Writing a performance-portable matrix multiplication

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

[a] *Computer Architecture Group, Universidade da Coruña, Spain*

Abstract

There are several frameworks that, while providing functional portability of code across different platforms, do not automatically provide performance portability. As a consequence, programmers have to hand-tune the kernel codes for each device. The Heterogeneous Programming Library (HPL) is one of these libraries, but it has the interesting feature that the kernel codes, which implement the computation to be performed, are generated at run-time. This run-time code generation (RTCG) capability can be used, in conjunction with generic parameterized algorithms, to write performance-portable codes. In this paper we explain how these techniques can be applied to a matrix multiplication algorithm. The performance of our implementation is compared to two state-of-the-art adaptive implementations, clBLAS and ViennaCL, on four different platforms, achieving average speedups with respect to them of 1.74 and 1.44, respectively.

Keywords: GPGPU, Heterogeneous Systems, OpenCL, Performance portability, Embedded languages

1. Introduction

Performance portability is an open problem in heterogeneous systems. As a consequence, programmers usually have to hand-tune the code of a given algorithm for each platform where it will be executed in order to maximize its performance [1, 2, 3]. The Heterogeneous Programming Library (HPL) [4] is a C++ framework that simplifies the portable programming of heterogeneous systems. This library puts emphasis on improving the programmability of these systems by providing an interface that is noticeably simpler than other alternatives like OpenCL [5].

The library provides a programming model similar to OpenCL, where a kernel, which expresses the parallel computation, is spawned to a given

device generating several threads. In fact, the current backend of this library is built on top of OpenCL. An interesting characteristic of the library is that the kernel code is translated into OpenCL at run-time. This run-time code generation (RTCG) capability can be used to adapt the code to the properties of the computing device where the code is going to be executed, since they are known at run-time.

This work illustrates the creation of performance-portable HPL kernels. These kernels receive a set of optimization parameters that are used inside the kernel to guide RTCG and generic optimization techniques. For example, RTCG can be used to unroll a loop, being the input optimization parameter the unroll factor. Another example would be a tiling transformation, a generic tile size being the input parameter of this optimization. Our approach complements these RTCG kernels with a a genetic algorithm that chooses the values of the input optimization parameters guided by the execution time of the versions generated at run-time.

The usage of some of these techniques has been shown and evaluated in [6]. The current work focuses on the matrix multiplication code, improving [6] in several points: (1) two new techniques, vectorization and instruction scheduling, are applied to generate performance-portable kernels, which generates a best-kernel 12 times faster than in [6] (2) the illegal combinations of parameters are discarded during the generation phase of the genetic algorithm, which reduces the search time on average 2.57 times with respect to [6], and (3) 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 [7], as it relies on clBLAS for its OpenCL BLAS routines.

Our matrix multiplication implementation is based on existing implementations for NVIDIA GPUs [8], AMD GPUs [9], and any kind of devices supporting OpenCL [10]. 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 than those previous ones.

The rest of this paper is organized as follows. Section 2 introduces the basic concepts of the Heterogeneous Programming Library (HPL). Then, Section 3 summarizes the new optimization techniques introduced in this paper with respect to [6]. Section 4 explains the implementation details

of our matrix multiplication, and how a genetic search is used to tune its parameters. This is followed by the experimental results in Section 5 and a discussion of related work in Section 6. Finally, Section 7 is devoted to our conclusions and future work.

2. The Heterogeneous Programming Library library

The Heterogeneous Programming Library (HPL), available for download under GPL license at `http://hpl.des.udc.es`, improves the programmability of heterogeneous systems. Codes written using HPL can be executed across a wide range of devices. In addition, programmers can exploit performance portability on top of HPL using its run-time code generation (RTCG) mechanism. This mechanism is present in HPL kernels, which are written using a language embedded in C++. This code is executed at run-time and it translates the HPL computational kernel into the HPL's intermediate representation (IR), currently OpenCL.

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

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

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

the host, and (4) the private memory, which is only accessible within each thread.

