ptr[0]` in this order, points to the index of `c[1]`. These values command HPL to generate a kernel like the `mxv_loopinter_col` one in Listing 3.15, which performs an access per columns. In turn, when the arrays `ord` and `ptr` are set to

their complementary values, the kernel generated is like the `mxv_loopinter_row` one shown in Listing 3.16, which performs an access per rows. Notice how the order of the loops and the occurrences of the counters (lines 7-9 of both listings) are exchanged among both kernels. This scheme can be generalized for any arbitrary number of loops. Notice that some loop exchanges may be illegal. Thus, the programmer is responsible for checking the legality of the orders tried or at least, for enumerating the set of legal orderings.

The loops exchanged in the example are HPL `for_` loops (lines 9-10), thus they give place to `for` loops when they are translated into OpenCL code. If in this example C++ `for` loops were used instead of `for_` loops, these loops would be executed during the HPL code generation process, which would give place to a fully unrolled

```
1  __kernel void mxv_loopinter_col(__global float *a, __global float *x,
2                                  __global float *y)
3  {
4    size_t szx = get_global_size(0);
5    size_t idx = get_global_id(0);
6    __private int c[2];
7    for((c[0]=0); (c[0]<N); (c[0]+=1)) {
8      for((c[1]=0); (c[1]<M/szx); (c[1]+=1)) {
9        y[((idx*(M/szx))+c[1])]+=(a[((idx*(M/szx))+c[1])*N+c[0]]*x[c[0]]);
10     }
11   }
12 }
13
14 __kernel void mxv_sched_col(__global float *a, __global float *x,
15                             __global float *y)
16 {
17   size_t szx = get_global_size(0);
18   size_t idx = get_global_id(0);
19
20   y[((idx*(M/szx))+0)]+=(a[((idx*(M/szx))+0)*N+0]*x[0]);
21   y[((idx*(M/szx))+1)]+=(a[((idx*(M/szx))+1)*N+0]*x[0]);
22   ...
23   y[((idx*(M/szx))+((M/szx)-1))]+=(a[((idx*(M/szx))+((M/szx)-1))*N+0]*x[0]);
24
25   y[((idx*(M/szx))+0)]+=(a[((idx*(M/szx))+0)*N+1]*x[1]));
26   y[((idx*(M/szx))+1)]+=(a[((idx*(M/szx))+1)*N+1]*x[1]));
27   ...
28   y[((idx*(M/szx))+((M/szx)-1))]+=(a[((idx*(M/szx))+((M/szx)-1))*N+1]*x[1]);
29
30   ...
31
32   y[((idx*(M/szx))+0)]+=(a[((idx*(M/szx))+0)*N+(N-1)]*x[(N-1)]);
33   y[((idx*(M/szx))+1)]+=(a[((idx*(M/szx))+1)*N+(N-1)]*x[(N-1)]);
34   ...
35   y[((idx*(M/szx))+((M/szx)-1))]+=(a[((idx*(M/szx))+((M/szx)-1))*N+(N-1)]*x[(N-1)]);
36 }
```

Listing 3.15: MxV code: OpenCL kernels with loops in column-major order

version of the original loop nest. In addition array `c` should be transformed into a native C++ array. In this case, the loop exchange technique would turn into a instruction scheduling technique, as different loop orders give place to a different order of the same sequence of instructions. The OpenCL kernels `mxv_sched_col` and `mxv_sched_row` from Listings 3.15 and 3.16 show the generic schedules for arbitrary `M` and `N` sizes derived from unrolling the column-major and row-major loop nests, respectively. This instruction scheduling technique is applied on top of a loop reordering mechanism in our matrix multiplication implementation, which will be introduced in Section 3.3.1.

```
1  __kernel void mxv_loopinter_row(__global float *a, __global float *x,
2                                  __global float *y)
3  {
4    size_t szx = get_global_size(0);
5    size_t idx = get_global_id(0);
6    __private int c[2];
7    for((c[0]=0); (c[0]<M/szx); (c[0]+=1)) {
8      for((c[1]=0); (c[1]<N); (c[1]+=1)) {
9        y[((idx*(M/szx))+c[0])]+=(a[((idx*(N/szx))+c[0])*N+c[1]]*x[c[1]]);
10     }
11   }
12 }
13
14 __kernel void mxv_sched_row(__global float *a, __global float *x,
15                             __global float *y)
16 {
17   size_t szx = get_global_size(0);
18   size_t idx = get_global_id(0);
19
20   y[((idx*(M/szx))+0)]+=(a[((idx*(M/szx))+0)*N+0]*x[0]);
21   y[((idx*(M/szx))+0)]+=(a[((idx*(M/szx))+0)*N+1]*x[1]);
22   ...
23   y[((idx*(M/szx))+0)]+=(a[((idx*(M/szx))+0)*N+(N-1)]*x[N-1]);
24
25   y[((idx*(M/szx))+1)]+=(a[((idx*(M/szx))+1)*N+0]*x[0]);
26   y[((idx*(M/szx))+1)]+=(a[((idx*(M/szx))+1)*N+1]*x[1]);
27   ...
28   y[((idx*(M/szx))+1)]+=(a[((idx*(M/szx))+1)*N+(N-1)]*x[N-1]);
29
30   ...
31
32   y[((idx*(M/szx))+((M/szx)-1))]+=(a[((idx*(M/szx))+((M/szx)-1))*N+0]*x[0]);
33   y[((idx*(M/szx))+((M/szx)-1))]+=(a[((idx*(M/szx))+((M/szx)-1))*N+1]*x[1]);
34   ...
35   y[((idx*(M/szx))+((M/szx)-1))]+=(a[((idx*(M/szx))+((M/szx)-1))*N+(N-1)]*x[N-1]);
36 }
```

Listing 3.16: MxV code: OpenCL kernels with loops in row-major order

Vectorization

When vector instruction are used in a code, selecting the appropriate vector size for them for each architecture is very relevant in terms of performance. HPL allows to rewrite at run-time a vectorized kernel using arbitrary vector sizes. This feature is accomplished by combining C++ templating and the `AliasArray` HPL data type, which allows to access an existing HPL `Array` made up of scalars using either scalar or vector data types.

The code in Listing 3.17 is a vectorized version of the grain-adjustable matrix-vector product from Listing 3.11, but it uses a generic vector type `vectype`. With this purpose, the HPL kernel in lines 1-20 is templated for this `vectype`. On the host side, the `MxV` class is properly instantiated using the desired vector type (line 23). On the kernel side, matrix `a` and vector `x` are wrapped in lines 6-7 using the `AliasArray` class provided by HPL, which allows to access them using vector types of the de-

```
1  template<typename vectype>
2  class MxV { // Other portions of the class have been elided
3    void operator()(Array<float,2> a, Array<float,1> x,
4                    Array<float,1> y)
5    {
6      AliasArray<vectype, 2> a_vec(a[0][0]);
7      AliasArray<vectype, 1> x_vec(x[0]);
8      Array<vectype, 0> tmp;
9      Int k;
10
11     for_(i=0, i<(M/szx), i++) {
12       for_(k=0, k<=(N/vectype::veclen), k++){
13         tmp += (a_vec[idx*(M/szx)+i][k] * x_vec[k]);
14       }
15       for_(k=0, k<vectype::veclen, k++){
16         y[idx*(M/szx)+i] += tmp[k];
17       }
18     }
19   }
20 };
21 int main(...) {
22   ...
23   MxV<vectype> mxv;
24   eval(mxv).global(M)(av, xv, yv);
25 }
```

Listing 3.17: MxV code: vectorized version

sired size. The loop in lines 12-14 is a vectorized version of the inner loop of the original version of the algorithm. This loop generates a resulting vector `tmp` with `vectype::veclen` positions. Finally, the values of `tmp` are accumulated in the position `y[idx*(M/szx)+i]` by the loop in lines 15-17.

3.2.4. Outlining a search process for the parameters

The parametrized optimization techniques just introduced are the building blocks of our self-adaptive HPL kernels. Thanks to the run-time code generation capabilities of HPL, any kernel written by combining these blocks can be translated into multiple OpenCL versions depending on the values given to the parameters of each technique applied. Therefore, any algorithm devoted to finding an optimized set of values for these parameters will also lead to an optimized OpenCL implementation of the original HPL kernel.

The fundamentals of both breadth-first search (BFS) and genetic algorithms (GA), and how these strategies could be applied to generate optimized versions of an input code, were introduced in Section 2.3.2. Unsurprisingly, the BFS approach turned out to be a very time-consuming option. Nevertheless, a genetic algorithm designed in a similar vein to that used in OCLoptimizer could be an interesting alternative to find an optimized combination of values for the parameters driving a self-adaptive HPL kernel. Thus, generally, each parameter introduced by each optimization technique will be a gene, all these genes being combined into the chromosome of each individual. By setting different values for these genes, different individuals can be generated, each individual eventually producing a different version of the kernel. Other operating aspects such as the fitness function used to evaluate the individuals, the termination condition of the algorithm, or the valid ranges that the values of each gene can take, may vary depending on the particular properties of each case, such as the problem implemented by the kernel or the capabilities of the target device.

3.3. Case Study: Matrix Multiplication

Matrix multiplication is a common time-consuming operation that is implemented by a wide range of parallel libraries. As it is an extensively studied and important problem, we have generated a highly optimized HPL implementation of this algorithm. Our implementation has several parameters that can be tuned through a genetic search guided by the kernel execution time.

Our performance-portable HPL kernel implements the operation $C = A \times B$. The code has been written in such a generic way that either A or B or both can be either directly loaded in private memory from global memory, or previously copied to local memory to optimize these further loads into private memory. Moreover, thanks to the aforementioned RTCG capabilities of HPL, it is possible to select the most appropriate combination of usage for both kinds of memory depending on the device selected at run-time. In addition, the granularity of the work to be performed by each thread can be adjusted by changing the global domain size. The size of the local domain can be changed depending on the capabilities of the device, and, within each thread, the tiling technique is applied. Moreover, the inner loops of the algorithm are fully unrolled and the instructions are reordered using the instructions scheduling technique, and then this inner code is vectorized for a generic vector type that can be configured at run-time. All these optimizations give place to a set of parameters that can be tuned for each device at runtime.

The rest of this section is organized as follows. First, the parametrized algorithm that our HPL matrix multiplication kernel implements is described in Section 3.3.1. Then, Section 3.3.2 explains how a genetic search has been used to find the best values for the parameters of our algorithm in each device.

3.3.1. Kernel implementation

The implementation of our kernel relies on a number of tunable parameters that will be introduced during the explanation and which are summarized in Table 3.2 for ease of reference. As explained in Section 3.1, the first two elements in the table are the standard HPL variables that provide the size of the global domain, which describes the total number of threads that execute the kernel in parallel, in

Name	Description
szy	# of rows of global domain
szx	# of columns of global domain
lszy	# of rows of local domain
lszx	# of columns of local domain
bszy	# of rows of each block of C calculated by one thread
bszx	# of columns of each block of C calculated by one thread
tW	Tile width to distribute the work among work groups
uf	Unroll factor to be applied over the tile width loop
copyA	Local memory copy flag for matrix A
copyB	Local memory copy flag for matrix B
vA	Vector size for copying matrix A from global to local memory
vB	Vector size for copying and/or manipulation of matrix B
vC	Vector size for copying and/or manipulation of matrix C
order	Order of the three innermost nested loops

Table 3.2: Parameters of the matrix multiplication algorithm

the second (`szy`) and the first dimension (`szx`). Similarly, the next two items in the table describe the corresponding dimensions of the local domain, which provide the size of the groups of threads, or work-groups following OpenCL terminology. In our kernel the domains are associated to the dimensions of the destination matrix, and as we can see from the description in Table 3.2, its rows are distributed across the second dimension of the domain, while the columns are mapped on the first dimension.

Figure 3.2 shows how the work is partitioned in tiles across the threads and how global, local and private memory regions are used. The top part, Figure 3.2.a, shows that each thread calculates a tile of `bszy` $\times$ `bszx` elements of the resulting matrix `C` by multiplying `bszy` rows of matrix `A` and `bszx` columns of matrix `B`. The tiling technique is also applied to the work to be performed in this computation. The shared dimension of matrices `A` and `B` (the columns of `A` and the rows of `B`) is partitioned into tiles of size `tW`. The local memory shared among the threads of the same group can be used to accelerate data loading. Figure 3.2.b shows how a tile of `lszx` $\times$ `bszy` rows and `tW` columns of matrix `A` is loaded into local memory collaboratively by the threads of the same group. Using the same method, a tile of `tW` rows and `lszy` $\times$ `bszx` columns of matrix `B` can be loaded into local memory. Let us notice that the dimensions of the block size and the local size are crossed. This combination consistently delivers better performance than its complementary, and

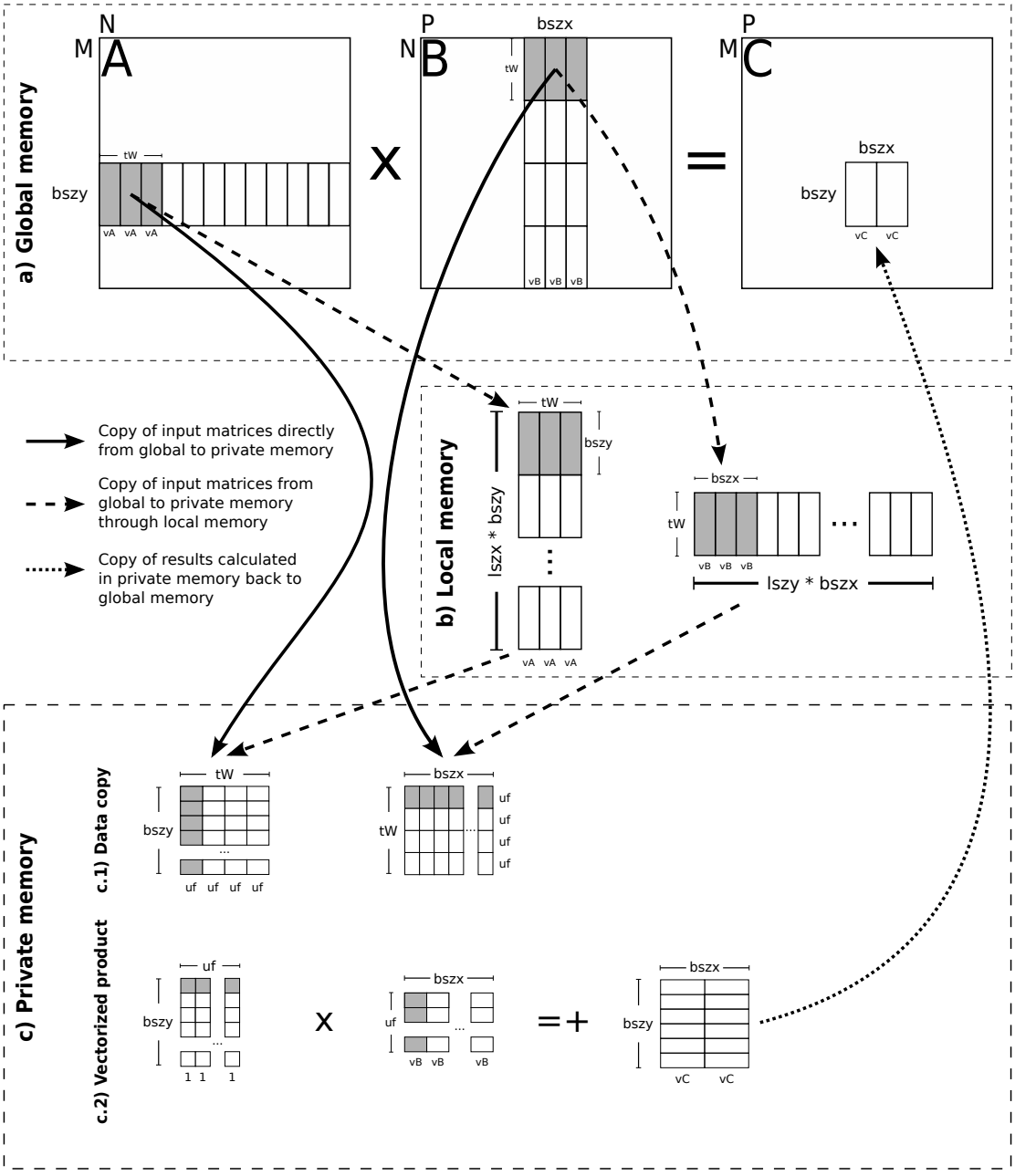


Figure 3.2: Matrix multiplication generic algorithm

more natural, alternative. The information of matrices `A` and `B` is loaded vectorially using vectors of size `vA` and `vB`, respectively. Once this information is collaboratively loaded into local memory, each thread calculates its tile of the resulting matrix `C`.

This is a good point to introduce the parameters in Table 3.2 related to vectorization. The values `vA`, `vB` and `vC` define the vector size used to move data from `A` and `B`, and to `C`, respectively. The two latter ones, `vB` and `vC`, are also used to define the lengths of the vectors used in the innermost loops that perform the computation. Figure 3.2.c.1 shows that matrix `A` is loaded into private memory in tiles of `bszy` $\times$ `uf` elements and matrix `B` in tiles of `uf` $\times$ `bszx` elements. Figure 3.2.c.2 shows that these tiles are multiplied vectorially. At tile level, the innermost loop iterates on the `N/tW` tiles of size `bszy` $\times$ `tW` in which the set of `bszy` rows of `A` assigned to the thread can be partitioned, multiplying each one of them by the same tile of `tW` $\times$ `bszx` elements of `B`. Similarly, the product of `bszy` complete rows of `A` and `bszx` complete columns of `B` that is required to calculate a complete tile of `bszy` $\times$ `bszx` elements of `C` is processed across different iterations of another outer loop.

Notice that each input matrix can be loaded into local memory prior to having it copied into private memory. The usage of local memory theoretically accelerates the loading of the matrices. However, in some architectures there may not be enough local memory or its usage can slow down the application [98, 99]. For this reason, the local memory can be bypassed, in which case data will be directly loaded from global to private memory. For each architecture, local memory can be used for loading both, one, or none of the input matrices. This is selected by the parameters `copyA` and `copyB` in Table 3.2. Namely, they determine whether matrices `A` and/or `B` have to be copied first to local memory or directly to private memory. For each matrix, the corresponding flag can take values either of 0, when no data is going to be copied to local memory, or 1 or 2, otherwise. In this two latter cases, when the flag takes the value 1 our kernel implementation will try to allocate exactly the local memory space needed to store tiles of `A` of size (`lszx` $\times$ `bszy`) $\times$ `tW` or tiles of `B` of size `tW` $\times$ (`lszy` $\times$ `bszx`). If the flag takes the value 2, it tries to allocate space for an additional column for each tile in order to avoid possible bank conflicts. The pseudo-code in Listing 3.18 shows a simplified version of the algorithm followed by each thread to calculate a complete `bszy` $\times$ `bszx` tile of `C`. For simplicity, this algorithm assumes that the local memory is used as a gateway between global and private memory and that vector lengths `vB` and `vC` are equal. The local variables to load a (`lszx` $\times$ `bszy`) $\times$ `tW` tile of `A` and a `tW` $\times$ `(lszy` $\times$ `bszx)` tile of `B` are declared in lines 3 and 6. Lines 9 and 11 declare the private variables to load `bszy` $\times$ `uf` elements of `A` and `uf` $\times$ `bszx` elements of `B`. Finally, the private variable `c` where

```
1  // Local submatrix of A
2  lA_sz = lszx*bszy; // Rows of local submatrix of A
3  local float localA[lA_sz][tW];
4  // Local submatrix of B
5  lB_sz = lszy*bszx; // Columns of local submatrix of B
6  local float localB[tW][lB_sz];
7
8  // Private submatrix of A
9  private float a[bszy][uf];
10 // Private submatrix of B
11 private float<vB> b[uf][bszx/vB];
12 // Private submatrix of C
13 private float<vC> c[bszy][bszx/vC];
14
15 A_gp = gidx*lA_sz;  // First row in A for group (gidx,gidy)
16 B_gp = gidy*lB_sz;  // First column in B for group (gidx,gidy)
17 lA_pos = lidx*bszy; // First row in localA
18 lB_pos = lidy*bszx; // First column in localB
19
20 for_(t=0, t<N, t+=tW){ // foreach tile of width tW in N
21   // Collaborative copies of A and B to local memory
22   localA[0:lA_sz][0:tW] <- A[A_gp:A_gp+lA_sz][t:t+tW]
23   localB[0:tW][0:lB_sz] <- B[t:t+tW][B_gp:B_gp+lB_sz]
24   barrier(); // Group barrier
25   for_(tt=0, tt<tW, tt+=uf){ // foreach tile of width uf in tW
26     b[0:uf][0:bszx] <- localB[tt:tt+uf][lB_pos:lB_pos+bszx]
27     a[0:bszy][0:uf] <- localA[lA_pos:lA_pos+bszy][tt:tt+uf]
28     // Vectorized product of a and b private memory slices
29     for(i=0; i<bszy; i++){       // loop 0
30       for(j=0; j<bszx/vC; j++){ // loop 1
31         for(k=0; k<uf; k++){    // loop 2
32           c[i][j] += a[i][k] * b[k][j];
33     }}}
34   }
35   barrier(); // Group barrier
36 }
37
38 C_row=gidx*lA_pos; // First row in C for a block
39 C_col=gidy*lB_pos; // First column in C for a block
40 C[C_row:C_row+bszy][C_col:C_col+bszx] <- c[0:bszy][0:bszx]
```

Listing 3.18: Calculation of a single block of C using local memory

the resulting `bszy` × `bszx` tile of `C` is stored is declared in line 13. Notice that each element of arrays `b` and `c` is a vector of size `vB` and `vC`, respectively. This enables vectorization when the multiplication is done.

Lines from 15 to 18 calculate the first position in `A` and `B` accessed for a given group, and the first position in `localA` and `localB` accessed by a given thread, respectively. Here it is important to explain that the tuple `(gidx,gidy)` corresponds to HPL predefined variables that provide the identifier of the thread group to which the current thread belongs in the first and the second dimensions of the domain, respectively. The loop between lines 20 and 36 iterates on each tile of size `tW` in the common dimension of `A` and `B`. Inside this loop, the corresponding slices of `A` and `B` are collaboratively copied by the threads of the same group into their local counterparts, `localA` and `localB` (see lines 22 and 23). The local barrier in line 24 waits until every member of the group has completed its part of this copy. Then, the inner loop between lines 25 and 34 iterates on subtiles of size `uf` within each tile of width `tW`. Lines 26 and 27 transfer the appropriate subtiles from `localA` and `localB` to their private counterparts, `a` and `b`, respectively.

