How to Write Performance Portable Codes using the Heterogeneous Programming Library

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

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

Abstract Parallel programming in heterogenous environments faces two important problems: (1) The writing of parallel codes requires a big effort from the programmers, and (2) In order to achieve the maximum obtainable performance, programmers have to hand-tune parallel codes for each device. The Heterogeneous Programming Library (HPL) tackles the first problem offering an easy way to write parallel codes that can be run on a large number of heterogeneous devices. This paper shows how this library can tackle the second one, as it describes the usage of the run-time code generation (RTCG) feature of the library to write a performance portable version of a parallel code. This performance portable version has the ability to adapt automatically to any device.

The paper focuses in the matrix multiplication algorithm as a case study. The adaptability of the resulting implementation relies on the tuning of a dozen of parameters. The search of the best values for these parameters is guided by a genetic algorithm where each individual is evaluated using its execution time.

The performance of this implementation has been compared to two state-of-the-art OpenCL adaptive implementations of the matrix product, namely, clBLAS and ViennaCL. The kernels used by clBLAS can be adapted to the platform where they are going to be run by means of a prior profiling. The ViennaCL implementation can be tuned through a set of parameters, but their values are selected through an exhaustive search. Except in a single test, where clBLAS takes the lead for a single matrix size in an AMD GPU, our implementation systematically outperforms the other adaptive libraries in four platforms: an NVIDIA GPU, an AMD GPU, a multicore Intel CPU and an Intel Xeon Phi accelerator. The average speedup of our implementation respect to clBLAS and ViennaCL is 1.74 and 1.44, respectively. In addition, on average our genetic search is 1.18 times faster than the clBLAS profiling and 160 times faster than the exhaustive search implemented by ViennaCL, and it finds faster versions of the matrix multiplication.

Keywords: GPGPU, performance portability, OpenCL

1 Introduction

One of the most important problems that hamper the wider use of heterogeneous systems is the current poor portability of the codes for these devices. The truly portable programming of heterogeneous system needs: (1) a unified programming language for any kind of device and, (2) a method to achieve performance portability. OpenCL [1] solves the first challenge as it enables the programming of a wide variety of devices. The second requirement, performance portability, has been widely addressed in the bibliography. For example, the framework [2] separates functionality from implementation details using specialized functions that allow to explore a great variety of implementations and to select the optimal one for a certain platform. VForce [3] provides performance portability in a transparent way across different kinds of accelerators to programs written in a domain-specific language focused on image and signal processing.

Performance portability can also be achieved through iterative processes. For example, [4] uses iterative compilation to select the optimal parameters for GPU codes according to a set of pre-defined, parameterized templates for linear algebra problems. An auto-tuning approach that selects the best execution plan for the SkePU skeleton programming framework in multi-GPU systems based on predictions is presented in [5]. The PARTANS framework [6], which is specifically designed to express stencil computations in multi-GPU systems, includes auto-tuning mechanisms to optimize this kind of computations.

Focusing on OpenCL, uCLbench [7] characterizes the properties of the device and the OpenCL implementation where the code is intended to run, 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 [8][9]. GLOpenCL [10] is a development framework consisting of a compiler and a runtime library that supports OpenCL on different types of multicores. OCLoptimizer [11] searches optimal unroll factors for OpenCL kernels based on compiler directives and a configuration file. Finally, Dolbeau et al [12] discuss the performance that the same OpenCL code achieves on different platforms. They use the CAPS compiler to generate auto-tuned OpenCL code.

The Heterogeneous Programming Library (HPL) [13] is a C++ framework that improves the programmability of heterogeneous systems by combining special data types and an embedded language to write kernels, which express the parallelized computations to run in the devices. HPL is a unified approach for programming heterogeneous systems as it uses as backend OpenCL, so that its kernels can run on any device. It also provides appropriate tools to provide performance portability, as the combination of its embedded language and C++ to write the kernels enables run-time code generation (RTCG), which can be used to write self-adaptive generic kernels. While other tools enable RTCG using similar mechanisms [14][15], they only target regular CPUs, and therefore they have sought other purposes. This way, this paper explores the development of kernels with portable performance by combining C++ and the HPL embedded language to generate parameterized generic kernels. The configuration parameters of each

kernel change certain aspects of how its code is optimized, and they are adjusted using a genetic algorithm through an iterative process. The performance of the kernel generated using each combination of values of its parameters is evaluated by executing the code. The configuration parameters select optimized unroll factors for some loops, an optimized granularity for the work performed by each instance of the kernel, the base version of the algorithm used, which data structures are stored in local memory, the best loop ordering and the best vector size. The performance of our kernels is compared to two state-of-the-art adaptive implementations, clBLAS and ViennaCL. These two implementations were chosen because (a) they use OpenCL, and thus, they target the same range of platforms as HPL, and (b) they provide adaptive mechanisms to enable performance portability. Our study also covers the OpenCL-based clMAGMA library [16], as it relies on clBLAS for its OpenCL BLAS routines.