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

The host API. The most important component of this API is the `eval` function, which requests the execution of a kernel `f` with the syntax `eval(f)(arg1, arg2, ...)`. The execution of the kernel can be parameterized by inserting methods calls between `eval` and the argument list. For example, by default, the global size is equal to the size of the first argument, whereas the local size is automatically selected by the library. Yet, this default behavior can be overridden by specifying alternative global and local sizes, using methods called `global` and `local` respectively. This way, if we want to define a 200×400 global domain divided into 2×4 local domains, the function `eval` should be invoked as follows `eval(f).global(200, 400).local(2, 4)(a, b)`. Listing 1 contains an HPL implementation of a matrix-vector product. The main procedure of this code contains an example host code for a matrix-vector product, where a global domain of M threads and local domains grouping 10 threads each are defined.

The template class `Array`. The variables used in a kernel must have type `Array<type, ndim [, memoryFlag]>`. This type represents an n-dimensional array of elements of a C++ `type`, or a scalar for `ndim`=0. Scalars and vectors can also be defined with special data types like `Int`, `Float`, `Int4`, `Float8`, etc. The `Array` optional `memoryFlag` either specifies one of the kinds of memory supported (`Global`, `Local`, `Constant` or `Private`). The default value of the `memoryFlag` is `Global`, the exception being the `Array`s declared inside the body of kernels, which are placed by default in `Private` memory. The elements that compose an array may be any of the usual C++ arithmetic types or a struct. The arrays passed as parameters to the kernels must be declared in the host using the same syntax. These variables are initially stored in the host memory, but when they are used as kernel parameters they are automatically transferred to the device. Similarly, the outputs are automatically transferred back to the host when needed.

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

Listing 1: 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_(i=0, i<(M/szx), i++)
7        for_(k=0, k<N, k++)
8          y[idx*(M/szx)+i] += a[idx*(M/szx)+i][k] * x[k];
9    }
10 };
11
12 int main(...) {
13   //Declare and initialize ax, xv and yv arrays
14   MxV matvec;
15   eval(matvec).global(M).local(10)(av, xv, yv);
16 }
```

(`if_`, `for_`, ...). Also, the arguments passed to `for_` loops are separated by commas instead of semicolons. In addition, the library provides an API based on predefined variables to obtain the global, local and group identifiers as well as the sizes of the domains and numbers of groups. For example, `idx` provides the global identifier of the first dimension, while `szx` provides the global size of that dimension. Adding the `l` prefix to this keywords allows to obtain their local counterparts, whereas replacing the letter `x` with `y` or `z` the same values are obtained for the second and the third dimensions respectively.

Kernels are written as regular C++ functions or functor classes that use these elements and whose parameters are passed by value if they are scalars, and by reference otherwise. The `MxV` class at the top of the code of Listing 1 contains an example of an HPL kernel implementing a matrix-vector product by means of a functor. In this kernel, each thread processes the multiplication of `M/szx` consecutive rows of matrix `a` by vector `x`.

3. Performance portability on OpenCL

The HPL library can be used to provide performance portability across different architectures. The work presented in [6] already showed how the run-time code generation (RTCG) capabilities of the library, in combination

with generic programming techniques, can be used to perform loop unrolling, to choose among different implementations of the same algorithm, to adjust the granularity of the work to be performed by each thread and to decide whether or not local memory is used. The algorithm presented in this paper uses all these techniques to build a performance-portable matrix multiplication. Moreover, it uses new methods to provide performance portability in HPL: a technique that allows to reorder several loops at run-time, and another technique that allows to dynamically change the vector length in vectorized codes. The matrix-vector product (MxV) HPL kernel in Listing 1 will be used as starting point to introduce these two techniques.

Loop interchange and instructions scheduling. Loop interchange, whenever it is 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 2 shows an example of how this technique is applied to the matrix-vector product HPL kernel. In the original version presented in Listing 1, 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 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

Listing 2: 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 matvec;
21   matvec.set_order(0,1); // sets o[0]=1 and p[o[0]]=p[1]=0
22   matvec.set_order(1,0); // sets o[1]=0 and p[o[1]]=p[0]=1
23   eval(matvec).global(szx)(av, xv, yv);
24 }
```

