An automatic optimizer for heterogeneous devices

Jorge Fernández-Fabeiro[a], Diego Andrade[b,*], Basilio B. Fraguela[b], Ramón Doallo[b]

[a] *Departamento de Informática, Universidad de Valladolid, Spain*
[b] *Universidade da Coruña, Spain*

Abstract

Codes written in a naive way seldom effectively exploit the computing resources, while writing optimized codes is usually a complex task that requires certain levels of expertise. This problem is further increased in the presence of heterogeneous devices, which present more tunable parameters than regular CPUs and high sensitivity to the optimization decisions taken. Furthermore, portability is an added concern given the wide variety of accelerators available. This paper tackles this problem adding an automatic optimizer to a library that already provides an easy and portable way to program heterogeneous devices, the Heterogeneous Programming Library (HPL). Our optimizer takes as input a simple version of a code and then tunes it for the device where it is going to be executed by performing the most usual set of optimizations applicable in heterogeneous devices. These optimizations are parametrized using a set of optimization parameters that need to be tuned for the device. The HPL library has also been equipped with an autotuner that can be used to this purpose. The effectiveness of the autotuner and the optimizer has been tested on several codes and devices. The results show that the combination of the autotuner and the optimizer make the tested codes 16 times faster on average than the original codes written by the programmer.

Keywords: Heterogeneous systems; Performance Portability, Performance Tuning; OpenCL

1. Introduction

Nowadays there is a wide range of heterogeneous devices available for general purpose computing. Thus, programmers have to face two important problems. The first one is that there is a large number of libraries, languages or extensions that allow to program these devices. While some of these alternatives are only usable on a limited range of devices, standards like OpenCL [1] or OpenACC [2] allow to develop codes that can be executed on a wide range of devices. The second problem is that, even if they use one of these portable technologies, the code has to be manually optimized to exploit effectively the capabilities of each device in order to obtain a good, and sometimes just even a reasonable performance. This is true even in the case of the approaches supported by optimizing compilers that retarget the binary generated to the specific device to use, as they can miss important optimization opportunities [3].

One of the programming tools that addresses the development of portable codes for heterogeneous devices is the Heterogeneous Programming Library (HPL) [4]. This framework provides a simple API and semantics, and the fact that its current backend relies on OpenCL implies that HPL codes can be run in all the devices that support this standard. The HPL environment generates and compiles OpenCL code at run-time, which opens the possibility to dynamically apply optimizations on it. This work takes advantage of this feature in order to extend HPL with an automatic optimizer, which is particularly valuable given the diverse nature of the devices that can be targeted by this framework.

The main contributions of the current paper are two important additions to the HPL environment which were not present in any previous version of this library:

- An optimizer that can automatically optimize a naive code written by a programmer. The

*Corresponding author
Email addresses: jorge@infor.uva.es (Jorge Fernández-Fabeiro), diego.andrade@udc.es (Diego Andrade), basilio.fraguela@udc.es (Basilio B. Fraguela), doallo@udc.es (Ramón Doallo)

©2020. This manuscript version is made available under the CC-BY-NC-ND 4.0 license
http://creativecommons.org/licenses/by-nc-nd/4.0/
The formal publication is available at https://doi.org/10.1016/j.future.2020.01.018

set of optimizations included in this optimizer are some of the usually applied on heterogeneous systems, such as local memory exploitation, the adjustment of the amount of work executed by each thread, or tiling. As we will see, this addition required important changes in the internals of HPL.

- An autotuner that sets the values of these parameters on a device using an iterative search process guided by the execution time. In our current implementation the search space can be traversed using a random policy, a genetic algorithm or an exhaustive search.

The rest of this paper is organized as follows. Section 2 introduces HPL. Then, Section 3 gives an overview about how HPL has been equipped with an optimizer, and it explains in detail the set of optimizations performed by that optimizer. Section 4 summarizes the parameters that control the operation of the optimizer and describes briefly the autotuner. Section 5 presents the experimental results, which are compared in Section 6 to those obtained by other approaches. Some related work is discussed in Section 7 and, finally, Section 8 concludes.

2. The Heterogeneous Programming Library

The Heterogeneous Programming Library (HPL) [4], available at `http://hpl.des.udc.es`, largely simplifies the development of portable heterogeneous systems with respect to other alternatives such as OpenCL while allowing fine grained control of the code, and thus enabling good performance compared to other alternatives [5].

The HPL library supports the same programming model as CUDA and OpenCL. This way, the hardware model is composed of a standard CPU host with a number of computing devices attached. The host runs the sequential parts of the code and it dispatches the parallel parts, which are codified as HPL kernels, to the devices. The CPU of the host can be itself a computing device. Devices are composed of a number of processors that execute SPMD parallel code on data present in the memory of their device. As kernels can only work with data available in the devices, data must be transferred between host and devices, but this process is totally automated by the library.

Several instances of each kernel are executed as threads and they are unequivocally identified using a tuple of non-negative integers, called global identifiers. These identifiers, and their associated threads, form a global domain with up to 3 dimensions. In turn, these threads can be associated in groups. With this purpose, local domains can be defined as equal portions of the global domain. Threads inside a group are also identified using tuples of local identifiers and they can be synchronized through barriers and share a small scratchpad memory.

The HPL memory model distinguishes four types of memory regions in the devices (from largest to smallest): (1) the global memory, which is read-/written and shared by all the processors, (2) the local memory, which is a read/write scratchpad shared by all the processors in a group, (3) the constant memory, which is a read-only memory for the device processors and can be set up by the host, and (4) the private memory, which is only accessible within each thread.

Programmers using HPL have to write a code to be executed in the host, and one or several kernel codes, which will be dispatched to the heterogeneous device(s). To do that, the library provides three main components: the template class `Array`, the kernels and the host API. They are now discussed in turn.

The variables used in a kernel must have type `Array<type, ndim [, memoryFlag]>`. This type represents an n-dimensional array of elements of a C++ `type`, or a scalar for `ndim`=0. Scalars and vectors can also be defined with special data types like `Int`, `Float`, `Int4`, `Float8`, etc. The `Array` optional `memoryFlag` specifies one of the kinds of memory supported (`Global`, `Local`, `Constant` or `Private`). The default value of the `memoryFlag` is `Global`, the exception being the `Array`s declared inside the body of kernels, which are placed by default in `Private` memory. The elements that compose an array may be any of the usual C++ arithmetic types or a struct. The arrays passed as arguments to the kernels must be declared in the host code using the same syntax. HPL automatically allocates the memory space for these data structures in the different memories where they are used and keeps the copies required in a coherent state by performing the minimum possible number of transfers.

HPL kernels use special control flow structures. They are similar to those available in C++, but their name finishes with an underscore (`if_`, `for_`,

```
#include "hpl.h"

using namespace HPL;

void saxpy(Array<float,1> y,
           Array<float,1> x, Float a) {
  y[idx] = a * x[idx] + y[idx];
}

int main(int argc, char *argv) {
  Float a;
  Array<float, 1> x(1000), y(1000);
  // x, y and a are filled in with data
  eval(saxpy).global(1000).local(10)(y, x, a);
}
```

Figure 1: SAXPY HPL code

...). Also, the arguments passed to `for_` loops are separated by commas instead of semicolons. In addition, 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 number of groups. For example, `idx` provides the global identifier of the first dimension, while `szx` provides the global size of that dimension. Adding the `l` prefix to these keywords allows to obtain their local counterparts, whereas replacing the letter `x` with `y` or `z` the same values are obtained for the second and the third dimensions respectively.

