Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 00 (2013) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2013

OCLoptimizer: an iterative optimization tool for OpenCL

Jorge F. Fabeiro[a], Diego Andrade*[a], Basilio B. Fraguela[a]

[a]*Computer Architecture Group, University of A Coruña, Spain*

Abstract

Nowadays, computers include several computational devices with parallel capacities, such as multicore processors and Graphic Processing Units (GPUs). OpenCL enables the programming of all these kinds of devices. An OpenCL program consists of a host code which discovers the computational devices available in the host system and it queues up commands to the devices, and the kernel code which defines the core of the parallel computation executed in the devices. This work addresses two of the most important problems faced by an OpenCL programmer: (1) hosts codes are quite verbose but they can be automatically generated if some parameters are known; (2) OpenCL codes that are hand-optimized for a given device do not get necessarily a good performance in a different one. This paper presents a source-to-source iterative optimization tool, called OCLoptimizer, that aims to generate host codes automatically and to optimize OpenCL kernels taking as inputs an annotated version of the original kernel and a configuration file. Iterative optimization is a well-known technique which allows to optimize a given code by exploring different configuration parameters in a systematic manner. For example, we can apply tiling on one loop and the iterative optimizer would select the optimal tile size by exploring the space of possible tile sizes. The experimental results show that the tool can automatically optimize a set of OpenCL kernels for multicore processors.

Keywords: OpenCL ; genetic algorithms ; iterative optimization

1. Introduction

Current computers include a heterogeneous set of computational devices, such as multicore processors and Graphic Processing Units (GPUs). Different technologies have been used to extract parallelism from different kinds of devices. For example, OpenMP [1] or Intel TBBs [2] are used to extract parallelism from multicore processors, while CUDA [3] is widely used for GPUs. OpenCL [4] is a standard that allows to write programs that can be executed on a wide range of devices, such as multicores, GPUs or heterogenous processors like the Cell. However, the performance of programs written in OpenCL is not portable, thus, an OpenCL program can obtain a good performance in one device but a poor one in a different device, and vice-versa. As a consequence, OpenCL programs have to be hand-tuned for each device where they may be executed.

Iterative optimization has been widely used to automatically tune a code for a given architecture [5, 6]. Additionally, the performance portability in the context of parallel languages has been studied for a long time [7]. However, lately, it has regained interest due to the heterogeneity of the available accelerators. For example, Du et al [8] use iterative compilation to select the optimal parameters for an OpenCL code in a given platform. This work is focused in the obtention of a portable linear algebra library and in the selection of the optimal tile size

*Corresponding author. Tel.: +34 981 167000 ext. 1298.
E-mail address: diego.andrade@udc.es.

for the tiling technique. Moore et al [9] introduce VForce, which allows the efficient usage of accelerators. The performance portability is completely transparent to the programmer. The work is focused in the field of the image and signal processing as the input is a program written in a domain-specific language called VSIPL++ (Vector Signal Image Processing Library extension). Wernsing et al [10] introduce the elastic computing framework, which allows to separate functionality from the implementation details using specialized functions. These functions allow to explore a great variety of alternative implementations and to select the optimal for a certain platform. The main limitation is that the code has to be expressed using the available specialized functions. Finally, Augonnet et al [11] present StarPU, which automates the efficient mapping of tasks in heterogeneous environments. The problem is that while this tool efficiently dispatch tasks, it does not tune the performance of each task individually.

This work introduces OCLoptimizer, a source-to-source iterative optimization tool that automatically optimizes an OpenCL program for the platform where the tool is executed. The election of OpenCL serves our purpose to execute the same kernel code on a wide range of platforms. Iterative optimizers are used to explore a wide range of possible optimization decisions. Namely, we can try to apply tiling on one loop and the tool would select the optimal tile size. The search space can consist of all the possible tile sizes, or a representative subset that will likely contain the optimal value. The version of the code built using each possible tile size can be evaluated by means of an analytical model, heuristics or by executing the resulting code.

