Automatic generation of optimized OpenCL codes using OCLoptimizer

Jorge F. Fabeiro, Diego Andrade, Basilio B. Fraguela and Ramón Doallo

Computer Architecture Group, Universidade da Coruña, Spain
Email: {jorge.fernandez.fabeiro,diego.andrade,basilio.fraguela,ramon.doallo}@udc.es

The eruption of multicore processors and several kinds of accelerators has generalized the interest in parallel programming. The OpenCL standard is very appealing because it provides code portability across most of these platforms. It defines a programming model where a host code requests the execution of kernels in computational devices. Unfortunately, the host API of OpenCL is quite verbose, which makes the development of its host code tedious and error-prone. More importantly, OpenCL does not provide automatic performance portability. As a result, users have to hand-tune OpenCL codes for each specific device, which implies trying different versions of the kernels and task partition granularities. As an answer to this situation we present OCLoptimizer, a tool that automatically generates host codes and optimizes OpenCL kernels for each specific target device based on a user provided configuration file. This configuration file describes basic kernel characteristics and annotations in the kernels that indicate the code transformations to test. Our tool can explore different granularities for the problem decomposition as well as different alternatives for the kernel. This exploration is performed by means of an iterative optimization process whose parameters and search strategy are defined by the user specifications. Support for OpenCL codes composed of multiple kernels is also provided by the tool. Experiments performed on multicore CPUs and different accelerators show that the tool is very effective, generating codes with an average speedup of 2.54 with respect to baseline hand-tuned implementations, in single kernel codes, and 1.79 in a code with multiple kernels.

Keywords: OpenCL ; performance portability ; genetic algorithms ; iterative optimization

Received 00 January 2009; revised 00 Month 2009

1. INTRODUCTION

The proliferation of several kinds of accelerators, with very interesting properties in terms of speedup and power consumption per flop with respect to traditional multicore processors, has sparked great interest in heterogeneous computing. The large variety of accelerators have given rise to different programming environments [1, 2] particularly adapted to the characteristics and limitations of each platform, thereby being specific to one vendor or family of devices. This restricts the portability of the codes and places into question the effort needed for their development. The exception is the OpenCL standard [3], which can be used to program a wide range of devices, covering the most important systems used nowadays.

OpenCL provides code portability by breaking applications into two well defined portions. First, a host code, which runs in a general purpose CPU. This code discovers the computational devices available in the system and interacts with them using a very generic API provided by the standard. Second, the computational kernels to run in the devices, which are compiled at run-time for each specific device under the request of the host. A weakness of OpenCL is that its host API is intended for low-level purposes, which coupled with the large number of items that the host has to manage (devices, kernels, memory objects, command queues, etc.), gives place to verbose and error-prone host codes [4]. Recently, new higher level APIs such as Computing-Language-Utility (CLU) or the SYCL specification [5] have emerged to facilitate the access to OpenCL capabilities. However, the largest shortcoming of OpenCL is that while its programs enjoy code portability, this is not the case for the performance. OpenCL provides a framework that enables performance portability on top of it, but this is not accomplished automatically. In fact OpenCL applications that perform adequately on a given device often require major changes even to just perform reasonably well in another device [6, 7, 8]. As a result,

the current situation is that users must hand-tune and write different kernels for different devices. These optimizations affect not only the kernel codes, but also the host code, as this is the one in charge of defining the workspace for each kernel run. This workspace consists of two index spaces, one for the global number of threads to use and another one for the number of threads in each work-group. A work-group is a set of threads that can cooperate. Both index spaces control the granularity of the parallel executions.

OCLoptimizer [9] is a source-to-source iterative optimization tool that deals with this situation. This tool is based on the annotated source code of a kernel, and a configuration file describing its most important characteristics. The tool uses this information to generate an optimized version of the kernel for a selected device as well as a suitable host code for it. The annotations in the kernel take the form of compiler directives that specify code transformations to test, and ranges of parameters to explore for those optimizations. OCLoptimizer follows an iterative compilation process seeking the best combination of compiler transformations and their parameters for each specific platform. The preliminary evaluation in [9], based only on applying the unroll transformation to the kernels and using a single CPU, yielded promising results.

In this paper we describe OCLoptimizer, particularly focusing on its extension to couple the selection of the best global and work-group index spaces to use in the kernel runs with the optimization of the annotated kernel sources. The resulting tool is evaluated on a multicore CPU (Intel Xeon E5-2660 Sandy Bridge), a GPU (NVIDIA Tesla Kepler K20m) and the new Intel Xeon Phi. The tool has also been extended to optimize OpenCL codes composed of multiple kernels through an iterative process where each kernel is optimized independently. Meanwhile, inter-dependent kernels are optimized taking into account their dependencies. The experimental results show that OCLoptimizer is effective for generating optimized OpenCL codes for all the platforms. Namely, it provides kernels with an average speedup of 2.54 with respect to baseline hand-tuned OpenCL implementations, in single kernel codes, and 1.79 in a code composed of several kernels.

The rest of this paper is organized as follows. Section 2 describes the OCLoptimizer tool and its inputs. Section 3 describes the support for OpenCL codes composed of multiple kernels. Then, Section 4 presents our experimental results, followed by a discussion on related work in Section 5. Finally, Section 6 concludes the paper.

2. THE OCLOPTIMIZER TOOL

OCLoptimizer is a source-to-source iterative optimization tool for OpenCL. As Fig. 1 depicts, our tool performs three main steps based on an annotated source code of a kernel and a configuration file, both required as inputs. First, it generates a suitable host code for the kernel. Then, it performs two searches driven by the exeenqcution time in order to optimize the OpenCL code for the platform where the tool is executed. The first search constitutes the second step of the tool. Its aim is to select the best global index space for the kernel, which determines the number of threads that run it in parallel, and an optimized local index space, which controls the number of threads per work-group. We call the combination of these two index spaces, which have between one and three dimensions, the workspace of the kernel. Finally, in the third step, OCLoptimizer runs an iterative compilation process driven by the annotations of the user in the source code, generating an optimized version of the kernel as a result.

The host generation stage only requires the specifications in the configuration file, described in Appendix A. The generated host code is a stand-alone program with all stages required to run an OpenCL kernel (a short description of them can be found in [4]). The initialization of the kernel inputs whose value is not specified in the configuration file may be random or performed by means of a code provided by the user, as Appendix A explains. The host code receives as arguments the parameters that define the workspace configuration of the kernel, namely, the global and the local sizes for each dimension of the workspace. This facilitates the search of an optimized workspace configuration, as it eliminates the need to recompile the host code. We now explain in detail the two search processes performed by the tool.

Example 1. A vector addition code ($A = B + C$) will be used as a running example through this paper. The input configuration file for this example is shown in Appendix A. This file indicates that the sizes of the arrays are 1024, the code must be optimized for a GPU, the input arrays must be initialized randomly, the local and global workspaces have one dimension, and the kernel receives four parameters: an scalar `n`, which takes the value of the arrays size, and the three arrays involved in the computation, `A`, `B` and `C`.