Our matrix multiplication implementation is based on existing implementations for NVIDIA GPUs [17], AMD GPUs [18], and any kind of devices supporting OpenCL [19]. This latter implementation also enables performance portability. Our implementation uses not only similar techniques to those introduced in these previous works but also new ones. As a consequence, our implementation turns out to be more effective that those previous ones.

The rest of the paper is organized as follows. Section 2 briefly introduces the HPL library. Section 3 explains how RTCG can be used in HPL to write parameterized generic kernels. Section 4 focuses on the case study, the matrix multiplication. Section 5 explains the method derived to select an optimized set of values for the configuration parameters of the kernel using iterative optimization. Section 6 shows the experimental results and Section 7 concludes.

2 The Heterogeneous Programming Library

The Heterogeneous Programming Library (HPL), which is publicly available at `http://hpl.des.udc.es`, intends to improve the programmability of heterogeneous systems while providing portability through an approach where the computational kernels that exploit heterogeneous parallelism are written in a language embedded in C++. This characteristic enables run-time code generation (RTCG), which is a powerful tool to provide performance portability, as we will see through this paper. HPL provides portability because OpenCL is the intermediate representation (IR) it currently generates, thus this library targets the same range of devices supported by OpenCL.

The HPL library supports the same programming model as CUDA and OpenCL. Its hardware model is composed by a host equipped with a standard CPU and memory, with a number of computing devices attached. The host runs the sequential parts of the code, while the devices run the parallel parts. Each device has processors that execute SPMD parallel code on data present in the memory of their device. As in OpenCL or CUDA, we can create groups of threads that can be synchronized through barriers and share a small scratchpad memory.

The memory model distinguishes the same kinds of memory as OpenCL (global, local, constant and private) and with the same properties. 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, or work-items using OpenCL terminology, can be executed in parallel, each instance being univocally identified. The number of instances of the kernels and their identifiers are defined by a global domain of non-negative integers with up to 3 dimensions. This way, instances are identified inside this domain with tuples of global ids. In turn, these instances can be associated in groups. With this purpose, we can define local domains as equal portions of the global domain. Instances are identified inside its group using tuples of local ids. Now, Section 2.1 explains how to program using HPL.

2.1 Programming using HPL

The library provides three main components to the programmers:

- A template class `Array` to define both the variables to be transferred between the host and the devices, and the variables that are local to the kernels.
- The kernels, which are functions written in a language embedded in C++. This embedded language is an API in C++ consisting of data types, functions, macros and predefined variables.
- An API that will be used by the code to inspect the devices available in a given platform and to order the execution of the kernels.

All the kernel variables must have type `Array<type, n [, memFlag]>`, which represents an `n`-dimensional array of elements of a C++ `type`, or a scalar for `n`=0. Scalars and vectors can also be defined with special data types like `Int`, `Float`, `Int4`, `Float8`, etc. The optional `memFlag` can specify one of the kinds of memory supported (`Global`, `Local`, `Constant` or `Private`). The arrays passed as parameters to the kernels must be declared in the host using the same type. These variables are initially stored in the host memory, but when they are used as kernel parameters they are automatically transferred to the device. The outputs are also automatically transferred to the host when needed.

HPL kernels also require that their control flow structures are written using special keywords. The embedded language uses the same constructs as C++ but their name finishes with an underscore (`if_`, `for_`, ...). Also, the arguments to `for_` loops are separated by commas instead of semicolons. The library provides an API based on predefined variables to obtain the global, local and group identifiers as well as the sizes of the domains and numbers of groups. For example, `idx` provides the first dimension of the global identifier of a work-item, while `szx` provides the global work size for that dimension. If we add the `l` prefix to these keywords we obtain their local counterparts, and if we replace the letter `x` with `y` or `z`, we obtain the same values for the second and the third dimensions respectively.