The three innermost nested loops in lines 29 to 33 perform the multiplication of a subtile of `bszy` $\times$ `uf` elements of `a` by another subtile of `bszy`$\times$ `bszx` elements of `b` using vector types. The result is stored in a private matrix `c`. These three loops are native C++ `for` loops, thus, they will be fully unrolled at run-time. In our implementation, these loops can be also dynamically reordered, according to the `order` parameter in Table 3.2, which is a vector of three elements that encodes the selected order. Once a thread has completed the calculation of its tile of `C`, the instruction in line 40 copies back the resulting matrix from the private copy in `c` to the appropriate positions of the global matrix `C`.

3.3.2. Genetic search of the kernel parameters

In Section 3.2.4 we outlined how a genetic search process could be applied to find an optimized version of a self-adaptive HPL kernel. That outline is extended here in order to particularize the algorithm for the current matrix multiplication case use. Therefore, in this case we are going to tune the values for the parameters summarized in Table 3.2 by means of a genetic algorithm. Thus, the individuals of the population represent different versions of the matrix multiplication self-adaptive kernel, and each gene in the chromosome of an individual represents a parameter from the aforementioned Table 3.2. The initial population is generated randomly and

individuals for the subsequent generations are the result of the known reproduction, crossover and mutation techniques. The fitness function to maximize is defined in terms of the inverse of the execution time of a kernel. This time is obtained by running each version three times and getting the average kernel time. The algorithm stops after five iterations without finding any kernel improving the fastest one ever found. When this happens, that kernel is returned as the most optimized version.

Regarding the values that the genes of each individual can take, they have to match certain mandatory conditions. These constraints, which are summarized in Table 3.3, are derived from restrictions imposed by HPL, the matrix multiplication algorithm, or the properties of the target device, and the violation of any of them will lead to the generation of an illegal version of the kernel. For example, HPL restricts the local size to be not greater than the global size, whereas the algorithm used to implement the matrix multiplication requires the tile width `tW` to be not greater than the common dimension `N` of matrices `A` and `B`. In addition, the device must have enough free memory space to perform the multiplication, and this restriction is directly related to the selected sizes for the global and the local domains and tile width, among other parameters. Other situations prevented by these conditions are, for instance, the definition of too large workspaces that can generate too many idle threads, or the selection of vector sizes or unroll factors that are incompatible with the block size, the tile size or the problem size. Thus, any operation of the algorithm involved in both random generation and mating reproduction of new individuals is refined to check first whether the parameters match these conditions. If this is not the case, the individual is discarded.

These conditions also introduce strong dependences among the optimization parameters of the matrix multiplication, which considerably restricts the ranges of valid values that they can take. This may seem a troublesome issue, since it increases the probability that a generated individual is not valid. However, this inconvenience ends up being an advantage. Table 3.2 shows that we are tuning 14 parameters, a number large enough to lead to a combinatorial explosion in a worst-case scenario. Thus, these restrictions contribute to reduce a considerably wide search space, which results in a more effective search process. Still, further tests revealed that it was advisable to set additional conditions in order to narrow the search space even further. Namely, these conditions intend to keep the values of some parameters within ranges

Condition	Explanation
$szy \leq P$ $szx \leq M$	Global workspace is not greater than C matrix
$lszx \leq szx$ $lszy \leq szy$	Local workgroups fit into global workspace
$tW \leq N$	Tile width for row-column product loop not greater than N
$uf \leq tW$	Unroll factor over tile not greater than tW
$vA \leq tW$	Vector size for row-column product loop not greater than tW
$vB \leq bszx$ $vC \leq bszx$	Vectors used to manipulate B and C are not greater than $bszx$
sizeof(A) + sizeof(B) + sizeof(C) $\leq$ g_mem_avail	Enough free space in global memory for matrices A, B and C
sizeof(local$A$) + sizeof(local$B$) $\leq$ l_mem_avail	Enough space in local memory for slices local$A$ and local$B$

Table 3.3: Minimum conditions of validity for GA individuals

that have heuristically shown to contain well-performing solutions to our problem, which helps to both reach better versions as well as to reduce the search time. The mutation of newborn individuals is implemented also with the intention of leading the search process to such solutions. In detail, both dimensions of the global domain have been limited to a minimum size of 128 when the algorithm is run in GPUs, and to a minimum size of 64 otherwise. These heuristic conditions are added to the mandatory conditions shown in Table 3.3 and they are also taken into account to qualify an individual as valid.

Finally, just like in OCLoptimizer, the genetic search implemented in this use case is also built on top of the GAlib genetic algorithm package [118].

3.4. Experimental results

In this section the performance and the search time of our adaptive implementation of the matrix multiplication is evaluated for different problem sizes, and compared with other approaches, in four very different platforms:

- **CPU**: A dual-socket system with two Intel Xeon E5-2660 Sandy Bridge with eight 2.2Ghz cores and Hyper-Threading (8×2 threads per processor, for a total of 32) and 64 GB of RAM. Intel OpenCL driver version 1.2-4.5.0.8. Single-precision theoretical peak performance of 563 GFLOPS.
- **Nvidia**: An NVIDIA Tesla K20m with Kepler GPU architecture and 5 GB GDDR5. NVIDIA OpenCL driver version 340.58. Single-precision theoretical peak performance of 3524 GFLOPS.
- **AMD**: An AMD FirePro S9150 with Hawaii GPU architecture and 16 GB GDDR5. AMD OpenCL driver version 1702.3. Single-precision theoretical peak performance of 5070 GFLOPS.
- **Accelerator**: An Intel Xeon Phi 5110P with sixty 1.053GHz cores with 8 GB of RAM. Intel OpenCL driver version 1.2-4.5.0.8. Single-precision theoretical peak performance of 2022 GFLOPS.

The test performs the multiplication of two square matrices of single-precision floating point values taking into account four different matrix sizes, 1024×1024, 2048×2048, 4096×4096 and 8192×8192. All test programs were compiled using `g++-4.7.2`. Also, in order to assess the quality of our approach, the performance of our HPL implementation tuned by means of a genetic search process is compared to the performance of two OpenCL state-of-the-art implementations, namely clBLAS 2.4 [19] and ViennaCL 1.5.1 [110]. We have selected these implementations because HPL is also currently based on OpenCL, they can be executed in the same range of platforms as our HPL adaptive code, and they also support some kind of adaptive behavior depending on the underlying hardware. We now briefly describe these libraries.

First, clBLAS is the implementation used by AMD in its clMath suite and thus it is the official BLAS library in the AMD platform. It includes a profiling tool that queries some of the properties of the platform where the matrix multiplication will be run. This information is used to select some candidate values for parameters such as the granularity of the work, both group and thread-level tile widths, and vector lengths, and to decide whether local memory is used or not. Using these ranges of values, the tool generates a set of representative kernels, which are run for different

problem sizes and it chooses the best one as the single optimized version for the platform. Originally, the tool only supports GPU profiling. We have modified it to be able to profile also the hardware of the rest of our testing platforms.

The ViennaCL implementation has several parameters that can be tuned for each platform. The latest distributions of ViennaCL, from 1.6.2 on, provide heuristically tuned values of these parameters for some of these platforms, but they deliver bad performance compared to our implementation. Previous versions of ViennaCL, such as 1.5.1, contained an auto-tuning tool that performs an exhaustive search for the values of these parameters, within a heuristically defined vast range, guided also by kernel execution time. On average, the performance of ViennaCL using this auto-tuner is 5 times the performance using the heuristically selected values, but on exchange, it requires a very large search time. The performance results reported in this work for ViennaCL are those resulting of this exhaustive search.

Table 3.4 shows the performance results for the three implementations in the four platforms tested. The third column contains the execution time in milliseconds and the performance measured in GFLOPS of the best kernel found by our genetically tuned HPL implementation. The fourth and fifth columns shows the speedup

Platform	Size	Best kernel performance Execution time (GFLOPS)	Speedup clBLAS	Speedup ViennaCL
CPU	1024	6.75 ms (318.00)	2.12	1.34
	2048	56.45 ms (304.33)	1.92	1.33
	4096	568.52 ms (241.75)	2.35	1.11
	8192	4768.57 ms (230.57)	2.57	1.13
Nvidia	1024	2.22 ms (969.52)	1.53	1.05
	2048	17.19 ms (999.64)	1.47	1.00
	4096	133.89 ms (1026.54)	1.55	1.02
	8192	1069.18 ms (1028.37)	1.55	1.03
AMD	1024	1.01 ms (2126.22)	2.50	2.07
	2048	6.53 ms (2630.91)	1.35	1.28
	4096	63.49 ms (2164.73)	0.93	1.06
	8192	839.19 ms (1310.21)	1.19	1.10
ACC	1024	7.43 ms (288.91)	1.81	2.08
	2048	44.38 ms (387.11)	1.70	2.22
	4096	350.95 ms (391.62)	1.54	2.17
	8192	3213.56 ms (342.15)	1.82	2.02

Table 3.4: Speedups achieved by best versions found

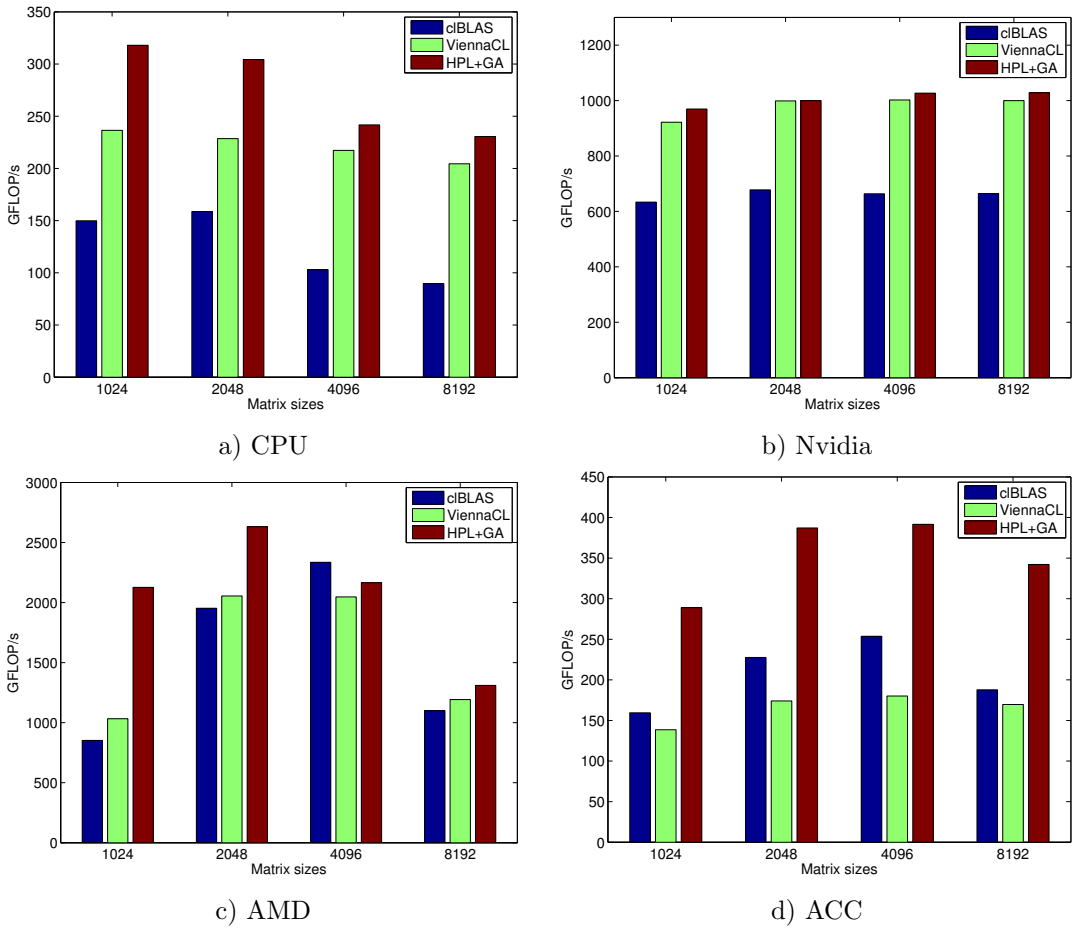


Figure 3.3: Performance in GFLOPS of clBLAS, ViennaCL and HPL best versions

achieved with respect to the clBLAS and ViennaCL implementations. Figures 3.3.a) to 3.3.d) compare the performance in GFLOPS of clBLAS and ViennaCL to that of our implementation for each problem size and platform. Let us recall that the kernels of all the implementations have been previously adapted to the underlying hardware by means of their respective profiling and tuning procedures. The results show that our implementation outperforms these two implementations for all matrix sizes and on the four platforms with the sole exception of matrix multiplication of size 4096 in the AMD platform. In this case, our HPL implementation is beaten narrowly by the clBLAS implementation. The average speedup of our approach is 1.74 with respect to clBLAS and 1.44 with respect to ViennaCL. Compared to clBLAS, our implementation achieves a peak speedup of 2.57 in the CPU platform for the

8192 size. The peak speedup with respect to ViennaCL is 2.22 and it is achieved in the ACC platform for the 2048 size. All the comparisons were done against the corresponding optimized versions generated by both clBLAS and ViennaCL for each different problem size.

Table 3.5 shows the best values of the parameters of the HPL generic matrix multiplication kernel found by the genetic algorithm. These parameters have been explained in Table 3.2. The Table shows that the values selected for each platform and for each problem size are different, and they are difficult to predict using a single general heuristic. A pattern can be observed in the values taken by some parameters within the same platform, but they cannot be easily found a priori.

Table 3.6 contains the time consumed by the tuning procedures conducted by our genetic algorithm, the clBLAS profiler and the ViennaCL auto-tuner. On average, our genetic search is 1.18 times faster than the clBLAS profiler. For the CPU and ACC platforms, the sum of times consumed by our genetic search for each matrix size is competitive in relation to that consumed by the clBLAS profiler. In the Nvidia and AMD platforms, both composed of GPUs, the clBLAS search procedure is quite faster, which is understandable taking into account that it is specifically directed to this kind of devices. The results also show that the ViennaCL auto-tuner is 160

Device	Size	sz(x,y)	lsz(x,y)	bsz(x,y)	(tW,uf)	v(A,B,C)	copy(A,B)	order
CPU	1024	(256,64)	(8,64)	(16,4)	(32,1)	(8,8,8)	(2,0)	201
	2048	(512,128)	(8,128)	(16,4)	(32,1)	(8,8,8)	(2,0)	201
	4096	(1024,256)	(2,256)	(16,4)	(256,8)	(16,16,16)	(1,0)	012
	8192	(2048,512)	(32,32)	(16,4)	(32,4)	(16,16,16)	(2,0)	201
Nvidia	1024	(128,256)	(2,64)	(4,8)	(32,2)	(2,4,4)	(2,0)	210
	2048	(512,256)	(4,64)	(8,4)	(256,4)	(2,4,4)	(2,0)	102
	4096	(512,512)	(16,16)	(8,8)	(32,2)	(2,2,2)	(2,0)	102
	8192	(1024,1024)	(2,128)	(8,8)	(32,2)	(4,8,8)	(2,0)	210
AMD	1024	(256,128)	(4,32)	(8,4)	(128,1)	(4,8,8)	(2,0)	102
	2048	(256,256)	(1,128)	(8,8)	(256,2)	(4,8,8)	(2,0)	120
	4096	(512,512)	(4,16)	(8,8)	(32,2)	(4,8,8)	(2,0)	012
	8192	(1024,1024)	(1,128)	(8,8)	(32,4)	(4,8,8)	(2,0)	012
ACC	1024	(256,64)	(1,16)	(16,4)	(8,2)	(1,16,16)	(0,0)	120
	2048	(256,128)	(1,8)	(16,8)	(512,8)	(8,16,16)	(0,0)	120
	4096	(2048,256)	(16,32)	(16,2)	(32,1)	(8,16,16)	(2,0)	201
	8192	(4096,512)	(16,16)	(16,2)	(32,1)	(16,2,2)	(2,0)	021

Table 3.5: Configuration of best versions found using our self-adaptive kernel

Device	Size	Total tuning time (s)		
		GA	clBLAS	ViennaCL
CPU	1024	120.57	42947.26	32428.25
	2048	339.99		60438.13
	4096	1729.80		500775.18
	8192	19286.90		4186086.80
Nvidia	1024	242.04	1225.53	18836.30
	2048	331.40		38292.62
	4096	4429.57		186041.36
	8192	17127.50		1394675.71
AMD	1024	1579.74	5425.97	1911.00
	2048	2422.34		6221.00
	4096	4587.55		60595.37
	8192	5792.07		$>$ 3 days
ACC	1024	260.32	86501.20	121891.58
	2048	915.69		211610.18
	4096	4401.47		1145630.97
	8192	31973.30		$>$ 3 days

Table 3.6: Total times for tuning self-adaptive kernels

times slower than our genetic search procedure. This large difference is undoubtedly due to the time-consuming exhaustive search it conducts.

3.5. Conclusions

In this chapter, we have presented a set of techniques to generate self-optimizing codes in HPL. These techniques are based on generic programming and the RTCG capabilities of the HPL embedded language for kernels. The resulting codes can be automatically optimized for a given device by finding the appropriate values for a set of optimization parameters. These parameters decide whether a given optimization technique is going to be applied or nor and/or the way it is going to be applied. The search of the best values for these parameters is guided by a genetic algorithm where each individual is evaluated using its execution time. This way, the techniques described in this chapter offer an alternative to complex auto-tuning libraries or complex source-to-source compilation tools.

As a case study, we have generated a generic adaptable version of the matrix multiplication algorithm. In our implementation a dozen of parameters allow to

tune the kernel for the different platforms and problem sizes. The performance of our adaptive kernel has been compared to two state-of-the-art OpenCL adaptive implementations of the matrix product, namely, clBLAS and ViennaCL. The kernels used by clBLAS can be adjusted to the device where they are going to be run by means of a prior profiling. The ViennaCL implementation can be tuned through a set of parameters, but their values are selected by means of an exhaustive search. Except in a single test, where clBLAS takes the lead for a single matrix size in an AMD GPU, our implementation systematically outperforms the other adaptive libraries in four systems: an NVIDIA GPU, an AMD GPU, a multicore Intel CPU and an Intel Xeon Phi accelerator. The average speedup of our implementation with respect to clBLAS and ViennaCL is 1.74 and 1.44, respectively. Compared to clBLAS, our implementation achieves a peak speedup of 2.57 in the CPU platform for the 8192 size. The peak speedup with respect to ViennaCL is 2.22, and it is achieved in the Xeon Phi for the 2048 size. Besides finding faster versions of the matrix multiplication, our genetic search is on average 1.18 times faster than the clBLAS profiling and 160 times faster than the exhaustive search implemented by ViennaCL.

3.6. Related work

Matrix multiplication is an algorithm extensively studied in the bibliography for multiple kinds of devices, including Nvidia [59] and AMD [68] GPUs. Some of these works focus on the study of several linear algebra operations. For example, ViennaCL [110] provides an OpenCL implementation of several linear algebra routines, including the matrix multiplication. Their approach is based in a generic version of the matrix multiplication where the parameters are either fixed heuristically or using an auto-tuner driven by the execution time. ViennaCL is, to the best of our knowledge, the best-performing OpenCL implementation of the matrix multiplication. Their auto-tuner obtains worse performance results than our implementation and, in addition, the search times are several orders of magnitude larger than ours. The reason for this latter problem is that they run an exhaustive search process, instead of an informed one like our HPL implementation does by means of a genetic algorithm.

clMAGMA [15] introduces an OpenCL version of the MAGMA library [111]. They use clBLAS to implement BLAS routines, including the matrix multiplication operation. Thus, our comparison to clBLAS is valid for this library. There are more approaches that try to achieve performance portability of linear algebra problems through iterative processes. For example, [23] uses iterative compilation to select the optimal parameters for GPU codes according to a set of pre-defined parameterized templates. They have 10 parameters, while we tune 14 parameters. They do not report the execution times of their autotuner. We obtain a better performance, although obviously we are using newer architectures. Matsumoto et al [69] automatically generate and tune several parametrized OpenCL versions of the A^TB variant of the GEMM routine. These versions are implemented following different algorithms devoted to exploit specific features of different kinds of devices. Moreover, the search process conducted consisted in an exhaustive search of the fastest kernels among tens of thousands of versions that had been previously chosen by means of heuristics. Notice that the execution time measured for each kernel included the time consumed by the transposition of matrix A.

This kind of linear algebra problems are also used to prove the abilty of rewrite-based methods to generate optimized code for accelerators. Steuwer el al [107] offer a high-level functional language language embedded in Scala to write kernels which are internally translated into an intermediate representation based on λ-calculus [103]. The language also includes heuristically defined rules that rewrite its functions as compositions of primitives, which are in turn linked to parametrized OpenCL code snippets. The implementation properties covered by these parameters are similar to those we cover in our self-adaptive kernels, such as workspaces sizes, vector lengths, or unroll factors. Optimized versions of kernels are found by means of an exhaustive search over a previously pruned subset of all the possible OpenCL implementations of an input kernel. That pruning is performed by keeping the values of the aforementioned parameters in a range of heuristically fixed values which are expected to produce the best code versions.

Other approaches are more general and they focus on a wider range of applications. For example, a simple model based on both hardware and application parameters is used in [35] to build an OpenCL performance-portable implementation of data streams clustering and to generate tuned versions of it for several

NVIDIA and AMD GPUs. More complex computations can be tuned by selecting the best implementations for the different numerical routines of which they are composed. Nitro [74] is a framework that provides programmers with a mechanism to manage collections of these building blocks and also information related to their performance in different platforms and for different applications. This information is used to train the framework about how to select optimal combinations of variants of those routines in order to solve different kinds of problems, such as sparse matrix operations, conjugate gradient solvers, breadth-first search algorithms, histogram calculations, and sorting operations.

Last but not least, there are solutions that intend too to provide self-adaptive implementations no matter the problem addressed in the kernel. A relevant work following this approach is CLTune [78], which is contemporary to our self-adaptive HPL kernels and consists in an auto-tuner for OpenCL kernels. Programmers must identify the parameters they consider that may affect the performance of their codes, and then refactor their kernels in terms of such parameters. The tool also provides a C++ programming interface through which the users must register the parameters to tune, set a range of valid values for each parameter and launch the optimization process. Internally, this tool deals with the parameters by means of macros and another generic programming techniques. Our approach, in turn, is not only based on this latter paradigm but also thoroughly exploits the run-time code generation capabilities provided by the HPL embedded kernel language. The search strategies implemented in [78] to tune the parameter values are a randomized search, a simulated annealing technique and a particle swarm evolutionary algorithm, all of which evaluate the versions they generate using the kernel execution time. The authors validate their approach by means of two use cases, a two-dimensional convolution and a matrix multiplication. Likewise ours, their implementation of this latter kernel is inspired in those from clBLAS and the aforementioned works of Matsumoto et al. Thus, it is optimized similarly to our self-adaptive kernel, although their local memory caching procedure for the input matrices is implemented in a more refined way. The devices targeted were several NVIDIA and AMD GPUs, on which they also outperform clBLAS, although no tests were run either on CPUs or on other kinds of accelerators such the Intel Xeon Phi.