The generated host code, which is not included in the paper due to space reasons, starts with the initialization of the inputs of the kernel. Then, all the steps of a usual OpenCL host code are performed (platforms discovery, context creation and devices discovery). Once a device of the type specified in the configuration file is selected, the kernel is loaded and compiled. Then, the array inputs specified in the configuration file are transferred to the device through a command queue. The generated host code and the kernel are written for a generic workspace configuration which is passed as a parameter to the host code. This workspace configuration is used to enqueue the kernel. Finally, the host code enqueues the commands to read the results generated by the kernel ■

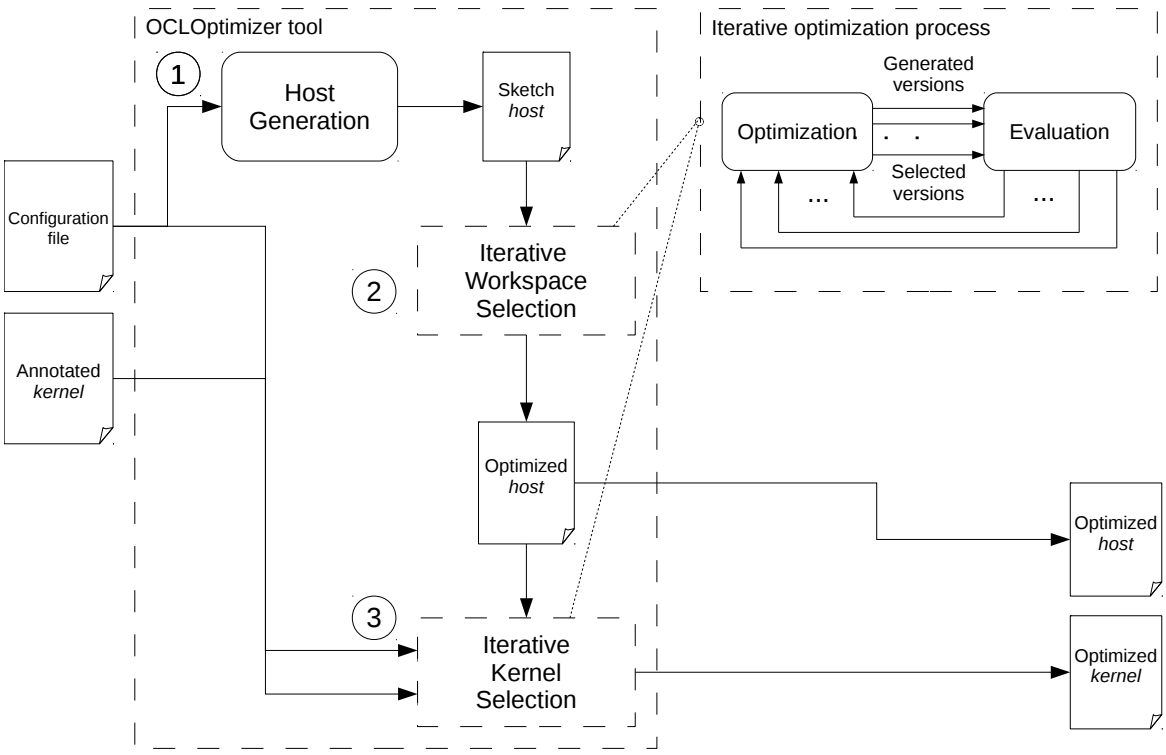

FIGURE 1. General workflow of OCLoptimizer

2.1. Workspace optimization

The configuration file described in Appendix A indicates the number of dimensions of the kernel workspace as well as the minimum and the maximum values that the tool has to explore for each dimension of the global and the local index spaces. OCLoptimizer chooses the best combination of sizes in each dimension of both index spaces within these limits by means of a search process guided by the execution time of the kernel in the device chosen in the configuration file. The user can choose between two search strategies for this process, exhaustive search (ES) or a genetic algorithm (GA) [10]. We now discuss them in turn.

Figure 2 summarizes the steps followed by the exhaustive search of an optimized kernel workspace configuration. First, the tool generates a list of possible workspace configurations taking into account the ranges of values specified by the user in the configuration file. In order to reduce the number of combinations, the points considered for each dimension i of a given index space are $m_i, 2m_i, 4m_i, 8m_i, \ldots M_i$, where m_i and M_i are the minimum and the maximum value specified for dimension i of that index space in the configuration file, respectively. Some of the configurations generated are impossible and they are thus discarded in advance. For example, the local workspace must be a divisor of the the global one in every dimension. Also, some devices or OpenCL implementations limit the maximum work-group size and its maximum value for each one of its dimensions. OCLoptimizer gathers these limitations using the API provided by OpenCL and discards the invalid configurations based on them. All the surviving configurations are evaluated using the execution time to find out which is the best one. This process can take a long time because the number of versions to evaluate is usually large. The tool shortens this process as the execution of the version associated to a given configuration is killed when its execution time reaches the minimum time observed until that moment.

The iterative optimization process guided by a GA is depicted in Fig. 3. In this case the chromosomes of the GA, which are potential solutions to our problem, have one gene per dimension of both the local and the global size, and this gene stores the size of that dimension. 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. The chromosomes are generated using random combinations of the values that the different genes can take. If the number of chromosomes randomly generated is not enough, they are cloned until the required number of chromosomes is achieved. Then, one workspace configuration per chromosome is generated. The execution time of each version, which is the fitness function of the GA, is measured.

The minimum execution time obtained by a member of the population is used to evaluate whether the GA search must finish. The condition for this is that the

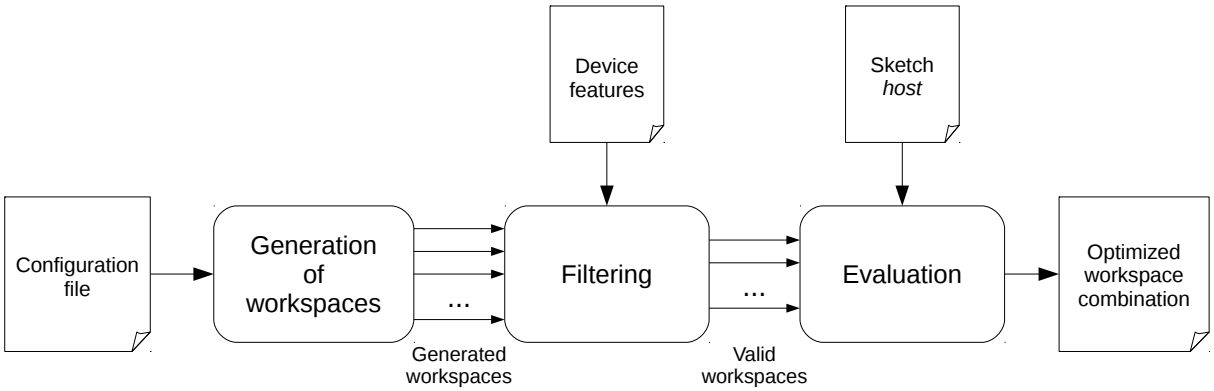

FIGURE 2. Iterative process to select an optimized workspace configuration in OCLoptimizer tool using an exhaustive search.