`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

Listing 3: 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> matvec;
24   eval(matvec).global(M)(av, xv, yv);
25 }
```

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 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 appropri-

ate 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 3 is a vectorized version of the original matrix-vector product of Listing 1 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` 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 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. Also, the new techniques described in Section 3 are applied. Firstly, the inner loops of the algorithm are fully

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

unrolled and the instructions are reordered using the instructions scheduling technique. Secondly, 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.

Section 4.1 describes the details of the implementation of our HPL matrix multiplication kernel. Next, Section 4.2 explains how a genetic search is used to find the best values for the parameters of our algorithm in each device.

4.1. Kernel implementation

The implementation of our kernel relies on a number of tunable parameters that will be introduced across the explanation and which are summarized in Table 1 for ease of reference. As explained in Sect. 2, the first two elements in the table are the standard HPL variables that provide the size of the global domain, which describes the total number of threads that execute the kernel in parallel, in the second (`szy`) and the first dimension (`szx`). Similarly, the next two ones describe the corresponding dimensions of the local domain, which provide the size of the groups of threads, or work-groups following OpenCL terminology. In our kernel the domains are associated to the dimensions of the destination matrix, and as we can see from the description in Table 1, its rows are distributed across the second dimension of the domain, while the columns are mapped on the first dimension.

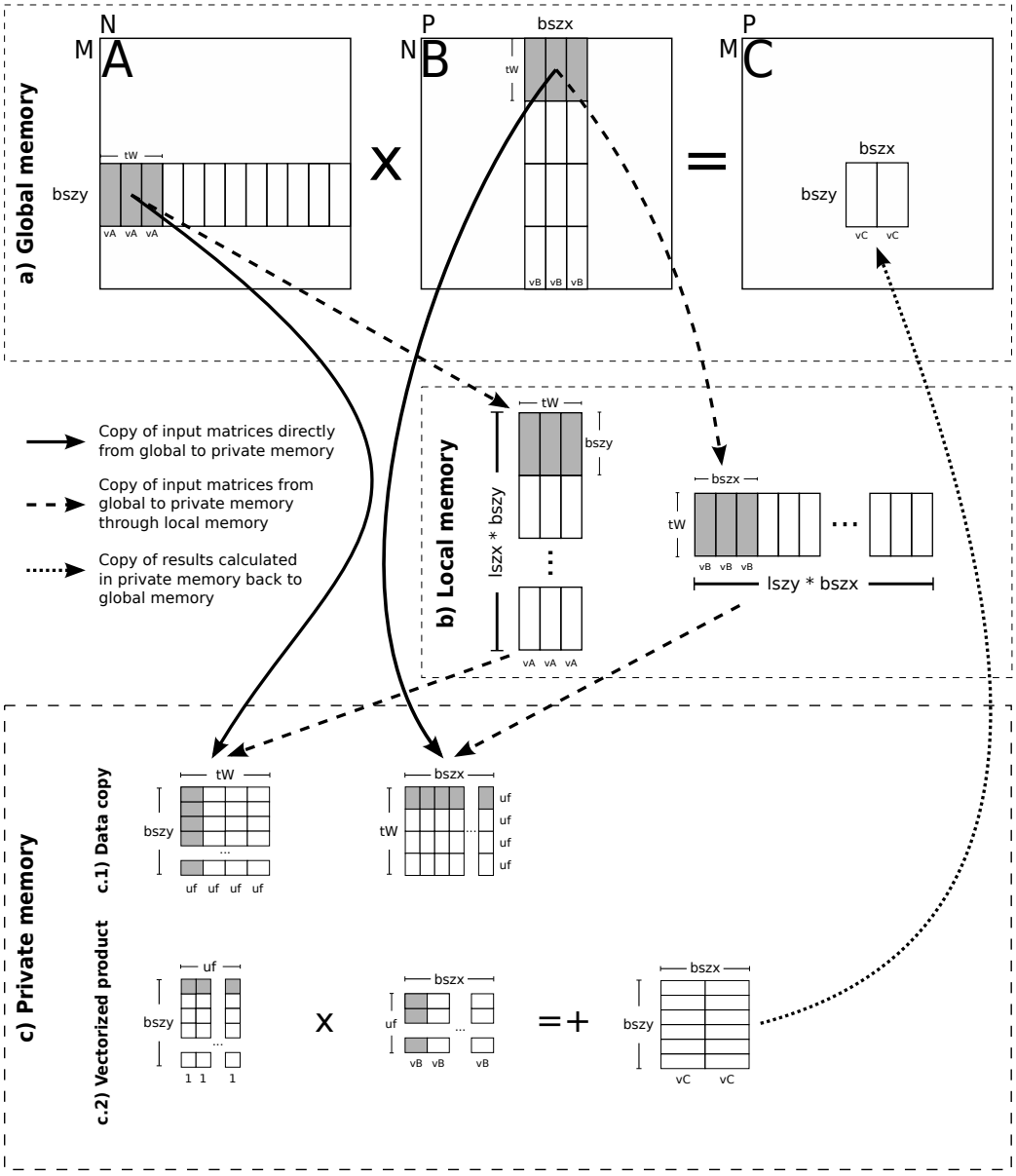

Figure 1: Matrix multiplication generic algorithm

Figure 1 shows how the work is partitioned in tiles across the threads and how global, local and private memory regions are used. The top part, Figure 1.a, shows that each thread calculates a tile of `bszy` × `bszx` elements of the resulting matrix `C` by multiplying `bszy` rows of matrix `A` and `bszx` columns of matrix `B`. The tiling technique is also applied to the work to be performed in this computation. The shared dimension of matrices `A` and `B` (the columns of `A` and the rows of `B`) is partitioned into tiles of size `tW`. The local memory shared among the threads of the same group can be used to accelerate data loading. Figure 1.b shows how a tile of `lszx` × `bszy` rows and `tW` columns of matrix `A` is loaded into local memory collaboratively by the threads of the same group. Using the same method, a tile of `tW` rows and `lszy` × `bszx` columns of matrix `B` can be loaded into local memory. Let us notice that the dimensions of the block size and the local size are crossed. This combination consistently delivers better performance than its complementary, and more natural, alternative. The information of matrices `A` and `B` is loaded vectorially using vectors of size `vA` and `vB`, respectively. Once this information is loaded into local memory, each thread calculates its tile of the resulting matrix `C`. This is a good point to introduce the parameters in Table 1 related to vectorization. Values of `vA`, `vB` and `vC` are used to define the vector size used to move data from `A` and `B`, and to `C`, respectively. The two latter ones, `vB` and `vC`, are also used to define the lengths of vectors used in the innermost loops that perform the computation. Figure 1.c.1 shows that matrix `A` is loaded into private memory in tiles of `bszy` × `uf` elements and `B` in tiles of `uf` × `bszx` elements. Figure 1.c.2 shows that these tiles are multiplied vectorially. At tile level, the innermost loop iterates on the `N/tW` tiles of size `bszy` × `tW` in which the set of `bszy` rows of `A` assigned to the thread can be partitioned, multiplying each one of them by the same tile of `tW` × `bszx` elements of `B`. Similarly, the product of `bszy` complete rows of `A` and `bszx` complete columns of `B` that is required to calculate a complete tile of `bszy` × `bszx` elements of `C` is processed across different iterations of another outer loop.

Notice that each input matrix can be loaded first into local memory, and then into private memory. The usage of local memory theoretically accelerates the loading of the matrices. However, in some architectures there may not be enough local memory or its usage can slow down the application [3, 11]. For this reason, the local memory can be bypassed, in which case data will be directly loaded from global to private memory. For each architecture, local memory can be used for loading both, none or one of the

input matrices. This is selected using `copyA` and `copyB` parameters in Table 1. These parameters are used to determine whether matrices `A` and/or `B` have to be copied first to local memory or directly to private memory. For each matrix, the corresponding flag can take values either of 0, when no data is going to be copied to local memory, or 1 or 2, otherwise. In this latter case, when the flag takes the value 1 our kernel implementation will try to allocate exactly the local memory space needed to store tiles of `A` of size (`lszx` $\times$ `bszy`) $\times$ `tW` or tiles of `B` of size `tW` $\times$ (`lszy` $\times$ `bszx`). If the flag takes a value of 2, it tries to allocate space for an additional column for each tile, to avoid possible bank conflicts.

The pseudo-code in Figure 2 shows a simplified version of the algorithm followed by each thread to calculate a complete `bszy` $\times$ `bszx` tile of `C`. For simplicity, this algorithm assumes that the local memory is used as a gateway between global and private memory and that vector lengths `vB` and `vC` are equal. The local variables to load a (`lszx` $\times$ `bszy`) $\times$ `tW` tile of `A` and a `tW` $\times$ `(lszy` $\times$ `bszx)` tile of `B` are declared in lines 3 and 6. Lines 9 and 11 declare the private variables to load `bszy` $\times$ `uf` elements of `A` and `uf` $\times$ `bszx` elements of `B`. Finally, the private variable `c` where the resulting `bszy` $\times$ `bszx` tile of `C` is stored is declared in line 13. Notice that each element of arrays `b` and `c` are vectors of size `vB` and `vC` respectively. This enables vectorization when the multiplication is done.

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

The three innermost nested loops in lines 29 to 33 perform vectorially the multiplication of a subtile of `bszy` $\times$ `uf` elements of `a` by another subtile of `bszy`$\times$ `bszx` elements of `b`. The result is stored in a private matrix `c`. These

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

Figure 2: Calculation of a single block of C using local memory

three loops are native C++ `for` loops, thus, they will be fully unrolled at run-time. In our implementation, these loops can be dynamically reordered, as we have applied the techniques described in Section 3. The selection is made by the `order` parameter in Table 1, which is a vector of three elements that encodes the selected order . Once a thread has completed the calculation of its tile of `C`, the instruction in line 40 copies back the resulting matrix from the private copy in `c` to the appropriate positions of the global matrix `C`.

4.2. Genetic search