Chapter 4

Performance-portable HPL

The approach presented in Chapter 3 to enable performance portability was built on top of the Heterogeneous Programming Library (HPL). This solution consists of a set of techniques to write self-optimizing HPL codes that use the run-time code generation (RTCG) capabilities of this library. By themselves, these techniques are not specially difficult to implement. However, blending them in order to achieve a parametrized implementation for a given problem is a more complex process that leads to quite long kernels. For instance, the matrix multiplication HPL self-adaptive kernel we implemented as use case in that chapter is about 350 lines long. Also, HPL kernels written following this approach were claimed to adapt themselves automatically to perform well in a particular device. To achieve this, proper values for the optimization parameters must be found. In the matrix multiplication use case a genetic algorithm was implemented to find these values, such informed search methods being clearly more affordable than any exhaustive alternative. However, these strategies are still based on the execution of a considerable number of versions, which makes them quite time-consuming.

In this chapter we go a step further and try to overcome these inconveniences so that performance portability can be provided on top of HPL without requiring almost any intervention from the programmer. In order to achieve this, we equipped HPL with a just-in-time optimizer that automatically tunes the code at run-time for the device where it is going to be executed. This just-in-time optimization process has two interesting characteristics: (1) it is lightweight, so that it does not delay too

much the execution of the code, and (2) it performs a set of optimizations typically applied in heterogeneous systems to tune a code for a target device. The flowchart shown in Figure 4.1 offers an overview of this optimization process. Regarding the input, the programmers have to write their HPL kernels in a naive way (without using vectorization, local memory or other optimization features), just encoding the calculation of one point of the solution. Moreover, the programmers also have to enclose that code inside a *compute* section, leaving variable declarations and other parts of the kernel out of it. This hint gives valuable information to the optimizer and it simplifies the optimization process. The naive input kernel is then loaded into an abstract syntax tree (AST) representation. Transformations such as exploiting local memory when available, the adjustment of the amount of work executed by each thread or loop tiling are applied on the tree in the order depicted in the flowchart. A set of parameters, such as the tile size if the tiling technique is applied, or the exact amount of work that is going to be assigned to each thread, drive both the application of each individual transformation and the conditions under which it is applied. The values for the parameters are tuned for a given device according to some heuristics based in general guidelines to optimize codes for heterogeneous devices. By tuning these values, the code generated from the transformed AST is expected to maximize parallel execution and both memory and instruction throughput when it is run in a target device.

The rest of this chapter is organised as follows. First, the code generation internals of HPL are introduced in Section 4.1, focusing on how kernels written using the HPL embedded language are translated into OpenCL C source code. Section 4.2 describes how that code translation process is overridden in order to load an HPL kernel into an AST manageable by the just-in-time optimizer. Then, in Section 4.3 the just-in-time optimization process is explained, which includes the description of the transformation techniques implemented, the optimization parameters derived from those techniques and the heuristics followed to give values to these parameters at run-time. This proposal is validated in Section 4.4 by optimizing several HPL kernels for multiple target devices and then discussing both the performance of the optimized kernels and the impact that the generation process of those codes has in such a just-in-time approach. Section 4.5 contains the conclusions drawn from the work presented in this chapter. Finally, the main features of some optimization tools that also follow to some extent a just-in-time approach are reviewed in Section 4.6.

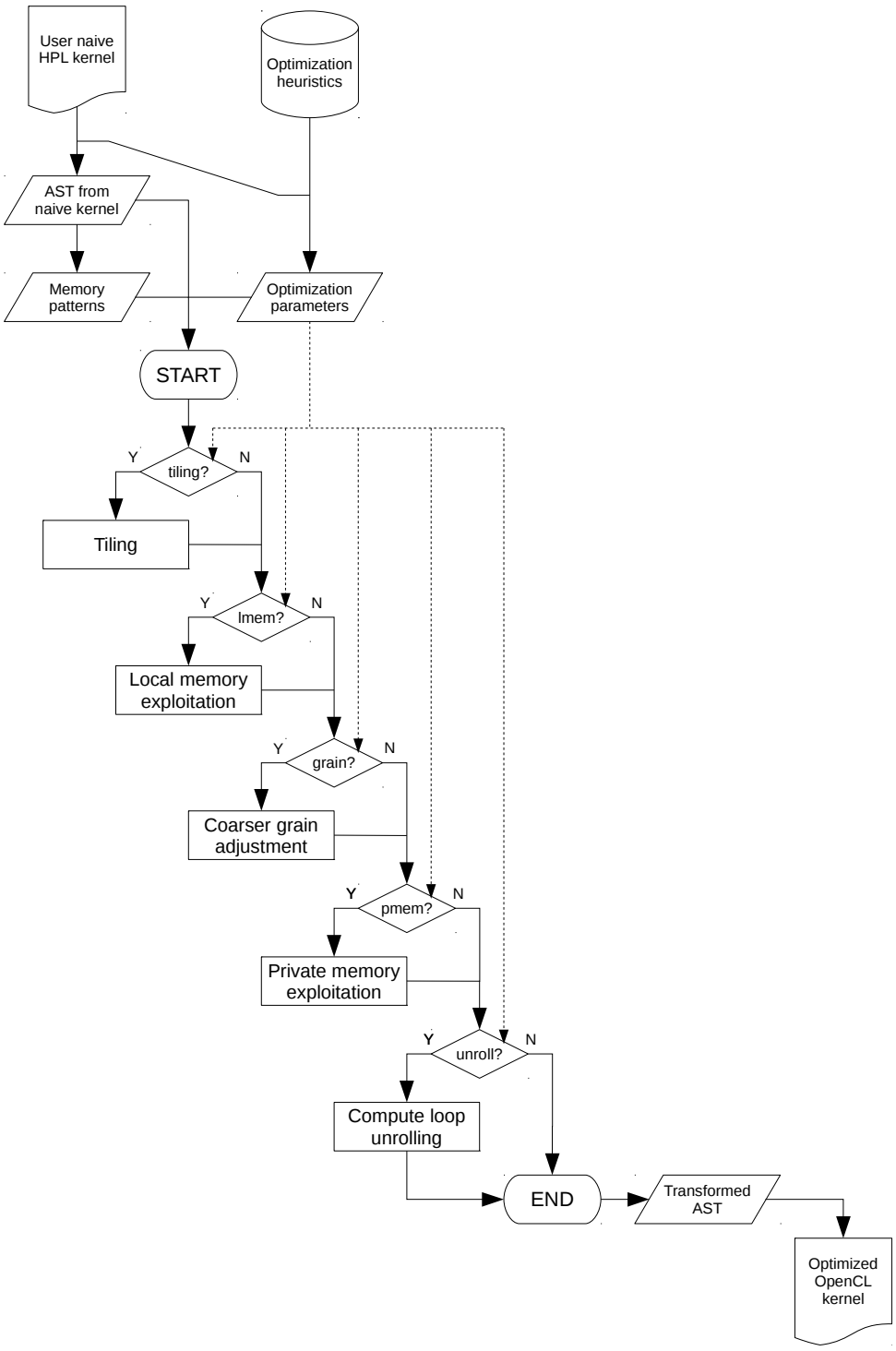


Figure 4.1: Workflow of the just-in-time optimizer

4.1. HPL code generation internals

When both the kernel embedded language and the host API of HPL were introduced in Section 3.1.2, we mentioned that its back-end currently generates OpenCL source code. Namely, HPL translates its kernels at run-time into OpenCL C kernels using the Portable Expression Template Engine (PETE) [43]. PETE is a portable C++ framework that lets users easily add expression-template functionality to container classes and perform complex expression manipulations. The *expression templates* technique allows to exploit the C++ templates to create parse trees of array expressions at compile time [113]. Along this section we will explain how HPL uses and extends PETE to parse the expressions composed of `Array` references and found on kernel instructions and to evaluate them as strings in order to compose the equivalent OpenCL code.

Let us start by introducing how the members of such expressions are evaluated. By default, PETE supports 45 built-in operators, including all the C/C++ mathematical operators and a collection of common mathematical functions. Moreover, any custom function needed can be added to the operator set supported by the expression-template system of PETE. This set is automatically built by means of a helper tool called `MakeOperators`, whose inputs are text files that include the specification of the operators. Listing 4.1 contains an extract of `StringOps.in`, a file

```
1  unaryOps
2  -----
3    TAG = "OpNs"
4    FUNCTION = "native_sqrt"
5    EXPR = "return \"native_sqrt(\" + a + ')';"
6  -----
7  ...
8  -----
9  binaryOps
10 -----
11   TAG = "OpAdd"
12   FUNCTION = "operator+"
13   EXPR = "return '('+ a + '+' + b + ')';"
14 -----
15 ...
```

Listing 4.1: PETE operator specification file: `StringOps.in` example

from the HPL library that specifies for PETE the mathematical and logical functions provided by the embedded kernel language. The operators specified in such files must be classified in terms of the number of operands they expect. Thus, notice how the headings `unaryOps` in line 1 and `binaryOps` in line 9 lead the lists of specifications for unary and binary operators respectively. The specifications for a couple of operators, one of each type, have been excerpted from the `StringOps.in` file. The unary operator `native_sqrt()`, which supports its OpenCL homonym function is defined in lines 3-5, while lines 11-13 contain the specification for `operator+`, which supports the common binary addition (`+`) operator. Both specifications follow the same structure, expecting three properties to be defined. Namely, `TAG` identifies each operator, `FUNCTION` is the name of the operator function expected, and `EXPR` contains a description of how to evaluate the operator on specific elements. The arguments to the functions must be referred with `a` in unary operators, `a` and `b` in binary ones, and `a`, `b` and `c` in trinary ones. No example of this latter trinary case is shown in the file excerpt provided. Let us recall that the operators specified in `StringOps.in` must be evaluated to their OpenCL string equivalents rather than to the result of the operation they represent, which is the default behavior of PETE. In this latter case, for instance, the `EXPR` description for `OpAdd` would be simply `(a + b)`, which eventually asks C++ to perform the $a + b$ operation. Nevertheless, in our example we are asking C++ to build and return a string that codifies such operation in OpenCL.

The operands of these functions can be literals, non-terminal nodes of an expression, or objects from any user-defined container class, so that the operators could be combined to incrementally build up the parse tree of an expression. PETE implements such combination mechanisms, but it must be told how to evaluate that container objects when passed as arguments to the operator functions. Regarding the HPL kernel embedded language, the `Array` templated class hierarchy, used either as `Array<T,ndim>` or through its convenience types (`Int`, `Float`, ...) for `Array<T,0>` scalars, plays such a container role. Listing 4.2 contains the specializations that PETE needs to emit the corresponding OpenCL string equivalent when any variable of such types is referred in a kernel. The code in lines 1-8 generates the OpenCL string for any `IndexedArray<T,ndim>` occurrence found in an HPL kernel. `Array<T,ndim>` objects return that specialized interface when they are referenced using the `[]` operator. For example, an `Array<int,2> a` will return the

```
1  template<typename T, int NDIM>
2  struct LeafFunctor<HPL::IndexedArray<T, NDIM>, StringizeExpr> {
3    typedef HPL::String_t Type_t;
4    static Type_t apply(const HPL::IndexedArray<T, NDIM> &v,
5        const StringizeExpr &)  {
6      return v.string();
7    }
8  };
9
10 template<typename T>
11 struct LeafFunctor<HPL::Array<T, 0>, StringizeExpr> {
12   typedef HPL::String_t Type_t;
13   static Type_t apply(const HPL::Array<T, 0> &v,
14       const StringizeExpr &) {
15     return v.string();
16   }
17 };
```

Listing 4.2: PETE specializations for container classes: HPL `Array` example

```
1  template<>
2  struct LeafFunctor<int, StringizeExpr> {
3    typedef HPL::String_t Type_t;
4    static Type_t apply(int a, const StringizeExpr &) {
5      return HPL::stringize(a);
6    }
7  };
```

Listing 4.3: PETE specializations for literals: `int` literal example

`IndexedArray<int,2>` interface when accessed as `a[row][col]` in a kernel, and through that interface a `string()` method is invoked (line 6) to get the string which encodes such memory access in OpenCL. In a similar vein, the code in lines 10-17 specializes the evaluation mechanism to generate the OpenCL equivalent for `Array<T,0>` scalar references in a kernel. Such objects also offer a `string()` method, which is invoked in line 15. Let us add that when an HPL array is declared as either a private or a local memory structure inside a kernel, the library captures such declaration, assigns an identifier to the structure and uses that identifier to generate a string with the corresponding OpenCL declaration. This string is appended to a code string buffer maintained by a class called `Codifier`, which is in charge of eventually emitting the full working OpenCL translations of kernels. Moreover, the evaluation mechanism has to be specialized also for literals, since HPL needs PETE

to print the value of a given literal in a string instead of just taking the value directly to operate with it. The code from Listing 4.3 shows such a specialization for `int` (integer) literals. For these cases, HPL implements a `stringize()` method, invoked in line 5, that takes the literal as an argument and prints it in the string returned.

At this point, the refinements and extensions performed by the HPL implementation on the PETE default behavior allow this latter framework to generate OpenCL code strings for each expression parsed from an HPL kernel. When such expressions are in the top level of a kernel body, the aforementioned `Codifier` class just takes their respective translations and pushes them to the code buffer. However, the expressions might be also inside HPL code blocks such as `if_` or `for_`. When the kernel embedded language was introduced in Section 3.1.2, we made a distinction between such HPL blocks, which the programmers must use to implement alternative and repetitive constructs in their kernels, and those from C++ used to exploit the run-time code generation capabilities of the library. The main differences identified then were, first, the addition of the underscore to avoid the usage of reserved C++ keywords, and second, the arguments being separated with commas instead of semicolons when needed. The HPL code constructs, as the rest of the kernel embedded language, can be accessed by the programmers through the `HPL.h` header file. Regarding this kind of constructs, they are provided by means of some macros defined in that file, and which follow the aforementioned distinctive format. As an example, the definition of the `for_` macro is shown in Listing 4.4. Arguments `a`, `b` and `c` in line 6 expect respectively the initialization statement of the loop counter, the boolean ending condition and the counter update instruction. When such a `for_(<init>,<end>,<step>)` is used in an HPL kernel, the code from that macro is inlined and run. Thus, the OpenCL equivalents for `a`, `b` and `c` (lines 8-10) are generated and then passed as arguments to a method `for_` of the `Codifier` object (lines 7-11). This method, whose implementation is excerpted in Listing 4.5, builds a string encoding the header of the equivalent OpenCL `for` loop, and pushes it to the code buffer (line 6). Then, in line 8 the `beginBlock()` method is called to inform the codifier that any expression parsed from now on belongs to the new `for` loop opened. Notice that before running the instructions from the body of the HPL kernel `for_` loop, the code from `HPL_common_block_macro` is inlined just before that loop. This macro defines the header of a C++ `for` (lines 2-4 in Listing 4.4). Such an inlining makes the body of the HPL `for_` loop become also the body of the loop

```
1  #define HPL_common_block_macro
2    for(int _hpl_tmp = 0;
3      _hpl_tmp < 1;
4      TheGlobalState().getCodifier().endBlock(), ++_hpl_tmp)
5
6  #define for_(a,b,c)
7    TheGlobalState().getCodifier().for_(
8      stringize(a),
9      stringize(b),
10     stringize(c)
11   );
12   HPL_common_block_macro
```

Listing 4.4: HPL interfaces for code constructs: `for_` example

```
1  void Codifier::for_(const String_t& init,
2                      const String_t& cond,
3                      const String_t& update)
4  {
5    <...>
6    add("for(" + init + "; " + cond + "; " + update + ") {", true);
7    <...>
8    beginBlock();
9  }
```

Listing 4.5: HPL `Codifier` class: `for_` loop processing

header added by the macro. Thus, in its first iteration, it runs the code in its body, which is therefore translated into OpenCL. When that iteration ends, the loop runs its counter update instructions set (line 4). Here, the `endBlock()` invocation closes the block in the OpenCL code buffer and informs the codifier about that. This way, the library is able to properly open and close, and also nest when needed, the multiple code constructs offered by the embedded kernel language.

4.2. Building an AST from an HPL kernel

As the flowchart depicted in Figure 4.1 shows, a user naive HPL kernel must be converted first into an abstract syntax tree (AST) in order to be processed by the just-in-time optimizer. Such an AST is built by capturing the expressions that PETE parses from the kernel and loading them into nodes of the tree. This AST

representation of the kernel was designed following the classical composition design pattern [39]. In order to compose such a tree, the original kernel code generation process of HPL has been tweaked. The modifications performed are explained in Section 4.2.1. In addition to an abstract representation of the kernel syntax, the optimizer also needs to collect information about the access patterns derived from the memory references found in the kernel. A classification for such patterns is presented in Section 4.2.2.

4.2.1. Overriding the original code generation process

Following the same path as in Section 4.1 to explain how HPL uses PETE, now we introduce the changes needed to use PETE in order to emit AST nodes and subtrees instead of composing OpenCL code strings.

Regarding the PETE specification files, the `EXPR` property of each operator defined must be modified to return an instance of the AST node class that represents the corresponding operation. Such instances are created by calling the proper node constructor. Lines 5 and 13 from Listing 4.6 show these changes in relation to the operators described in the original `StringOps.in` excerpt contained in Listing 4.1. Now, the operand arguments expected by the node constructors are instances of any class of the AST hierarchy, so that both literals and `Array` objects must be also loaded into AST nodes. Listings 4.7 and 4.8 contain the code specializations needed so that PETE emits such nodes for `Array` operands and `int` literals, respectively. In the `Array` operands case, notice how a `generateASTNode()` method is invoked in lines 6 and 15. This method, which is the counterpart of `string()`, has been added to both the `IndexedArray<T,ndim>` and `Array<T,0>` interfaces and it returns the AST node representation of such memory accesses. As line 5 from Listing 4.8 shows, for literals just a node containing its string representation is returned.

Unsurprisingly, similar changes must be performed on both macros and `Codifier` methods that deal with HPL code constructs. The new version of the `for_` macro definition is shown in Listing 4.9. In this case, the arguments are first transformed into AST nodes (lines 8-10) and then passed to a modified implementation of `Codifier::for_()`. Now this method, instead of emitting a string encoding the OpenCL `for` header, it instantiates an AST node representing a `for` loop (line 6 of

```
1  unaryOps
2  -----
3    TAG = "OpNs"
4    FUNCTION = "native_sqrt"
5    EXPR = "return new FunctionUnaryOpASTNode(\"native_sqrt\",a);"
6  -----
7  ...
8  -----
9  binaryOps
10 -----
11   TAG = "OpAdd"
12   FUNCTION = "operator+"
13   EXPR = "return new BinaryOpASTNode(\"+\",a,b);"
14 -----
15 ...
```

Listing 4.6: PETE modifications to emit an AST: `StringOps.in` example

```
1  template<typename T, int NDIM>
2  struct LeafFunctor<HPL::IndexedArray<T, NDIM>, GenerateAST> {
3    typedef HPL::String_t Type_t;
4    static Type_t apply(const HPL::IndexedArray<T, NDIM> &v,
5        const GenerateAST &)  {
6      return v.generateASTNode();
7    }
8  };
9
10 template<typename T>
11 struct LeafFunctor<HPL::Array<T, 0>, GenerateAST> {
12   typedef HPL::String_t Type_t;
13   static Type_t apply(const HPL::Array<T, 0> &v,
14       const GenerateAST &) {
15     return v.generateASTNode();
16   }
17 };
```

Listing 4.7: PETE modifications to emit an AST: HPL `Array` example

Listing 4.10). Then, that node is appended to its parent in the tree by means of the `appendNode()` method called in line 7. At this point, it is worth mentioning that, initially, a node representing the kernel function body is set as parent and therefore, any top-level expression found is appended to it when `appendNode()` is called. In order to support nested code constructs, a stack of parent nodes is maintained. In this case, the stack is updated in line 8 by pushing the new node created. Then, the new top element of the stack is set as the current parent node of the `Codifier`

```
1  template<>
2  struct LeafFunctor<int, GenerateAST> {
3    typedef HPL::String_t Type_t;
4    static Type_t apply(int a, const GenerateAST &) {
5      return HPL::LeafASTNode(HPL::stringize(a));
6    }
7  };
```

Listing 4.8: PETE modifications to emit an AST: `int` literal example

```
1  #define HPL_common_block_macro
2    for(int _hpl_tmp = 0;
3      _hpl_tmp < 1;
4      TheGlobalState().getCodifier().endBlock(), ++_hpl_tmp)
5
6  #define for_(a,b,c)
7    TheGlobalState().getCodifier().for_(
8      getASTNode(a),
9      getASTNode(b),
10     getASTNode(c)
11   );
12   HPL_common_block_macro
```

Listing 4.9: PETE modifications to emit an AST: `for_` example

```
1  void Codifier::for_(const ASTNode* init,
2                      const ASTNode* cond,
3                      const ASTNode* update)
4  {
5    <...>
6    ForLoopASTNode* forNode = new ForLoopASTNode(init,cond,update)
7    appendNode(forNode);
8    parentNodes_.push(forNode);
9    parentNode_ = parentNodes_.top();
10   <...>
11   beginBlock();
12 }
```

Listing 4.10: PETE modifications to emit an AST: `for_` loop processing

object (line 9), so that the subtrees representing further expressions found inside the `for_` block are appended to it. Once the code of the HPL `for_` loop is processed, a modified version of the `endBlock()` method is called. This version pops the `for` node from the stack of parents, and then sets back the top of the stack as the current parent node of the codifier.

With these modifications, the result of evaluating an HPL kernel function is an AST representing it, instead of its OpenCL translation. However, as the HPL backend is OpenCL, we still need a mechanism to emit an OpenCL implementation for a given tree. Such implementation is built by means of a `stringize()` method, whose implementation is mandatory for all the classes of the AST hierarchy. In leaf nodes, this method directly emits the equivalent OpenCL string, whereas in non-terminal nodes it visits the children to invoke the same method and then combine the strings received to generate its translation. Thus, the invocation of the `stringize()` method of the root node of the tree will eventually generate an OpenCL implementation of the kernel. Notice how now the expression parsing and the code generation processes are two separate steps, which enables the ability to manipulate the AST in any desired way before translating it into OpenCL code. This allows the library to apply to the input kernel a set of source-to-source code optimizations in form of transformations performed on the AST representation. Thus, when the optimization process is finished, the transformed AST is translated into an optimized OpenCL version of the input HPL kernel.