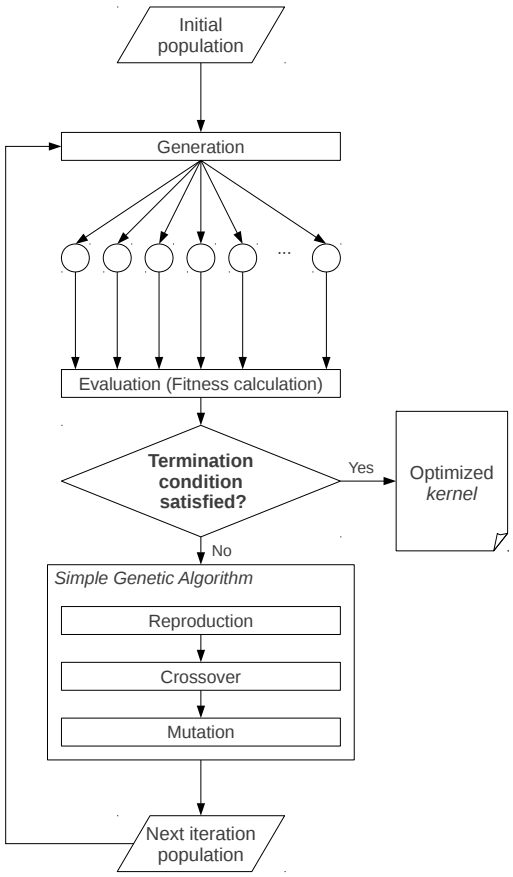

FIGURE 3. Iterative process in OCLoptimizer using a genetic algorithm

fitness function, i.e., the best execution time achieved, has not improved in the last three generations. In that case, the workspace configuration associated to this minimum execution time is provided as the output of the iterative process. Otherwise, a new population is generated. Concretely, the best individuals from the current population are selected to produce the next generation through reproduction applying the mechanisms of crossover and mutation [10]. 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 order to implement the mentioned GA search approach we have used the GAlib genetic algorithm package [11], written in C++ by Matthew Wall at the Massachusetts Institute of Technology.

It deserves to be mentioned that since this process of OCLoptimizer runs the kernel with multiple workspace configurations, the kernel must be written to support a generic workspace configuration. OCLoptimizer provides macros that ease the programming of such

a generic version. These macros are named `GENSTEP`, `GENINIT` and `GENLIMIT` and they are declared in a header file which is automatically included in the kernel by the tool. The programmer can use these macros in one or several loops selected to distribute their iterations among the threads of the workspace. In multidimensional workspaces, each loop will be assigned to a given dimension of the workspace. This way, the three macros receive three parameters: (1) `n` the number of iterations of the original loop, (2) an indication about wether the iterations are distributed among a number of points of the global (g) or the local (l) size, and (3) the number of the dimension of the workspace that is assigned to this loop.

The `GENSTEP` macro calculates the best step for the loop while the `GENINIT` and `GENLIMIT` macros calculate the lowerbound and upperbound of the loop, respectively. These three macros generate the distribution that fits better to the current platform. For example, when a GPU is detected the macros give place to a cyclic distribution of the iterations among the threads. This way, the access pattern followed by the references generate interleaved accesses, which improve the performance in this kind of platforms. In CPUs or accelerators such as the Xeon Phi, the macros give place to a consecutive distribution, which favors cache locality and auto-vectorization.

2.2. Optimized kernel code generation

Once the best global and local index spaces have been chosen, OCLoptimizer performs an iterative compilation process in which a series of code transformations are applied to the kernel code. These optimizations are suggested by the user by means of compiler directives inserted in the kernel code. This stage has been built on top of the CLANG frontend for LLVM [12]. Our tool works on the Abstract Syntax Tree (AST) representation of the input kernel, rather than on the LLVM intermediate representation (IR). The reason is that this enables us to generate an output optimized kernel much more human-readable and similar to the input kernel. An additional benefit is that the resulting code is easier to maintain than the one that can be obtained from the LLVM IR, which is very important for many users.

The input kernel is annotated with special OCLoptimizer directives. These annotations precede the piece of code affected by the transformation and include the parameters of the technique and the search space. The general form of an annotation is

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

where `<name>` is the name of the optimization technique to be applied and `<params>` stands for its parameters, which are different for each technique. They usually define the range of values to be tested in the iterative optimization process. The optional field `tolerance` establishes that only the versions whose execution times are below a tolerance `t` ($0 \leq$ `t` ≤ 100) per cent above the best time found, proceed to the next level of the iterative process. Finally, the argument `number` can optionally set the maximum number of versions that proceed to the next level of the iterative process.

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

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

where `init`, `end` and `step` are the first, the last and the step values of the range of potential unroll factors that the tool has to evaluate, respectively.

EXAMPLE 2. The input kernel of the vector addition running example is shown in Figure 4. The kernel receives the four parameters mentioned in Example 1. It consists of one isolated loop where the macros `GENINIT`, `GENLIMIT` and `GENSTEP` are used to calculate the boundaries and the step of the loop, because we want to distribute its iterations among the work-items. Let us notice that the iterations of this loop are distributed among the work-items of the only dimension of the global space. This loop is annotated with an `unroll` pragma that instructs OCLoptimizer to test all the unroll factors between 2 and 10 with step 2.

The code of Figure 5 is the output kernel generated by the tool for the GPU when an unroll factor of 2 is selected. Let us notice that when the tool detects that it is generating code for a GPU, it generates a distribution of the iterations among the work-items that favours interleaving. In the case of a CPU, the macros would assign consecutive iterations to a thread, in order to favour locality and auto-vectorization. The boundaries and the step of the loop are calculated using an optimized size of the global space selected in the previous stage ■

Just as in the case of the search of an optimized workspace, OCLoptimizer allows the user to choose between two strategies to guide the iterative compilation process that optimizes the kernel code. In this case the two possibilities are the breadth first search (BFS) and a genetic algorithm (GA).