As an use case, the version of OCLoptimizer presented in this work can apply the unroll and unroll-and-jam optimization to one or several loops in an OpenCL kernel. All the possible unroll factors in a range are tested and the decision is taken based on the execution time obtained using each unroll factor. The space of possible unroll factors can be explored using an exhaustive search or it can be guided using genetic algorithms [12]. This second alternative improves the quality of the generated version when the code contains several unrollable loops and it dramatically reduces the total execution time of the tool, as it explores the search space in a more intelligent manner.

The OCLoptimizer is executed on the target platform for which the code has to be optimized and it takes two inputs: (1) an annotated OpenCL code, where the programmer has included a set of directives that indicate the parts of the code that must be optimized and the techniques to apply to each one, and (2) a configuration file that includes information to generate automatically a working host code. The output of the tool is the working host code and an OpenCL kernel highly optimized for the target platform. The host code generated contains a random initialization of the data structures that appear in the code, which can be easily replaced by the programmer with its own initialization. The automatic generation of the host code releases OpenCL programmers from the painful task of generating it manually. Experimental results show the effectiveness of the tool to automatically optimize a set of different OpenCL kernels.

This paper is organized as follows. Sections 2 and 3 give an overview of OpenCL and CLANG, the two technologies on top of which OCLoptimizer is built. Section 4 describes how the OCLoptimizer tool works and how it is internally organized. Section 5 presents the experimental results and Section 6 concludes.

2. OpenCL

The Open Computing Language (OpenCL) is an open industrial standard for the programming of heterogeneous platforms, including multicore processors, GPUs, Digital Signal Processors (DSPs) or Field Programmable Gate Arrays (FPGAs). The standard consists of a framework composed of a C99-based programming language for writing kernels and a set of application programming interfaces (APIs). The hardware manufacturers that support the standard provide access to their hardware accelerators on their own implementations of the standard.

OpenCL distinguishes the host system and the devices (accelerators) available in that host system. An OpenCL device is logically composed of one or more computing units (CUs) which are divided into one or more processing elements (PEs).

OpenCL programs run on the host system, which discovers the OpenCL platforms available in that host system, then it finds the devices that can be accessed through each platform, it creates a queue associated to each device and it queues up commands through these queues. The host orders the execution of kernels, which are the core of the parallel computation, to the devices. Host and devices have separated memory spaces, so data must be explicitly transferred to the devices. In addition, the synchronization of data transfers and kernels may be explicitly

managed in the queue associated to each device. Thus, the kind of commands enqueued to the devices include: data transfers, synchronization operations and kernels.

The host defines a workspace which is composed of several work-items. An instance of the kernel is generated by each work-item. These instances are executed by the PEs of the device. The workspace can have one or several dimensions, this way, each work-item is identified by a sequence of identifiers, one per dimension. The number of points of each dimension in the global workspace is the global size.

Work-items are grouped into work-groups. The number of points per dimension of a work-group is called the local size. These work-groups are composed of work-items that can be synchronized and they share a special memory region, called local memory. All work-items in one work-group are executed on the same CU.

This way, work-items are identified in the workspace using one identifier per dimension, called global identifiers, and they are identified inside their work-group using one identifier per dimension, called local identifiers. As it was said, each instance of the kernel is associated to a work-item. Inside each instance of the kernel the values of the global and local identifiers can be obtained, thus, they can be used to exploit data-parallelism easily. For example, a vector addition can be easily programmed with OpenCL by defining a one-dimensional workspace where the number of work-items is equal to the number of elements of the vectors to be added. The kernel would perform the addition of two individual elements, and the global identifier would be used to select the pair of elements to be added.

3. CLANG and the LLVM compiler infraestructure

The Low Level Virtual Machine (LLVM) [13] project is a compiler infrastructure designed for compile-time, link-time and idle-time optimization of programs. CLANG is a C and C++ frontend for LLVM. It translates C or C++ code, with support for OpenCL to the LLVM intermediate representation (IR). CLANG and LLVM are often used by most programmers to compile OpenCL codes.