4.2.2. Gathering memory access information

Before the optimization process starts, the AST has to be populated with information on the access patterns that appear in the code. In order to do that, the references to data structures located in global memory are searched in the branch of the AST corresponding to the *compute* section. These references are classified according to its memory access pattern. In order to do that, the optimizer implements a simplified version of the analyzer described in [32], which uses the index expressions of each reference to classify them in one of these seven types, ordered from the simplest to the most complex one:

1. `NoPat`: It is the default pattern. The expression(s) that index the data structure do(es) not include global identifiers or loop counters.

2. `SinglePat`: The indexing expressions only contain global identifiers, each position of the data structure being accessed by one single work item. HPL allows the users to define global domains with up to three dimensions, which leads to the three different cases depicted in Figure 4.2.

3. `InnerPat`: The indexing expressions only contain inner computing loop counters, each dimension of the structure being traversed using a loop counter with stride 1. As Figure 4.3 depicts, the following cases arise depending on both the number of inner computing loops found in the code and the dimensionality of the structure traversed:

   - For 1D structures, the `(kx)` counter of a single inner computing loop iterates along the only dimension of the structure.
   - For 2D structures, the `(ky,kx)` counters of a two-loop nest iterate respectively along the rows (`y` dimension) and the columns (`x` dimension) of the structure.
   - For 3D structures, the `(kz,ky,kx)` counters of a three-loop nest iterate respectively along the `z`, `y` and `x` dimensions of the structure.

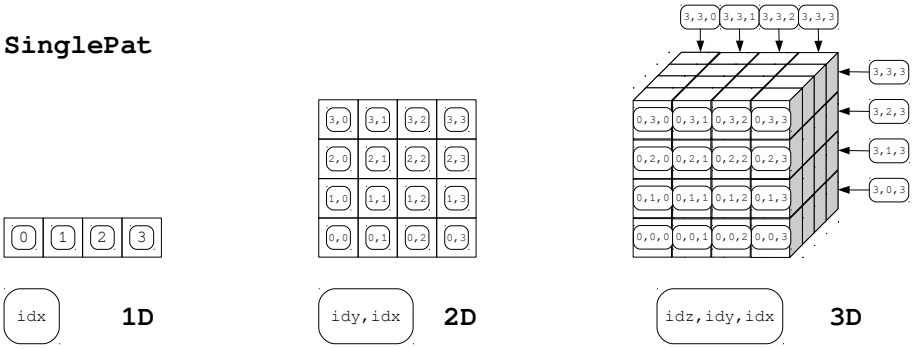


Figure 4.2: Just-in-time optimizer memory patterns: `SinglePat`

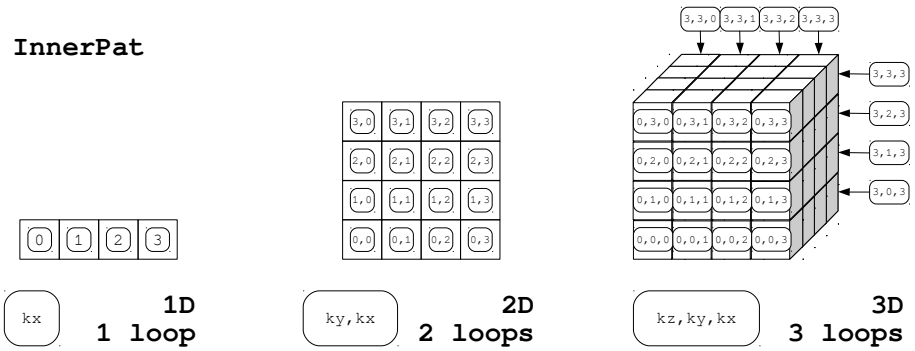


Figure 4.3: Just-in-time optimizer memory patterns: `InnerPat`

4. `RowPat`: The rightmost dimension of a structure is indexed using a loop counter with stride 1, whereas the rest of its dimensions must be indexed by the components of the global identifiers corresponding to the remaining dimensions. These conditions lead to three possible cases, the following two being depicted in Figure 4.4:

   - For 2D structures, each `(idy,idx)` thread in the 2D domain iterates on the `idy`-th row of the structure.
   - For 3D structures, each `(idz,idy,idx)` thread in the 3D domain iterates on the `(idz,idy,:)` row of the structure.

   Regarding 1D structures, in such cases all the threads in the 1D domain would iterate along the only dimension of the structure, this situation being already identified as a 1D `InnerPat` memory access.

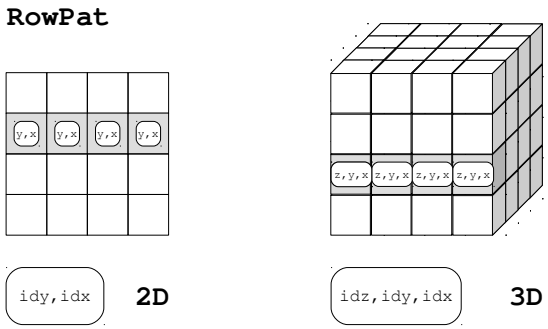


Figure 4.4: Just-in-time optimizer memory patterns: `RowPat`

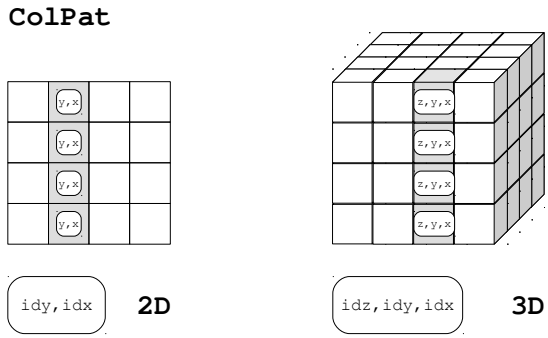


Figure 4.5: Just-in-time optimizer memory patterns: `ColPat`

5. `ColPat`: In this pattern, the loop counter indexes the second rightmost dimension of the structure, and the rest are indexed by the corresponding global identifiers. This pattern represents a work-item traversing slices in a column-major order, so it can only appear in structures whose dimensionality is greater than 1. Figure 4.5 shows the two possibilities of this pattern:

   - For 2D structures, each `(idy,idx)` thread in the 2D domain iterates on the `idx`-th column of the structure.
   - For 3D structures, each `(idz,idy,idx)` thread in the 3D domain iterates on the `(idz,:,idx)` column of the structure.

6. `DepthPat`: In this pattern, the loop counter indexes the third rightmost dimension of the structure, while the rest are indexed by the corresponding global identifiers. This pattern, shown in Figure 4.6, represents a work-item traversing slices across planes in a 3D structure, which are the only for which this pattern can appear. Thus, each `(idz,idy,idx)` thread in the 3D domain iterates on a slice `(:,idy,idx)` across the planes of the structure.

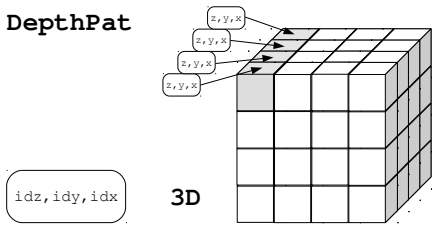


Figure 4.6: Just-in-time optimizer memory patterns: `DepthPat`

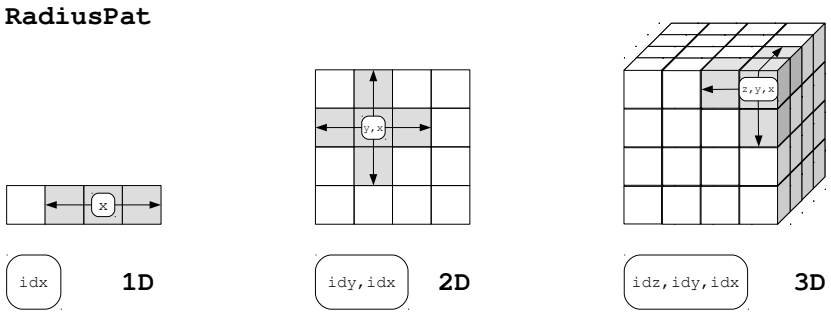


Figure 4.7: Just-in-time optimizer memory patterns: `RadiusPat`

7. `RadiusPat`: The expressions that index one or several dimensions operate a global identifier and a loop counter. Thus, a single work item visits several positions around a pivot position of the data structure defined by the global identifiers. 1D, 2D and 3D cases of this pattern are shown in Figure 4.7.

    - For 1D structures, each thread `(idx)` in the 1D domain pivots on the `(idx)` position of the structure to visit several positions along the `x` dimension using a loop counter.
    - For 2D structures, each thread `(idy,idx)` in the 2D domain pivots on the `(idy,idx)` position of the structure to visit several positions along both the `x` and `y` dimensions using two loop counters.
    - For 3D structures, each thread `(idz,idy,idx)` in the 3D domain pivots on the `(idz,idy,idx)` position of the structure to visit several positions along the `x`, `y` and `z` dimensions using three loop counters.

Once the pattern of a single memory reference has been identified, the data structure accessed is classified as having the same type of pattern. When a data structure is accessed by multiple references with different patterns, it will be classified as of the type of the most complex one. We will see the utility of this classification along the explanation of the optimization process of the AST, which is introduced in the next section.

4.3. Just-in-time optimization process

By now we have introduced how an HPL kernel can be loaded into an abstract syntax tree and how additional information about the memory accesses performed by the kernel can be extracted. As the flowchart depicted in Figure 4.1 shows, those are two of the three inputs required to run the just-in-time optimization process. This process consists on the application at run-time of several strategies that are known to be effective to optimize code for heterogeneous systems. Such strategies are implemented by means of transformations that, likewise those included on self-adaptive kernels, are driven by a number of parameters. Depending on the values given to such parameters, both the set of transformations applied and the way each

one is performed individually on the AST will vary. By properly tuning such values, the library is able to build at run-time different AST representations for an input kernel and, thus, to generate OpenCL versions optimized for different target devices. Both the strategies followed and the transformations implemented to apply them are detailed in Section 4.3.1, whereas the parameters that control them are described in Section 4.3.2. As shown in Figure 4.1, the values for these parameters are the third input expected by the optimization process. In the tools presented in Chapters 2 and 3, multiple search algorithms were used to find tuned values for the optimization techniques supported. These algorithms were time-consuming to different extents, which made them incompatible with the just-in-time approach followed by this tool. Thus, the optimizer needs to be provided with a mechanism able to quickly find or, at least, to retrieve those values at run-time. To meet such a fundamental requirement, we have opted for defining some heuristics based on the general capabilities of different kinds of heterogeneous devices, rather than implementing a specific search algorithm. These heuristics are presented in Section 4.3.3.

4.3.1. Code transformation techniques

According to the CUDA C Programming Guide [81], the optimization of a CUDA code has to focus on three basic strategies:

- Maximize parallel execution to achieve maximum utilization.
- Optimize memory usage to achieve maximum memory throughput.
- Optimize instruction usage to achieve maximum instruction throughput.

These strategies, although explicitly recommended in this guide for Nvidia GPUs, are also applicable to the optimization of the GPUs of other manufacturers and of any other heterogeneous device capable of executing parallelized codes. In its aim of tuning codes for any kind of device, our optimizer tries to apply these three strategies by following these five steps:

1. The **tiling stage**, where the tiling technique is applied to the code in the compute section of the HPL kernel.

2. The **local memory exploitation** stage, that performs a set of transformations in the code aimed at using the local memory of the device, when available.

3. The **coarser grain adjustment** stage, where the code is generalized to allow the adjustment of the amount of work made by each thread.

4. The **private memory exploitation** stage, where some of the data structures are copied to private memory to decrease the pressure on the global memory.

5. The **compute loop unrolling** stage, where the innermost loop of the code in the compute section can be unrolled.

This way, the maximization of the parallel execution is targeted by stage 3. The optimization of the memory usage is targeted by stages 1, 2 and 4, and the maximization of the throughput is targeted by stages 3 and 5, although we will see that stage 4 also implies a loop unrolling optimization which also supports this strategy. The transformations made in each one of these stages are explained now in turn. The naive matrix multiplication kernel from Listing 4.11 will be used as a running example throughout this explanation.

Tiling

This step applies the well-known tiling optimization technique to all the loops in the compute section. This technique can only be applied when the kernel has at least one loop in its compute section. For example, a naive version of the SAXPY code will not have such a loop but a naive matrix multiplication, like the one in our running example, will have it. The purpose of this technique is to split the computation in tiles to ensure that the information used by the kernel can be maintained in the top levels of the memory hierarchy. Listing 4.12 shows the tiled version of the loop of the running example using a generic tile size of `tW0` iterations.

Local memory exploitation

The next step tries to exploit the local memory of the device when available. The local memory is shared among the threads of the same group. As a result,

```
1  void mxm(Array<float,2> c, Array<float,2> a,
2        Array<float,2> b, Int K)
3  {
4    Int k;
5    compute {
6      c[idy][idx] = 0.0f;
7      for(k=0;k<K;k++)
8        c[idy][idx] += a[idy][k] * b[k][idx];
9    }
10 }
```

Listing 4.11: MxM running example: input HPL kernel

```
1  ...
2  c[idy][idx] = 0.0f;
3  for(kk=0;kk<K;kk+=tW0) {
4    for(k=kk;k<kk+tW0;k++) {
5      c[idy][idx] += a[idy][kk] * b[kk][idx];
6    }
7  }
8  ...
```

Listing 4.12: MxM running example: application of loop tiling

in order to use it effectively, we have to choose which data structures will make use of local memory, copy to the local counterpart of each data structure the slices of them traversed by the threads of the same group, and rewrite the computation section by replacing the references to the global data structures by references to the aforementioned local counterparts.

In order to select the data structure that will be copied to local memory, the optimizer makes use of the information about the access patterns followed by each memory reference derived when the AST was built. Let us recall that in addition each data structure was classified as of the same type of access pattern as the most complex memory reference associated to it. The optimizer inspects this information and it selects the data structures having access patterns more complex than `SinglePat` to be loaded into local memory. In our running example the result of this classification is:

- `c[idy][idx]` is classified as `SinglePat`.
- `a[idy][k]` is classified as `RowPat`.

- `b[k][idx]` is classified as `ColPat`.

Thus, as both the `ColPat` and the `RowPat` access patterns are more complex than `SinglePat`, matrices `b` and `a` are selected to be loaded into local memory in this example. When the selection is done we have to follow four steps to transform the code: (1) the local memory counterpart structures have to be declared, (2) code snippets copying data from global to local memory must be generated, (3) the global references must be replaced by local ones in the compute section of the kernel, and (4) if any of the local structure is updated, the information must be copied back to global memory. Now, we give more details about these four steps.

The most challenging task of the first step, the declaration of the local array, is to find out which is the appropriate size of each dimension of a local data structure. These sizes are going to depend on the type of access pattern followed by the memory references associated to the data structure, and on whether tiling and coarser grain adjustment transformations are going to be applied to the code. This coarser grain adjustment transformation is applied in a subsequent step of the optimizer, but it decides which transformations are going to be applied at the beginning of the optimization process. Therefore, the information on whether this technique is going to be applied or not and the grain size are already available at this point.

Table 4.1 shows the expressions used to calculate the size of each dimension of the local data structure. In this table, `lszx`, `lszy` and `lszz` are the size of the local space for dimensions 0, 1 and 2 respectively. The parameters `tW0`, `tW1` and `tW2` are the tile sizes for inner computing loops 0, 1 and 2 respectively, if tiling has been applied to them. If not, their values will be the length of these loops. The `bszx`, `bszy` and `bszz` parameters appear when the coarser grain adjustment

dims		InnerPat	RowPat	ColPat	DepthPat	RadiusPat
1D	0	[tW0]	-	-	-	[lszx*bszx+tW0]
2D	1	[tW1]	[lszy*bszy]	[tW0]	-	[lszy*bszy+tW1]
	0	[tW0]	[tW0]	[lszx*bszx]	-	[lszx*bszx+tW0]
3D	2	[tW2]	[lszz*bszz]	[lszz*bszz]	[tW0]	[lszz*bszz+tW2]
	1	[tW1]	[lszy*bszy]	[tW0]	[lszy*bszy]	[lszy*bszy+tW1]
	0	[tW0]	[tW0]	[lszx*bszx]	[lszx*bszx]	[lszx*bszx+tW0]

Table 4.1: Expressions for each dimension size of the local data structure

transformation is applied. In this transformation the iterations of several loops are assigned in a block-cyclic manner to threads, and these parameters are the size of a block of iterations assigned to a given thread. Each parameter is associated to the loop whose counter indexes a given dimension. Like in the previous cases, `x` is associated to dimension 0, `y` to 1 and `z` to 2. The rationale of these expressions is that the optimizer has to copy to local memory only the slice of the data structure that is going to be traversed by the threads of the current group. In our running example, the declarations of the local memory counterparts of the data structures `a` and `b` are:

```
__local float lmem_a[lszy][tW0];
__local float lmem_b[tW0][lszx];
```

The second step of the transformation consists in copying the information from global to local memory. We use copy mechanisms similar to those described in [34], which make use of the access pattern information that we already have. Also, these mechanisms make sure that the copied data is organized as its copy in global memory, which simplifies the third step.

Then, in the third step, we have to modify all the references to the global version of each data structure in the compute section of the kernel, so that they refer to their local counterparts. In addition, the indexing of these references has to be adjusted to use local identifiers instead of global ones.

If the data structures that have been copied to the local memory are written, the optimizer has to perform a fourth step to copy the information back to global memory. In this case we use the complementary code to the one used in the second step for the reverse copy.

Finally, the optimizer has to introduce at certain points of the code the local barriers required to synchronize the operation of the different threads of the same group. For example, after a collaborative copy is done, a local barrier must be performed to make sure that the copy is completed before the computation starts.

The code snippet in Listing 4.13 shows how our running example is adapted to use local memory. In this case, the information does not have to be copied back to global memory, as the information mapped to local memory is only read.

```
1  ...
2  for(kk=0;k<K;kk+=tW0) {
3
4    for(lr=lidy;lr<lszy;lr+=lszy)
5      for((lc=lidx);lc<tW0;lc+=lszx))
6        lmem_a[lr][lc] = a[((idy/lszy)*lszy)+lr][kk+lc];
7
8    for(lr=lidy;lr<tW0;lr+=lszy)
9      for((lc=lidx);lc<lszx;lc+=lszx))
10       lmem_b[lr][lc] = b[kk+lr][((idx/lszx)*lszx)+lc];
11
12   barrier(CLK_LOCAL_MEM_FENCE);
13
14   for(k=0;k<tW0;k++)
15     c[idy][idx] += lmem_a[lidy][k]*lmem_b[k][lidx];
16
17   barrier(CLK_LOCAL_MEM_FENCE);
18 }
19 ...
```

Listing 4.13: MxM running example: local memory exploitation

Coarser grain adjustment

The next step tries to adjust the number of threads and, conversely, the amount of work made by each thread. In order to do this, important modifications must be performed in the code, as we have to add loops that allow to change the number of points of the result that are going to be computed by each thread. Let us recall that in order to benefit from the optimizer, the HPL programmer has to provide a naive version of the kernel that computes just one point of the result. Therefore, this naive version minimizes the grain size and maximizes the number of threads required. As a result, the kernel will have less loops than its sequential version because the loops have been replaced by parallel executions of the kernel.

A sequential version of our matrix multiplication is shown in Listing 4.14. Let us notice that the naive kernel shown in Listing 4.11 removes the two outermost loops, those that index the resulting matrix, and keeps the innermost one, because it is required to calculate a single point of the result.

As a first step of this transformation, the optimizer is going to recover these loops, but written in a normalized way. To do this, the existing loops in the compute

```
1 for(j=0;j<M;j++) {
2   for(i=0;i<N;i++) {
3     for(k=0;k<K;k++) {
4       c[j][i] += a[j][k] * b[k][i];
5     }
6   }
7 }
```

Listing 4.14: MxM running example: sequential version

```
1 ...
2 for(zz=idz*bszz; zz<Z; zz+=szz*bszz)
3   for(yy=idy*bszy; yy<Y; yy+=szy*bszy)
4     for(xx=idx*bszx; xx<X; xx+=szx*bszx)
5       for(z=zz;z<min(zz+bszz,Z);z++)
6         for(y=yy;y<min(yy+bszy,Y);y++)
7           for(x=xx;x<min(xx+bszx,X);x++)
8             [...]
9 ...
```

Listing 4.15: MxM running example: coarser grain adjustment generic loop nest

section are going to be enclosed in new loops, a pair per dimension of the global space, and the global identifiers are going to be replaced by the counters of these loops in all the indexing expressions. This transformation enables the distribution of the work among a reduced number of threads so that it is possible to reduce the number of threads that perform the computation. Listing 4.15 shows a generic form of the loops that would enclose the existing computation if the three dimensions of the global work-space were used in the naive version of the code. Each pair of loops in lines 2 and 5, lines 3 and 6, and lines 4 and 7, assigns the iterations following a block-cyclic distribution, where the block sizes are `bszz`, `bszy` and `bszx`, respectively.

Listing 4.16 shows a version of our running example with the loops added, where `szx` = `N`/4, `szy` = `M`/4, and `bszx` = `bszy` = 2. Let us also recall that `N` and `M` are respectively the number of columns and rows of the resulting matrix `c`. In this case, the parallel execution requires 16 times less threads and each thread is going to execute two blocks of two iterations each for each one of the two pair of loops added. Previous research from [29] showed that the overhead introduced by these loops is not negligible. For this reason, the optimizer applies small optimizations

```
1  for(yy=idy*2; yy<M; yy+=(M/4)*2) {
2    for(xx=idx*2; xx<N; xx+=(N/4)*2) {
3      for(y=yy;y<min(yy+2,M);y++) {
4        for(x=xx;x<min(xx+2,N);x++) {
5          c[y][x] = 0.0f;
6          for(k=0;k<K;k++){
7            c[y][x] += a[y][k] * b[k][x];
8          }
9        }
10     }
11   }
12 }
```

Listing 4.16: MxM running example: application of coarser grain adjustment

on top of this technique, like removing the inner loop of a pair when the block size is 1, or removing the outer one when only one block of iterations is assigned to one thread.