Kernels are written as regular C++ functions or functor classes that use these elements and whose parameters are passed by value if they are scalars, and by reference otherwise. The `MxV` class at the top of the code of Figure 1 contains an example of an HPL kernel implementing a SAXPY computation. In this kernel, each thread computes one position of the resulting array `y`.

Finally, the HPL host API contains functions that allow to discover the devices available and their properties as well as to request the execution of kernels on them. The most important function is `eval`, which runs a kernel in a device allowing to specify the global and local domains and the list of arguments. This is exemplified in the `main` function of Figure 1, which requests the execution of the SAXPY kernel using a global domain of 1000 threads divided in subdomains of 10 threads.

3. An optimizer for HPL

The HPL backend currently translates the HPL kernels invoked into OpenCL kernels at run-time. This is performed with the assistance of the Portable Expression Template Engine (PETE) [6], which is a portable C++ framework that lets users easily add expression-template functionality to container classes and perform complex expression manipulations [7]. HPL uses and extends this library to parse the tokens that compose an HPL kernel and turn them into strings containing the equivalent OpenCL kernel code.

In this work, this translation stage is equipped with an optimizer that is applied to a region of code selected by the user. As we will see, our tool applies five critical optimizations whose parameters are autotuned by means of iterative compilation. This requires major changes in the implementation of this stage. Figure 2 shows an overview of the workflow of the autotuner and the underlying optimizer. Within the optimizer, instead of using PETE to translate directly the HPL code into OpenCL, now the HPL code is loaded into an Abstract Syntax Tree (AST) representation of the kernel, which is step 1 of Figure 2. On top of this AST representation the optimizer applies a set of classical source-to-source code optimizations for codes in heterogeneous systems. These optimizations are tuned using a set of parameters whose values are set by the autotuner (step 1.1). Then, the optimizations are applied on the AST (step 2). Once the code is optimized, this AST representation is turned into the equivalent OpenCL kernel code (step 3). As last step, the new code version is compiled and its performance is measured. After this, the autotuner either proceeds to generate new combinations of tuning parameters in order to further explore the search space, or finishes and provides the best code version found. Currently our autotuner supports random and genetic algorithm searches, while the criteria for terminating the search is based on the time spent in the process, which is controlled by the user.

In order to take advantage of this optimizer, an HPL kernel must be written in a simple way, that is, without using vectorization, local memory or other optimization features. Rather, it must just encode the calculation of one point of the solution. In addition, the code that performs the computation must be enclosed inside a *compute* section, leaving initializations and other parts of the kernel outside this section. This hint, which just requires using the keyword `compute` before the code to optimize enclosed by curly brackets, as shown in Figure 3, gives valuable semantic information to the optimizer and it simplifies the optimization process.

Before the optimization process starts, the opti-

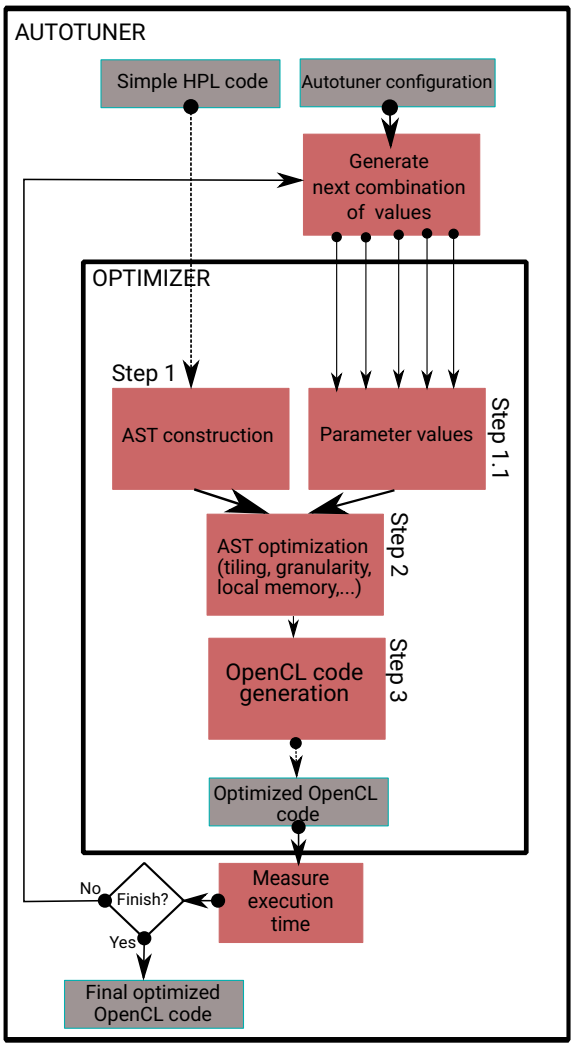


Figure 2: Workflow of the autotuner and the optimizer

mizer populates the AST with information of the access patterns that appear in the code. Such patterns are extracted from the analysis of the references to data structures located in global memory found within the compute section marked by the user. These references are classified according to their memory access patterns. In order to do that, the optimizer implements a simplified version of the analyzer described in [8], which uses the index expressions of each reference to classify them in one of these seven types, 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, so all the threads access the same point of this data structure.
2. `SinglePat`: The indexing expressions only contain global identifiers, so each thread accesses a different position of the data structure.
3. `InnerPat`: The indexing expressions only contain inner loop counters, thus, all the threads access the same slice of the data structure.
4. `RowPat`: One dimension is indexed using a global identifier and another one (the rightmost one) using a loop counter with stride 1. As a consequence, each thread traverses sequentially a different row of the data structure.
5. `ColPat`: A loop counter with stride 1 indexes the second rightmost dimension of the structure, and the rest are indexed using global identifiers. Each thread traverses a different slice of the data structure in column-major order.
6. `DepthPat`: A loop counter with stride 1 indexes the third rightmost dimension of the structure, while the rest are indexed by the corresponding global identifiers. Thus, each thread traverses a different 1D slice across planes in a 3D structure.
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 of the data structure around another one working as a pivot.

Once the pattern of each reference has been identified, the data structure accessed by the reference is classified as associated to the pattern of the reference. A data structure accessed by references with different patterns will be classified as associated to the most complex pattern found in its references. We will see the utilization of this classification during the explanation of the optimization process of the AST that follows.

It must be noted that codes that include references whose access pattern does not conform to one of the set previously described, the optimizer will not be able to process it. This may happen in a wide range of situations such as triangular loops, conditionals, indirections, complex linear indexing expressions, etc. Relatedly, our tool expects a clean simple coding style to recognize the patterns, so that possibilities such as for example flattening a multidimensional data structure and linearizing its access, may hide the underlying pattern from the analyzer and thus preclude the analysis.