The OCLoptimizer is a source-to-source tool as, this way, programmers can inspect the resulting code and maybe make some final adjustments on it. This goal can be achieved using two alternative methods:

- We can transform the C code into LLVM IR, make some transformations on top of the LLVM IR and generate back the transformed C code.
- We can transform the C code into an Abstract Syntax Tree (AST) and make some transformations on top of the source code using the information included in the AST, to generate a transformed version of the original source C code.

The first alternative can be implemented using CLANG to generate the LLVM IR and then the LLVM infrastructure to transform the LLVM IR. The resulting C code can be generated using the LLVM static compiler (`llc`). The problem with this alternative is that the C code generated by `llc` is difficult to understand by the programmer.

The second alternative can be built on top of CLANG as its architecture includes the following components:

- A syntatic analyzer (Parser) and lexycal analyzer (Lexer).
- An Abstract Syntax Tree (AST) generator (using a C or C++ as an input).
- An LLVM intermediate representation generator (using an AST as an input).
- Source code managing support. Specifically, the librewrite module can be used to know the source code file range where the source code associated to a given node of the AST is located. These ranges can be used to modify the source code.

The AST generator can be used to generate the AST, while the source code managing support can be used to transform a program on top of the source code. This approach suits our purposes better, as it generates a human-readable C code and it seems the most natural way to implement source-to-source transformations.

4. The OCLoptimizer tool

The OCLoptimizer tool is a source-to-source iterative optimization tool for OpenCL programs. The inputs of the tool are a configuration file and an OpenCL kernel. The OCLoptimizer generates a working OpenCL host

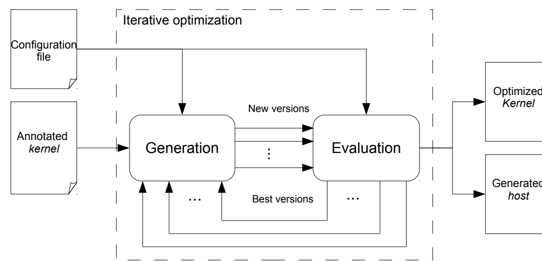

Fig. 1. General workflow of OCLoptimizer

```
# common parameters
N=1024,2048
device=gpu
# compiler parameters
mode=system
ocllibpath=/usr/local/lib
oclincludepath=/usr/local/include
# workspace
ndims=1
[dim0]
globalsize=N
localsize=1
# kernel arguments
nargs=1
[arg0]
name=A
size= N
type= float *
mode=w
```

Fig. 2. Configuration file example

code for the kernel, and it optimizes the kernel for the platform where the tool is executed. The rest of this Section is organized as follows. First, Section 4.1 introduces the workflow of the tool, then, Section 4.2 summarizes its main components, after that, Section 4.3 describes the inputs of the tool and Section 4.4 introduces the two search algorithms that can be used to driver the iterative optimization process and finally Section 4.5 describes the outputs generated by the tool.

4.1. Workflow of OCLoptimizer

Figure 1 shows the general workflow followed by the tool. As it was said, the tool receives as inputs: an annotated OpenCL kernel and a configuration file. The configuration file is mainly used to drive the generation of a working host code and to define the different problem sizes that must be taken into account by the tool in the optimization process. This working host code contains a random initialization, that the user of the tool can replace with its own initialization. This host code is internally used by the tool to execute the intermediate versions of the kernel generated. The annotated kernel contains pragmas that indicate the parts of the code that must be optimized and the techniques to apply to each one. Pragmas are processed in an iterative process that gives place to an optimized version of the kernel. In OCLoptimizer, this iterative process can be guided by a breadth first search (BFS) or a genetic algorithm (GA). In BFS, pragmas are processed one by one in an iterative process where the processing of each pragma gives place to a new level of the process. In each level, a set of versions of the kernel are generated as a result of the application of a pragma, these versions are ordered and filtered to decide which ones proceed to the next level. In the GA, several versions of the kernel are generated by applying the transformations associated to all the pragmas present in the code at the same time, if one of these versions is satisfactory, then it is selected as the output version of the kernel, else, the GA generates a new population by reproduction, crossover and mutation.