Private memory exploitation

One of the consequences of the transformation applied in the previous step is that as each thread accesses a larger global memory area, hence there is an increase in the pressure on the global memory. One way to alleviate this pressure is to make use of the private memory of each processor. In order to do that, good candidate references must be identified to target with this transformation. Thus, global memory positions whose content is updated with new values and are clearly eligible. Exploiting the private memory in such a way contributes to maximize the usage of processor registers, which is expected to largely increase the performance of the kernel. The structure of this transformation is similar to the one related to the exploitation of local memory. First, a private data structure of the appropriate size has to be declared, then the contents of this private data structure has to be initialized. After that, the global memory references have to be replaced by private ones, and finally, the contents of the private data structure must be copied back to the corresponding positions of the global memory. These four steps of this transformation are now explained in turn.

First, the declaration of the private memory data structure has to be placed at

the beginning of the kernel. This private memory declaration can take two different forms. Namely, it can be either an array with the appropriate number of dimensions and of the appropriate size, or it can be a set of independent scalar variables. The first option is the most logical one and it will simplify the transformation, as the code will be more natural. However, some device architectures do not support addressing such private memory regions [40] and some compilers do not map arrays in private memory to registers but to arrays in global memory, which is counterproductive. Thus, the explicit declaration in private memory of this set of scalar variables seems more artificial, but it solves the aforementioned issue.

In our running example, using the version with coarser grain adjustment in Listing 4.16 as an starting point, the best candidate data structure to be stored in private memory is the result matrix `c`. Each thread is going to generate 2 blocks of 2×2 elements of the result, totalling 4×4 elements. Thus, that is the size of the private data structure that must be declared. If we opt for a single array of the appropriate size, the corresponding declaration would be as follows:

```
float pBlock_c[4][4];
```

In turn, if we opt for declaring 16 scalars, the declaration would be:

```
float pBlock_c_000, pBlock_c_001;
float pBlock_c_010, pBlock_c_011;
float pBlock_c_100, pBlock_c_101;
float pBlock_c_110, pBlock_c_111;
float pBlock_c_200, pBlock_c_201;
float pBlock_c_210, pBlock_c_211;
float pBlock_c_300, pBlock_c_301;
float pBlock_c_310, pBlock_c_311;
```

Furthermore, sometimes the value updated in the original global memory position depends on previous calculations, their results being stored in intermediate variables already declared as private in the naive kernel. In these cases, a set of as many private memory positions as the grain size has been adjusted to must be also allocated for each intermediate variable. These situations are detected by means of a simplified dependence analysis routine that we have implemented. This situation does not arise in our matrix multiplication running example.

Second, the private data structure must be initialized in the same way as the

```
1  // Block initialization loop
2  for(y=0;y<4;y++)
3    for(x=0;x<4;x++)
4      pBlock_c[y][x] = 0.0f;
5
6  // Initialization using several private scalars
7  float pBlock_c_000 = 0.0f;
8  float pBlock_c_001 = 0.0f;
9  ...
10 float pBlock_c_311 = 0.0f;
```

Listing 4.17: MxM running example: private memory initialization options

```
1  for(yy=idy*2; yy<M; yy+=(M/4)*2) {
2    for(xx=idx*2; xx<N; xx+=(N/4)*2) {
3      br=0;
4      for(y=yy;y<min(yy+2,M);y++) {
5        bc=0;
6        for(x=xx;x<min(xx+2,N);x++) {
7          for(k=0;k<K;k++) {
8            pBlock_c[br][bc] += a[y][k] * b[k][x];
9          }
10         bc++;
11       }
12     br++;
13     }
14   }
15 }
```

Listing 4.18: MxM running example: compute section using private arrays

global data structure in the original code. Before that, if this initialization is done inside a loop and the optimizer chooses to generate several private scalars, this loop must be unrolled with an unroll factor equal to the grain size used in the previous step. Listing 4.17 shows the result of the application of this step in our running example. Code initialization snippets derived from both declaration options are shown in the same figure. First, if the optimizer decides to generate a single private data structure, and second, if it opts for several scalar variables.

Third, global memory references of the compute section have to be replaced with their counterparts accessing the private memory structure. If the optimizer chose to use a single data structure, this process involves using the new private data structure but with the appropriate indexes. Listing 4.18 shows the new compute section of

our running example using the private arrays. Notice the instructions added to initialize (lines 3 and 5) and update (lines 10 and 12) properly the counters used to reference the private memory structure. Nevertheless, if the optimizer opted for using several private scalars, the transformation involves unrolling the whole loop nest that was added to adjust the iteration distribution of the kernel. Let us recall that scalar variables cannot be indexed using loop counters. This unrolling is usually more complicated than the one of the initialization, as the original compute section of the naive kernel may be an imperfect loop nest or a combination of multiple code constructs. In such cases, instead of applying a simple unrolling the optimizer should perform an unroll-and-jam transformation, the implementation of this latter one being a bit more complex. Listing 4.19 shows the same example but using private scalars. In this code the loops that originally indexed references to the

```
1  ...
2  for(k=0;k<K;k++) {
3    pBlock_c_000 += a[(idy*2)+(0*(M/4)*2)+0][k]*b[k][(idx*2)+(0*(N/4)*2)+0];
4    pBlock_c_001 += a[(idy*2)+(0*(M/4)*2)+0][k]*b[k][(idx*2)+(0*(N/4)*2)+1];
5    pBlock_c_010 += a[(idy*2)+(0*(M/4)*2)+1][k]*b[k][(idx*2)+(0*(N/4)*2)+0];
6    ...
7    pBlock_c_301 += a[(idy*2)+(2*(M/4)*2)+0][k]*b[k][(idx*2)+(2*(N/4)*2)+1];
8    pBlock_c_310 += a[(idy*2)+(2*(M/4)*2)+1][k]*b[k][(idx*2)+(2*(N/4)*2)+0];
9    pBlock_c_311 += a[(idy*2)+(2*(M/4)*2)+1][k]*b[k][(idx*2)+(2*(N/4)*2)+1];
10 }
11 ...
```

Listing 4.19: MxM running example: compute section using private scalars

```
1  \\ Block copy-back loop
2  for(yy=0;yy<4;yy+=2) {
3    for(xx=0;xx<4;xx+=2) {
4      for(y=0;y<2;y++) {
5        for(x=0;x<2;x++) {
6          c[(idy*2)+(yy*(M/4)*2)+y][(idy*2)+(yy*(M/4)*2)+y] = pBlock[yy+y][xx+x];
7        }
8      }
9    }
10 }
11
12 \\ Copy-back from several private scalars
13 c[(idy*2)+(0*(M/4)*2)+0][(idx*2)+(0*(N/4)*2)+0] = pBlock_c_000;
14 c[(idy*2)+(0*(M/4)*2)+0][(idx*2)+(0*(N/4)*2)+1] = pBlock_c_001;
15 c[(idy*2)+(0*(M/4)*2)+1][(idx*2)+(0*(N/4)*2)+0] = pBlock_c_010;
16 ...
17 c[(idy*2)+(2*(M/4)*2)+0][(idx*2)+(2*(N/4)*2)+1] = pBlock_c_301;
18 c[(idy*2)+(2*(M/4)*2)+1][(idx*2)+(2*(N/4)*2)+0] = pBlock_c_310;
19 c[(idy*2)+(2*(M/4)*2)+1][(idx*2)+(2*(N/4)*2)+1] = pBlock_c_311;
```

Listing 4.20: MxM running example: private memory copy-back options

private memory structure have been fully unrolled first, and then each reference has been replaced with its corresponding scalar variable. Notice also how the indexes of the references that still access global memory must be also unrolled.

This application of the unrolling technique can generate a couple of potential issues. First, the size of the kernel code is limited in most devices, and when large grain sizes are set this technique can increase considerably the kernel code size. Second, such grain sizes may speed up the kernel in some platforms, but the time consumed by the transformation process might hide that improvement. Thus, it is important that the optimizer chooses wisely the grain size in order to avoid exceeding these limitations after the unrolling is applied.

Finally, in the fourth step the information of the private variables is copied back their corresponding global memory positions. In turn, the private variables that might have been allocated to perform intermediate calculations were not related to any global memory position and, hence, they would not copied back anywhere. Listing 4.20 contains the copy-back sections of our running example using both private arrays and private scalars.

Let us remind that this transformation technique is able to increase the performance largely as a result of the maximization of the usage of the processor registers. Nevertheless, we must also note that it can also generate registers spilling if more private memory positions than registers available are allocated, which would cause the opposite effect. Thus, it is important to carefully select the grain size when applying the coarser grain adjustment transformation.

Compute loop unrolling

Loop unrolling is another well-known optimization technique. In this step, the optimizer can unroll the innermost loop of the compute section of the naive kernel. Such a transformation increases the number of independent statements available to be scheduled and may help the processor to discover groups of instructions that can be packed and automatically vectorized. Listing 4.21 shows one version of the innermost compute loop unrolled with a generic factor `uf`. This technique could be applied just after tiling the loop but, as Figure 4.1 shows, it is has been relegated to the last step of the optimization process. That made the cumulative application of

```
...
for(kk=0;kk<K;kk+=tW0) {
  ...
  for(k=kk;k<kk+tW0;k+=uf) {
    c[idy][idx] += a[idy][kk+0] * b[kk+0][idx];
    c[idy][idx] += a[idy][kk+1] * b[kk+1][idx];
    ...
    c[idy][idx] += a[idy][kk+(uf-1)] * b[kk+(uf-1)][idx];
  }
  ...
}
...
```

Listing 4.21: MxM running example: unrolling a previously tiled loop

```
...
for(kk=0;kk<K;kk+=tW0) {
  ...
  for(k=kk;k<kk+tW0;k+=uf) {
    pBlock_c_000 += a[(idy*2)+(0*(M/4)*2)+0][kk+0]*b[kk+0][(idx*2)+(0*(N/4)*2)+0];
    pBlock_c_000 += a[(idy*2)+(0*(M/4)*2)+0][kk+1]*b[kk+1][(idx*2)+(0*(N/4)*2)+0];
    ...
    pBlock_c_000 += a[(idy*2)+(0*(M/4)*2)+0][kk+(uf-1)]*b[kk+(uf-1)][(idx*2)+(0*(N/4)*2)+0];

    pBlock_c_001 += a[(idy*2)+(0*(M/4)*2)+0][kk+0]*b[kk+0][(idx*2)+(0*(N/4)*2)+1];
    pBlock_c_001 += a[(idy*2)+(0*(M/4)*2)+0][kk+1]*b[kk+1][(idx*2)+(0*(N/4)*2)+1];
    ..
    pBlock_c_001 += a[(idy*2)+(0*(M/4)*2)+0][kk+(uf-1)]*b[kk+(uf-1)][(idx*2)+(0*(N/4)*2)+1];

    pBlock_c_010 += a[(idy*2)+(0*(M/4)*2)+1][kk+0]*b[kk+0][(idx*2)+(0*(N/4)*2)+0];
    pBlock_c_010 += a[(idy*2)+(0*(M/4)*2)+1][kk+1]*b[kk+1][(idx*2)+(0*(N/4)*2)+0];
    ...
    pBlock_c_010 += a[(idy*2)+(0*(M/4)*2)+1][kk+(uf-1)]*b[kk+(uf-1)][(idx*2)+(0*(N/4)*2)+0];
    ..
    pBlock_c_301 += a[(idy*2)+(2*(M/4)*2)+0][kk+0]*b[kk+0][(idx*2)+(2*(N/4)*2)+1];
    pBlock_c_301 += a[(idy*2)+(2*(M/4)*2)+0][kk+1]*b[kk+1][(idx*2)+(2*(N/4)*2)+1];
    ...
    pBlock_c_301 += a[(idy*2)+(2*(M/4)*2)+0][kk+(uf-1)]*b[kk+(uf-1)][(idx*2)+(2*(N/4)*2)+1];

    pBlock_c_310 += a[(idy*2)+(2*(M/4)*2)+1][kk+0]*b[kk+0][(idx*2)+(2*(N/4)*2)+0];
    pBlock_c_310 += a[(idy*2)+(2*(M/4)*2)+1][kk+1]*b[kk+1][(idx*2)+(2*(N/4)*2)+0];
    ..
    pBlock_c_310 += a[(idy*2)+(2*(M/4)*2)+1][kk+(uf-1)]*b[kk+(uf-1)][(idx*2)+(2*(N/4)*2)+0];

    pBlock_c_311 += a[(idy*2)+(2*(M/4)*2)+1][kk+0]*b[kk+0][(idx*2)+(2*(N/4)*2)+1];
    pBlock_c_311 += a[(idy*2)+(2*(M/4)*2)+1][kk+1]*b[kk+1][(idx*2)+(2*(N/4)*2)+1];
    ...
    pBlock_c_311 += a[(idy*2)+(2*(M/4)*2)+1][kk+(uf-1)]*b[kk+(uf-1)][(idx*2)+(2*(N/4)*2)+1];
    ...
  }
}
...
```

Listing 4.22: MxM running example: tiled loop unrolling and private scalars usage

both techniques easier, the code in Listing 4.22 being the result of such a combined transformation.

4.3.2. Optimization parameters

As the flowchart depicted in Figure 4.1 shows, the optimization process is driven by a decision tree in which each branch implies a set of transformations to be performed or not on the input AST. As Section 4.3.1 explains, these transformations affect multiple aspects of the kernel execution, and they are implemented in a parametrized way. We now introduce those parameters grouped by the aspects they directly affect.

First, the optimization parameters related to the workspace configuration are listed in Table 4.2. Thus, both the number of threads that will be created and the groups gathering these threads must be defined. The work is going to be distributed equally among these threads in a block-cyclic basis, being consecutive and pure-cyclic distributions sub-cases of this one that can be configured giving the block sizes the proper values. Notice also that these global, local and block sizes can have up to three dimensions, as they affect the workspace configuration of the kernel. There is also an additional boolean flag to indicate whether the work distribution loops must be totally unrolled. As commented in Section 4.3.1, performing this unroll along with a proper selection of the values for the rest of the workspace-related parameters may lead to a more efficient exploitation of the private memory of the devices.

Another relevant property of a naive input kernel is how a single point of the solution space of the problem is computed. That calculation is usually implemented as one or more nested loops that iterate on both input and output memory structures. The parameters related to the optimization of this part of the code are shown in Table 4.3. Thus, the optimizer is able to transform each one of these loops by tiling it with some width, for which it must be provided with as many tile sizes as nested computing loops the naive kernel has. Moreover, the innermost loop of that nest can be unrolled too, this unroll factor being hence an additional optimization parameter.

Finally, the local memory can be exploited when available by transforming the kernel in the terms described in Section 4.3.1. The usage or not of the local memory depends on the boolean parameter shown in Table 4.4. Let us insist on the fact that this flag only commands the optimizer to apply or not this transformation. A glimpse

of Table 4.1 from the aforementioned Section 4.3.1 shows that this optimization depends indeed on the values given to the parameters related to both the workspace configuration and the nested computing loops. Namely, that table defines the sizes

Name	Workspace dimensions			Explanation
	1D	2D	3D	
Global size	szx	szx	szx	Global workspace sizes. One value per dimension of the work-space.
	-	szy	szy	
	-	-	szz	
Local size	lszx	lszx	lszx	Local workspace sizes. One value per dimension of the work-space.
	-	lszy	lszy	
	-	-	lszz	
Block size	bszx	bszx	bszx	Block sizes for block-cyclic distribution. One value per dimension.
	-	bszy	bszy	
	-	-	bszz	
Full block unrolling				Boolean indicating whether the work distribution loops must be fully unrolled.

Table 4.2: Workspace-related parameters of the just-in-time optimizer

Name	Nested computing loops			Explanation
	1 loop	2 loops	3 loops	
Tile size	tW0	tW0	tW0	Tile sizes for the inner computing loops. One value per each loop.
	-	tW1	tW1	
	-	-	tW2	
Innermost loop unroll factor				Factor to unroll the innermost computing loop. One single value.

Table 4.3: Computing loops-related parameters of the just-in-time optimizer

Name	Explanation
Local memory usage	Boolean indicating if local memory has to be exploited or not.

Table 4.4: Local memory-related parameters of the just-in-time optimizer

of the local memory structures declared by this optimization, which depend on the global, local, block and tile sizes.

4.3.3. Optimization heuristics

We have just described the parameters that drive the code transformations performed during the just-in-time optimization process of a naive kernel. Such a parametrized approach is clearly inspired in those followed to implement the optimizations applied by the tools presented in Chapters 2 and 3. In these two former tools, either exhaustive or informed search methods were used to determine adequate values for the optimization parameters on multiple target devices. Although these algorithms were proven to be effective to find those values, the time they consumed was too long to apply them in a just-in-time solution like the one introduced in this chapter. Because of that, in this case we opt for defining some heuristics able to provide values for the parameters of the optimizer without paying any search time. Broadly speaking, these heuristics follow the basic guidelines for optimizing codes in heterogeneous environments introduced at the beginning of Section 4.3.1, although in some cases they are considerably affected by practical aspects such as the implementation each vendor offers for the OpenCL standard.

The first strategy commands users to tune codes in order to maximize the parallel execution with the purpose of achieving a maximum utilization of the processing elements of the target device. Thus, theoretically in CPUs creating less threads that do more work each is the best option, while in GPUs more but lighter threads are preferred. Therefore, higher values are set for the global sizes in GPUs, usually near to the naive ones, than in CPUs. The optimizer can infer the block sizes from the values set for the global sizes. Regarding the local sizes, vendors generally ask users to let the runtime decide them automatically. However, as mentioned in Section 1.3 in relation to the the motivating example, the Intel OpenCL optimization guide for their multicore CPUs recommends programmers to pack many GPU-like threads into a same group and let the runtime use these groups to distribute the workload among the cores available [51], and also to try other group sizes different from that automatically set by the runtime.

The second strategy encourages an optimized exploitation of the memory hierar-

chy of the devices, trying to maximize its data throughput. Virtually all the parameters driving the code transformations affect this optimization strategy. In GPUs, the block sizes must be tuned in such a way that threads in a group can perform coalesced accesses to global memory. The tiling technique is going to be applied whenever possible, since a proper selection of values for the tile sizes is expected to favor cache locality in CPUs, and they are also part of the parameters that drive the local memory usage in GPUs. Focusing on local memory exploitation, the optimizer will be commanded to perform the associated code transformations if the target device is a GPU. In this case, the optimizer will transform the code in order to explicitly cache arbitrary data structures on the on-chip memory to which this OpenCL region is usually mapped. At this point, values for the local, block and tile sizes must be fixed taking into account that the threads in a group must be able to collaboratively copy their slices of each structure in a coalesced way and, of course, all these slices must also fit in the local memory. In CPUs, in turn, the local memory region is usually mapped somehow to the cache levels of the processor, which makes more advisable not to override its default management. Climbing up the memory hierarchy, private memory exploitation also depends both on the global and the block sizes, as well as on the tile size. With this optimization, results that in the naive code are originally returned straight away to global memory, are written first in private variables in order to reduce the memory contention. These variables are mapped to processor registers. Thus, as the block sizes grow, it is more likely to need more registers than available, a spilling problem arising then. By default, the optimizer tries to allocate that private space as an array. However, as we noted when this optimization was introduced, some device architectures and compilers do not deal properly with such private memory allocations, mapping directly these arrays to global memory even if the space required could fit in the registers available. The full block unrolling option can be activated to overcome this issue, since when the work distribution loop nest is fully unrolled the private space must be allocated as a collection of single scalar variables. When the only pattern access detected for a given structure in a naive kernel is `SinglePat`, it usually means that each individual thread is updating a single position of this global memory structure, this situation making the structure a candidate to be directly mapped to the private memory of the device. As a consequence, only structures that are accessed following patterns more complex than `SinglePat` are selected to be cached in local memory.

Finally, the third strategy recommends to write high-level codes in such a way that the instruction throughput of the processing elements is maximized. To achieve this, the optimizer offers two loop unroll transformations. First, we have just introduced the full block unroll flag, that was originally thought to tune the private memory exploitation. However, activating it along with setting a proper block size leads to an increase of the independent instructions available in the generated kernel, and this should reveal multiple automatic optimization options for the underlying OpenCL compiler. Second, there is also the option to unroll the innermost computing loop, although the factor applied in this case is limited by either the loop length, or the tile size if that loop has been previously tiled. Moreover, notice that in GPUs, and sometimes also in CPUs, global sizes may be left with their naive values, which implies a value of 1 for the block sizes. If each thread computes one position of the result, there are no work distribution loops to unroll. In these cases, such an increase in the number of independent instructions can be achieved by unrolling the innermost computing loop provided that the naive kernel had any.

Notice how all the parameters described somehow affect virtually all the code transformations performed by the optimizer. Because of that, it is not uncommon that values that maximize the benefits derived from one optimization may hamper those achieved by another one, or even make illegal the application of the latter. For instance, we have just seen how essential it is to find a trade-off between the values set for the local, block and tile sizes when we try to exploit the local memory in a GPU, since there are multiple related constraints affecting them. Thus, the values cannot either exceed the workspace configuration limits of the device, or imply the allocation of more local memory space than available, or hide coalescent accesses to global memory data in order to cache them in the local memory structure allocated. Such situations are not unfamiliar, as the validity conditions listed in Table 3.3 for the matrix multiplication self-adaptive kernels remind. Nevertheless, the heuristics described can be applied to obtain first a set of candidate parameter values for different device types and, then, perform by hand a fine adjustment on these values in order to find such balances, avoid illegal combinations and pin as far as possible to both each problem properties and each device architecture. In the future we plan to design some algorithms to fix them automatically.

4.4. Experimental results

This section contains the validation of the effectiveness of our just-in-time optimizer. The purpose of this validation is to prove that the optimizer can generate faster versions of a set of benchmarks for different types of platforms. The input kernels used in this validation process are HPL single-point implementations for one-, two-, and three-dimensional signal convolutions (1DCONV, 2DCONV, 3DCONV), a Direct Coulomb Summation [108] (DCS3D), a matrix multiplication (MATMUL), a single time step of an N-body simulation [1] (NBODY) and symmetric k- and 2k-rank update matrix operations (SYRK, SYR2K). 1DCONV and NBODY are defined in one-dimensional workspaces, 2DCONV, MATMUL, SYRK and SYR2K have two-dimensional workspaces and, finally, 3DCONV and DCS3D solution spaces have three dimensions. Regarding the input naive kernels that implement these problems, 1DCONV, NBODY, DCS3D, MATMUL, SYRK and SYR2K naive versions consist of a single inner computing loop, whereas 2DCONV and 3DCONV are computed by a two-loop and a three-loop nest, respectively. Moreover, as Table 4.5 shows, three different test classes named as "small" (S), "medium" (M) and "large" (L) have been defined by setting different combinations of sizes for the global workspace and the nested loops of each problem.