3.1. Applying optimizations on the AST

According to the CUDA C Programming Guide [9], the optimization of a CUDA code has to focus on three basic strategies:

- Maximizing parallel execution to achieve maximum utilization.
- Optimizing memory usage to achieve maximum memory throughput.
- Optimizing instruction usage to achieve maximum instruction throughput.

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

1. The **tiling stage**, where the tiling technique is applied to the main loop nest 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 **loop unrolling** stage, where the innermost loop of the code in the compute section can be unrolled.

The maximization of the benefit from parallel execution based on granularity is targeted by stage 3. The strategy of optimizing 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 basic matrix multiplication kernel in Figure 3 will be used as a running example throughout this explanation.

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

Figure 3: MxM running example

```
1 ...
2 c[idy][idx] = 0.0f;
3 for(kk=0; kk<limk; kk+=tW0) {
4   limtile = min(kk+tW0, limk);
5   for(k=kk; k<limtile; k++) {
6     c[idy][idx] += a[idy][k] * b[k][idx];
7   }
8 }
9 ...
```

Figure 4: MxM running example tiled

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 simple version of the SAXPY code (see Figure 1) will not have such a loop, but a 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. Figure 4 shows the tiled version of the loop of the running example using a generic tile size of `tW0` iterations. The differences between Figures 3 and 4 exemplify the changes on the code involved by this transformation:

- The loop in the original kernel (`for k`) is replaced by two loops (`for kk` and `for k`). The two new loops traverse the same iteration space as the original loop in a coordinated fashion. Namely, the outer loop proceeds in steps of the size of the tile considered, thus marking the beginning of each new tile, while the inner loop iterates within the current tile.
- The tile size may not be a divisor of the size of the original loop. As a result, before proceeding to iterate within the current tile in `for k`,

we must compute a maximum correct upper bound for such loop (`limtile`).

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, 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 of these data structures 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.

The optimizer chooses the data structures that will be copied to local memory based on the information gathered in the AST about the access patterns. Namely, it selects the data structures with access patterns that are more complex than `SinglePat` to be loaded into local memory. Let us recall that in the `SinglePat` access pattern each thread accesses a different element of a data structure, thus there is no point in loading that information in 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, since both the `ColPat` and the `RowPat` access patterns are more complex than `SinglePat`, matrices `a` and `b` are selected to be loaded into local memory in this example. Once the selection is done, the optimizer follows four steps to transform the code: (1) the local memory counterpart data structures are declared, (2) the code snippets that copy data from global to local memory are generated, (3) the global references are replaced by local ones in the compute section of the kernel, and (4) if any of the local structure is updated, a code snippet to copy back that information to global memory is also generated. More details are now given about these steps.

The most challenging task of the first step, the declaration of the local array, is to find out the appropriate size of each dimension of the local data structure. These sizes depend on the type of access pattern followed by the memory references associated to the data structure, and on the tile and the grain sizes. Although the coarser grain adjustment transformation is applied in a subsequent step of the optimizer, the grain size is already set in step 1.1 of Figure 2.

Table 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 sizes of the local space for dimensions 0, 1 and 2 respectively. The parameters `tW0`, `tW1` and `tW2` are the tile sizes for the inner computing loops 0, 1 and 2, respectively, if tiling has been applied to them. If not, their values will be the length of the corresponding loops. The `bszx`, `bszy` and `bszz` parameters appear when the coarser grain adjustment 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 the block of iterations assigned to a given thread. If the technique is not applied, these parameters just take the value 1. Each parameter is associated to the loop whose counter indexes a given dimension. As 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 the local memory only the slice of the data structure that is going to be traversed by the threads of the current group. In our run-

Pattern	Dims		Size expression
InnerPat	1D	0	[tW0]
	2D	1	[tW1]
		0	[tW0]
	3D	2	[tW2]
		1	[tW1]
		0	[tW0]
RowPat	2D	1	[lszy*bszy]
		0	[tW0]
	3D	2	[lszz*bszz]
		1	[lszy*bszy]
		0	[tW0]
ColPat	2D	1	[tW0]
		0	[lszx*bszx]
	3D	2	[lszz*bszz]
		1	[tW0]
		0	[lszx*bszx]
DepthPat	3D	2	[tW0]
		1	[lszy*bszy]
		0	[lszx*bszx]
RadiusPat	1D	0	[lszx*bszx+tW0]
	2D	1	[lszy*bszy+tW1]
		0	[lszx*bszx+tW0]
	3D	2	[lszz*bszz+tW2]
		1	[lszy*bszy+tW1]
		0	[lszx*bszx+tW0]

Table 1: Size of each dimension of the local data structure

ning 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 of copying the information from global to local memory. We use copy mechanisms similar to those described in [10], 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, the optimizer turns all the references to the global version of each data structure in the compute section of the kernel to references to their local counterparts. In addition, the indexing of these references is adjusted to use local identifiers instead of global ones.

If the data structures that have been copied to the local memory are updated, the optimizer performs a fourth step to copy the information back to global memory. In this case, it uses 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 Figure 5 shows how our running example is adapted to use local memory. This example does not require the fourth step to copy

```
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 ...
```

Figure 5: MxM kernel using local memory

```
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];
```

Figure 6: Sequential version of a MxM operation

back the information to global memory, because the localized data structures are not updated.

In summary, the steps followed to apply this technique are:

- Choose which data structures will be loaded into local memory depending on their access pattern.
- Declare the local arrays (see Table 1).
- Replace the global references by local ones.
- Copy-back to global memory the updated local arrays.

Coarser grain adjustment

The next optimization tries to adjust the number of threads and, conversely, the amount of work made by each one. In order to do this, the optimizer has 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 programmer must provide a version of the kernel that computes just one point of the result. Therefore, this simple version minimizes the grain size and maximizes the number of threads required. As a result, such a kernel has fewer loops than its sequential version because the loops have been replaced by the parallel executions of the kernel. This can be seen in Figure 6, which is a sequential version of our running example. The single-point version of the same code, written as an HPL kernel in Figure 3, lacks the two outermost loops, those that index the resulting matrix, and keeps the innermost one, because it is required to compute a single point of the result.

As a first step of this transformation, the optimizer encloses the existing loops in the compute section in new loops, a pair per dimension of the global space, and the global identifiers are replaced by the counters of these loops in all the indexing expressions. This part of the transformation resembles the tiling optimization, but in this case different loops traverse different parts of the iteration

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

Figure 7: Coarser grain adjustment generic loop nest

```
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,Y); y++) {
4        for(x=xx; x<min(xx+2,X); 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 }
```

Figure 8: MxM kernel with coarse grain adjustment

space, which is distributed in a block-cyclic manner. This transformation enables the distribution of the work among fewer threads. Figure 7 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 simple version of the code. Each pair of loops in lines 1 and 4, lines 2 and 5, and lines 3 and 6, assigns the iterations following a block-cyclic distribution, where the block sizes are `bszz`, `bszy` and `bszx`, respectively.