The values of the parameters summarized in Table 1 are tuned using a genetic algorithm (GA) [12]. In order to implement this GA search we have used the GAlib genetic algorithm package [13]. Genetic algorithms initially create a population of individuals which are characterized by a set of genes. If the current population meets the fitness criteria, the genetic algorithm finishes, if not, the genetic algorithm generates a new population by generation, crossing and/or mutation.

In our case, individuals are versions of the matrix multiplication and their genes are each one of the parameters of Table 1. The initial population is generated randomly. Individuals for the subsequent generations are generated using random values for each parameter (generation) or by combining the genes of two individuals (crossing). Also, once these new individuals have been generated, some of their genes can be mutated following some rules (mutation). These mutations are not just random, as they intend to generate combinations around the area of the search space where the best solutions are usually found. The fitness criteria is that an individual with a faster kernel execution time have not been found for the last five generations. When the search concludes, such individual with the best kernel execution time is selected.

The values taken by genes of one individual have to match certain conditions. These conditions can be imposed by HPL, the matrix multiplication algorithm, or the properties of the device where the generated kernel will be executed. For example, HPL restricts the local size to be not greater than the global size, whereas the algorithm used to implement the matrix multiplication forces the tile width `tW` to be not greater than the common dimension `N` of matrices `A` and `B`. In addition, the device must have free memory space enough to perform the multiplication, and this restriction is directly related to the selected sizes for the global and the local domains and tile width, among other parameters.

The default generation and combination operations of the GA are overloaded to check in advance if the parameters match these conditions. If not, the individual is discarded. Notice, that the high number of parameters increase the probability that a generated individual is invalid. This checking mechanism discards these individuals before they are tested, thus, the effect on the search time of these defective individuals is negligible.

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

Table 2: Minimum conditions of validity for GA individuals

Table 2 summarizes the conditions that must be matched by the values taken by the genes of an individual. These conditions prevent things like: too large workspaces that can generate too many idle threads, local workspaces larger than the global ones, and vector sizes or unroll factors that are incompatible with the block size, the tile size or the problem size.

Despite the limitations imposed by the conditions included in Table 2, the genetic algorithm still has a large range of possible values to explore for each gene, which gives place to a large number of possible individuals. In order to increase the effectiveness of the genetic search, additional conditions have been imposed in order to keep the values of some parameters within ranges that have heuristically shown to contain the optimal solutions to our problem. This reduces the search time and it helps to reach a better solution. In detail, both dimensions of the global domain have been limited to a minimum size

of 128 when the algorithm is run in GPUs, and to a minimum size of 64 otherwise. These heuristic conditions are added to the mandatory conditions shown in Table 2 and they are checked before an individual is qualified as valid.

5. 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 [14] and ViennaCL 1.5.1 [10]. 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 GPU profiling. We have modified it to be able to profile also the hardware of the rest of our testing platforms.

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