Optimized versions of these kernels have been generated running tests for the aforementioned three size classes on three different computing platforms, namely a CPU and two GPUs from different vendors:

- **CPU**: A dual-socket system with two Intel Xeon E5-2660 Sandy Bridge with eight 2.2Ghz cores each and Hyper-Threading (8×2 threads per processor, for a total of 32) and 64 GB of RAM. Intel OpenCL driver version 1.2-4.5.0.8. Single-precision theoretical peak performance of 563 GFLOPS.
- **Nvidia**: An NVIDIA Tesla K20m with Kepler GPU architecture and 5 GB GDDR5. NVIDIA OpenCL driver version 367.57. Single-precision theoretical peak performance of 3524 GFLOPS.
- **AMD**: An AMD FirePro S9150 with Hawaii GPU architecture and 16 GB GDDR5. AMD OpenCL driver version 1702.3. Single-precision theoretical peak performance of 5070 GFLOPS.

	Small (S) class		Medium (M) class		Large (L) class	
	Dimension sizes	Computing loops sizes	Dimension sizes	Computing loops sizes	Dimension sizes	Computing loops sizes
1DCONV	32768	32768	65536	65536	131072	131072
2DCONV	1024×1024	256×256	2048×2048	256×256	4096×4096	256×256
3DCONV	$64 \times 64 \times 64$	$64 \times 64 \times 64$	$128 \times 128 \times 128$	$64 \times 64 \times 64$	$256 \times 256 \times 256$	$64 \times 64 \times 64$
NBODY	32768	32768	65536	65536	131072	131072
DCS3D	$64 \times 64 \times 64$	4096	$128 \times 128 \times 128$	8192	$256 \times 256 \times 256$	16384
MATMUL	2048×2048	2048	4096×4096	4096	8192×8192	8192
SYRK	2048×2048	2048	4096×4096	4096	8192×8192	8192
SYR2K	2048×2048	2048	4096×4096	4096	8192×8192	8192

Table 4.5: Size classification of test cases run in the experiments

Now, a brief explanation of the work that each kernel performs and the results obtained after running their respective optimization test cases are discussed. Let us start with the 1DCONV kernel, which implements the convolution of two unidimensional signals of the same length that are stored in global memory. Each instance of the naive kernel computes a point of the result iterating on one of the signals following a 1D `RadiusPat` pattern, and following an 1D `InnerPat` pattern on the other one. So, when local memory is exploited, these two input signals are selected to be cached in local memory. The values provided to the optimization parameters to run each test of this kernel are detailed in Table 4.6, which lists them in the same order as Tables 4.2, 4.3 and 4.4 from Section 4.3.2. This way, the first parameters shown are the sizes of the global and local workspaces. Then, the parameters related to the grain adjustment are listed, namely the block sizes and whether its computing has been fully unrolled or not. Regarding the nested computing loops, the widths set to tile them are shown first, followed by the unroll factor applied to the innermost one. The last parameter shown indicates whether the generated code tries to exploit local memory or not. The columns are grouped by target platform, showing for each one the parameter values set to run the tests for each size class. An N/A value for a parameter means that the affected transformation is not applicable. Furthermore, for the tile sizes and the innermost loop unroll factor, values of N/T (not tiled) and N/U (not unrolled) respectively mean that these techniques have not been applied despite being possible to do so. Table 4.7 contains the performance results obtained by the versions generated using the aforementioned parameter values. Namely, it shows, for each test case, the execution time of both the naive version and the

optimized version generated, and the speedup obtained by the latter. Two additional times are shown in a separate group of rows inside the same table. The first row of this group contains the time consumed by the whole optimization process, which includes the building of an AST from a naive HPL kernel, the transformations performed on the AST to obtain an optimized version, and the translation of this optimized AST into an OpenCL kernel code. The second row of the group shows the time consumed by the OpenCL runtime of the corresponding target device to build an OpenCL program from the kernel code generated by the optimizer. The columns in this table are grouped in the same way as those of Table 4.6. An analogous pair of tables following the same layout will be also provided to support the discussion of the results obtained for the rest of the kernels.

Due to the memory access patterns followed, the naive implementation of the kernel suffers from an important memory access contention, as the threads are continuously accessing overlapping positions in global memory. Thus, the execution times of the naive versions run in both GPUs do not seem to be as good as expected when compared to those obtained in the CPUs, taking into account the the peak performances of each device. This issue seems to specially hamper the performance of such cases in the Nvidia GPU. After applying the optimizations detailed in Table 4.6, the performances obtained in both GPUs considerably increase,

Platform		Nvidia			AMD			CPU		
Size class		S	M	L	S	M	L	S	M	L
Global sizes	`szx`	32768	65536	131072	32768	65536	131072	32768	65536	131072
Local sizes	`lszx`	256	256	256	256	256	256	32	32	32
Block sizes	`bszx`	1	1	1	1	1	1	1	1	1
Full block unrolling		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Tile sizes	`tW0`	128	256	512	1024	1024	1024	16	16	16
Innermost loop unroll factor		128	256	512	N/U	N/U	N/U	N/U	N/U	N/U
Local memory usage		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE

Table 4.6: Optimization parameters set for 1DCONV

Platform	Nvidia			AMD			CPU		
Size class	S	M	L	S	M	L	S	M	L
Naive kernel time (ms)	41.04	152.80	531.49	17.38	37.76	258.07	19.82	77.69	311.16
Optimized kernel time (ms)	7.87	29.22	110.23	3.81	12.92	40.46	11.72	46.14	184.15
Speedup	5.21	5.23	4.82	4.56	2.92	6.38	1.69	1.68	1.69
Code generation time (ms)	3.38	6.20	11.90	0.80	0.80	0.84	0.29	0.30	0.30
OpenCL compilation time (ms)	8.74	9.34	19.09	79.13	117.51	80.84	59.32	59.58	57.82

Table 4.7: Performance results obtained for 1DCONV

as the aforementioned contention problems are mitigated by means of a previous collaborative caching of the input signals in local memory. Regarding the CPU, a proper selection of both the local workspace size and the width used to tile the inner computing loop of the kernel is contributing to achieve slight improvements on the performances obtained.

Let us note that for the AMD GPU and the Intel CPU platforms, the time consumed by their respective OpenCL runtimes to compile the kernel codes generated by the optimizer is considerably longer than the time the Nvidia runtime needed to perform the same task, probably because more thorough code analyses are run by the former two compilers. Moreover, sometimes these compilation times are several orders of magnitude greater than the kernel times of some optimized versions. However, in general, the larger the test size is, the larger the kernel times are and, therefore, also the lower the impact of optimized code compilation is. Furthermore, this is only a problem the first time a kernel is used in a program. As the flowchart depicted in Figure 3.1 from Section 3.1.2 shows, HPL keeps caches in which both OpenCL translations of kernels and their subsequent compiled programs are stored.

Platform		Nvidia			AMD			CPU		
Size class		S	M	L	S	M	L	S	M	L
Global sizes	szx	1024	2048	4096	1024	2048	4096	1024	2048	4096
	szy	1024	2048	4096	1024	2048	4096	1024	2048	4096
Local sizes	lszx	32	32	32	16	16	16	256	512	1024
	lszy	32	32	32	16	16	16	1	1	1
Block sizes	bszx	1	1	1	1	1	1	1	1	1
	bszy	1	1	1	1	1	1	1	1	1
Full block unrolling		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Tile sizes	tW0	32	32	32	32	32	32	N/T	N/T	N/T
	tW1	32	32	32	32	32	32	N/T	N/T	N/T
Innermost loop unroll factor		N/U	N/U	N/U	4	4	4	4	4	4
Local memory usage		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE

Table 4.8: Optimization parameters set for 2DCONV

Platform	Nvidia			AMD			CPU		
Size class	S	M	L	S	M	L	S	M	L
Naive kernel time (ms)	1850	7364	29406	1169	5205	23675	1250	4990	19908
Optimized kernel time (ms)	432	1725	6883	315	1577	7068	761	3043	12184
Speedup	4.29	4.27	4.27	3.72	3.30	3.35	1.64	1.64	1.63
Code generation time (ms)	0.93	0.86	0.98	1.49	1.26	1.29	0.83	0.52	0.47
OpenCL compilation time (ms)	2.80	3.47	4.36	190	256	261	67	67	66

Table 4.9: Performance results obtained for 2DCONV

Thus, the first time an HPL kernel is evaluated, both the OpenCL translation and the OpenCL compiled program are cached, they being ready for further uses along the same user application.

The 2DCONV kernel implements the convolution of a two-dimensional input matrix with a filter of size 256×256. To compute a point of the filtered matrix, the input is accessed from global memory following a 2D `RadiusPat`, whereas the filter is traversed according to a 2D `InnerPat` pattern. Thus, both the input matrix and the filter are selected to be cached in local memory when exploited. The values selected for the optimization parameters in each test case of this kernel are detailed in Table 4.8, whereas Table 4.9 shows the performance results obtained by the original versions and the ones generated using those parameter values. Similar issues to those suffered by the 1DCONV naive implementation arise in this case when its performance on both GPUs is compared to that obtained on the CPU. The optimizations listed in Table 4.8 were able to mitigate contention problems in GPUs, and also to obtain again slight improvements on the performance on the CPU.

The 3DCONV kernel implements the convolution of a three-dimensional input matrix with a $64 \times 64 \times 64$ filter. To compute a point of the result, the input is accessed from global memory following a 3D `RadiusPat`, whereas the filter is traversed according to a 3D `InnerPat` pattern. Thus, both the input matrix and the filter are selected to be cached in local memory when exploited. The values used for the optimization parameters in each test case of this kernel are detailed in Table 4.10, whereas Table 4.11 shows the performance results obtained by the original versions and the ones generated using those parameter values. The issues already commented for the 1DCONV and 2DCONV kernels also appear in this problem, although now in the large test case the naive version is considerably slower in the CPU than in both GPUs, probably because the CPU is not dealing well with the memory structures that do not properly fit in the cache. Regarding the optimized kernels, similar speedups to those obtained for the 2DCONV problem are achieved now by optimizing the 3DCONV kernel in similar terms.

The NBODY kernel computes the gravitational interactions of a set of particles randomly distributed in a three-dimensional space. Particle data are stored in a one-dimensional array of `float4` vectors, with each vector element containing the (x, y, z) coordinates and the mass of a particle. Each instance of the naive kernel

Platform		Nvidia			AMD			CPU		
Size class		S	M	L	S	M	L	S	M	L
Global sizes	szx	64	128	256	64	128	256	64	128	256
	szy	64	128	256	64	128	256	64	128	256
	szz	64	128	256	64	128	256	64	128	256
Local sizes	lszx	32	32	32	16	16	16	64	128	256
	lszy	8	8	8	8	8	8	1	1	1
	lszz	2	2	2	2	2	2	1	1	1
Block sizes	bszx	1	1	1	1	1	1	1	1	1
	bszy	1	1	1	1	1	1	1	1	1
	bszz	1	1	1	1	1	1	1	1	1
Full block unrolling		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Tile sizes	tW0	8	8	8	8	8	8	N/T	N/T	N/T
	tW1	8	8	8	4	4	4	N/T	N/T	N/T
	tW2	8	8	8	2	2	2	N/T	N/T	N/T
Innermost loop unroll factor		N/U	N/U	N/U	N/U	N/U	N/U	4	4	4
Local memory usage		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE

Table 4.10: Optimization parameters set for 3DCONV

Platform	Nvidia			AMD			CPU		
Size class	S	M	L	S	M	L	S	M	L
Naive kernel time (ms)	2251	14937	119205	1315	13185	121181	1269	10471	235243
Optimized kernel time (ms)	572	4542	36430	415	4570	40936	810	6899	202902
Speedup	3.94	3.29	3.27	3.17	2.89	2.96	1.57	1.52	1.16
Code generation time (ms)	1.30	1.34	1.33	1.94	1.91	1.96	0.62	0.64	0.67
OpenCL compilation time (ms)	2.99	3.45	9.64	155	182	212	89	90	96

Table 4.11: Performance results obtained for 3DCONV

calculates the acceleration experienced by a particle due to its interaction with the whole system. Then, that acceleration is used to compute the new position and speed of the associated particle. Notice that although particles are distributed in a 3D space, the problem lies in traversing that one-dimensional vector array, which follows a 1D `InnerPat` access pattern. This array is selected to be cached in the local memory when exploited. The values set to the optimization parameters in each test case of this kernel are detailed in Table 4.12, whereas Table 4.13 displays the performance results. A distinctive characteristic of this benchmark compared to the other unidimensional problems is that its code generation times are slightly longer, particularly for the GPU test cases, the reason being the complexity of its compute section and the high unroll factor applied. These two issues also seem to make the AMD GPU OpenCL runtime to considerably increase the time it needs to compile the kernels optimized for this device.

The DCS3D kernel computes the electrostatic potential of each position of a three-dimensional grid due to the interactions of 4096 randomly distributed charged

Platform		Nvidia			AMD			CPU		
Size class		S	M	L	S	M	L	S	M	L
Global sizes	`szx`	32768	65536	131072	32768	65536	131072	32768	65536	131072
Local sizes	`lszx`	256	256	256	256	256	256	256	512	1024
Block sizes	`bszx`	1	1	1	1	1	1	1	1	1
Full block unrolling		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Tile sizes	`tW0`	64	64	64	256	256	256	N/T	N/T	N/T
Innermost loop unroll factor		64	64	64	64	64	128	N/U	N/U	N/U
Local memory usage		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE

Table 4.12: Optimization parameters set for NBODY

Platform	Nvidia			AMD			CPU		
Size class	S	M	L	S	M	L	S	M	L
Naive kernel time (ms)	31.76	124.09	445.11	16.72	54.61	312.73	182.82	729.90	2879.99
Optimized kernel time (ms)	16.73	62.98	243.83	10.68	37.85	259.01	177.67	709.35	2838.30
Speedup	1.90	1.97	1.83	1.57	1.44	1.21	1.03	1.03	1.01
Code generation time (ms)	5.06	5.17	5.06	7.75	7.46	12.90	1.33	1.34	1.39
OpenCL compilation time (ms)	3.70	3.67	4.36	668	664	647	72.12	72.80	73.64

Table 4.13: Performance results obtained for NBODY

particles. Energy in each point of the grid depends on the (x, y, z) position and the potential of each particle. Thus, the problem is defined in a three-dimensional space, but the naive kernel traverses a `float4` 4096-particle array by means of a single compute loop. That array is cached in GPU local memory since it is accessed following a 1D `InnerPat` pattern. The values used for the optimization parameters in each test case of this kernel are detailed in Table 4.14, whereas Table 4.15 contains the associated performance results. Notice that particle properties in both NBODY and DCS3D kernels are stored from the outset as `float4` arrays, with each vector packing the (x, y, z) particle position and the fourth coordinate representing either mass or potential. This works as an optimization by itself, because it allows not only to exploit vector instructions to compute the distances between particles but also to get simultaneously the values for the aforementioned magnitudes. Moreover, it seems that letting both the Intel OpenCL runtime exploit its work-item grouping capabilities and the device manage itself the cache hierarchy does not leave much room for the optimizer to take advantage of any additional transformation. Because of that, neither code generation nor OpenCL compilation times are given in Table 4.15 for this device. The GPUs seem to tolerate much better than the CPU the 1D `InnerPat` pattern followed by the naive kernels of both NBODY and DCS3D problems, probably because such access pattern combined with a `bszx` of 1

is allowing coalesced reads. Furthermore, an adequate definition of both the local workspace and the width used to tile the compute loops leads to an optimized usage of the local memory and, thus, to more significant speedups, specially for the Nvidia GPU.

The MATMUL kernel performs a typical $C = A \times B$ matrix multiplication, both A and B being suitable candidates to be cached in local memory when exploited. The loop computing the dot product of a point of C in the naive kernel iterates on both matrices following different patterns. Namely, matrix A is read by rows in a 2D `RowPat` pattern, whereas matrix B is read by columns in a 2D `ColPat` pattern. According to the heuristics implemented in the optimizer, both structures are picked to be cached in local memory when exploited. The values set for the optimization parameters in each test case of this kernel are detailed in Table 4.16, whereas Table 4.17 shows the performance results for this benchmark. The performance of the versions generated for this MATMUL kernel are on average 2.01 times behind that of the best ones obtained by the self-adaptive test case described in Section 3.4. How-

Platform		Nvidia			AMD			CPU		
Size class		S	M	L	S	M	L	S	M	L
Global sizes	szx	64	128	256	64	128	256	64	128	256
	szy	64	128	256	64	128	256	64	128	256
	szz	64	128	256	64	128	256	64	128	256
Local sizes	lszx	64	128	256	32	32	32	AUTO	AUTO	AUTO
	lszy	1	1	1	4	4	4	AUTO	AUTO	AUTO
	lszz	1	1	1	2	2	2	AUTO	AUTO	AUTO
Block sizes	bszx	1	1	1	1	1	1	1	1	1
	bszy	1	1	1	1	1	1	1	1	1
	bszz	1	1	1	1	1	1	1	1	1
Full block unrolling		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Tile sizes	tW0	128	256	512	512	512	512	N/T	N/T	N/T
Innermost loop unroll factor		N/U	N/U	N/U	N/U	N/U	N/U	N/U	N/U	N/U
Local memory usage		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE

Table 4.14: Optimization parameters set for DCS3D

Platform	Nvidia			AMD			CPU		
Size class	S	M	L	S	M	L	S	M	L
Naive kernel time (ms)	48.96	519.36	8294	13.92	367.57	8497	177.70	2841.17	45416
Optimized kernel time (ms)	16.23	248.56	3973	11.26	267.97	6886	177.70	2841.17	45416
Speedup	3.02	2.09	2.09	1.24	1.37	1.23	1.00	1.00	1.00
Code generation time (ms)	1.09	1.11	1.12	1.85	1.54	1.51	-	-	-
OpenCL compilation time (ms)	3.50	3.13	3.05	170	170	170	-	-	-

Table 4.15: Performance results obtained for DCS3D

ever, code transformations, scope and capabilities differ in each solution. Regarding the local memory exploitation, the self-adaptive kernel chooses among matrices A, B, or both, to be cached. This just-in-time optimizer, however, implements a more general approach, automatically selecting all the input structures following memory access patterns more complex than `SinglePat`. Moreover, the self-adaptive kernel also implements a quite optimized version of the matrix multiplication algorithm that is able to explicitly vectorize, unroll and reorder the loops that compute in private memory the dot product operations for each position of C. For the sake of a broader scope, the just-in-time optimizer is agnostic about which particular operation the compute section runs. Regarding the OpenCL compilation times of the optimized versions tested for this problem, it is noticeable how the full unrolling of the computation performed on the 32×8 private memory variables led to such a complex code that the Intel OpenCL runtime needed about 10 seconds to compile it. Let us recall that thanks to the HPL kernels cache, this would be a problem only the first time the code is generated and compiled.

The SYRK kernel performs the symmetric rank one update $C = \alpha AA^T + \beta C$, the optimizer detecting a 2D `RowPat` access pattern for A. Thus, A is cached in lo-

Platform		Nvidia			AMD			CPU		
Size class		S	M	L	S	M	L	S	M	L
Global sizes	szx	512	1024	2048	512	1024	2048	256	512	1024
	szy	512	1024	2048	512	1024	2048	64	128	256
Local sizes	lszx	16	16	16	16	16	16	128	128	128
	lszy	16	16	16	16	16	16	1	1	1
Block sizes	bszx	4	4	4	4	4	4	8	8	8
	bszy	4	4	4	4	4	4	32	32	32
Full block unrolling		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Tile sizes	tW0	16	16	16	32	32	32	32	64	128
Innermost loop unroll factor		16	16	16	N/U	N/U	N/U	N/U	N/U	N/U
Local memory usage		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE

Table 4.16: Optimization parameters set for MATMUL

Platform	Nvidia			AMD			CPU		
Size class	S	M	L	S	M	L	S	M	L
Naive kernel time (ms)	244.48	3297	26364	226.45	2302	23737	351	16521	216742
Optimized kernel time (ms)	29.04	238	1915	18.19	156	1872	113	1011	7933
Speedup	8.42	13.86	13.77	12.45	14.82	12.68	3.09	16.35	27.32
Code generation time (ms)	11.26	11.29	11.29	4.17	4.14	4.14	32.23	32.67	31.80
OpenCL compilation time (ms)	3.42	3.57	3.91	159	157	164	10803	10806	10821

Table 4.17: Performance results obtained for MATMUL

cal memory when this feature is exploited. The values selected for the optimization parameters in each test case of this kernel are detailed in Table 4.18, whereas Table 4.19 shows the performance results. In order to compute $A \times A^T$, each instance of the naive implementation of the kernel must access simultaneously two rows of A. The threads that compute a diagonal position just access a single row of A, but the more the position computed by a given thread moves away from the diagonal, the more the stride between the pair of rows of A read increases. Such reads from global memory lead to quite unfriendly situations for GPUs, to the extent that the naive kernel performs considerably worse in both GPUs than in the CPU. As commented, when the naive kernel accesses columns of A^T, it is accessing rows of A indeed, but these accesses do not follow a 2D `RowPat` pattern exactly in the same terms as the optimizer initially expects. Namely, in a pure 2D `RowPat` pattern the row dimension must be indexed using the `idy` global thread identifier, whereas in this case it is being indexed using the `idx` one. However, the optimizer is also able to detect such pattern variations and generate the code needed to copy to local memory an additional slice gathering the positions of A read by these accesses. Thanks to this extended caching mechanism, no global memory reads are performed by the inner

Platform		Nvidia			AMD			CPU		
Size class		S	M	L	S	M	L	S	M	L
Global sizes	szx	512	1024	2048	512	1024	1024	256	512	1024
	szy	512	1024	2048	512	1024	1024	2048	4096	8192
Local sizes	lszx	16	16	16	16	16	16	32	32	32
	lszy	16	16	16	16	16	16	32	32	32
Block sizes	bszx	4	4	4	4	4	4	8	8	8
	bszy	4	4	4	4	4	4	1	1	1
Full block unrolling		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Tile sizes	tW0	32	32	32	64	64	64	512	1024	2048
Innermost loop unroll factor		32	32	32	64	64	64	8	8	8
Local memory usage		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE

Table 4.18: Optimization parameters set for SYRK

Platform	Nvidia			AMD			CPU		
Size class	S	M	L	S	M	L	S	M	L
Naive kernel time (ms)	2636	28463	379550	4096	33475	291594	444	3630	31035
Optimized kernel time (ms)	103	829	6637	51	663	6134	144	1149	12306
Speedup	25.55	34.32	57.19	80.30	50.46	47.54	3.08	3.16	2.52
Code generation time (ms)	22.14	22.31	22.44	58.04	58.84	59.50	4.35	4.59	4.53
OpenCL compilation time (ms)	12	12	36	994	1013	1049	177	177	175

Table 4.19: Performance results obtained for SYRK

loop that computes $A \times A^T$, and thus considerable speedups are achieved in both GPUs. In relation to the code generation process, notice how, for the test cases in both GPUs, the complexity of some of the transformations applied, mainly the combination of a 4×4 block size and a high unroll factor of the innermost compute loop, leads the optimizer to consume higher times to generate these kernels.