4.2. Internal structure of OCLoptimizer

The OCLoptimizer is written in C++ on top of the libraries provided by the CLANG front-end. The code is organized in four packages:

- The *PragmaPreprocessor* package contains the classes that scan the input kernel file searching for the pragmas introduced by the user. The relevant information from these pragmas is extracted and maintained in a separate structure.
- The *OcloptsGenerator* package contains the classes that implement the code generation phase of the iterative process. They generate the AST associated to the input kernel using CLANG and they process each annotation found in the code to generate a number of transformed versions. These versions are generated using the `librewrite` module of CLANG, which allows to locate the string of the source code associated to each part of the AST. This module provides a natural method to perform source-to-source transformations.
- The *HostGenerator* package contains the classes that parse the configuration file and generate a working host code that can be used in the evaluation phase of the iterative process.
- The *VersionSelector* package contains the classes that implement the evaluation stage of the iterative process. It executes each version of the code using each problem size specified in the configuration file. The classes in this package also manage an ordered list of versions, a score system used when several problem sizes have to be taken into account, and the versions filtering when the tolerance and number parameters appear in a pragma.

4.3. Inputs of OCLoptimizer

The OCLoptimizer has two input files, the configuration file and the annotated kernel, whose content is covered separately. The configuration file contains the definition of several variables which will be used mainly to drive the generation of the host code. This file has four sections:

- The *common parameters* section initializes the values of some variables that will be used through the rest of the configuration file and it configures other general settings of the OpenCL host code to be generated. Figure 2 contains a configuration file example. In this example, this section contains the initialization of the variable `N` to an array with two values 1024 and 2048. This means that the tool has to take into account two possible values 1024 and 2048, whenever this variable appears. Finally, it establishes that the host code has to use a GPU to perform the computation. Alternatively, the `device` variable could take the value `CPU`.
- The *compiler parameters* section sets several values related to the compilation process, like the location of the library and headers files of the OpenCL implementation to be used, or the compilation mode which sets the way the intermediate versions of the code are compiled by the tool. The current version of OCLoptimizer only supports the `system` compilation mode, which performs the compilation using a system call to the g++ compiler. In the example of Figure 2, the compilation mode is set to `system`, while the locations of the library and header files are provided.
- The *workspace definition* section defines the workspace. Namely, the number of dimensions of the workspace (`ndims`), and for each dimension, the global and the local size. The values associated to dimension `X` are preceded by a `[dimX]` clause. In the example of Figure 2, the workspace has only one dimension composed of `N` work-items and each work-item is in a different work-group. As `N` has two possible values (1024 and 2048) the tool has to consider two different problem sizes. These problem sizes will be evaluated separately to select the optimal version of the kernel.
- The *kernel arguments* section defines the number, the type and the size of each argument of the kernel. The information associated to the argument `X` is preceded by a `[argX]` clause. In the example of Figure 2, the kernel has only one argument called `A` (`name=A`) which is an array of `N` elements (`size=N`) of type float (`type= float *`) that can be modified inside the kernel (`mode=w`).

The other input of OCLoptimizer is an OpenCL kernel containing annotations of the user. These annotations are pragmas that mark regions of the code with the optimization techniques to apply to them. The annotations include the parameters of the technique and the search space. The general form of an annotation is

```
# pragma oclopts <name> <params> [ tolerance <0 - 100>] [ number n]
```

```
...
# pragma oclopts unroll 2 16 2 number 2
for (int i=0;i <1000; i++) {
        a[i] = a[i] + s;
}
...
```