Figure 8 shows a version of our running example with the loops added, where $\texttt{szx} = \texttt{N}/4$, $\texttt{szy} = \texttt{M}/4$, and $\texttt{bszx} = \texttt{bszy} = 2$. Let us recall that `N` and `M` are respectively the number of columns and rows of the resulting matrix `c`, while `szx` and `szy` are the total number of threads in the corresponding dimensions of the global domain of the kernel execution. In this case, the parallel execution requires 16 times fewer threads and each thread is going to execute four blocks (2×2) of four iterations (2×2) each, for each one of the two pairs of loops added. Previous research from [11] showed that the overhead introduced by these loops is not negligible. For this reason, the optimizer applies on top of this technique small optimizations such as 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

The efficient exploitation of the memory hierarchy usually implies an equally efficient usage of the private memory, which is commonly mapped to processor registers, as this alleviates the pressure on the slower memory levels. In order to do this, our optimizer maps to private memory the references that operate on memory positions that are both read and written by the kernel. The rationale for this heuristic is that write-only references offer no temporal locality. However, read-and-write ones exhibit sometimes great temporal locality and, anyway, its associated data need to be loaded into registers in order to be used as inputs. In addition, these data sets are often smaller than the read-only working sets, which makes them more amenable to placement in the small private memories. The aforementioned circumstances can be observed in the matrix product running example. This code updates each element of the destination matrix `c` using as inputs a full row from the input matrix `a` and a column of the input matrix `b`.

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. This structure is declared as a group of independent scalars. This complicates the transformed code but we have found that it guarantees that the private memory will be mapped to registers. Then, these private scalars must be filled in. After that, the global memory references must be replaced by private ones, and finally, if updated, the contents of the private scalars must be copied back to the corresponding positions of the global memory. Due to its similarity with the local memory exploitation stage, the details of this transformation are not included in this paper.

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 input kernel. This 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. Returning to our running example, Figure 9 shows an unrolled version of the innermost compute loop using a generic unroll factor, `uf`, which is assumed to divide exactly the `tW0` tile size previously applied.

```
...
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];
  }
  ...
}
...
```

Figure 9: Unrolling a previously tiled MxM loop

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 if the work distribution loops must be fully unrolled.

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

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

Table 2: Input parameters of the optimizer

4. Optimization parameters

The optimization stages described in the previous section depend on the values of a set of parameters. While some of these parameters decide whether an optimization is applied or not, others tune certain aspects of the optimization (for example, the unroll factor used in the unrolling technique). The values of these parameters largely influence the performance of the resulting code on a given device, and thus their values should be tuned for each platform.

Table 2 summarizes, for each possible number of dimensions of the problem, the parameters used by the optimizer that need to be fixed before the optimization process starts. A dash symbol (−) means that this value is not required in that case. The optimization parameters related to the workspace definition and the work distribution are listed first. There are parameters to define the global and local workspace (one per dimension of the workspace), as well as the block size (also one per dimension), since the work is going to be distributed in a block-cyclic manner among the threads. There is also a boolean that defines whether the work distribution loops must be totally unrolled or not.

The second group of parameters is related to the application of the tiling technique. They define the tile size (one per each nested compute loop), with an additional parameter to set an unroll factor for the innermost loop of that nest.

The last group of parameters is related to the exploitation of the local memory. In this case, only a flag that establishes whether the local memory is going to be used or not is required.

While these are the values that must be set before the optimization process starts, there are other decisions that are taken automatically during the optimization process. For example, the data structures that are going to be mapped to local memory are selected following the algorithm explained in Section 3.1, while the sizes of the local memory arrays depend on the tile size and the work distribution parameters, as explained in the same section.

Selecting good values for the aforementioned parameters is not a trivial task. Previous works [11, 12] have tried exhaustive or informed search strategies guided by the execution time that obtained good results. Our autotuner, represented in Figure 2, injects in the optimizer different combinations of values of the optimization parameters. The search is guided by the execution time, and the combinations can be generated randomly or using a genetic algorithm. The experiments in Section 5 show that both strategies are very effective, as they can generate good results in a reasonable time.

It is worth mentioning that our autotuner is completely integrated in the HPL framework and that we offer a command-line interface (CLI) to it. The parameters of this CLI are: basic data on the problem to optimize such as its number of dimensions or the size of the global domain in each dimension, and indications on the search process to apply such as the search strategy or the maximum search time allowed. For instance, `-d GPU -n 1024,1024 -k 1024 -s g[enetic] -t 3600` would indicate that: the code has to be run on the GPU (`-d`), it has two

dimensions, the size of each dimension of threads of the problem is 1024 (`-n`), the number of iterations to be processed by each thread is 1024 (`-k`), the genetic search is requested (`-s`), and the maximum search time allowed is 3600 seconds (`-t`). The user must build an object of type `OptimizerSettings` to hold this data, which is easily achieved by providing its constructor with the well-known `argc` and `arv` arguments of C++ programs, from which the object obtains the data provided by the command-line arguments. After this step, the user must just perform the `eval` invocation (see example in Figure 1) using the modifier `optimize`, which takes as input the `OptimizerSettings` built in the previous step to control and define the optimization process. This way, the optimization can be simply launched in a couple of lines such as

```
OptimizerSettings opt_settings(argc, argv);
eval(kernel).optimize(opt_settings)(args);
```

It also deserves to be mentioned that, while transparent to the user, the `OptimizerSettings` object is also used to store the optimization parameters described in Table 2, which are for internal use of the autotuner. The runtime updates the values of these fields in order to command the optimizer to generate the kernel version required in each step of the search algorithm.

5. Experimental results

This section contains the validation of the effectiveness of our optimizer. The purpose of this validation is to prove that the optimizer can generate faster versions of a wide 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 [13] (DCS3D), the matrix multiplication (MATMUL) used as a running example along Section 3.1, a single time step of an N-body simulation [14] (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, the solution spaces of 3DCONV and DCS3D have three dimensions. Regarding the input simple kernels that implement these problems, 1DCONV, NBODY, DCS3D, MATMUL, SYRK and SYR2K 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 3 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, called dimension sizes in the table, and the nested inner loops of each problem, which are the computing loop sizes in the table.

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

- **K20**: 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.
- **FirePro**: 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.
- **Xeon Phi**: 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.

Let us first analyze the speedups achieved. Figure 10 shows the speedups of the optimized version with respect to the simple version provided by the user. There is a separate graph for each one of the three problem sizes: small, medium and large. Within each graph, the speedups for the three platforms using different search strategies are found. Namely, as mentioned before, our optimizer can currently perform either a random or a genetic algorithm search, and we have tried 5, 15 and 60 minutes long searches of both kinds. The bar with the color associated to a specific device and search strategy shows the speedup achieved after a 5 minutes search. The additional speedup obtained thanks to extending the search to 15 and 60 minutes is represented with the stacked magenta and cyan bars, respectively. Notice that a logarithmic scale

	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	16384	16384	32768	32768	65536	65536
2DCONV	512 × 512	256 × 256	1024 × 1024	256 × 256	2048 × 2048	256 × 256
3DCONV	32 × 32 × 32	32 × 32 × 32	64 × 64 × 64	64 × 64 × 64	128 × 128 × 128	128 × 128 × 128
NBODY	16384	16384	32768	32768	65536	65536
DCS3D	32 × 32 × 32	2048	64 × 64 × 64	4096	128 × 128 × 128	8192
MATMUL	1024 × 1024	1024	2048 × 2048	2048	4096 × 4096	4096
SYRK	1024 × 1024	1024	2048 × 2048	2048	4096 × 4096	4096 × 4096
SYR2K	1024 × 1024	1024	2048 × 2048	2048	4096 × 4096	4096 × 4096

Table 3: Size classification of test cases run in the experiments

		Average speedup
Global		16.01
Kernels	1DCONV	5.62
	2DCONV	3.58
	3DCONV	3.86
	DCS3D	1.94
	MATMUL	9.56
	NBODY	1.55
	SYRK	46.23
	SYR2K	55.74
Search methods	Genetic	15.84
	Random	16.18
Platforms	K20	12.21
	FirePro	32.31
	Xeon Phi	5.09
Problem sizes	Small	17.29
	Medium	19.25
	Large	11.48

Table 4: Average speedup of the optimized versions

was used so that the large diversity of speedups achieved could be observed.