The SYR2K kernel performs a symmetric rank two update $C = \alpha AB^T + \alpha BA^T + \beta C$. The optimizer detects a 2D `RowPat` access twice, first for A when computing $A \times B^T$, and also for B when $B \times A^T$ is computed. Thus, both matrices are selected to be cached in local memory. The values assigned to the optimization parameters in each test case of this kernel are detailed in Table 4.20, whereas Table 4.21 shows the performance results obtained. In order to perform $A \times B^T$ and $B \times A^T$, each instance of the naive implementation must perform strided accesses to a pair of rows from both input matrices. This gives place to GPU-unfriendly situations quite similar to those we have just described for the SYRK naive kernel. In this case, the optimizer detects the transposed variations of the 2D `RowPat` patterns arisen when the kernel reads A^T and B^T, and thus the code needed to cache two additional slices for the positions of A and B read by these transposed accesses is generated. Since both the

Platform		Nvidia			AMD			CPU		
Size class		S	M	L	S	M	L	S	M	L
Global sizes	szx	512	1024	2048	512	1024	2048	256	512	1024
	szy	512	1024	2048	512	1024	2048	2048	4096	8192
Local sizes	lszx	16	16	16	16	16	16	32	32	32
	lszy	16	16	16	16	16	16	32	32	32
Block sizes	bszx	4	4	4	4	4	4	8	8	8
	bszy	4	4	4	4	4	4	1	1	1
Full block unrolling		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Tile sizes	tW0	32	32	32	32	32	32	64	128	256
Innermost loop unroll factor		32	32	32	32	32	32	8	8	8
Local memory usage		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE

Table 4.20: Optimization parameters set for SYR2K

Platform	Nvidia			AMD			CPU		
Size class	S	M	L	S	M	L	S	M	L
Naive kernel time (ms)	10815	78807	756930	9494	92087	878013	968	11122	99757
Optimized kernel time (ms)	316	2551	20353	114	1554	16238	364	3399	36855
Speedup	34.21	30.89	37.19	82.98	59.27	54.07	2.66	3.27	2.71
Code generation time (ms)	38.39	39.09	39.00	55.35	54.88	54.48	7.10	7.43	7.39
OpenCL compilation time (ms)	12	13	33	1519	1444	1533	246	246	246

Table 4.21: Performance results obtained for SYR2K

global memory issues and the solution applied to them are similar than those from the SYRK kernel, the performance obtained by the versions of the current SYR2K problem optimized for GPUs are also similar.

4.5. Conclusions

In Chapter 3 we described a mechanism to program performance-portable HPL kernels that exploit the run-time code generation capabilities of the library. These codes are able to generate several versions depending on the values given to a number of optimization parameters. However, writing kernels in such a way requires a considerable programming effort. Furthermore, the search algorithms needed to find proper values for those parameters on a target device were found to be too time-consuming to some extent. Thus, in this chapter we present a just-in-time approach that intends to overcome these inconveniences.

This just-in-time optimizer is embedded in the workflow of HPL, which implies relevant modifications in the way the library translates the kernels written in its C++-based language into working OpenCL codes. Originally these translation process was performed directly at run-time, whereas now the kernels are loaded in an AST representation. Once the tree is ready, the optimizer can transform it by applying common strategies to tune codes for heterogeneous devices. Finally, an optimized OpenCL kernel is generated from the transformed AST. Namely, the optimizer is able to tile and unroll the compute loops, to cache shared data on the local memory of the devices when available, to coarsen the granularity of the naive kernel, and to reduce memory access contention by computing results in the private memory regions of the devices. Along their implementation, all these transformations performed on the naive syntax tree emerged as interdependent. Moreover, this pool of transformations is not usually applied as a whole, rather some optimizations are more suitable than others depending on the device properties and the problem characteristics. These two sets of constraints determined the order fixed in the workflow to perform them. A set of parameters is defined to drive the optimization process, and the values given to them determine which transformations are going to be applied and under which conditions. These parameters decide the sizes of the global and the local workspaces, the tile widths for inner computing loops and an unroll

factor for the innermost one, whether local memory is going to be exploited, the private memory block size, and whether the computing performed over that region must be fully unrolled. By setting different values for these parameters, the library generates different versions from a same input code. Thus, when these values are chosen taking into account the capabilities of a device, the version generated will be optimized for it. By now, these values are fixed heuristically, trying to adapt those general guidelines to optimize codes for heterogeneous platforms to the concrete properties of each target device.

Naive implementations of eight different problems have been optimized for three platforms: an Nvidia GPU, an AMD GPU and a multicore Intel CPU. Each naive implementation has been tested in each platform for three different problem sizes: small, medium and large. The optimizer generated versions with speedups from 1.83 up to 57.19 in the Nvidia GPU, from 1.21 to 82.98 in the AMD GPU and up to 27.32 in the Intel CPU. In this latter device, the optimizer was not able to generate versions faster than the baseline for the DCS3D problem. Moreover, the optimization process showed to be quite lightweight, requiring just between 1 and 59 milliseconds to transform naive HPL kernels into OpenCL optimized codes. Such code generation times can be considered negligible, specially for large test cases. In turn, in some cases the OpenCL runtimes of both the AMD GPU and the Intel CPU needed compilation times which can be several orders of magnitude greater than the kernel times of the corresponding optimized versions, probably due to thorough code analysis tasks being performed by the compilers on these codes. In relation to this issue, we remind that since HPL stores the evaluated kernels in an internal cache, once an optimized version is obtained, it can be reused without having to generate and compile it again.

In the future we plan to implement some kind of algorithm able to select the optimization values by itself, which will automatically provide the performance portability. There are several sources from which knowledge for that algorithm could be extracted and then encoded, such as heuristics based on the bibliography and results obtained in prior experiments, micro-benchmarking, or analytical performance models. This work can be also extended by enriching the optimization pool, both making the current transformations more generic and by implementing other well-known techniques, like explicit loop vectorization or enabling local memory

exploitation to compute intermediate results.

4.6. Related work

As the name itself suggests, the main purpose of implementing just-in-time optimization capabilities in any programming solution is to reduce the time invested in the generation of optimized versions. Multiple strategies are usually applied to overcome this problem, ranging from previous processes of platform profiling to complex algorithms that try to replicate the human expertise about code optimization. Moreover, these programming frameworks also try to reduce as much as possible the intervention of users in the optimization process. In a best-case scenario, the framework can be simply fed with the code to optimize, but it is very common to ask the users to give, at least, some light indications to guide the optimization process. Also, as a consequence of the addition of such capabilities, it is also common that these frameworks provide users with some kind of high-level programming interface to interact, instead of having to write complex *host* codes like those of OpenCL.

This way, some approaches work as black boxes that just expect programs originally written in classic high-level languages like C, C++ or FORTRAN. This is the case of a multi-objective auto-tuning framework developed by Jordan et al. [52] on top of the Insieme [86] compiler infrastructure. This framework receives programs written in C, C++, OpenMP, MPI, and OpenCL as inputs, and loads them into an intermediate representation provided by the Insieme compiler. Then the code is analyzed, which yields as outcomes both a number of regions susceptible to be optimized and a set of feasible transformations to apply. As in our just-in-time optimizer, these transformations depend on multiple parameters such as unroll factors, tile sizes or the number of iterations to distribute among the threads. An iterative algorithm based on evolutionary methods and search space pruning mechanisms were purposely designed for this framework in order to find tuned configurations for those parameters. Code versions are evaluated by running them on the target device during the search process, which is performed just-in-time as a part of the program compilation. The result is a number of configurations that are translated by the compiler back-end into optimized code regions written in C, OpenCL or MPI code, depending on the input. These specialized code regions are bundled into a

multi-version executable, although the decision about the combination of regions to run remains application-specific and could be forwarded to the user if needed.

Other tools rely on the functional portability provided by OpenCL and then try to overcome its well-known performance portability gap by implementing code transformations able to optimize OpenCL kernels in multiple ways. For instance, Fang et al. propose Sesame [36], a performance-portable framework for OpenCL that gathers a number of techniques derived from a comprehensive systematic study on the optimization space for many-core devices performed by the authors. In that study they evaluate the impact that a proper usage of the vector capabilities of processors [33] or the local memory hardware usually included in many-core architectures [32] have in the performance of OpenCL kernels. The memory access pattern classification implemented in our just-in-time optimizer is based in this latter study. Regarding the proper exploitation of local memory, they propose two tools that complement each other: one enables local memory usage in OpenCL kernels [34], whereas the other is able to rewrite OpenCL codes that already used local memory in a quite architecture- or device-specific way [31]. The OpenCL code analysis and transformation operations performed by these tools are implemented by means of LLVM and Clang. One of the future research directions they proposed to effectively implement such a framework is to find a generic order to apply optimizations. The workflow followed by our just-in-time optimizer tackles this issue.

By design, any heterogeneous programming framework that, like HPL, provides its own high-level kernel language, must ask users to translate their codes into it. Hence, the efforts made to alleviate the user intervention are commonly focused on the optimization stages, either by taking charge of the whole process or, at least, guiding the programmers along a feasible optimization path and helping them to transform the input code. An example of this is Many-Core Levels (MCL), a framework oriented to different kinds of many-core devices that was built by Hijma et al. as an implementation of their *stepwise-refinement for performance* methodology [46]. It is composed of the Many-Core Programming Language (MCPL), which is an embedded language to write kernels, and a compiler able to improve these kernels with optimizations with different levels of abstraction. These optimizations range from general transformations usually applicable on many-cores to quite architecture-specific tweaks, and they are presented to the user along an iterative process from

the more general to the more specific. Thus, their compiler first proposes some general optimizations and provides feedback about the potential performance they can yield. Then, the programmer takes a decision according to that feedback and commands the compiler to optimize the code. Depending on the transformations applied and the capabilities of the device, a different set of more specific optimizations is presented. As the abstraction level of the optimizations decreases, the framework might discharge on the programmer the transformation of the MCL kernel. At the end of the process, this kernel is translated into OpenCL or C++ source code. Thus, starting with a naive input kernel, this framework is able to follow different optimization paths depending on the properties of both the problem and the device, but user feedback is needed to some extent in each step forward. Our just-in-time optimizer, in turn, only requires users to intervene at the beginning of the process, since they must indicate the *compute* section in their naive input HPL kernels and, by now, to perform manually a fine adjustment the optimization parameters.

Steuwer et al. propose in [103] a high-level functional language embedded in Scala to implement simple problem descriptions. This forces the programmers not only to rewrite their kernels but also to leap from the imperative to the functional paradigm, which may result uncomfortable for the most inexperienced users. In their framework, the authors include a set of rewrite rules based on λ-calculus that must be properly combined and applied to optimize each input code. After that, an optimized low-level expression composed of several primitives is obtained. These primitives are mapped to parametrized routines that generate OpenCL snippets that implement the operations the primitives represent. The parameter values passed to those routines are used to specify OpenCL low-level details such as the global and local workspaces sizes or vector lengths. Thus, at the end of the evaluation process, an OpenCL code optimized according to the rules applied and the parameter values passed to the primitives is obtained. Three consecutive exploration processes are needed to generate optimized versions from an input code. First, search space of general optimization rules is heuristically pruned by selecting just those that are expected to perform well. Then, another heuristic is applied to prune the options to implement such rules in OpenCL. Finally, the values for OpenCL primitive parameters are found by pruning again the search space and generating all the remaining kernel versions. The authors followed this approach to implement a performance-portable matrix multiplication, executing exhaustively all the generated OpenCL

versions to test it on CPUs and on several desktop [92] and mobile GPUs [107].

There are also domain-specific programming frameworks that provide just-in-time optimization capabilities. HALIDE [91] is a domain-specific language for image processing that is built on top of C++. Halide programmers must describe a high-level strategy to map image processing pipelined applications to heterogeneous platforms. The compiler provided by the framework is in charge of generating the code that implements that strategy, OpenCL being one of the supported back-ends for GPU code. The process is driven by an iterative auto-tuner that performs a genetic search to find an optimized schedule for a given user strategy. The configuration of each schedule defines the parameter values for the optimizations, which include code transformations such as work distribution, vectorization or loop unrolling. In [73], the HALIDE development team presents a just-in-time version of the original algorithm that encodes both a refined mechanism of function bounds analysis for the parameters as well as additional human expertise.

Chapter 5

Conclusions

Heterogeneous computing platforms are ubiquitous nowadays. Such platforms consist of different kinds of parallel devices, each including multiple processing elements that sometimes are of different nature. Architectural differences among these devices are related not only to the capabilities of their processing elements, but also to another details like the memory hierarchy organization. Some of these architectures, like the x86-based multicore processors, intend to provide backwards compatibility in relation to prior designs. Others propose novel approaches to extract parallelism from existing devices, like the current GPUs, which became general-purpose computing devices. There are also some proposals that try to take advantage of the best features of both worlds. This way, devices such as the Intel Xeon Phi or the Accelerated Processing Units (APUs) from AMD are able to run code originally thought for traditional CPUs, but providing at the same time the massively data parallel capabilities of GPUs. Such a variety of architectural designs from different vendors gave place to multiple solutions to program these devices. Some of these solutions were specifically devoted to a single device type from a particular vendor, such as CUDA, created by NVIDIA to exploit the parallel capabilities of its GPUs. Other solutions, however, tried to cover a wider range of device types, which turned them into real heterogeneous programming frameworks. For example, the OpenMP standard was born as a directive-based approach to extract parallelism in shared memory systems. However, since its 4.0 version it can be also used to decompose an existing program in multiple tasks and offload them onto the several devices

available in any heterogeneous system. The OpenACC initiative also followed this directive approach to provide a similar solution.

Both OpenMP 4.0 and OpenACC offer code portability as far as the user-annotated code remains the same no matter the devices on which it will be run. However, the directives added on top of the code must be modified in order to choose the target devices and to adapt the work decomposition to the underlying platform. In this context, the OpenCL standard proposes a different programming model based on kernels that can be run in any device supporting the standard, and a host code that defines the environment to run them. These host programs are written by means of an API that exposes some low-level details that may be a bit cumbersome for inexperienced users. Such a trade-off leads to the eruption of multiple proposals to program heterogeneous systems that are built on top of OpenCL, some of them also trying to improve its programmability by hiding these details to the users. An example of the latter is the Heterogeneous Programming Library (HPL), on which some of the tools developed in this thesis are based.

There is no doubt that functional portability is one of the strengths of OpenCL but, unfortunately, architectures and capabilities differ among devices, so that an OpenCL program is often not performance-portable straightaway. However, there are some general strategies that can be followed to tune an OpenCL kernel for different devices: maximize the number of running threads, optimize the usage of the several levels of the memory hierarchy, and reveal situations that may lead to the extraction of instruction-level parallelism. There are multiple parametrized transformations that, when applied to codes, make them follow these strategies. The parameter values will vary depending on the device targeted. The tools developed in the context of this PhD thesis generate optimized OpenCL codes for several kinds of devices by applying such parametrized techniques and implementing multiple search procedures to set proper values for those parameters.

The rest of this chapter is devoted firstly to reminding how the tools developed implement these approaches and, thus, how they contribute to improve performance portability in heterogeneous systems, and then to describing some feasible future research directions for each one.

OCLOptimizer

The first solution we describe is OCLoptimizer, a source-to-source optimizer for OpenCL codes built on top of the LLVM-Clang compiler infrastructure. The inputs of the optimizer are a configuration file and a kernel annotated with indications on the optimizations to try. The outputs are a host code suitable for the input kernel, an optimized workspace configuration and a kernel properly tuned for the platform where the tool is executed. The tool implements search processes driven by the execution times of kernels to find both the optimized workspaces and the optimized kernel versions. The search of an optimized workspace can be either exhaustive, taking into account the whole set of legal combinations of parameters, or guided by a genetic algorithm (GA). Regarding the kernels, users can annotate the loops in their kernels to try different factors to unroll them. Depending on the number of loops annotated and the range of factors set for each loop, a combinatorial explosion can occur when generating the search space for the kernel optimization process. Because of that, the exhaustive algorithm followed to search an optimized workspace is replaced here with a breadth-first search (BFS) that processes the directives one by one. A genetic search was implemented also, each individual tested by the algorithm being a combination of unroll factors for all the directives. The code analysis and transformation operations needed to optimize kernels are built on top of the LLVM-Clang compiling infrastructure.

The tool is also able to optimize applications composed of several kernels, although in this case the workflow it follows varies depending on how the kernels are related each other. There are applications whose kernels are totally independent and they can be run in workspaces with different configurations. In such cases, the tool launches as many independent optimization processes as kernels are included in the application. Because of that, the user has to write each kernel in its own file and extract from the host code of the original application all the information needed to configure each process. Since several independent optimization process are launched, the outputs are the respective optimized workspace configurations and kernel versions for each input code. The user must merge back the optimized kernels into the application, and also modify it to apply the optimized workspace configurations. Nevertheless, when the kernels in the application are interdependent to some extent, it is quite usual that kernels must run on a common workspace. Moreover,

sometimes these applications are built in such a way that a given kernel produces intermediate results that must be used as inputs by another one. In such cases, the workflow of the tool is divided into three stages. First, an optimized workspace configuration is obtained for each kernel separately. One of the constraints being that all the kernels must run on the same workspace, all these workspace configurations are valid candidates to be selected as the optimized one. Thus, in a second step, optimized versions of all the kernels are generated using each workspace configuration candidate. Finally, the fastest combination of a candidate workspace configuration and their corresponding optimized kernels is returned as the output of the whole application optimization process. Again, the user is required to merge back the optimized kernels into the application and also to modify it in order to apply the optimized workspace configuration selected.

The performance of the tool was evaluated in a CPU, a GPU and an Intel Xeon Phi accelerator. This validation showed that OCLoptimizer successfully tunes single and multiple kernel OpenCL codes for the different platforms. Regarding single kernel OpenCL codes, the achieved speedup was 2.22 when using the GA in the workspace and kernel code search processes and 2.86 when using ES+BFS, but the searches guided by the GA were about ten times faster than those guided by ES+BFS. Focusing on ES+BFS, the average speedups achieved were respectively 1.59, 2.54 and 4.46 for CPU, GPU and Xeon Phi, which shows that all the platforms benefit from the usage of the tool. Support for codes composed of several kernels was validated using the SNU NPB IS benchmark in CPU and GPU, achieving speedups of 2.45 for CPU and 1.19 for GPU. No tests were run for the Xeon Phi as the SNU NPB suite does not have an implementation specially optimized for this device.

Self-adaptive HPL kernels

This is the first solution of the two based on the Heterogeneous Programming Library (HPL) developed in this PhD thesis. Namely, this tool follows a generic programming approach to implement self-optimizing HPL codes by exploiting the run-time code generation capabilities of the library. A remarkable result of this work is the description of a collection of techniques to generate performance-portable versions of an HPL code, and a set of parameters that drive the application of such techniques.

This way, kernels with different work granularities can be generated by giving values to global and local workspaces and iteration block sizes. Loops can be tiled with a given tile size or unrolled to some factor, and if they are nested, different orders and schedules can be tried for their instructions. Moreover, the generation of different snippets of code can be driven by any given condition. For instance, different algorithms can be used depending on the type of the target device, or different data structures can be chosen to be cached in slices in local memory. In this last situation the code selection mechanism would be used several times, since multiple code snippets would have to be toggled or replaced along the kernel. This is the case of the declarations of local memory structures, the copy of slices from global to local memory or the new accesses to local memory replacing the old ones.

We used these parametrized optimizations as building blocks to write a generic HPL kernel tunable by means of a dozen of parameters for a matrix multiplication. The search of best values for these parameters is driven by a genetic algorithm configured in a way similar to that of OCLoptimizer. Due to legality reasons and also in order to narrow the search space, different ranges and inter-dependent conditions were set for the values that these parameters can take. By properly choosing these constraints, the time consumed by the search algorithm decreases considerably with no loss in the quality of the optimized version generated. This strategy of programming self-adaptive kernels by means of the HPL embedded language is feasibly adaptable to other problems different from the matrix multiplication.

The performance of this use case was evaluated in an NVIDIA GPU, an AMD GPU, a multicore Intel CPU and an Intel Xeon Phi accelerator, and it was compared to two state-of-the-art OpenCL adaptive implementations of the matrix product, namely, clBLAS and ViennaCL. The average speedup of our implementation was 1.74 respect to clBLAS, and 1.44 respect to ViennaCL. In terms of search time, our genetic algorithm was on average 1.18 times faster than the clBLAS profiling, and 160 times faster than the exhaustive search performed by ViennaCL.

HPL-embedded just-in-time optimizer

The implementation of self-adaptive kernels by means of the run-time code generation capabilities of HPL, in the terms it is proposed in Chapter 3, may result

quite low-level and verbose for users with elementary programming skills. Thus, the just-in-time optimizer for HPL was designed having them mind. Since the framework is built on top of OpenCL, both the programming and the execution of HPL kernels are also defined in terms of work-items running in a workspace. To exploit this tool, users have to map a problem to such a workspace and write a naive HPL kernel that computes a single point of it. The optimizer, which is embedded into the workflow of the library, analyses the code of the input kernel and generates just-in-time an optimized version for a target device. This approach simplifies considerably the task of programming a self-adaptive HPL kernel, since now a naive implementation is able to eventually give place to multiple versions.

The analysis and transformation tasks involved are performed on an abstract syntax tree (AST) on which the input kernel is previously loaded. This forced to modify the original OpenCL code generation mechanism of HPL in order to store the components of the parsed expressions in the nodes of our own tree-shaped composite class hierarchy, instead of translating them directly into their OpenCL equivalent string. Each node is required to implement a method that generates its OpenCL equivalent. Thus, if such a method is invoked for the root node, eventually the whole tree is recursively translated into a full OpenCL kernel. Once generated, this intermediate representation is enriched with information about the different memory access patterns followed by the kernel.

This design allowed to implement several optimization techniques as tree transformations. These techniques were defined according to well-known optimization strategies for heterogeneous devices. Namely, the optimizer is able to tile and unroll to some extent the compute loops, to cache shared data in the local memory of the devices when available, to coarsen the granularity of the naive kernel, and to reduce memory access contention by computing results in the private memory of the devices. Since they are inter-dependent, these transformations are applied following an order that was experimentally fixed. Moreover, all of them depend on a set of parameters. The values given to those parameters decide which transformations and under which conditions they are going to be applied. Namely, these parameters are used to determine the sizes of workspaces, the tile widths for the inner computing loops, the unroll factor for the innermost one, whether local memory is going to be exploited, the private memory block size, and whether the space for this block is

allocated as an array or as a set of private variables. When this latter allocation method is chosen, the computation performed in this private memory space is also fully unrolled. By setting different values, the library generates different versions from a same input code.