Fig. 3. Annotated kernel

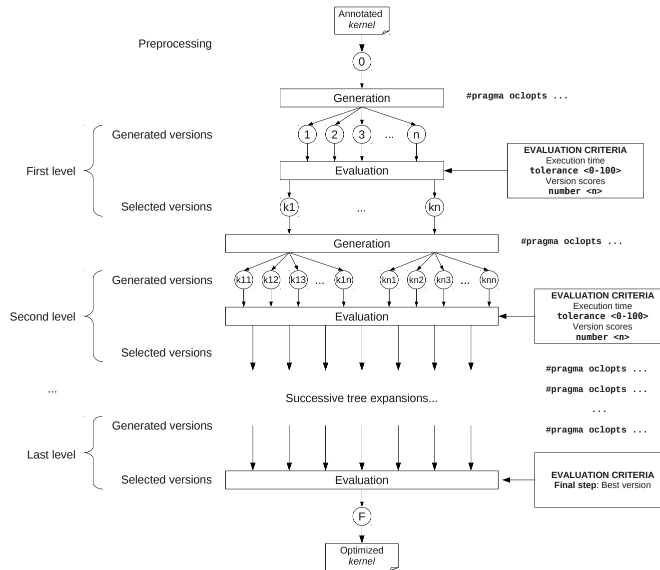

Fig. 4. Iterative process in OCLoptimizer tool using a breadth search first

The keyword `<name>` must be replaced by the name of the optimization technique to be applied. `<params>` is replaced by the parameters, which are different for each technique. They usually define the range of values to be tested in the iterative optimization process. The field `tolerance`, when it is defined, establishes that only the versions whose execution times are less than a `tolerance` per cent above the average value, proceed to the next level of the iterative process. Finally, the argument `number` can optionally set the maximum number of versions which proceed to the next level of the iterative process. These two fields, (`tolerance` and `number`), are only used by BFS. As it was said previously, this version of the tool can only apply automatically the unroll and unroll-and-jam technique and it selects the optimal unroll factor. The general form of the pragma associated to this technique is as follows.

```
# pragma oclopts unroll [init end step] [ tolerance <0 - 100>] [ number n]
```

The parameters associated to this technique (`init`, `end` and `step`) set the first, the last and the step value of the range of potential unroll factors that the tool has to evaluate.

Figure 3 contains a section of an OpenCL kernel in which a loop is annotated with an `unroll` pragma. All the possible unroll factors between 2 and 16 with step 2 must be tested and at most the best two versions of the kernel has to proceed to the next stage of the optimization process.

4.4. Search algoritms used by OCLoptimizer

The iterative process performed by the tool can be guided by a breadth first search (BFS) or a genetic algorithm (GA). These two alternatives are commented separately in Sections 4.4.1 and 4.4.2.

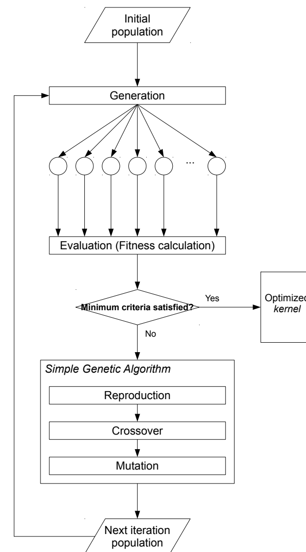

Fig. 5. Iterative process in OCLoptimizer tool using a genetic algorithm

4.4.1. Breadth First Search

The BFS is illustrated in Figure 4. This search has two main stages: optimization and evaluation. The annotations in the kernel are processed one by one. In the optimization stage, one of the annotations of the kernel is processed giving place to a number of versions of the kernel. For example, if an unrolling pragma is found, a version of the kernel using each unroll factor requested is generated.

In the evaluation stage, the tool creates a list of versions ordered in increasing order of their execution time. The `tolerance` argument of the pragma is used to remove from the list those versions with an excessive execution time. Then, if the processed pragma has also a `number(n)` argument, only a maximum of `n` versions will be maintained in this list of versions. The resulting list of versions will be used as a base for the next level of the iterative optimization process.