Kernels are written as regular functions or functors that use these elements and whose parameters are passed by value if they are scalars, and by reference

Listing 1.1. SAXPY HPL code

```
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 (not shown)
  eval(saxpy).global(1000).local(10)(y, x, a);
}
```

otherwise. The `saxpy` routine in Listing 1.1 implements using this language the SAXPY (Single-precision real Alpha X Plus Y) vector BLAS routine, which computes $Y = a \times X + Y$. In this kernel, each instance `idx` computes a different position of the result `y[idx]`.

Regarding the host interface, its most important component is the function `eval`, which requests the execution of the kernel `f` with the syntax `eval(f)(arg1, arg2, ...)`. The execution of the kernel can be parameterized by inserting specifications, in the form of methods, between `eval` and the argument list. For example, the global and the local sizes can be specified using methods called `global()` and `local()` respectively. This way, the `saxpy` routine is invoked in Listing 1.1 with a global domain of 1000 elements and a local domain of 10 elements.

3 Performance portability in HPL

HPL generates the internal representation (IR) of its kernels by running them as regular code in the host when an `eval` requests their execution for the first time. Subsequent requests just reuse the IR generated the first time, which is stored in an internal cache, unless this cache is erased in order to force the regeneration of the IR. The HPL macros and data types capture all the expressions in which they are involved during the execution of the kernel in the host, allowing the runtime to generate the associated IR. However, regular C++ sentences found within the kernel are simply executed and they do not appear in the resulting IR. This characteristic enables RTCG, which can be used, for example, to choose between different versions of the same code, or to parameterize the generation of code. The method proposed in this paper combines RTCG and generic kernels to generate different versions of the same kernel based on different input parameters. In this context, generic kernels are those written for generic values of some parameter, such as the granularity, which can be adjusted at run-time.

First, we describe the strategy we have followed to parameterize the kernels. We have defined the HPL kernels using functors, so that for each kernel we

Listing 1.2. MxV code: original version

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

define a class with the name of the kernel that defines the `operator()`. The arguments and the body of this method are the arguments and the body of the kernel, respectively. The parameters that will be used to parameterize the kernel at runtime, are defined as properties of this class, thus, they can be accessed from the `operator()` method. Besides, they can be set from the host before the generation of the kernel code is initiated by an `eval` invocation.

Based on a set of parameters, we have used RTCG and generic kernels to generate codes that at the same time: (1) apply the unrolling technique to one or several loops using a given unroll factor, (2) select the best granularity of the computation performed by each instance of the kernel, (3) select the most suitable variant of an algorithm depending on the device that will be used, (4) decide which data structures are stored in local memory, (5) select an optimized loop order, and (6) choose an optimized vector size in the vectorized portions of code. The methods used to introduce these features in the kernels are now explained in turn.

`Unrolling:` Loop unrolling is a popular optimization technique whose main benefits are that it unveils instruction level parallelism, minimizes branch penalty and reduces the number of control instructions executed. Loop unrolling using arbitrary unroll factors can be introduced in HPL kernels using RTCG. The C++ code will be used in conjunction with the embedded language to generate the unrolled loops. Let us see an example starting from the matrix-vector product (MxV) code shown in Listing 1.2. This code defines the HPL kernel in lines 2-8. Each instance of the kernel processes one row from the input matrix, thus a single loop is required to multiply each element of the row by the corresponding element of the input vector.

Listing 1.3 shows an unrolled version of the kernel. The loop between lines 6-9 is an unrolled version of the original loop, thus, its stride is now the unroll factor (`uf`). The body of the loop is replicated `uf` times by a native C++ loop

Listing 1.3. MxV code: unrolled version

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

(lines 7-8). As the number of iterations of the loop `N` may not be a multiple of `uf`, to prevent out of range array accesses, the loop limit is `N-uf`. If there are some iterations left after that loop, they are processed without unrolling by the code in lines 10-11. The value for the unroll factor is passed to the kernel from the main procedure by setting the appropriate attribute of the class that defines the kernel (line 17).