The BFS strategy is depicted in Fig. 6. This strategy processes the pragmas in the kernel one by one in an iterative process where the processing of each pragma gives place to a new level of the tree of versions. The figure shows that each level corresponds to the processing of one directive and has two stages: generation and evaluation. In the generation stage the currently considered pragma gives 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 for the associated loop is generated. In the evaluation stage all the versions are

```
__kernel void addvec(const unsigned int n, __global float *a,
                     __global float *b, __global float *c) {

  #pragma oclopts unroll 2 10 2
  for(i=GENINIT(n,g,0), i<GENLIMIT(n,g,0);i=i+GENSTEP(n,g,0))
    a[i]=b[i]+c[i]

}
```

FIGURE 4. Input kernel for the vector addition example

```
__kernel void addvec(const unsigned int n, __global float *a,
                     __global float *b, __global float *c) {

  gid=get_global_id(0);
  sid=get_global_size(0);

  for(i=gid; i<N; i=i+(sid*2)) {
    a[i+(sid*0)]=b[i+(sid*0)]+c[i+(sid*0)];
    a[i+(sid*1)]=b[i+(sid*1)]+c[i+(sid*1)];
  }

}
```

FIGURE 5. Output kernel for the vector addition example

timed. If the user has specified the `tolerance` and/or `number` modifiers in the pragma, they are applied to filter the versions and keep only the fastest ones according to the criteria indicated. The surviving versions are used as a base for the next level of the iterative optimization process, which considers the next pragma found in the kernel. This process is repeated until all the pragmas in the kernel have been processed. Then, the fastest version of the final step is chosen as the best kernel version.

We must note that BFS can be potentially very demanding due to the large number of versions that it can generate, as if there are x pragmas on a code and the user wants to test y variations for each one of them, it will generate and evaluate x^y kernels. For this reason the `tolerance` and/or `number` modifiers are very useful to keep the search time for this strategy within reasonable limits for codes with many directives and/or when we want to test many possibilities for each directive.

The GA search for an optimized kernel code follows the same procedure described for the GA search of the best workspace configuration in Section 2.1. The difference is that in this case the chromosomes have one gene per directive to apply. This gene stores the value of the parameters of the code transformation designated in its corresponding directive. For example, in the case of the unroll optimization, the genes will be the unroll factors of the corresponding loops. The search process evolves by generating and evaluating the kernel versions associated to the chromosomes of the GA. As in the selection of an optimized workspace, we have used the GAlib package [11] to implement this GA search process.

As we will see in Section 4, the GA search is much faster than BFS. For this reason, and because it is critical for the GA to keep a population with a reasonable number of chromosomes, OCLoptimizer dismisses the `tolerance` and `number` modifiers when this search strategy is applied. It is also interesting to mention that because BFS only allows to explore different values for the optimization parameters of the different code transformations one after another, it can only consider a limited set of combinations for those transformations. In contrast, the GA considers all the directives at once, which enables it to explore a larger space of kernel versions. For this reason GA can reach faster kernels than BFS in some situations.

3. SUPPORT FOR MULTIPLE KERNELS OPENCL CODES

The tool can also optimize OpenCL applications composed of multiple kernels. In this case, the tool loses its ability to generate a working host code for the whole application. Keeping this ability would require a much more complex configuration file where information such as the relations and the data flow between the different kernels should be reflected. Nevertheless, our tool generates an optimized workspace configuration and code for each kernel of the application. These outputs can be integrated by the user in its own code.

The process followed by OCLoptimizer to optimize codes with multiple kernels, which consists of several fully automated steps, is now described. Some of the kernels that compose the application may have to use the same workspace configuration, while others can use their own configuration. These two types of kernels are

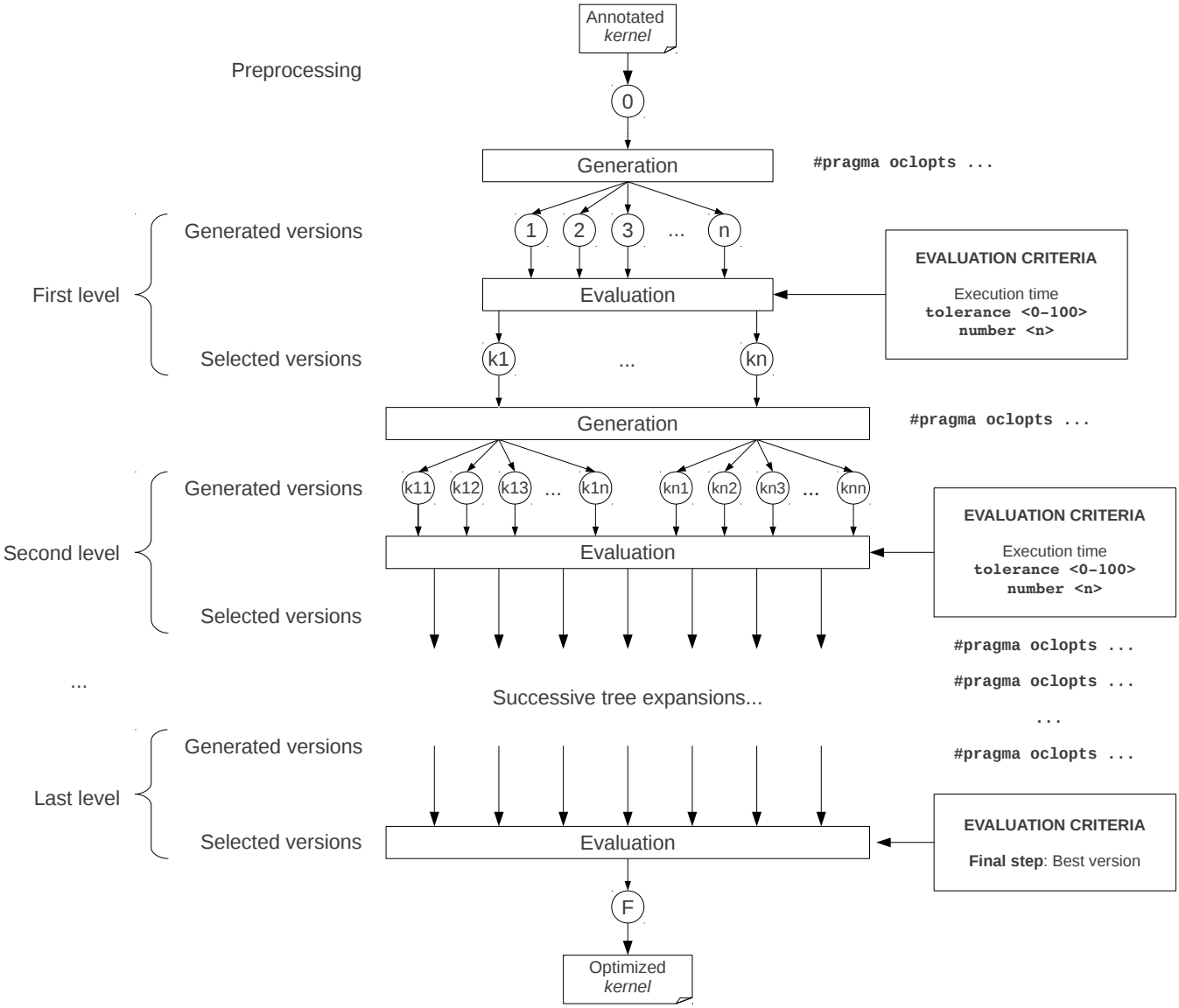

FIGURE 6. Iterative process to select the optimized kernel in OCLoptimizer tool using a breadth search first

processed differently by the tool. The kernels that can use their own workspace configuration are optimized one by one. The user has to provide a separate configuration file and an annotated code for each kernel, and she obtains an optimized workspace configuration and a kernel code for each one, which can be invoked by the user from her own host file.