Table 3 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 3.a) to 3.d) compare the performance in GFLOPS of clBLAS and ViennaCL to that of our implementation for each problem size and platform. Let us recall that the kernels of all the implementations have been previously adapted to the underlying hardware by means of their respective profiling and tuning procedures. The results show that our implementation outperforms these two implementations for all matrix sizes and on the four platforms with the sole exception of matrix multiplication of size 4096 in the AMD platform. In this case, our HPL implementation is

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

Table 3: Speedups achieved by best versions found

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 made against the corresponding optimized versions generated by both clBLAS and ViennaCL. Let us notice that clBLAS and ViennaCL also tunes the code for each different problem size. These best-kernels are, on average, 12 times faster than those found in [6]. 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 4 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

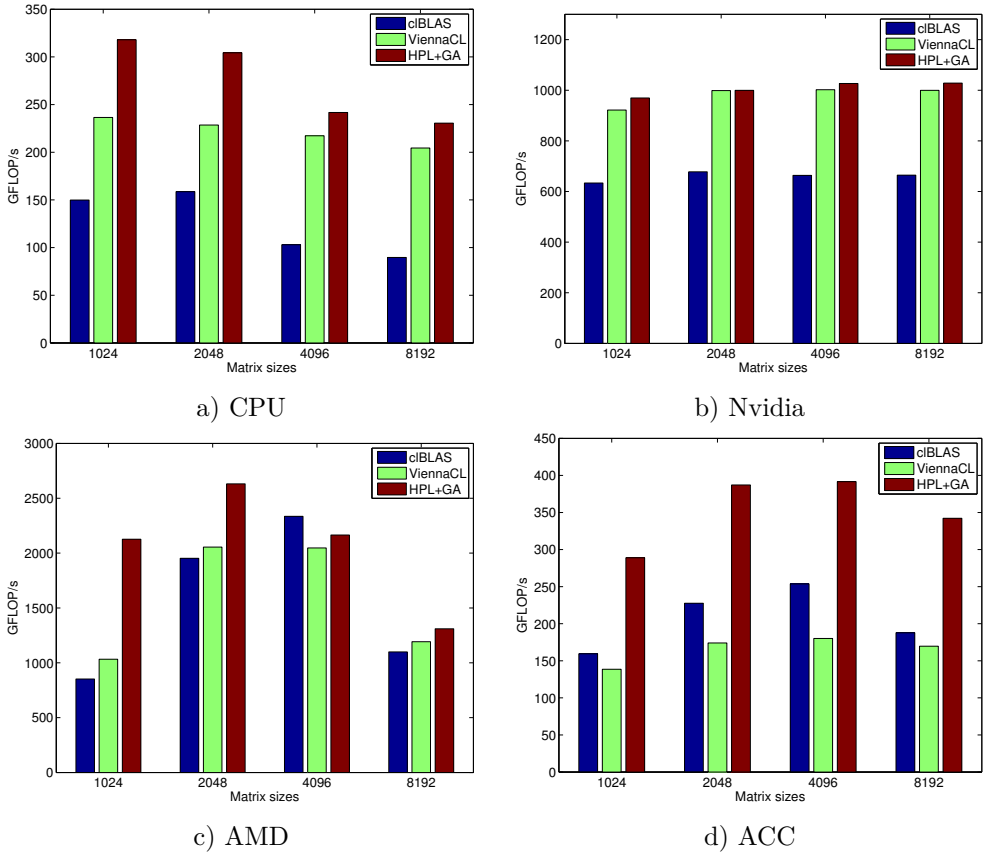

Figure 3: Performance in GFLOPS of clBLAS, ViennaCL and HPL best versions

predict using a single general heuristic. A pattern can be observed in the values taken by some parameters within the same platform, but they cannot be easily found a priori.

Table 5 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

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 4: Configuration of the best versions found using our approach

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 [6], they are 2.57 times shorter than those reported in [6]. This is a consequence of the improvements in the search process that have been explained in Section 4.2.

6. Related work

Matrix multiplication is an algorithm extensively studied in the bibliography for multiple kinds of devices, including Nvidia [8] and AMD [9] GPUs. Some of these works focus on the study of several linear algebra operations. For example, ViennaCL [10] provides an OpenCL implementation of several linear algebra routines, including the matrix multiplication. Their approach is based in a generic version of the matrix multiplication where the parameters are either fixed heuristically or using an auto-tuner driven by the execution time. ViennaCL is, to the best of our knowledge, the best-performing OpenCL implementation of the matrix multiplication. Their auto-tuner obtains worse performance results than our implementation and, in addition,

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 5: Total times for tuning procedures

the search times are several orders of magnitude larger than ours. The reason is that they run an exhaustive search process, instead of an informed one like our HPL implementation does by means of a genetic algorithm.

clMAGMA [7] introduces an OpenCL version of the MAGMA library [15]. They use clBLAS to implement BLAS routines, including the matrix multiplication operation. Thus, our comparison to clBLAS is valid for this library.

There are more approaches that try to achieve performance portability of linear algebra problems through iterative processes. For example, [16] uses iterative compilation to select the optimal parameters for GPU codes according to a set of pre-defined parameterized templates. They have 10 parameters, while we tune 14 parameters. They do not report the execution times of their autotuner. We obtain a better performance, but obviously we are using newer architectures.

Matsumoto et al [17] automatically generate and tune several parametrized OpenCL versions of the $A^T B$ variant of GEMM routine. These versions are implemented following different algorithms devoted to exploit specific fea-