`Granularity:` HPL creates one instance (or thread in HPL terminology) of the kernel for each point of the global domain. The optimal amount of work performed by each thread must be tuned for each platform in order to maximize the performance. For example, CPUs tend to be more effective using threads with larger workloads than GPUs. It is interesting to be able to tune that granularity at run-time depending on the type of device we are using. We can do that in HPL by changing the number of points in the global domain. For example, in our MxV code, the number of threads created is equal to the number of rows of the input matrix, thus, each thread processes one row of this matrix. If we reduce the number of threads, each thread should process several rows from the input matrix. This modification requires that the code is rewritten for a generic grain size, the grain size being in this case the number of rows of the input matrix processed by each thread. In our proposal, the rows are distributed using a block-cyclic policy, thus, grains of `bszx` rows are assigned cyclically to the threads available. An optimized value of `bszx` is found for each device. In the MxV code, this block size will not have a big influence in the performance, but

Listing 1.4. MxV code: auto-adjustable granularity version

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

in other problems some values of `bsz`s may benefit locality or coalescing, so, they will have a big impact in the performance.

In order to implement this distribution of the rows, the MxV kernel code must be changed to add two outer loops that process the blocks of `bszx` rows assigned to each thread. Loop headers in lines 6-7 of Listing 1.4 select the appropriate set of rows to be processed by each thread following a block-cyclic policy. The resulting kernel does not use RTCG but it is written in a generic way, so that if different values are provided for the size of the global domain and the block size, the granularity of the work performed by each thread is automatically adjusted at run-time.

`Algorithm selection:` The type of device used for a kernel execution is known at run-time. HPL can use this information to choose between different versions of the same algorithm, or portions of the algorithm, using RTCG. For example, a version that exploits local memory is good for GPUs but it may introduce unnecessary synchronization points in CPUs. The best strategy to divide the work among the threads varies depending on the type of device. RTCG can be used to select the appropriate base version or implementations of portions of the algorithm at run-time. Figure 1.5 shows the skeleton of a MxV vector kernel where a different variant of the algorithm is selected depending on the type of device. In the same vein, the size of the problem can advise the usage of different base versions of the algorithm.

`Local memory:` The usage of local memory is crucial for some devices like GPUs. We propose a technique to dynamically adjust the usage of local memory in HPL kernels. The idea is to write kernels where one or several data structures

Listing 1.5. MxV code: algorithm version selection

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

may optionally be stored in local memory or not. For example, in the MxV code, we can choose vector `x` for this purpose. A boolean parameter `copyX` will be set in the host to indicate whether we want to place that array in local memory. Listing 1.6 contains the MxV kernel modified to implement this behavior. The kernel uses RTCG to make the copy of `x` to local memory if `copyX` is activated, see lines 7-11. When the computation is done, the global array `x` or its local copy will be used depending on the value of the `copyX` parameter in line 13.

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

The code in Listing 1.7 shows an example of how this technique is applied to the matrix-vector product HPL kernel. In the original version presented in Listing 1.2, each thread performs the multiplication of one row of matrix `a` and the vector `x`. Let us recall that each thread processes the multiplication of `M/szx` consecutive rows of matrix `a` by vector `x`. The product within each thread can be done using the traditional order, where matrix `a` is accessed by rows, or it can be done by traversing per columns the chunk of `M/szx` rows of `a` processed by each thread. The order can be changed by swapping the two loops in the kernel. In HPL, this code transformation can be done at run-time using a new technique based on indirections. Arrays `init`, `e` and `s` have one position per loop (2 in the example) containing the initialization, limit and step of the counters of each one of the actual loops that we want to reorder. This way, we call actual loop `j` the one whose data is stored in the `j`-th position of these vectors. The loops with indices `c[0]` and `c[1]` are just container loops where the real loops are placed. The loop order can be changed modifying the contents of arrays `o` and `p`. This way, the number of the actual loop `j` to be implemented by the container loop, `i`, with index `c[i]` is stored in `o[i]`. Also, the references inside

Listing 1.6. MxV code: local memory usage

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

the loops have indexing functions that depend on the indices of the container loops, `c[i]`. Each `p[j]` contains the index of vector `c` that implements the actual loop `j`, that is, whenever `o[i]=j`, then `p[j]=i`. This way, any reference to the indexing variable of the actual loop `j` in the original code can be systematically replaced by `c[p[j]]`, ensuring that the appropriate loop index will be used no matter which the loop ordering chosen. In this example, the instruction in line 21 requests that the container loop 0 (`c[0]`) implements the actual loop 1 (`o[0]=1`). Similarly, the instruction in line 22 configures the container loop 1 (`c[1]`) so that it implements the actual loop 0, (`o[1]=0`). Regarding the `p` array, `p[o[0]]`, which is `p[1]` in this order, points to the index of container `c[0]`, and `p[o[1]]`, which is `p[0]` in this order, points to the index of `c[1]`. These values give place to the access per columns, while if arrays `o` and `p` are set to their complementary values, they would give place to an access per rows.