In the case of groups of kernels that have to use the same workspace configuration, OCLoptimizer follows a two-step process. In the first step, it is applied separately to each kernel of the group of inter-dependent kernels. For each kernel, it requires a configuration file, an annotated kernel, and if necessary, an initialization code. In the case of inter-dependent kernels this last optional parameter is very important, as the inputs of some kernels are intermediate results of the algorithm which have to comply with certain characteristics. This way, the inputs of these kernels must be generated using a code provided by the user. In addition, the workspace configuration of this kind of kernels usually has to match certain conditions. These conditions can be specified in the workspace restrictions section of the configuration file associated to each kernel (see Appendix A). The output of this step is an optimized workspace configuration for each kernel, while the optimized kernels generated are discarded.

In the second step, the tool tries, in turn, the workspace configurations chosen in the first step. In each turn, one workspace configuration is used for all the kernels in the group of inter-dependent kernels, and OCLoptimizer only performs the kernel code optimization stage (Section 2.2) of each kernel. The tool finally provides to the user the workspace configuration and associated optimized kernels that gave place to the shortest runtime for the group of kernels.

4. EXPERIMENTAL RESULTS

OCLoptimizer can be used on OpenCL codes composed of one single kernel or multiple kernels. Section 4.1 shows the validation performed with several OpenCL codes composed of one kernel, while Section 4.2 shows the experiments for an OpenCL application composed of five different kernels.

4.1. Single kernel OpenCL codes

This part of the validation is based on five computationally intensive single-kernel codes: an N-body simulation [13] (NBODY), a matrix multiplication (MATMUL), a discretization of the Laplacian operator with a nine-point stencil [14] (STENCIL), a Sobel Edge Detector [15] (SOBEL) and a Direct Coulomb Summation [16] (DCS). The NBODY kernel has two unrollable loops and the MATMUL kernel has three unrollable loops, whereas the STENCIL, SOBEL and the DCS kernels have four unrollable loops each. Regarding the worskpace configuration, NBODY has 1-dimensional (global and local) workspaces, MATMUL, STENCIL and SOBEL have 2-dimensional workspaces, and finally the workspaces of DCS have 3 dimensions. Hand-tuned versions of these kernels, which use local memory and have been vectorized wherever possible, have been used as inputs for this part of the validation process. Also, calculations common to several threads are performed collaboratively in order to improve the performance. The experiments were run on three different platforms:

- The **CPU**: A system with two Intel Xeon E5-2660 Sandy Bridge with eight 2.2Ghz cores and hyper-threading (8×2 threads per processor, for a total of 32) and 64 GB of RAM. OpenCL runtime: Intel OpenCL 1.2-3.2.1.16712
- The **GPU**: An NVIDIA Tesla Kepler K20m 5 GB GDDR5. OpenCL runtime: NVIDIA CUDA Toolkit 5.0.35
- The **Accelerator**: An Intel Xeon Phi 5110P with sixty 1.053GHz cores with 8 GB of RAM. OpenCL runtime: Intel OpenCL 1.2-3.2.1.16712

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

Tables 1 to 6 summarize the performance results obtained in the three platforms using two combinations of search processes: that with the longest search time, which is exhaustive search (ES) for the workspaces and BFS for the kernel versions (ES+BFS), and that with the shortest one, which uses GA search for both optimization processes.

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

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

Table 7 summarizes the execution time, the global speedup and the performance measured in GFLOP/s achieved in our experiments for each code, problem size, and platform using the ES+BFS search. The speedup achieved by the optimized single kernel codes generated using the configurations selected by ES+BFS search is on average a 29% better than those generated using the configurations selected by GA. On exchange, the whole execution time of the tool is much longer when using

TABLE 1. Speedups and configurations selected using ES+BFS in the CPU

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

TABLE 2. Speedups and configurations selected using the Genetic Algorithm in the CPU

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

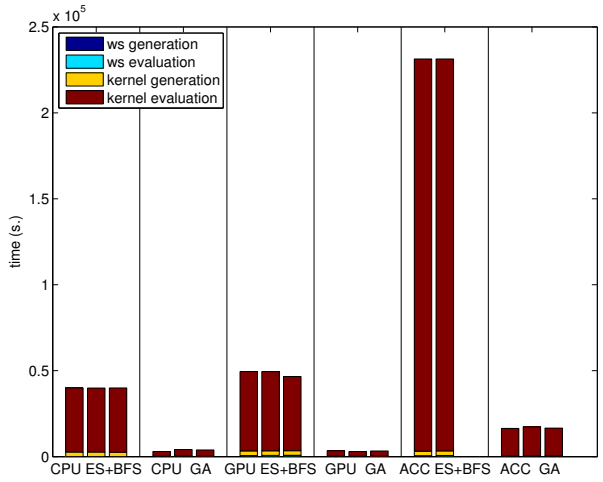

FIGURE 7. Search times for NBODY

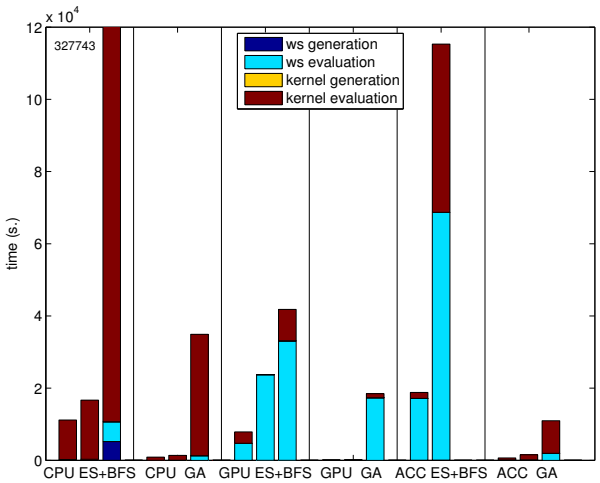

FIGURE 8. Search times for MATMUL

the ES+BFS search, as we will see now.

Figures 7 to 11 show the time required by the search processes discussed for NBODY, MATMUL, STENCIL, SOBEL and DCS, respectively. Each figure is divided into six sections, one for each combination of a device (CPU, GPU and Accelerator) and search process (ES+BFS or GA). Each section shows the search time for each tested problem size, from the smallest one to the largest one with bars divided into four stages: the workspace generation and evaluation times, which characterize the first optimization process, and the kernel optimization generation and evaluation times of the second search process.