When the parameter values are chosen taking into account the capabilities of a device, the version generated will be tuned to some extent for it. By now, the values are heuristically fixed trying to map the aforementioned optimization guidelines to the properties of each target device. Naive implementations of eight different problems were optimized for three platforms, an Nvidia GPU, an AMD GPU and a multicore Intel CPU. The optimizer generated versions with speedups from 1.83 up to 57.19 in the Nvidia GPU, from 1.21 to 82.98 in the AMD GPU and up to 27.32 in the Intel CPU. In this latter device, the optimizer was not able to generate versions faster than the baseline for the DCS3D problem. Moreover, the optimization process showed to be quite lightweight, requiring just between 1 and 59 milliseconds to transform naive HPL kernels into OpenCL optimized codes. This cost is further amortized across several kernel executions, as it must be noted that HPL stores the kernels generated in an internal cache. Thus, once an optimized version is obtained in an application, it can be reused without generating it again. These results show that, by means of a set of heuristically driven parametrized transformations, the embedded optimizer is able to take HPL naive kernels and generate, automatically and just-in-time, OpenCL versions of them tuned for different platforms.

5.1. Future work

The three tools developed in this PhD thesis have been proven to provide performance portability both to OpenCL and to HPL, since they were able to take input codes and generate tuned versions of them for different platforms. In all cases, the optimization processes consisted in applying parametrized techniques, the search of optimized values for the associated parameters being one of the most challenging issues addressed. The strategies implemented in OCLoptimizer and in the current approach to the self-adaptive HPL kernels explore the search space of these parameter values by generating and executing the codes of the corresponding versions, which makes them very-time consuming. This condition is particularly severe for

the exhaustive methods, since they visit all the search space, but it may be also troublesome for the informed ones, namely in their first steps, because these earlier kernels usually have quite long execution times. Thus, an interesting future research work for both tools would be the study of alternative ways to evaluate versions either with no real code execution, such as inferring their performance by means of analytical models, or by performing shortened runs from which a performance profile could be extracted. In relation to the the heuristics proposed for the HPL just-in-time optimizer, they follow some well-known recommendations for optimizing codes for heterogeneous devices that are effective to some extent, but they also have the drawback of being too general. Thus, one of the main future research efforts in relation to this tool will be devoted to improve these heuristics thoroughly.

The parametrized optimization techniques on which the matrix multiplication self-adaptive kernel is based were implemented using the run-time code generation (RTCG) capabilities of HPL. Namely, in this first approach the kernel was made self-adaptive by including on it verbose HPL code snippets that exploit such capabilities to generate different versions in run-time. Such low-level implementations of the parametrized optimization techniques could be encapsulated into C++ classes which offer the programmer a much more affordable high-level semantics and syntax. These classes would implement both optimization abstractions to make the incorporation of the techniques in the kernels easier, and search abstractions providing an interface to define optimization parameters and to choose among different search algorithms to find tuned values for these parameters. Furthermore, looking for more candidate problems from other domains to follow this approach would reveal a couple of future research directions. First, the study of a wider range of problems will likely lead us to identify more optimizations that might be implemented in HPL using the RTCG approach. Some of the new optimizations that might be found in this study could be also implemented for OCLoptimizer. Second, many problems are usually solved by means of iterative pipelined algorithms, each step of them being implemented with a single kernel. The possibility of embedding some kind of online optimization training arises in these cases, as different combinations of values for the parameters could be tested and refined during the iterations of the algorithm.

Regarding the set of techniques applied by the HPL-embedded just-in-time optimizer, it could grow by deriving new optimizations from others already included

on it. For instance, the loop vectorization could be implemented on top of the unrolling technique. Some techniques already implemented can be extended too. For example, the mechanism that selects the structures to be sliced and copied in local memory could be enhanced to take into account the local memory space available in the device. Depending on the memory access pattern detected on the structures copied, the addition of this constraint could imply variations in the sizes of the local workspace and the iterations block or in the compute loop tile width, among other parameters. This may help to find a balanced application of the affected techniques that maximizes the overall benefit obtained from all of them. Furthermore, as for OCLoptimizer and the RTCG approach, the study of a wider range of problems will likely lead us also to identify more optimizations to enrich the set of techniques included in the just-in-time optimizer.

Bibliography

[1] S. Aarseth. *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge Monographs on Mathematical Physics. Cambridge University Press, 2003.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[3] Altera Corporation. Press release: Altera Announces Industry's First OpenCL Program for FPGAs. `https://www.altera.com/about/news_room/releases/_2011/products/nr-opencl.html`, November 2011. [Online; accessed 19-December-2016].

[4] AMD. *AMD Stream Computing User Guide*, December 2008.

[5] AMD. Press Release: AMD Releases New Version of ROCm, the Most Versatile Open Source Platform for GPU Computing. `http://www.amd.com/en-us/press-releases/Pages/rocm-platform-2016nov14.aspx`, Nov 2016.

[6] AMD and Accelereyes. ACL - AMD Compute Libraries - clMath. `https://github.com/clMathLibraries`. [Online; accessed 20-December-2016].

[7] AMD Developer Central. APP SDK. A Complete Development Platform. `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/`. [Online; accessed 19-December-2016].

[8] ARM Ltd. ARM Mali GPU OpenCL Version 3.0 Developer Guide. `http://infocenter.arm.com/help/topic/com.arm.doc.100614_0300_00_en/arm_mali_gpu_opencl_developer_guide_100614_0300_00_en.pdf`, April 2016. [Online; accessed 19-December-2016].

[9] P. J. Ashenden. *The Designer's Guide to VHDL, Volume 3, Third Edition (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2008.

[10] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[11] O. Beckmann, A. Houghton, M. Mellor, and P. H. J. Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 291–306. Springer Verlag, 2004.

[12] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[13] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with ompss. In *Euro-Par 2011 Parallel Processing: 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part I*, pages 555–566, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[14] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. Productive programming of gpu clusters with ompss. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 557–568, May 2012.

[15] C. Cao, J. Dongarra, P. Du, M. Gates, P. Luszczek, and S. Tomov. clMAGMA: High performance dense linear algebra with OpenCL. In *International Workshop on OpenCL (IWOCL)*, pages 13–14, 2013.

[16] J. Cavanagh. *Digital Design and Verilog HDL Fundamentals.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2008.

[17] N. Chaimov, B. Norris, and A. Malony. Toward multi-target autotuning for accelerators. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 534–541, Dec 2014.

[18] C.J. Newburn and B. So and Z. Liu and M.D. McCool and A.M. Ghuloum and S. Du Toit and Z-G. Wang and Z. Du and Y. Chen and G. Wu and P. Guo and Z. Liu and D. Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *9th IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO 2011)*, pages 224–235, 2011.

[19] clBLAS. `https://github.com/clMathLibraries/clBLAS`, 2015. [Online; accessed 16-February-2017].

[20] K. Daloukas, C. D. Antonopoulos, and N. Bellas. GLOpenCL: OpenCL support on hardware- and software-managed cache multicores. In *Proce. 6th Intl. Conf. on High Performance and Embedded Architectures and Compilers*, pages 15–24, 2011.

[21] U. Dastgeer, J. Enmyren, and C. W. Kessler. Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems. In *Proc. 4th Intl. Workshop on Multicore Software Engineering*, IWMSE '11, pages 25–32, 2011.

[22] R. Dolbeau, F. Bodin, and C. de Verdiere. One OpenCL to rule them all?, 2013.

[23] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, Aug 2012.

[24] W. Engel. *ShaderX2: Introductions & Tutorials with DirectX 9.* Wordware Pub., 2004.

[25] J. Enmyren and C. W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth Interna-*

tional Workshop on High-level Parallel Programming and Applications, pages 5–14, 2010.

[26] J. F. Fabeiro, D. Andrade, and B. B. Fraguela. OCLoptimizer: An iterative optimization tool for OpenCL. In *Proc. Intl. Conf. on Computational Science (ICCS 2013)*, pages 1322–1331, 2013.

[27] J. F. Fabeiro, D. Andrade, and B. B. Fraguela. Writing a performance-portable matrix multiplication. *Parallel Computing*, 52:65–77, 2016.

[28] J. F. Fabeiro, D. Andrade, B. B. Fraguela, and R. Doallo. Writing self-adaptive codes for heterogeneous systems. In *Proc. 20th Intl. Conf. Euro-Par 2014 Parallel Processing*, pages 800–811, 2014.

[29] J. F. Fabeiro, D. Andrade, B. B. Fraguela, and R. Doallo. Automatic generation of optimized OpenCL codes using OCLoptimizer. *The Computer Journal*, 58(11):3057–3073, Nov 2015.

[30] J. F. Fabeiro, D. Andrade, B. B. Fraguela, and R. Doallo. How to Write Performance Portable Codes using the Heterogeneous Programming Library. In *19th Workshop on Compilers for Parallel Computing*, CPC 2016, July 2016.

[31] J. Fang, H. Sips, P. Jaaskelainen, and A. L. Varbanescu. Grover: Looking for performance improvement by disabling local memory usage in opencl kernels. In *2014 43rd International Conference on Parallel Processing*, pages 162–171, Sept 2014.

[32] J. Fang, H. J. Sips, and A. L. Varbanescu. Aristotle: A performance impact indicator for the OpenCL kernels using local memory. *Scientific Programming*, 22(3):239–257, 2014.

[33] J. Fang, A. L. Varbanescu, X. Liao, and H. J. Sips. Evaluating vector data type usage in OpenCL kernels. *Concurrency and Computation: Practice and Experience*, 27(17):4586–4602, 2015.

[34] J. Fang, A. L. Varbanescu, J. Shen, and H. Sips. ELMO: A User-Friendly API to Enable Local Memory in OpenCL Kernels. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2013.

[35] J. Fang, A. L. Varbanescu, and H. Sips. An Auto-tuning Solution to Data Streams Clustering in OpenCL. In *Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on*, Aug 2011.

[36] J. Fang, A. L. Varbanescu, and H. Sips. Sesame: A User-Transparent Optimizing Framework for Many-Core Processors. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 70–73, May 2013.

[37] B. B. Fraguela, M. G. Carmueja, and D. Andrade. Optimal Tile Size Selection Guided by Analytical Models. In *Parallel Computing (ParCo)*, pages 565–572, 2005.

[38] B. B. Fraguela, Y. Voronenko, and M. Püschel. Automatic Tuning of Discrete Fourier Transforms Driven by Analytical Modeling. In *Proc. 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT '09)*, pages 271–280, 2009.

[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[40] B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[41] P. Gerald Tennyson, G. Karthik, and G. Phanikumar. MPI+OpenCL implementation of a phase-field method incorporating CALPHAD description of Gibbs energies on heterogeneous computing platforms. *Computer Physics Communications*, 186:48–64, 2015.

[42] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[43] S. Haney, J. Crotinger, S. Karmesin, and S. Smith. PETE: The Portable Expression Template Engine, 1999.

[44] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–11, May 2009.

[45] HGPU Group. OpenCL resources in `hgpu.org`. `http://hgpu.org/?tag=opencl`. [Online; accessed 20-December-2016].

[46] P. Hijma, R. V. van Nieuwpoort, C. J. H. Jacobs, and H. E. Bal. Stepwise-refinement for performance: a methodology for many-core programming. *Concurrency and Computation: Practice and Experience*, 27(17):4515–4554, 2015.

[47] HSA Foundation. HSA Platform System Architecture Specification, Version 1.1, January 2016.

[48] IBM, Sony, and Toshiba. *C/C++ Language Extensions for Cell Broadband Engine Architecture*. IBM, 2006.

[49] Intel Developer Zone. OpenCL Drivers and Runtimes for Intel Architecture. `https://software.intel.com/en-us/articles/opencl-drivers`, November 2016. [Online; accessed 19-December-2016].

[50] Intel Programmable Solutions Group. Intel FPGA SDK for OpenCL. `https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html`, November 2016. [Online; accessed 19-December-2016].

[51] Intel Software Development Tools. Intel SDK for OpenCL Applications 2012: OpenCL Optimization Guide. `https://software.intel.com/sites/landingpage/opencl/optimization-guide/`, 2012. [Online; accessed 24-January-2017].

[52] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. A Multi-objective Auto-tuning Framework for Parallel Codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 10:1–10:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[53] N. Kazakova, M. Margala, and N. Durdle. Sobel edge detection processor for a real-time volume rendering system. In *Circuits and Systems, 2004. ISCAS*

'04. Proceedings of the 2004 International Symposium on, pages II – 913–16 Vol.2, 2004.

[54] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL Shading Language, Version 1.10. `https://www.opengl.org/registry/doc/GLSLangSpec.Full.1.10.59.pdf`, April 2004. [Online; accessed 11-December-2016].

[55] Khronos OpenCL Working Group. The OpenCL Specification, Version 1.0. `https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf`, October 2009. [Online; accessed 19-December-2016].

[56] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 237–246, 2000.

[57] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[58] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating performance and portability of OpenCL programs. In *Proc. Fifth Intl. Workshop on Automatic Performance Tuning (iWAPT 2010)*, June 2010.

[59] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning GEMM kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, Nov 2012.

[60] Q. Lan, C. Xun, M. Wen, H. Su, L. Liu, and C. Zhang. Improving performance of GPU specific OpenCL program on CPUs. In *Proc. 13th Intl. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'12)*, pages 356–360, 2012.

[61] C. Lattner. LLVM. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications: Elegance, Evolution and a Few Fearless Hacks*. Creative Commons, 2011.

[62] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. 2004 Intl. Symp. on Code Generation and Optimization (CGO'04)*, pages 75–86, Palo Alto, California, Mar 2004.

[63] O. S. Lawlor. Embedding OpenCL in C++ for Expressive GPU Programming. In *Proceedings of 1st International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, May 2011.

[64] T. Lutz, C. Fensch, and M. Cole. PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems. *ACM Transactions on Architecture and Code Optimization*, 9(4):59:1–59:24, January 2013.

[65] A. Mametjanov, D. Lowell, C.-C. Ma, and B. Norris. Autotuning stencil-based computations on GPUs. In *Proc. 2012 IEEE Intl. Conf. on Cluster Computing*, pages 266–274, 2012.

[66] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Trans. Graph.*, 22(3):896–907, July 2003.

[67] R. Marques, H. Paulino, F. Alexandre, and P. D. Medeiros. Algorithmic skeleton framework for the orchestration of gpu computations. In *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, pages 874–885, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[68] K. Matsumoto, N. Nakasato, and S. Sedukhin. Implementing a code generator for fast matrix multiplication in OpenCL on the GPU. In *2012 IEEE 6th Intl. Symp. on Embedded Multicore Socs (MCSoC)*, pages 198–204, Sept 2012.

[69] K. Matsumoto, N. Nakasato, and S. Sedukhin. Performance tuning of matrix multiplication in OpenCL on different GPUs and CPUs. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 396–405, Nov 2012.

[70] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 1.0, June 1994.

[71] N. Moore, M. Leeser, and L. Smith King. VForce: An environment for portable applications on high performance systems with accelerators. *J. Parallel Distrib. Comput.*, 72(9):1144–1156, 2012.

[72] V. M. Morales, P. H. Horrein, A. Baghdadi, E. Hochapfel, and S. Vaton. Energy-efficient FPGA implementation for binomial option pricing using OpenCL. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.

[73] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016.

[74] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro. Nitro: A framework for adaptive code variant tuning. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 501–512, May 2014.

[75] T. Ngo, L. Snyder, and B. Chamberlain. Portable Performance of Data Parallel Languages. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pages 1–20, 1997.

[76] R. V. Nieuwpoort and J. W. Romein. Correlating radio astronomy signals with many-core hardware. *International Journal of Parallel Programming*, 39(1):88–114, 2011.

[77] M. J. Norton. *Spells of Fury: Building Windows 95 Games Using Directx 2.* Sams, Indianapolis, IN, USA, 1996.

[78] C. Nugteren and V. Codreanu. CLTune: A Generic Auto-Tuner for OpenCL Kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 195–202, Sept 2015.

[79] NVIDIA Corporation. NVIDIA Accelerated Computing: OpenCL. `https://developer.nvidia.com/opencl`. [Online; accessed 19-December-2016].

[80] NVIDIA Corporation. CUDA Compute Unified Device Architecture, Version 1.0. `http://developer.download.nvidia.com/compute/cuda/1.0/`

NVIDIA_CUDA_Programming_Guide_1.0.pdf, July 2008. [Online; accessed 11-December-2016].

[81] NVIDIA Corporation. *CUDA C Programming Guide*. NVIDIA Corporation, 2013.

[82] OpenACC-Standard.org. The OpenACC Application Programming Interface, Version 2.5. http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf, October 2015. [Online; accessed 9-December-2016].

[83] OpenMP Architecture Review Board. OpenMP Fortran Application Programming Interface, Version 1.0. http://www.openmp.org/wp-content/uploads/fspec10.pdf, October 1997. [Online; accessed 11-December-2016].

[84] OpenMP Architecture Review Board. OpenMP C and C++ Application Programming Interface, Version 1.0. http://www.openmp.org/wp-content/uploads/cspec10.pdf, October 1998. [Online; accessed 11-December-2016].

[85] OpenMP Architecture Review Board. OpenMP Application Programming Interface, Version 4.5. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, November 2015. [Online; accessed 12-December-2016].

[86] Parallel Systems Group, University of Innsbruck. Insieme Compiler and Runtime Infrastructure. http://insieme-compiler.org. [Online; accessed 6-February-2017].

[87] M. Peercy, M. Segal, and D. Gerstmann. A Performance-oriented Data Parallel Virtual Machine for GPUs. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[88] H. Perkins. DeepCL. https://github.com/hughperkins/DeepCL, 2016. [Online; accessed 19-December-2016].

[89] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *Proc. Intl. Symp. on Code Generation and Optimization*, pages 144–156, 2007.

[90] Qualcomm Technologies. Qualcomm Adreno OpenCL Programming Guide. https://developer.qualcomm.com/download/adrenosdk/

`adreno-opencl-programming-guide.pdf`, October 2016. [Online; accessed 19-December-2016].

[91] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, 2013.

[92] T. Remmelg, T. Lutz, M. Steuwer, and C. Dubach. Performance Portable GPU Code Generation for Matrix Multiplication. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, GPGPU '16, pages 22–31, 2016.

[93] R. Reyes, I. López, J. J. Fumero, and F. de Sande. accULL: An User-directed Approach to Heterogeneous Programming. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 654–661, July 2012.

[94] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In *International Workshop on GPUs and Scientific Applications*, pages 51–56, 2010.

[95] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification, Version 1.0. `https://www.opengl.org/registry/doc/glspec10.pdf`, July 1994. [Online; accessed 11-December-2016].

[96] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 137–148, Nov 2011.

[97] H. Shan, S. Williams, W. de Jong, and L. Oliker. Thread-level Parallelization and Optimization of NWChem for the Intel MIC Architecture. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 58–67, New York, NY, USA, 2015. ACM.

[98] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. An application-centric evaluation of OpenCL on multi-core CPUs. *Parallel Computing*, 39(12):834 – 850, 2013.

[99] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance traps in OpenCL for CPUs. In *Proc. 21st Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP 2013)*, pages 38–45, 2013.

[100] S. S. Shende and A. D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, May 2006.

[101] G. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford applied mathematics and computing science series. Clarendon Press, 1985.

[102] T. Sorensen and A. F. Donaldson. The Hitchhiker's Guide to Cross-Platform OpenCL Application Development. In *Proceedings of the 4th International Workshop on OpenCL*, IWOCL '16, pages 2:1–2:12, 2016.

[103] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 205–217, 2015.

[104] M. Steuwer, M. Friese, S. Albers, and S. Gorlatch. Introducing and Implementing the Allpairs Skeleton for Programming Multi-GPU Systems. *International Journal of Parallel Programming*, 42(4):601–618, 2014.

[105] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1176–1182, May 2011.

[106] M. Steuwer, P. Kegel, and S. Gorlatch. Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library. In *2012 IEEE 26th Interna-*

tional Parallel and Distributed Processing Symposium Workshops PhD Forum, pages 1858–1865, May 2012.

[107] M. Steuwer, T. Remmelg, and C. Dubach. Matrix Multiplication Beyond Auto-tuning: Rewrite-based GPU Code Generation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, pages 15:1–15:10, 2016.

[108] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007.

[109] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer. Automatic OpenCL device characterization: Guiding optimized kernel design. In E. Jeannot, R. Namyst, and J. Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 438–452. Springer-Verlag, 2011.

[110] P. Tillet, K. Rupp, S. Selberherr, and C.-T. Lin. Towards performance-portable, scalable, and convenient linear algebra. In *5th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2013. USENIX.

[111] P. D. S. Tomov and R. Nath. MAGMA: Matrix Algebra on GPU and Multicore Architectures, 2011.

[112] TOP500.org. TOP500 List - 48th Edition. `https://www.top500.org/lists/2016/11/`, November 2016. [Online; accessed 9-December-2016].

[113] T. L. Veldhuizen. C++ Templates as Partial Evaluation. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, pages 13–18, 1999.

[114] M. Viñas, Z. Bozkus, and B. B. Fraguela. Exploiting heterogeneous parallelism with the Heterogeneous Programming Library. *Journal of Parallel and Distributed Computing*, 73(12):1627–1638, 2013.

[115] M. Viñas, B. B. Fraguela, D. Andrade, and R. Doallo. Towards a High Level Approach for the Programming of Heterogeneous Clusters. In *2016 45th Inter-*

national Conference on Parallel Processing Workshops (ICPPW), pages 106–114, Aug 2016.

[116] M. Viñas, B. B. Fraguela, D. Andrade, and R. Doallo. High productivity multi-device exploitation with the Heterogeneous Programming Library. *Journal of Parallel and Distributed Computing*, 101:51–68, 2017.

[117] M. Viñas, B. B. Fraguela, Z. Bozkus, and D. Andrade. Improving OpenCL Programmability with the Heterogeneous Programming Library. In *Proceedings of 2015 International Conference On Computational Science (ICCS 2015)*, pages 110–119, 2015.

[118] M. Wall. GAlib: A C++ Library of Genetic Algorithm Components, 1996.

[119] R. Weber and G. D. Peterson. Poster: Improved OpenCL Programmability with clUtil. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1451–1451, Nov 2012.

[120] J. R. Wernsing and G. Stitt. Elastic computing: A portable optimization framework for hybrid computers. *Parallel Computing*, 38(8):438–464, 2012.