The configuration file can define a set of different problem sizes (see parameter `N` in Figure 2). In that case, a separate ordered list must be generated for each problem size. Each per-size list is filtered using the `tolerance` and `number` parameters of the pragma. The tool composes a new list using the versions that are at least in one of the lists. This new list of versions will be used as a base for the next level of the iterative process. This process will be repeated until all the pragmas in the kernel have been processed. In this final step of the process, each version gets a score equal to its position in each per-size list. Then, we add the scores obtained by each version in the different per-size lists and we create a final list of versions ordered in increasing order of their overall scores. This way, when different problem sizes have to be taken into account, the versions that perform better for more problem sizes occupy the first positions in this final list. Then, the first version of the final list generated after the processing of the last pragma will be picked as the overall winner.

The problem of BFS is that if there are several loops to be unrolled in a kernel, the number of versions to be evaluated experiments a combinatorial explosion. For example, if 4 loops are annotated to be unrolled and the number of possible unroll factors to be tested in each loop is 10, then the final number of versions to be evaluated is 10^4. This number of versions can be reduced using the `tolerance` and `number` pragma parameters which reduce the number of versions that proceed to the next level. However, the problem of the BFS search is that either a limited subset of the search space is evaluated or, if the search space is explored exhaustively, the execution time of the tool is very large. As a consequence, this approach may not lead to an optimal solution as the unroll factors selected for each loop may be interdependent if several unrollable loops are part of the same loop nest.

4.4.2. Genetic Algorithm Search

Figure 5 gives an overview of the iterative optimization process guided by a GA. First, the chromosomes of the GA, which are potential solutions to our problem, have one gene per pragma in the input kernel. In the current version of the tool, the chromosome has one gene per loop to be unrolled, and this gene stores the unroll factor to be used by this chromosome for the particular loop.

The initial population of the algorithm is composed of a configurable number of individuals. This number has been fixed internally in the tool by experimentation. These chromosomes are generated using random combinations of the values that the different genes can take (different unroll factors). If the number of chromosomes randomly generated is not enough, they will be cloned until the required number of chromosomes is achieved.

One version of the kernel per chromosome is generated. These versions are evaluated and their speedup with respect to the original kernel is calculated. The maximum speedup obtained is used to evaluate if the current generation meets the minimum criteria and thus, it is a valid solution to our problem. In that case, the kernel version associated to this maximum speedup is provided as the output of our tool with the working host code.

If the algorithm has not yet achieved an acceptable solution, then, a new population is generated. The best individuals from the current population are selected to produce the next generation through reproduction. This new population of individuals is generated using two different operators:

- *Crossover*: The genes of two parent chromosomes are combined to generate a new child chromosome which will be incorporated to the next generation. A random cross point is selected which divides the chromosomes into two halves. The genes of the first half of the first parent chromosome and the genes of the second half of the second parent chromosome give place to the new child chromosome.
- *Mutation*: One random gene of the chromosome changes its status. This mutation gives place to a new individual of the next generation.

The new individuals generated will be used to perform a new iteration/generation of the genetic algorithm. This process will be repeated until the population achieves an acceptable value of the fitness function. In our tool this fitness function is obtained by calculating the speedup of the best generated version with respect to the original kernel. The GA finishes when the speedup obtained has not improved in the last three generations.

When several loops have to be unrolled in the same loop nest, this method, unlike BFS, evaluates a limited number of the possible combinations of unroll factors and it takes into account the unrolling of all the unrollable loops simultaneously. This approach may lead to a better solution than using BFS.

4.5. Outputs of OCLoptimizer

The OCLoptimizer has two outputs: a kernel highly-optimized for the platform where the tool has been executed and a working host code. The kernel is an optimized version of the input kernel where all the loops marked as unrollable have been unrolled and the optimal unroll factor have been selected. The working host code can be used to execute the output kernel, without any modification, it contains a random initialization of the data structures that can easily be replaced by the user of the tool with its own initialization. The user of the tool has to make sure that the algorithm works correctly with a random initialization. Still, the tool will be extended in the future to read an initialization code or directly the contents of the data structures from user provided files, as the user may want that the tool uses its own initialization during the iterative optimization.