This scheme can be generalized for any arbitrary number of loops. Notice that some loop interchanges may be illegal. Thus, the programmer is responsible for checking the legality of the orders tried or at least, for enumerating the set of legal orderings.

The loops interchanged in this example are HPL `for_` loops (lines 9-10). Thus, they will give place to `for` loops in the generated OpenCL kernel. If in this example, `for_` loops are transformed into `for` loops, these loops will be executed during the HPL code generation process, which will give place to a fully unrolled version of the original loop nest. In addition array `c` should be transformed into a native C++ array. In this case, the loop interchange technique turns into a

Listing 1.7. MxV code: version with interchangeable loops

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

instruction scheduling technique, as different loop orders will give place to a different order of the same sequence of instructions. This instruction scheduling technique is applied to our matrix multiplication implementation.

`Vectorization:` Vectorization is another usually applied optimization technique. When heterogeneous systems are considered, selecting the appropriate vector size for each architecture is very relevant in terms of performance. HPL allows to rewrite at run-time a vectorized kernel using arbitrary vector sizes. This feature is accomplished by combining C++ templating and the `AliasArray` HPL data type, which allows to access vectorially an existing HPL `Array` made up of scalars.

The code in Listing 1.8 is a vectorized version of the original matrix-vector product of Listing 1.2 that uses a generic vector type `vectype`. With this purpose, the HPL kernel in lines 1-20 is templated for this `vectype`. On the host side, the `MxV` class is properly instantiated using the desired vector type (line 23).

On the kernel side, matrix `a` and vector `x` are wrapped in lines 6-7 using the `AliasArray` class provided by HPL which allows to access them vectorially with a given vector size.

The loop in lines 12-14 is a vectorized version of the inner loop of the original version of the algorithm. This loop generates a resulting vector `tmp`

Listing 1.8. MxV code: vectorized version

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

with `vectype::veclen` positions. Finally, the values of `tmp` are accumulated in `y[idx*(M/szx)+i]` by the loop in lines 15-17. This vectorization technique is applied to our matrix multiplication implementation.

4 Case Study: Matrix Multiplication

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

Our performance-portable HPL kernel implements the $C = A \times B$ operation. The code has been written in such a generic way that either A or B or both can be either directly loaded in private memory from global memory, or previously copied to local memory to optimize these further loads into private memory. Moreover, thanks to the aforementioned RTCG capabilities of HPL, it is possible to select the most appropriate combination of usage for both kinds of memory depending on the device selected at run-time. In addition, the granularity of

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

Table 1. Parameters of the matrix multiplication algorithm

the work to be performed by each thread can be adjusted by changing the global domain size. The size of the local domain can be changed depending on the capabilities of the device, and, within each thread, the tiling technique is applied. The inner loops of the algorithm are fully unrolled and the instructions are reordered using the instructions scheduling technique. Finally, this inner code is vectorized for a generic vector type that can be configured at run-time. All these optimizations give place to a set of parameters that can be tuned for each device at runtime and that are summarized in Table 1.

5 HPL portable kernels through iterative optimization

The search of an optimized set of parameter values for the kernel is performed using an iterative optimization process guided by a Genetic Algorithm (GA). Concretely, we have built the iterative search on top of the sequential version of the GAlib genetic algorithm package [20]. The chromosomes of our GA, which are potential solutions to our problem, have one gene per configuration parameter of the kernel. The initial population of the algorithm is composed of a configurable number of individuals that have been fixed by experimentation. The individuals and chromosomes of the initial population are randomly generated. Each individual generates a different version of the kernel using the values selected for each configuration parameter. These versions are evaluated using their fitness function, which is its execution time.

The minimum execution time obtained by a member of the population is used to decide whether the search must finish. The condition for this is that the fitness function (the execution time) has not improved for five generations. When this happens, the chromosomes that provided the best solution are used to generate

an optimized version of the kernel. If the condition has not been reached, a new generation of individuals is generated. This offspring is created starting from the best individuals of the previous generation, and using mechanisms such as crossover and mutations. The process is repeated until the fitness function has not improved for five generations.