We summarize these results in two ways. First, Figure 11 shows the average speedups across all the benchmarks for each platform and search strategy using the same approach as Figure 10 to label them and identify the search time used. It is noticeable that just devoting only five minutes to the autotuning, the codes can become between 2.8 times faster on average, in the random search of large problems in the K20, and 38.4, in the random search of medium problems in the FirePro. This single result coupled with the ease of use of the tool justifies its interest. Investing a very reasonable ten additional minutes in the autotuning allows to reach average speedups between 3.1, again the random search of large problems in the K20, and 53.5, again in the random search of medium problems in the Firepro. Finally, when one hour is devoted to the autotuning, which is still a very reasonable time [12], the average speedup is in a range within 3.6, attained by the random search for medium problems in the Xeon Phi, and 55, again in the random search of medium problems in the FirePro.

A second summarization is provided by Table 4, which shows the average speedups across more axis of the experiment. Considering all the problem sizes, search strategies and search times tried, the optimizer gets a global average speedup of 16.01. If each code is considered individually, the speedups go from the average 1.55 of NBODY to the 55.74 of SYR2K. There is not a big difference between using a random search or a genetic one, although the genetic search behaves slightly better. The AMD FirePro is the platform that benefits the most from the optimizer, and the Xeon Phi is the one with the more discrete, although still noticeable, speedups. This last result, also clearly illustrated in Figures 10 and 11, makes sense since Xeon Phis have no actual software managed local memory, but a traditional hardware managed memory hierarchy with two levels of caches, the L2 cache being much more generous than the fast memories found in GPUs. As a result, in general this machine is less sensitive than GPUs to our optimizations focused on the exploitation of the fast memories. Another conclusion from Table 4 and the aforementioned figures is that Small and Medium size problems get better speedups than Large ones. This effect is related to some extent to our experimental methodology, since given a fixed search time, the number of code variants tested, and thus the ability to improve the baseline code, will be smaller the larger the problem size considered. This way, in our experiments, the average number of variants explored

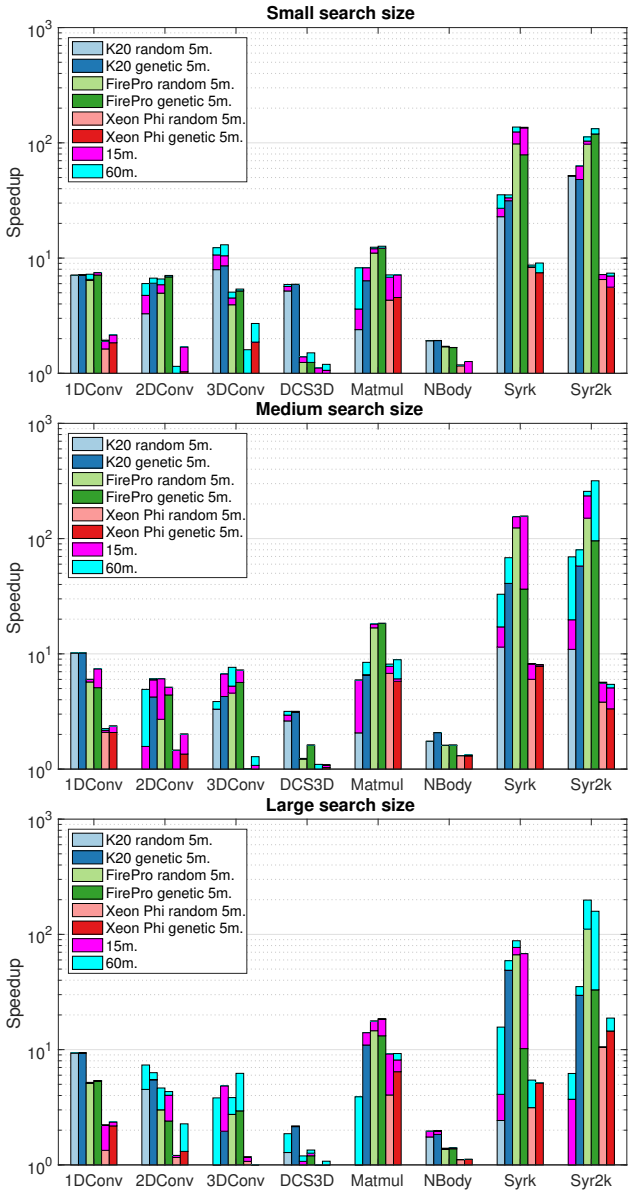


Figure 10: Speedup achieved using the optimizer for the 8 benchmarks, in the three platforms using the small, medium and large problems sizes. The search strategies are: Random (5 minutes), Genetic (5 minutes), 15 minutes and 60 minutes.

for the large problems across the three search times considered was 488, while this value grew to 1364 for the medium size problems and 2971 for the small problem sizes.

Another point of view is provided by Figure 12, which shows in which percentage of the optimization tests performed the speedup achieved was within a given range. There are separate bars for the random and the genetic searches, as well as for

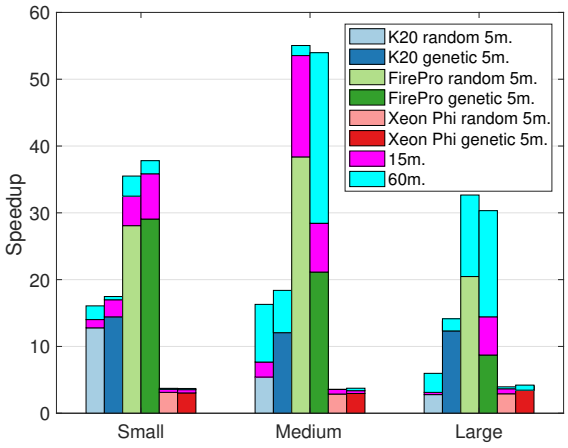


Figure 11: Average speedups

the global average. The darkest color corresponds to the percentage of optimization processes where the optimizer could not improve the baseline. The next three areas correspond to increasing but still modest levels of optimization, reaching speedups of up to 1.1, 1.25 and 1.5, respectively. The top yellow area is associated to the percentage of tests where the speedup was above 1.5. As we can see, the optimizer achieves considerable speedups around 80% of the times, no matter which search strategy is used. However, we can see that the genetic search is somewhat more likely to achieve a considerable speedup than the random one.