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

TABLE 3. Speedups and configurations selected using the ES+BFS in the GPU

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

TABLE 4. Speedups and configurations selected using the Genetic Algorithm in the GPU

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

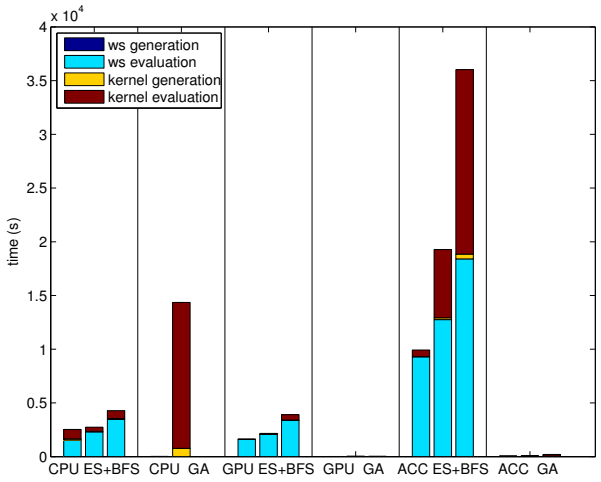

FIGURE 9. Search times for STENCIL

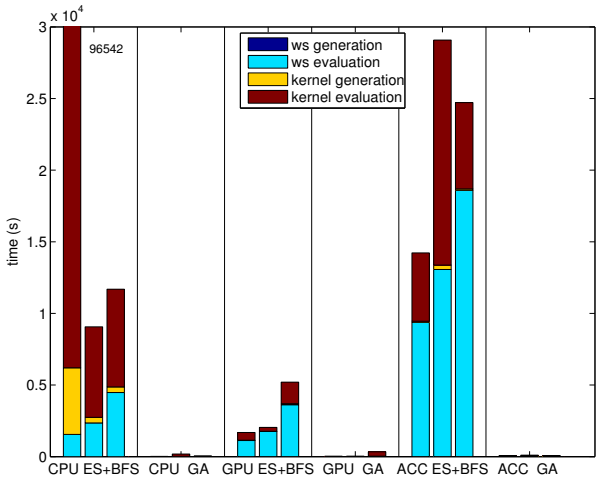

FIGURE 10. Search times for SOBEL

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

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

Figures 12 and 13 represent the speedups achieved using different workspace configurations and unroll factors, respectively. In both cases, the different

TABLE 5. Speedups and configurations selected using ES+BFS in the Accelerator

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

TABLE 6. Speedups and configurations selected using the Genetic Algorithm in the Accelerator

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

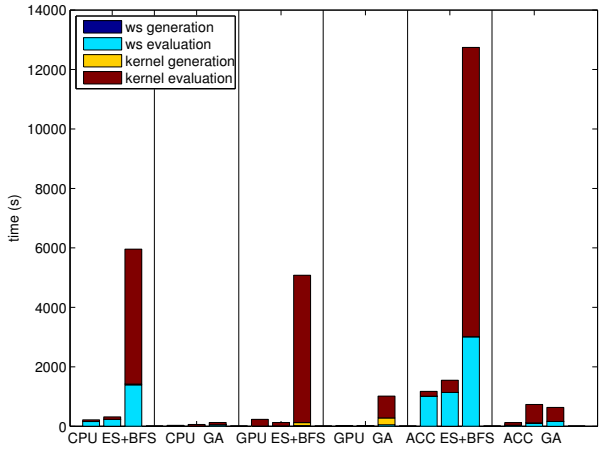

FIGURE 11. Search times for DCS

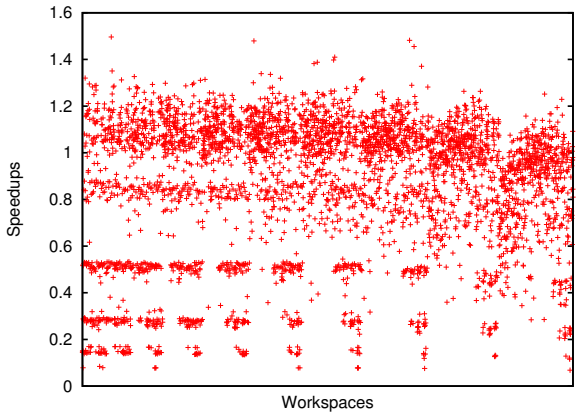

FIGURE 12. Speedups achieved using different workspace configurations for SOBEL using BFS in the CPU

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

4.2. Codes with several kernels

The tool has been tested on the Integer Sort (Benchmark) of the NAS Parallel Benchmarks (NPB). The baseline of these experiments is the (Seoul National University) SNU OpenCL NPB [17] implementation of this benchmark, which has 5 different kernels. There are two versions of this benchmark, one suitable for CPU and another one suitable for GPU. In

TABLE 7. Global speedups using ES+BFS

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

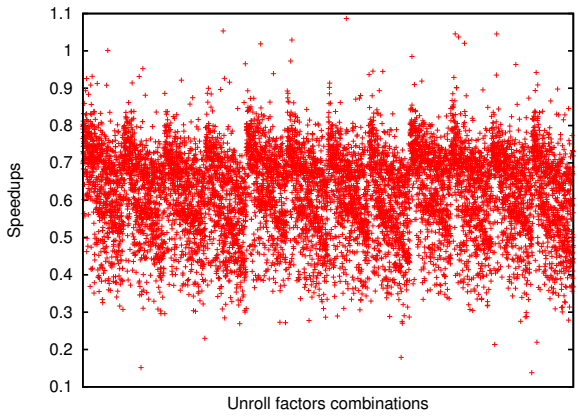

FIGURE 13. Speedups achieved using different unroll factors for SOBEL using BFS in the CPU

both implementations, several kernels use as inputs intermediate results which have to comply with certain characteristics, thus, special initialization codes had to be provided to the tool. In the GPU implementation, three of these kernels are inter-dependent and they must use the same workspace configuration, while in the CPU implementation the number of inter-dependent kernels is four. The experiments have been run on the same CPU and GPU used in the experiments of Section 4.1. SNU NPB CPU version is used as the baseline for the CPU experiments, and its GPU version for the GPU. The Accelerator platform has not been used in these experiments as the SNU NPB suite does not have an implementation optimized for the Xeon Phi.