6 Experimental results

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

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

The test performs the multiplication of two square matrices of single-precision floating point values taking into account four different matrix sizes, 1024×1024, 2048×2048, 4096×4096 and 8192×8192. All test programs were compiled using `g++-4.7.2`. Also, in order to assess the quality of our approach, the performance of our HPL implementation tuned by means of a genetic search process is compared to the performance of two OpenCL state-of-the-art implementations, namely clBLAS 2.4 [21] and ViennaCL 1.5.1 [19]. We have selected these implementations because HPL is also currently based on OpenCL, they can be executed in the same range of platforms as our HPL adaptive code, and they also support some kind of adaptive behavior depending on the underlying hardware. We now briefly describe these libraries.

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

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

Table 2. Speedups achieved by best versions found

GPU profiling. We have modified it to be able to profile also the hardware of the rest of our testing platforms.

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

Table 2 shows the performance results for the three implementations on the four tested platforms. The third column contains the execution time in milliseconds and the performance measured in GFLOPS of the best kernel found by our genetically tuned HPL implementation. The fourth and fifth columns shows the speedup achieved with respect to the clBLAS and ViennaCL implementations. Figures 1.a) to 1.d) compare the performance in GFLOPS of clBLAS and ViennaCL to that of our implementation for each problem size and platform. Let us recall that the kernels of all the implementations have been previously adapted to the underlying hardware by means of their respective profiling and tuning procedures. The results show that our implementation outperforms these two implementations for all matrix sizes and on the four platforms with the sole ex-

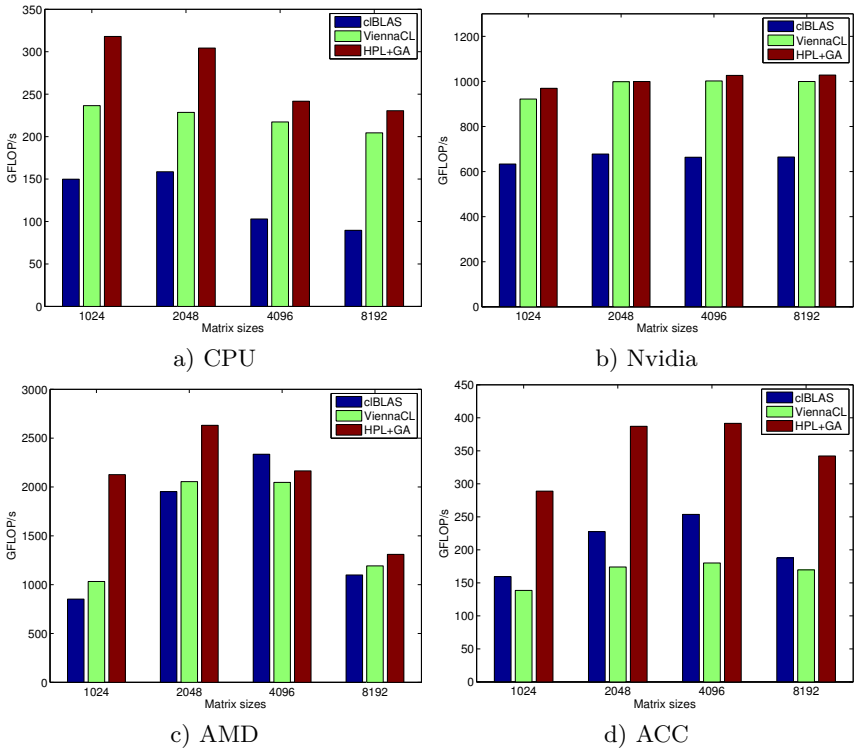

Figure 1. Performance in GFLOPS of clBLAS, ViennaCL and HPL best versions

ception of matrix multiplication of size 4096 in the AMD platform. In this case, our HPL implementation is beaten narrowly by the clBLAS implementation. The average speedup of our approach is 1.74 with respect to clBLAS and 1.44 with respect to ViennaCL. Compared to clBLAS, our implementation achieves a peak speedup of 2.57 in the CPU platform for the 8192 size. The peak speedup with respect to ViennaCL is 2.22 and it is achieved in the ACC platform for the 2048 size. All the comparisons were done against the corresponding optimized versions generated by both clBLAS and ViennaCL for each different problem size. These best-kernels are, on average, 12 times faster than those found in [22]. This improvement is a consequence of the application of new techniques to generate a performance-portable code and some generic optimizations applied to the matrix multiplication algorithm.