Table 5 gives us an idea of the values chosen by the autotuner for the parameters in Table 2. These values are displayed separately for each platform and problem size, for the SYR2K benchmark using the 60 minutes genetic search. As this benchmark implements a 2D problem, the `szz`, `lszz` and `tW2` parameters are always set to 0. Besides that, in general the size of the global space grows with the problem size, and some architectures (FirePro and Phi) benefit more from non-square global spaces than others (K20). Something similar happens with the local space. Regarding the tile size, it is worth to notice that the Phi prefers much larger values than the GPUs. Unroll factors are usually below 4, except for one case (Phi, Large) and the local memory is worth to be used just in the GPUs.

Another insightful view of the results is provided by Figure 13, which represents the impact of the application of each optimization on the final speedup. Namely, the contribution of each tuning stage is expressed as a ratio of the overall speedup. The

Platform (Problem Size)	Global size			Local size			Block size			Unroll factor	Local memory
	szx	szy	szz	lszx	lszy	lszz	tW0	tW1	tW2		
K20 (S)	256	256	0	8	32	0	8	0	0	4	Yes
K20 (M)	256	256	0	8	32	0	8	0	0	2	Yes
K20 (L)	2048	4096	0	32	16	0	16	0	0	4	Yes
FirePro (S)	256	256	0	8	32	0	8	0	0	4	Yes
FirePro (M)	128	256	0	8	16	0	16	0	0	0	Yes
FirePro (L)	128	4096	0	2	128	0	16	0	0	2	Yes
Phi (S)	64	1024	0	1	2	0	32	0	0	4	No
Phi (M)	256	2048	0	2	4	0	2048	0	0	4	No
Phi (L)	4096	1024	0	1	64	0	2048	0	0	8	No

Table 5: Values of the main optimization parameters of Table 2 for the SYR2K benchmark using the 60 minutes genetic search, for each problem size

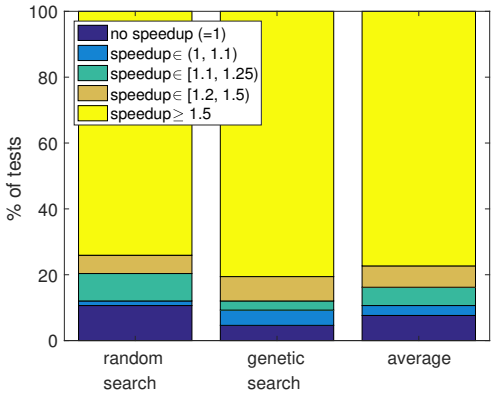


Figure 12: Contribution of the tuning of each optimization parameter to the overall speedup.

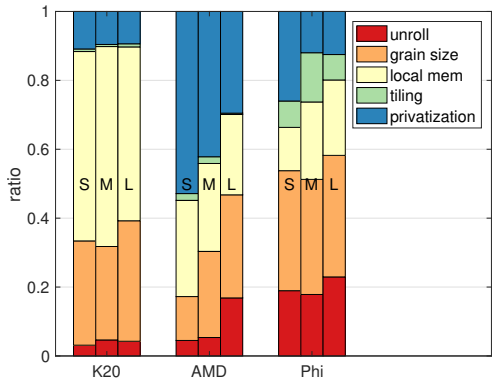


Figure 13: Relative impact of each optimization on the final speedup

figure contains the average values observed in all the benchmarks for each one of the three platforms and for each problem size considered. The contribution to the speedup of each stage was computed considering the additional speedup it provides when applied after the preceding stages in the order explained in Section 3. In this comparison neither of the two versions considered when measuring the additional impact of an optimization stage applies the stages that follow the one being measured. The reason is that sometimes the optimization stages of our optimizer depend on preceding ones, and thus they cannot be applied if those they depend on are missing, which makes this strategy the fair way to estimate the isolated impact of each optimization. It must be noted however, that the individual impact of an optimization measured in this way can be much smaller than the one it can provide when combined with subsequent ones, particularly when, as we have just explained, they are enablers for some of those optimizations.

6. Comparison to other approaches

The optimization tool presented in this paper consists of: (1) an autotuner that finds appropriate values for the optimization parameters, and (2), an optimizer that performs the optimizations of the HPL kernel code according to these values. It is reasonable to wonder how good is our tool in these two aspects. This is discussed in turn in Sections 6.1 and 6.2.

6.1. HPL built-in autotuner vs. OpenTuner

In order to assess the quality of our autotuner, we need to compare it with another well-established one such as OpenTuner [15]. The integration of our optimizer with OpenTuner required to write an ad-hoc Python tool that uses the OpenTuner API. The tool defines a search space with all the candidate values for the optimization parameters, and the validity constraints that define which combinations of values are valid and which are not. These

constraints generally depend on user-provided information (see Section 4), on the capabilities of the target device, queried via PyOpenCL [16] and on specific information that is extracted by the optimizer when it analyzes the input code. Contrary to what happens with our built-in autotuner, this latter information cannot be easily provided to OpenTuner, as it is generated once the code is analyzed by HPL. To overcome this issue, we chose the MATMUL problem as a single comparison case and we hardcoded this information for this specific kernel into the OpenTuner tool.

The OpenTuner Python tool uses the default search method, which is the best option according to the authors of that API. Finally, the information about the execution time of the kernel must be fed up to OpenTuner in a JSON-formatted file.

Figure 14 shows a comparison of the speedups achieved by the best kernels found by the genetic algorithm of our HPL autotuner and by those found by the evolutionary techniques of OpenTuner for the MATMUL benchmark. The comparison is done for each problem size and target device.

The performance of OpenTuner is on average better than that of our built-in autotuner. However, in our opinion this is not the only aspect to consider. Let us recall that, our built-in autotuner implements a plain genetic algorithm, whereas the default search method of OpenTuner combines several evolutionary techniques and it is able to leap from one to another on the go depending on the progress of the search. Thus, in a given search space, the more elaborated a method is, the more it is expected to benefit from the search time and obtain better versions. However, the random nature of evolutionary search techniques may explain that variability of the results. For instance, independent runs of either the same or different search processes do not ensure to explore the same versions in a given search space. This behavior may lead to well-known pitfalls of these techniques such as starting the search process evaluating quite bad solutions or reaching local optima too quickly. Also, the seamless integration of our HPL autotuner and the optimizer makes the overhead introduced by the launch of each kernel and the retrieval of its execution time is quite lower than in the OpenTuner Python tool. In this latter tool, the Python code must run the HPL optimizer in a special mode to request the evaluation of a kernel version, and then parse the standard output to get the execution time.