The host code is generated automatically and it is ready to use the kind of device specified in the configuration file, it creates a workspace with the characteristics specified in the configuration file, it compiles the kernel, it enqueues the data to be used by the kernel and the kernel itself, and it gathers the results generated by the kernel.

5. Experimental results

The validation is based in three computationally intensive codes: a matrix multiplication (MATMUL), a Sobel Edge Detector [14] (SOBEL) and a Direct Coulomb Summation [15] (DCS). The OpenCL kernel that implements MATMUL has three unrollable loops while the kernels that implement the SOBEL and the DCS algorithms have four unrollable loops each. The experiments were run on an Intel Xeon E5620 CPU with eight 2.4 Ghz cores and 16 GB of RAM.

Table 1. Speedups achieved and optimal and maximum unroll factors (UFs) using a BFS or a GA for the three codes and using thee different problems sizes

Code	Size	BFS		GA		
		Speedups	Optimal UFs	Speedups	Optimal UFs	Maximun UFs
MATMUL	1024	1.34013	1,3,3	1.16928	2,3,65	3,3,154
	2048	1.63384	1,1,93	1.24247	2,3,139	3,3,154
	4096	1.41205	1,2,71	1.36228	3,1,79	3,3,154
SOBEL	1024	1.02336	6,3,3,3	1.04856	1,10,1,3	13,13,3,3
	2048	1.03579	1,1,3,3	1.03958	1,1,3,3	13,13,3,3
	4096	1.19674	4,2,1,3	1.13234	7,4,1,3	13,13,3,3
DCS	64	1.39748	1,1,1,8	1.26691	1,2,2,41	2,2,2,64
	128	1.04998	1,1,1,5	1.02554	1,1,1,26	2,2,2,64
	256	1.01123	2,1,1,12	1.00635	2,2,1,7	2,2,2,64

Table 2. Tool execution times for CPU performance tuning

Code	Size	BFS				GA			
		Time	#ver	Percentages		Time	#ver	Percentages	
				Generation	Evaluation			Generation	Evaluation
MATMUL	1024	2793.11	1386	30.94%	69.06%	181.063	79	0.84%	99.16%
	2048	19971.4	1386	4.32%	95.68%	2127.52	133	1.44%	98.56%
	4096	324506	1386	0.08%	99.92%	16252.3	69	1.92%	98.08%
SOBEL	1024	1546.76	1521	57.52%	42.28%	120.599	75	44.58%	55.42%
	2048	3595.92	1521	26.09%	73.91%	151.791	76	43.20%	56.80%
	4096	11686.5	1521	8.03%	91.97%	652.53	222	31.58%	68.42%
DCS	64	1262.21	512	25.02%	74.98%	70.6902	42	13.08%	86.92%
	128	11421.6	512	2.76%	97.24%	216.242	25	5.81%	94.19%
	256	173117	512	0.18%	99.82%	2979.13	25	3.98%	96.02%

Table 1 summarizes the results obtained using either the BFS or the GA search. For each unrollable loop, a range of possible unroll factors (UFs) has been taken into account. The values in these ranges are generated always starting at 1 and using step 1, the maximum value for each case is contained in the Maximum UFs columns. Each cell of these columns contains one maximum unroll factor per loop separated by commas. The BFS search has been tested generating all the possible versions of the kernel considering the ranges established in the pragmas, without filtering the number of versions using the `number` or `tolerance` parameters. The GA search has been tested using populations with one gene per unrollable loop where each gene can take values inside the range of possible UFs. Thus, this table contains two groups of columns, one for the BFS and another one for the GA. Each one of these groups contain the speedup with respect to the original OpenCL version obtained by the tool, and the unroll factors for each loop (separated by commas) that gave place to this speedup. The results of the table show that the tool can generate a tuned version of the kernel.