Table 3 shows the best values of the parameters of the HPL generic matrix multiplication kernel found by the genetic algorithm. These parameters have been explained in Table 1. The Table shows that the values selected for each platform and for each problem size are different, and they are difficult to predict using a single general heuristic. A pattern can be observed in the values taken

Device	Size	(szx,szy)	(lszx,lszy)	(bszx,bszy)	(tW,uf)	(vA,vB,vC)	copy (A,B)	order
CPU	1024	(256,64)	(8,64)	(16,4)	(32,1)	(8,8,8)	(2,0)	201
	2048	(512,128)	(8,128)	(16,4)	(32,1)	(8,8,8)	(2,0)	201
	4096	(1024,256)	(2,256)	(16,4)	(256,8)	(16,16,16)	(1,0)	012
	8192	(2048,512)	(32,32)	(16,4)	(32,4)	(16,16,16)	(2,0)	201
Nvidia	1024	(128,256)	(2,64)	(4,8)	(32,2)	(2,4,4)	(2,0)	210
	2048	(512,256)	(4,64)	(8,4)	(256,4)	(2,4,4)	(2,0)	102
	4096	(512,512)	(16,16)	(8,8)	(32,2)	(2,2,2)	(2,0)	102
	8192	(1024,1024)	(2,128)	(8,8)	(32,2)	(4,8,8)	(2,0)	210
AMD	1024	(256,128)	(4,32)	(8,4)	(128,1)	(4,8,8)	(2,0)	102
	2048	(256,256)	(1,128)	(8,8)	(256,2)	(4,8,8)	(2,0)	120
	4096	(512,512)	(4,16)	(8,8)	(32,2)	(4,8,8)	(2,0)	012
	8192	(1024,1024)	(1,128)	(8,8)	(32,4)	(4,8,8)	(2,0)	012
ACC	1024	(256,64)	(1,16)	(16,4)	(8,2)	(1,16,16)	(0,0)	120
	2048	(256,128)	(1,8)	(16,8)	(512,8)	(8,16,16)	(0,0)	120
	4096	(2048,256)	(16,32)	(16,2)	(32,1)	(8,16,16)	(2,0)	201
	8192	(4096,512)	(16,16)	(16,2)	(32,1)	(16,2,2)	(2,0)	021

Table 3. Configuration of the best versions found using our approach

by some parameters within the same platform, but they cannot be easily found a priori.

Table 4 contains the time consumed by the tuning procedures conducted by our genetic algorithm, the clBLAS profiler and the ViennaCL auto-tuner. On average, our genetic search is 1.18 times faster than the clBLAS profiler. For the CPU and ACC platforms, the sum of times consumed by our genetic search for each matrix size is competitive in relation to that consumed by the clBLAS profiler. In the Nvidia and AMD platforms, both composed of GPUs, the clBLAS search procedure is quite faster, which is understandable taking into account that it is specifically directed to this kind of devices. The results also show that the ViennaCL auto-tuner is 160 times slower than our genetic search procedure. This large difference is undoubtedly due to the time-consuming exhaustive search it conducts. As for the search times of our tool, despite covering much more optimization parameters and techniques than [22], they are 2.57 times shorter than those reported in [22].

7 Conclusions

We have presented a generic implementation of the matrix multiplication based on RTCG techniques exploited thanks to the use of the HPL embedded language for kernels. As a result, a dozen of parameters allow to tune this implementation for the different platforms and problem sizes. The search of the best values for these parameters is guided by a genetic algorithm where each individual is evaluated using its execution time. This implementation illustrates and proves the effectiveness of a set of techniques to build a performance-portable implementa-

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

Table 4. Total times for tuning procedures

tion of any algorithm in HPL. They offer an alternative to complex auto-tuning libraries or complex source-to-source compilation tools.

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

As future work, we are planning to implement mechanisms that allow to automatically apply these techniques to any HPL code with a minimal intervention by the programmer.

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). The authors are also members of the CAPAP-H5 network, in whose framework the paper has been developed.