Tables 8 and 9 contain the speedups achieved by OCLoptimizer using the ES+BFS and the GA search processes respectively. The experiments were performed for three problem sizes: S, W and A. The tables show the speedups obtained from the workspace optimization and the unroll optimization, both calculated following the same approach as in Section 4.1. The last column contains the execution time of the best version of the benchmark generated and the speedup with respect to the baseline. The workspace configurations and unroll factors chosen are not reported because of the large amount of data they imply given the existence of up to 5 kernels in the codes. As expected, the ES+BFS search obtains better results than the GA. The speedups in the CPU (3.03 on average for ES+BFS) are larger than in the GPU (1.13 on average for ES+BFS), and most of the speedup comes from the workspace optimization. These observations are similar to those made for the single kernel codes. The main conclusion of this experiment is that OCLoptimizer not only supports codes with strong inter-dependencies between their kernels, but it can also achieve respectable speedups despite working on hand-tuned state of the art implementations such as these two IS SNU NPB codes.

5. RELATED WORK

Iterative search techniques based on actual runtime measurements [18, 19] or analytical models [20, 21] have been widely used to automatically tune codes for different architectures. On the other hand, while performance portability in the context of parallel languages has been studied for a long time [22], it has lately regained interest due to the heterogeneity of the available accelerators.

For example, the elastic computing framework [23] separates functionality from implementation details using specialized functions. These functions allow a user to explore a great variety of alternative implementations and to select the optimal one for a certain platform. This work is limited by the fact that the code has to be expressed using the available specialized functions. This important limitation is not so strong in OCLoptimizer, which works on top of native OpenCL code written using special macros and annotated with `#pragmas`.

From the point of view of providing an adaptive scheduling, StarPU [24] automates the efficient mapping of tasks in heterogeneous environments, although it cannot tune the performance of each individual task. However, OCLoptimizer does tune the performance of individual tasks.

Iterative compilation is used in [25] to select the optimal parameters for GPU codes in a given

TABLE 8. Speedups and execution times for the IS benchmark using the ES+BFS search

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

TABLE 9. Speedups and execution times for the IS benchmark using the Genetic Algorithm

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

platform according to a set of pre-defined parametrized templates. This work is specifically focused on obtaining a portable linear algebra library and selecting the optimal tile size for the tiling technique for GPUs. OCLoptimizer targets any OpenCL code and platform, and it selects optimized workspace configurations and unroll factors.

VForce [26] provides performance portability in a transparent way across different kinds of accelerators to programs written in the VSIPL++ (Vector Signal Image Processing Library extension), a domain-specific language focused on image and signal processing. OCLoptimizer is not domain-specific as it targets any kind of application.

The auto-tuner presented in [27] uses predictions to select the best execution plan for the SkePU skeleton programming framework in multi-GPU systems. The PARTANS framework [28], which is specifically designed to express stencil computations in multi-GPU systems, includes auto-tuning mechanisms to optimize this kind of computations. One more time, the main drawback of this work with respect to OCLoptimizer is the limited scope of application, which is not present in our tool.

The orCUDA autotuner [29] generates complete CUDA code using as an input an annotated C loop. The annotations are introduced in the code as commentaries and they drive an iterative optimization process to select the size of the grid of threads, the size of the the thread blocks and certain parameters of different optimization techniques, including unrolling. The search of the best variant of the code is driven using the execution time. This work is focused on CUDA, so it cannot be used to tune codes for CPUs or Accelerators, and, despite they present examples with more complex codes, the validation only uses small kernels, with a single loop, which are used in the the resolution of partial differential equations. OCLoptimizer also targets non-Nvidia GPUs, CPUs and Accelerators as it is based on OpenCL, and the codes included in our experimental results are more complex than those in this work.

Focusing on OpenCL, uCLbench [30] characterizes the properties of the target device and the OpenCL implementation found, seeking to guide programmers in the hand-tuning of their codes. The main changes required to port the performance of OpenCL codes that have been tuned for GPUs to CPUs are discussed in [7] and [8]. A common point in both papers is the importance of adapting the granularity of the parallelism for this port, which is supported by OCLoptimizer.

GLOpenCL [31] is a unified development framework which supports OpenCL on different types of multicores. This framework consists of a compiler and a runtime library. The compiler is based on LLVM and it performs a set of source-to-source transformations such as serialization of logical threads, elimination of synchronization operations and variable privatization. The validation is performed using 5 different kernels on different multicore platforms. The results show that the performance achieved using GLOpenCL is close to the performance obtained using the implementations provided by the vendor of each platform. Unlike OCLoptimizer, this work does not select an optimized workspace configuration.

Finally, Dolbeau et al [32] discuss the different performance obtained using the same OpenCL code on different platforms. They use the CAPS compiler to generate autotuned OpenCL code. This compiler can select the optimal group size but it does not change the global size. As we have seen, OCLoptimizer obtains important performance gains from the selection of both the global and the local workspace sizes.

6. CONCLUSIONS

Two of the main weaknesses of OpenCL are the low level of its host API, which makes the development of its host codes tedious and error-prone, and, more importantly, the lack of performance portability. This paper has presented OCLoptimizer, a tool that addresses

both issues with little programmer effort. Given a configuration file and a kernel annotated with indications on the optimizations to try, OCLoptimizer is able to generate a working host code, find an optimized workspace configuration and tune the kernel for the platform where the tool is executed. The tool also supports the automated optimization of kernels in applications with multiple inter-dependent kernels, which is a unique feature as far as we know, although in this case the current version does not generate the host code.

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

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

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

The IS benchmark of of the SNU NPB has been used to validate the support of the tool for OpenCL codes composed of several kernels. In this case, the experiments were run only on the CPU and the GPU, achieving an average global speedup of 1.79 (2.45 for CPU and 1.19 for GPU). These speedups are more modest than those observed in the single kernel benchmarks but the baseline used for these experiments is a hand-tuned state-of-the-art implementation of the benchmark. These experiments confirm that the ES+BFS is more effective than the GA search and that most of the speedup comes from optimizing the workspace configuration. However, in this benchmark the largest speedups are achieved in the CPU.

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

FIGURE A.1. Configuration file for the vector addition example

We plan to extend our tool to support more optimization techniques in the kernels. We also want to integrate the use of analytical models and heuristics to reduce the search times.

ACKNOWLEDGEMENTS

This work is supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2013-42148-P), and by the Galician Government under the Consolidation Program of Competitive Reference Groups (ref. GRC2013-055). This work is also partially supported by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

APPENDIX A. OCLOPTIMIZER CONFIGURATION FILE

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

example file in Fig. A.1.

The *common parameters* section initializes variables that will be used through the rest of the file and it configures some general settings of the OpenCL host code to generate. In Fig. A.1 this section initializes the variable `N` to 1024. Then, it establishes that the host code has to use a GPU to perform the computation. Alternatively, the `device` variable could take the values `CPU`, which is used to select the main processor, or `ACC`, which is used to select a Xeon Phi if available. If there are two devices of the same type, the current implementation of OCLoptimizer selects the first one. The `initialization` variable specifies how the kernel inputs whose value are not specified in the configuration file should be initialized in the host code. In the example, the `random` value indicates that it should be initialized with random values. In some cases, a random initialization would not be valid as the contents of the input data should fulfill certain conditions. In that case, the `initialization` variable should be set to `code` and the path of the file containing the initialization code should be provided as the third parameter of the tool.