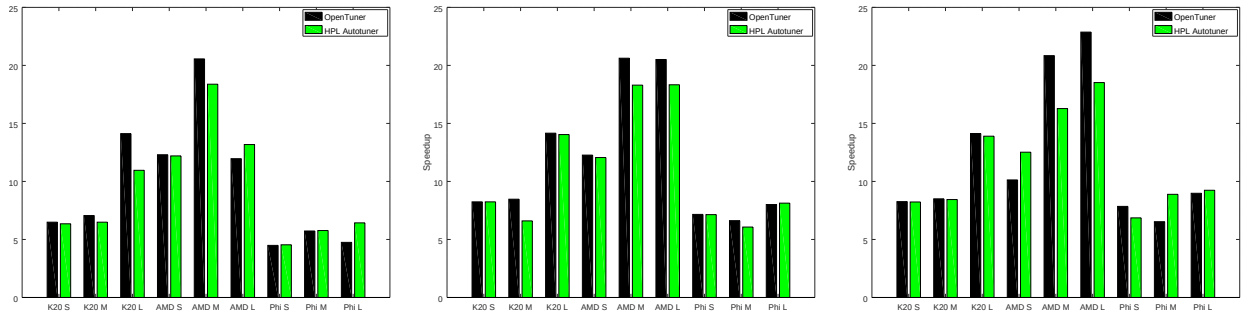
Summarizing, we find that OpenTuner offers more powerful evolutionary search methods than those provided by our autotuner. However, we consider that this does not balance out the additional programming effort needed to achieve a full and smooth integration of an OpenTuner-based search component with the HPL parts for kernel analysis, generation and evaluation. Moreover, the tighter integration of our autotuner allows it to perform a somewhat faster exploration of the search space that allow it to obtain results reasonably competitive with those of OpenTuner.

6.2. HPL optimizer vs. clBLAS

In addition, it is interesting to compare the performance of the optimized kernels generated by our tool with that of kernels provided by a state-of-the-art library in order to assess how good our automatic optimizer is. In this case, we have decided to use as reference the performance achieved by the same routines in the clBLAS library [17]. This domain-specific library provides hand-tuned implementations of basic linear algebra routines, thus, it does not provide baselines for all our benchmarks. Because of that, this comparison is based on the MATMUL, SYRK and SYR2K kernels, which are the ones also available in clBLAS.

Table 6 contains the speedup of our library with respect to clBLAS for the three aforementioned benchmarks on the three tested platforms. The cases where our optimizer is slower than clBLAS are in bold font. In these experiments the last clBLAS stable version (2.12) is used and the codes are based on the samples provided for each routine by clBLAS, which have been only modified to accept any problem size and to record the execution time of the routine. The speedup compares the performance of the best kernel obtained with our tool using the 60 minutes genetic search with the time required by clBLAS to complete the execution of the corresponding routine. In the case of clBLAS, ten executions of the routines are done. Then, the first one is disregarded, because it takes a longer time, and the average time of the remaining 9 executions is calculated to be taken as reference.

The results show that our optimizer generates competitive kernels, as in fact our kernel is slower than the clBLAS routine only in 4 out of the 27 cases (14.8% of the tests). The general tendency shows that clBLAS performance is worse for small problem sizes and more competitive for larger problem sizes. Regarding the differences in the performance on the different platforms, the performance



a. 5m search b. 15m search c. 60m search

Figure 14: Comparison of using OpenTuner vs using the HPL autotuner, within the HPL optimizer

of our kernels is clearly better than clBLAS in the Xeon Phi, almost the same in the K20, and better in the FirePro. The reason of the large difference in the performance for Xeon Phi is probably that this platform has not been considered at all by the clBLAS team.

7. Related work

We have already addressed the problem of automatic optimization and autotuning for heterogeneous systems in our previous works [11, 12, 18, 19]. In OCLoptimizer [11] the optimization process is driven through pragmas that are introduced in the code by the programmer itself, and the only optimizations that can be performed are unroll and unroll-and-jam. In [12, 18, 19] the run-time code generation capabilities of HPL are explicitly exploited by an expert programmer who has to manually write each kernel using a series of techniques that give place to the application of different code transformations. This way the resulting code can be self-tuned using an external search algorithm that operates on the parameters that control the transformations.

Other authors have addressed the problem of automatic code optimizations from many perspectives [20]. This way, some approaches work as black boxes whose input is a code written in a traditional language such as C, C++ or FORTRAN. This is the case of a multi-objective auto-tuning framework developed by Jordan et al. [21] on top of the Insieme [22] compiler infrastructure.

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 that can optimize OpenCL kernels in multiple ways. For instance, Fang et al. propose Sesame [23], 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 [24] or the local memory hardware usually included in many-core architectures [8] have in the performance of OpenCL kernels. The memory access pattern classification implemented in our optimizer is based on 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 [10], whereas the other is able to rewrite OpenCL codes that already used local memory in a quite architecture- or device-specific way [25]. 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 optimizer tackles this issue.

The Many-Core Levels (MCL) framework, oriented to different kinds of many-core devices, was built by Hijma et al. as an implementation of their stepwise-refinement for performance methodology [26]. 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.

Steuwer et al. propose in [27] a high-level functional language embedded in Scala to implement simple problem descriptions. This forces the pro-

Code/Size	K20			FirePro			Xeon Phi		
	S	M	L	S	M	L	S	M	L
MATMUL	1.9	1.0	**0.8**	8.2	5.3	1.2	18.2	9.7	6.1
SYRK	1.4	1.3	1.2	3.1	2.5	1.1	6.7	3.8	1.6
SYR2K	**0.6**	1.5	**0.2**	3.1	2.1	1.1	4.1	2.3	**0.9**

Table 6: Speedup of our best performing kernel w.r.t. the clBLAS routine

grammers not only to rewrite their kernels but also to leap from the imperative to the functional paradigm, which may become uncomfortable for many users.

There are also domain-specific programming frameworks that provide optimization capabilities. An example is Halide [28], 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 European AutoTune project has developed the Periscope Tuning Framework which combines automated performance analysis and performance tuning. A plugin of this framework, presented in [29], succeeded to optimize the performance of CPU, GPUs and Xeon Phi. They autotune the compilation flags and the workspace configuration for each specific platform. Regarding the compilation flags, the search space is explored using an individual search strategy, where parameters are autotuned one at a time in decreasing order of importance. Regarding the tuning of the workspace, the user provides an initial tuning specification, which is going to narrow intelligently the search space to be explored, and then the search space is traversed using a local or a global strategy, depending on whether the kernels in the application are autotuned separately or jointly. The validation is conducted using several benchmarks of the Rodinia Suite and the Direct Coulomb summation benchmark on several platforms. They report modest speedups, always below 2x.

Other works focus on traversing the search space of possible values of the parameters in the most efficient manner. For instance, the authors of [30] use machine learning techniques to explore the search space. In this work, the parametrized optimized benchmarks are written by hand. Thus, the authors focus on using machine learning techniques instead of performance models to narrow the search space. This smaller search space is then traversed exhaustively to find the best values for the parameters. The training of the neural network has to be done for each benchmark separately, and even for each problem size, if we want to increase the accuracy. That is an important limitation of this approach. Also, Bruel et al. [31] use an autotuner to find the most appropriate values for the optimization flags of the Nvidia compiler. Finally, in the same vein, Kernel Tuner [32] is an easy-to-use tool for testing and auto-tuning OpenCL, CUDA, and C kernels. In this proposal the user is responsible for generating a tunable version of the kernel and for indicating which are the tunable parameters. The contribution focuses on proposing and comparing the performance of different search strategies, including a genetic algorithm. The evaluation only relies on two codes (a 2D-convolution, and GEMM) and a single GPU.