References

1. Munshi, A., Gaster, B., Mattson, T.G., Fung, J.: OpenCL Programming Guide. Addison-Wesley Professional (2011)
2. Wernsing, J.R., Stitt, G.: Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. In: Proc. ACM SIGPLAN/SIGBED 2010 conf. on Languages, compilers, and tools for embedded systems. (2010) 115–124
3. Moore, N., Leeser, M., Smith King, L.: VForce: An environment for portable applications on high performance systems with accelerators. J. Parallel Distrib. Comput. **72**(9) (2012) 1144–1156
4. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing **38**(8) (Aug 2012) 391–407
5. Dastgeer, U., Enmyren, J., Kessler, C.W.: Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems. In: Proc. 4th Intl. Workshop on Multicore Software Engineering. IWMSE '11 (2011) 25–32
6. Lutz, T., Fensch, C., Cole, M.: PARTANS: An autotuning framework for stencil computation on multi-GPU systems. ACM Trans. Archit. Code Optim. **9**(4) (January 2013) 59:1–59:24
7. Thoman, P., Kofler, K., Studt, H., Thomson, J., Fahringer, T.: Automatic OpenCL device characterization: Guiding optimized kernel design. In Jeannot, E., Namyst, R., Roman, J., eds.: Euro-Par 2011 Parallel Processing. Volume 6853 of Lecture Notes in Computer Science. Springer-Verlag (2011) 438–452
8. Lan, Q., Xun, C., Wen, M., Su, H., Liu, L., Zhang, C.: Improving performance of GPU specific OpenCL program on CPUs. In: Proc. 13th Intl. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'12). (2012) 356–360
9. Shen, J., Fang, J., Sips, H., Varbanescu, A.: Performance traps in OpenCL for CPUs. In: Proc. 21st Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP 2013). (2013) 38–45
10. Daloukas, K., Antonopoulos, C.D., Bellas, N.: GLOpenCL: OpenCL support on hardware- and software-managed cache multicores. In: Proce. 6th Intl. Conf. on High Performance and Embedded Architectures and Compilers. (2011) 15–24
11. Fabeiro, J.F., Andrade, D., Fraguela, B.B., Doallo, R.: Automatic generation of optimized OpenCL codes using OCLoptimizer. The Computer Journal **58**(11) (Nov 2015) 3057–3073
12. Dolbeau, R., Bodin, F., de Verdiere, C.: One OpenCL to rule them all? (2013)

13. Viñas, M., Bozkus, Z., Fraguela, B.B.: Exploiting heterogeneous parallelism with the Heterogeneous Programming Library. J. Parallel Distrib. Comput. **73**(12) (December 2013) 1627–1638
14. Beckmann, O., Houghton, A., Mellor, M., Kelly, P.H.J.: Runtime code generation in C++ as a foundation for domain-specific optimisation. In: Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers. Volume 3016 of Lecture Notes in Computer Science. Springer Verlag (2004) 291–306
15. Newburn, C., So, B., Liu, Z., McCool, M., Ghuloum, A., Toit, S.D., Wang, Z.G., Du, Z., Chen, Y., Wu, G., Guo, P., Liu, Z., Zhang, D.: Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In: 9th IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO 2011). (2011) 224–235
16. Cao, C., Dongarra, J., Du, P., Gates, M., Luszczek, P., Tomov, S.: clMAGMA: High performance dense linear algebra with OpenCL. In: International Workshop on OpenCL (IWOCL). (2013) 13–14
17. Kurzak, J., Tomov, S., Dongarra, J.: Autotuning GEMM kernels for the Fermi GPU. IEEE Transactions on Parallel and Distributed Systems **23**(11) (Nov 2012) 2045–2057
18. Matsumoto, K., Nakasato, N., Sedukhin, S.: Implementing a code generator for fast matrix multiplication in OpenCL on the GPU. In: 2012 IEEE 6th Intl. Symp. on Embedded Multicore Socs (MCSoC). (Sept 2012) 198–204
19. Tillet, P., Rupp, K., Selberherr, S., Lin, C.T.: Towards performance-portable, scalable, and convenient linear algebra. In: 5th USENIX Workshop on Hot Topics in Parallelism, Berkeley, CA, USENIX (2013)
20. Wall, M.: GAlib: A C++ Library of Genetic Algorithm Components. (1996)
21. AMD: clBLAS. https://github.com/clMathLibraries/clBLAS (2015) [Online; accessed 17-May-2016].
22. Fabeiro, J.F., Andrade, D., Fraguela, B.B., Doallo, R.: Writing self-adaptive codes for heterogeneous systems. In: Proc. 20th Intl. Conf. Euro-Par 2014 Parallel Processing. (2014) 800–811