The results of Table 1 shows that the tool when guided by BFS provides a better solution than when guided by GA, as expected, now we will see at which cost. Table 2 contains the execution time of the tool for each case tested in Table 1. The results are provided separately for the BFS and the GA search. For each case, it is provided the total execution time of the tool, the number of intermediate versions generated (#ver), and which percentage of the execution time is due to the generation of the intermediate versions and which percentage is due to the evaluation of each version generated. The results of the table shows that the tool guided by BFS generates many more versions of the kernel than when guided by GA and as a consequence it increases largely the total execution time of the tool. Most of the tool execution time is devoted to evaluate each version generated. Let us notice that in the current version of the tool, the evaluation of each version is performed by executing the code generated, which is a very time-consuming approach. This evaluation could be performed using heuristics or performance models, which would allow to reduce drastically the total execution time of the tool.

6. Conclusions

This paper has introduced OCLoptimizer, a tool that performs two valuable actions for OpenCL programmers: it generates automatically a working host code and it tunes the performance of a kernel for the platform where

the tool is executed. The tool receives as inputs a configuration file and an annotated version of the kernel to be optimized. The annotations are introduced by the user of the tool and they mark regions of the code with the optimization techniques to apply to them.

The version of this tool presented in this paper automatically applies the unroll and unroll-and-jam technique on the loops annotated by the user. The tool selects the unroll factors, or combinations of unroll factors, that generate the shortest execution time. Two different methods are used to explore the search space: a breadth first search (BFS) and a genetic algorithm (GA). The experimental results show that the tool can tune automatically the performance of a set of OpenCL kernels for multicore processors. The speedup achieved is 19% on average with a peak speedup of 63%. The tool, when is guided by the BFS search, maximizes the speedup achieved but the execution time of the tool is a 2400% greater that using a GA, on exchange, the solution provided by BFS is a 64% better than the one provided by GA.

As future work, the tool will be extended to automatically select the optimal workspace configuration and to apply a wider range of optimization techniques. We also plan to extend OCLoptimizer so that the evaluation of the different versions generated can be also made using heuristics or analytical models rather than execution times [16]. This way, the tool will be faster and will be able to optimize code for a platform different from where the tool is executed.

Acknowledgements

This work has been supported by the Galician Government under projects Consolidation of Competitive Research Groups (ref 2010/6), INCITE08PXIB105161PR, UDC/GI-000265 and CN2012/211 (partially supported by FEDER funds), and the Ministry of Education and Science of Spain, FEDER funds of the European Union (Project TIN2010-16735).

References

[1] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, Parallel programming in OpenMP, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[2] J. Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism, 1st Edition, O'Reilly, 2007.

[3] NVIDIA Corporation, NVIDIA CUDA Compute Unified Device Architecture Programming Guide, NVIDIA Corporation, 2007.

[4] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, OpenCL Programming Guide, Addison-Wesley Professional, 2011.

[5] L.-N. Pouchet, C. Bastoul, A. Cohen, N. Vasilache, Iterative optimization in the polyhedral model: Part i, one-dimensional time, in: Proceedings of the International Symposium on Code Generation and Optimization, 2007, pp. 144–156.

[6] T. Kisuki, P. M. W. Knijnenburg, 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, 2000, p. 237.

[7] T. Ngo, L. Snyder, B. Chamberlain, Portable performance of data parallel languages, in: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM), 1997, pp. 1–20.

[8] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra, From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming, Parallel Comput. 38 (8) (2012) 391–407.

[9] N. Moore, M. Leeser, L. Smith King, Vforce: An environment for portable applications on high performance systems with accelerators, J. Parallel Distrib. Comput. 72 (9) (2012) 1144–1156.

[10] J. R. Wernsing, G. Stitt, Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing, in: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems, 2010, pp. 115–124.

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

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

[13] C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, 2004.

[14] N. Kazakova, M. Margala, 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, 2004, pp. II – 913–16 Vol.2. doi:10.1109/ISCAS.2004.1329421.

[15] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, Journal of Computational Chemistry 28 (16) (2007) 2618–2640.

[16] B. B. Fraguela, M. G. Carmueja, D. Andrade, Optimal tile size selection guided by analytical models, in: Parallel Computing (ParCo), 2005, pp. 565–572.