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

The *workspace definition* section sets the parameters related to the workspace configuration. These are the number of dimensions of the workspace (`ndims`), and for each dimension, the global and the local size. As the optimized global and local size will be found iteratively, the user has to specify, separated by commas, the minimum and the maximum value to test for the sizes of all the local and global dimensions. The values associated to dimension `X` are preceded by a `[dimX]` clause. In the example of Fig. A.1, the workspace only has one dimension composed of between 1 and N (`globalsize=1,N`) work-items and each work-group is composed of between 1 and 32 work-items (`localsize=1,32`).

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

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

REFERENCES

[1] IBM, Sony, and Toshiba (2006) *C/C++ Language Extensions for Cell Broadband Engine Architecture*. IBM, New York.

[2] NVIDIA Corporation (2013) *CUDA C Programming Guide*. NVIDIA Corporation, Santa Clara, CA.

[3] Munshi, A., Gaster, B., Mattson, T. G., and Fung, J. (2011) *OpenCL Programming Guide*. Addison-Wesley Professional, Boston, MA.

[4] Nieuwpoort, R. V. and Romein, J. W. (2011) Correlating radio astronomy signals with many-core hardware. *International Journal of Parallel Programming*, **39**, 88–114.

[5] Lee Howes (2014) SYCL Provisional Specification. Khronos Group, Beaverton, OR.

[6] Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., and Kobayashi, H. (2010) Evaluating performance and portability of OpenCL programs. *Proc. of Fifth Intl. Workshop on Automatic Performance Tuning (iWAPT)*, Madrid, Spain, 5-8 July, pp. 14-16. Universidad Carlos III, Madrid.

[7] Lan, Q., Xun, C., Wen, M., Su, H., Liu, L., and Zhang, C. (2012) Improving performance of GPU specific OpenCL program on CPUs. *Proc. of 13th Intl. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'12)*, Beijing, China, 14-16 December, pp. 356–360. IEEE, Washington.

[8] Shen, J., Fang, J., Sips, H., and Varbanescu, A. (2013) Performance traps in OpenCL for CPUs. *Proc. 21st Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP 2013)*, Belfast, Northern Ireland, February 27th-March 1th, pp. 38–45. IEEE, Washington.

[9] Fabeiro, J. F., Andrade, D., and Fraguela, B. B. (2013) OCLoptimizer: An iterative optimization tool for OpenCL. *Proc. Intl. Conf. on Computational Science (ICCS 2013)*, Barcelona, Spain, 5-7 June, pp. 1322–1331. Elsevier, New York.

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

[11] Wall, M. (1996) *GAlib: A C++ Library of Genetic Algorithm Components*.

[12] Lattner, C. and Adve, V. (2004) LLVM: A compilation framework for lifelong program analysis & transformation. *Proc. of Intl. Symp. on Code Generation and Optimization (CGO)*, Palo Alto, California, 21-24 March, pp. 75–86. IEEE, Washington.

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

[14] Smith, G. (1985) *Numerical Solution of Partial Differential Equations: Finite Difference Methods* Oxford applied mathematics and computing science series. Clarendon Press, Oxford, UK.

[15] Kazakova, N., Margala, M., and Durdle, N. (2004) Sobel edge detection processor for a real-time volume rendering system. *Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS)*, Vancouver, Canada, 23-26 May, pp. II – 913–16 Vol.2. IEEE, Washington.

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

[17] Seo S.,Jo G., and Lee J.. (2011) Performance Characterization of the NAS Parallel Benchmarks in OpenCL. *Proc. of IEEE Intl. Symp. on Workload Characterization (IISWC)*, Austin, Texas, 6-8 November, pp. 137–148. IEEE, Washington.

[18] Kisuki, T., Knijnenburg, P. M. W., and O'Boyle, M. F. P. (2000) Combined selection of tile sizes and unroll factors using iterative compilation. *Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Philadelphia, Pensylvania, 15-19 October, pp. 237–246. IEEE, Washington.

[19] Pouchet, L.-N., Bastoul, C., Cohen, A., and Vasilache, N. (2007) Iterative optimization in the polyhedral model: Part i, one-dimensional time. *Proc. Intl. Symp. on Code Generation and Optimization (CGO)*, San Jose, California, 11-14 March. pp. 144–156. IEEE, Washington.

[20] Fraguela, B. B., Carmueja, M. G., and Andrade, D. (2005) Optimal tile size selection guided by analytical models. *Parallel Computing (ParCo)*, Malaga, Spain, 13-16 September, pp. 565–572.John von Neumann Institute for Computing, Julich.

[21] Fraguela, B. B., Voronenko, Y., and Püschel, M. (2009) Automatic tuning of discrete Fourier transforms driven by analytical modeling. *Proc. 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Raleigh, North Carolina, 12-16 December, pp. 271–280. IEEE, Washington.

[22] Ngo, T., Snyder, L., and Chamberlain, B. (1997) Portable performance of data parallel languages. *Proc. of ACM/IEEE conf. on Supercomputing (SC)*, San Jose, California, 15-21 November, pp. 1–20. IEEE, Washington.

[23] Wernsing, J. R. and Stitt, G. (2010) Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. *Proc. of ACM SIGPLAN/SIGBED conf. on Languages, compilers, and tools for embedded systems (LCTES)*, Stockholm, Sweden, 13-15 April, pp. 115–124. ACM, New York.

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

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

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

[27] Dastgeer, U., Enmyren, J., and Kessler, C. W. (2011) Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems. *Proc. of 4th Intl. Workshop on Multicore Software Engineering* (IWMSE), Honolulu, Hawai, 11 May, pp. 25–32. German Computer Science Society, Germany.

[28] Lutz, T., Fensch, C., and Cole, M. (2013) PARTANS: An autotuning framework for stencil computation on multi-GPU systems. *ACM Trans. Archit. Code Optim.*, **9**, 59:1–59:24.

[29] Mametjanov, A., Lowell, D., Ma, C.-C., and Norris, B. (2012) Autotuning stencil-based computations on gpus. *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)*, Beijing, China, 24-28 September, pp. 266–274. IEEE, Washington.

[30] Thoman, P., Kofler, K., Studt, H., Thomson, J., and Fahringer, T. (2011) Automatic OpenCL device characterization: Guiding optimized kernel design. In Jeannot, E., Namyst, R., and Roman, J. (eds.), *Proc. of Euro-Par 2011 Parallel Processing*, Lecture Notes in Computer Science, **6853**, Bordeaux, France, August 29th-September 2nd. pp. 438–452. Springer-Verlag, New York.

[31] Daloukas, K., Antonopoulos, C. D., and Bellas, N. (2011) GLOpenCL: OpenCL support on hardware- and software-managed cache multicores. *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HIPEAC)* , Crete, Greece, 22 January. pp. 15–24. ACM, New York.

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