8. Conclusions

While there are a number of tools that provide varying degrees of functional portability to codes for heterogeneous devices, performance portability continues to be an elusive target. In this paper we extend one of such tools, the Heterogeneous Programming Library (HPL), with a runtime that applies some of the most common optimizations to a simple code provided by the user in order to generate a variant adapted to the device at hand. The implementation of our optimizer required important modifications in HPL, as we modified extensively the module that translates the kernels written in the HPL C++-based language into working OpenCL codes. The purpose of our changes was to first generate an AST representation of the code and later analyze and modify it in order to apply different code transformations. Namely, our optimizer can tile and unroll the compute loops,

cache shared data on the local memory of the devices when available, coarsen the granularity of the simple input kernel, and reduce memory access contention by computing results in the private memory regions of the devices.

A last component of our optimizer is an integrated autotuner engine that is in charge of finding proper values for the set of parameters that control the optimizer transformations. Such parameters range from booleans that determine whether a certain optimization must be applied or not to degrees of granularity per task or tile sizes per dimension. Our autotuner engine traverses the search space by applying iterative compilation and relying on runtime measurements in the target architecture in order to take its decisions. Currently, the engine supports random and genetic algorithm searches.

Experiments based on 8 benchmarks and three different platforms show that the optimized versions generated are on average 16 times faster than the input simple kernels. As expected, the genetic algorithm search provides better results than the random search. While all the platforms benefit from the optimizer, it is worth to mention the 32x average speedup obtained for the AMD FirePro platform, the averages found for a Nvidia K20 GPU and an Intel Xeon Phi being 12.2 and 5.1, respectively. We also used OpenTuner as an alternative autotuner for our optimizer. The integration of OpenTuner with our optimizer was not easy, as we had to patch by hand this integration in order to provide OpenTuner with the same information as our own autouner. The performance of the kernels generated for some of our benchmarks was also compared to the performance of the corresponding routines of a state-of-the-art optimized library such as clBLAS. The conclusion of this experiment is that the performance of the kernels generated by our optimizer is usually better than that of clBLAS.

An interesting future work would be to implement an algorithm able to select the optimization values by itself, which would 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, such as explicit loop vectorization or exploiting local memory to compute intermediate results. Finally, more search strategies as well as heuristics that prune the search processes, making them more efficient, could be also implemented.

Acknowledgements

This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the EU (TIN2016-75845-P), and by the Government of Galicia (Xunta de Galicia) co-founded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups (ED431C 2017/04) as well as under Xunta de Galicia and FEDER funds of the EU (Centro de Investigación de Galicia accreditation 2019-2022, ref. ED431G2019/01)

References

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

[2] OpenACC-Standard.org, The OpenACC Application Programming Interface Version 2.5 (Oct 2015).

[3] S. Ghike, R. Gran, M. J. Garzarán, D. Padua, Directive-based compilers for GPUs, in: 27th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC 2014), 2014, pp. 19–35.

[4] M. Viñas, Z. Bozkus, B. B. Fraguela, Exploiting heterogeneous parallelism with the Heterogeneous Programming Library, J. Parallel Distrib. Comput. 73 (12) (2013) 1627–1638.

[5] P. Faber, A. Größlinger, A comparison of GPGPU computing frameworks on embedded systems, IFAC-PapersOnLine 48 (4) (2015) 240–245, 13th IFAC and IEEE Conf. on Programmable Devices and Embedded Systems (PDES 2015).

[6] S. Haney, J. Crotinger, S. Karmesin, S. Smith, PETE: The Portable Expression Template Engine, in: Technical Report LA-UR-99-777, Los Alamos National Laboratory, 1999.

[7] 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), 1999, pp. 13–18.

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

[9] CUDA C Programming Guide, `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, [Online; accessed 29-december-2016] (2016).

[10] J. Fang, A. L. Varbanescu, J. Shen, 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, 2013.

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

[12] J. F. Fabeiro, D. Andrade, B. B. Fraguela, Writing a performance-portable matrix multiplication, Parallel Computing 52 (2016) 65–77.

[13] 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.

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

[15] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, S. Amarasinghe, OpenTuner: An extensible framework for program autotuning, in: 23rd Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'14), Edmonton, Canada, 2014, pp. 303–315.

[16] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation, Parallel Computing 38 (3) (2012) 157 – 174.

[17] clBLAS, `https://github.com/clMathLibraries/clBLAS`, [Online; accessed 3-July-2015] (2013).

[18] J. F. Fabeiro, D. Andrade, B. B. Fraguela, R. Doallo, How to write performance portable codes using the heterogeneous programming library, in: 19th Workshop on Compilers for Parallel Computing, Valladolid, Spain, 2016.

[19] J. F. Fabeiro, D. Andrade, B. B. Fraguela, R. Doallo, Writing self-adaptive codes for heterogeneous systems, in: Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings, 2014, pp. 800–811. doi:10.1007/978-3-319-09873-9_67.
URL `https://doi.org/10.1007/978-3-319-09873-9_67`

[20] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, R. Vuduc, Autotuning in High-Performance Computing Applications, Proceedings of the IEEE 106 (11) (2018) 2068–2083.

[21] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, H. Moritsch, A Multi-objective Auto-tuning Framework for Parallel Codes, in: Intl. Conf. on High Performance Computing, Networking, Storage and Analysis, SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 10:1–10:12.

[22] Parallel Systems Group, University of Innsbruck, Insieme Compiler and Runtime Infrastructure, `http://insieme-compiler.org`, [Online; accessed 6-February-2017].

[23] J. Fang, A. L. Varbanescu, H. Sips, Sesame: A User-Transparent Optimizing Framework for Many-Core Processors, in: 2013 13th IEEE/ACM Intl. Symp. on Cluster, Cloud, and Grid Computing, 2013, pp. 70–73.

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

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

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

[27] M. Steuwer, C. Fensch, S. Lindley, C. Dubach, Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code, in: 20th ACM SIGPLAN Intl. Conf. on Functional Programming, ICFP 2015, 2015, pp. 205–217.

[28] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, S. Amarasinghe, Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines, in: 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '13, 2013, pp. 519–530.

[29] E. Bajrovic, R. Mijakovic, J. Dokulil, S. Benkner, M. Gerndt, Tuning opencl applications with the periscope tuning framework, in: 2016 49th Hawaii International Conference on System Sciences (HICSS), 2016, pp. 5752–5761.

[30] T. L. Falch, A. C. Elster, Machine learning-based autotuning for enhanced performance portability of opencl applications, Concurrency and Computation: Practice and Experience 29 (8).

[31] P. Bruel, M. Amarís, A. Goldman, Autotuning cuda compiler parameters for heterogeneous applications using the opentuner framework, Concurrency and Computation: Practice and Experience 29 (22) (2017) e3973.

[32] B. van Werkhoven, Kernel tuner: A search-optimizing gpu code auto-tuner, Future Generation Computer Systems 90 (2019) 347 – 358.