tures of different kinds of devices. Moreover, the search process conducted consisted on an exhaustive search of the fastest kernels among tens of thousands of heuristically previously chosen versions. Notice that the execution time measured for each kernel included the time consumed by the transposition of matrix A.

Other approaches are more general and they focus on a wider range of applications. For example, a simple model based on both hardware and application parameters is used in [18] to build an OpenCL performance-portable implementation of data streams clustering and generate tuned versions of it for several NVIDIA and AMD GPUs.

The OCLoptimizer [19] source-to-source optimization tool searches optimal unroll factors for OpenCL kernels based on compiler directives and a configuration file. It also selects the optimal global and local workspaces.

The Periscope Tuning Framework [20] provides an automated evaluation of the search space to tune performance and energy efficiency. In GPGPUs it targets applications written in the pipeline patterns framework, and it tries to increase the throughput of the pipeline.

Orio [21] is an open-source extensible framework for the generation and autotuning of code for several hardware architectures. It targets code written in programming languages such as C or Fortran. Functions to be tuned by Orio must be implemented in a parametrized way similar to that of our HPL kernels, and then annotated with complex directives in order to provide the framework with the information needed to conduct the autotuning process. Both OrCUDA [22] and OrCL [23] are built on top of Orio, expanding its functionalities in order to generate respectively CUDA and OpenCL optimized code. OrCUDA [22] is used to tune some stencil-based computations for different NVIDIA GPUs, whereas OrCL [23] produces OpenCL optimized codes for several NVIDIA and AMD GPUs, and an Intel Xeon Phi accelerator. It targets several numerical kernels used in iterative sparse linear systems solution and in parallel simulations of solid fuel ignition

Complex computations are usually tuned by selecting the best implementations for the different numerical routines of which they are composed. Nitro [24] is a framework that provides programmers with a mechanism to manage collections of these building blocks and also information related to their performance in different platforms and for different applications. This information is used to train the framework about how to select optimal combinations of variants of those routines in order to solve different kinds of problems, such as sparse matrix operations, conjugate gradient solvers, breadth-

first search algorithms, histogram calculations, and sorting operations.

Another relevant topic is the study of the interest of performance portability. In this vein, Dolbeau et al [25] discuss the variations in the obtained performance using the same OpenCL code on different platforms. They also use the CAPS compiler to generate auto-tuned OpenCL code.

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 implementation 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] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, H. Kobayashi, Evaluating performance and portability of OpenCL programs, in: Proc. Fifth Intl. Workshop on Automatic Performance Tuning (iWAPT 2010), 2010.

[2] Q. Lan, C. Xun, M. Wen, H. Su, L. Liu, C. Zhang, Improving performance of GPU specific OpenCL program on CPUs, in: Proc. 13th Intl. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'12), 2012, pp. 356–360.

[3] J. Shen, J. Fang, H. Sips, A. Varbanescu, Performance traps in OpenCL for CPUs, in: Proc. 21st Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP 2013), 2013, pp. 38–45.

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

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

[6] J. F. Fabeiro, D. Andrade, B. B. Fraguela, R. Doallo, Writing self-adaptive codes for heterogeneous systems, in: Proc. 20th Intl. Conf. Euro-Par 2014 Parallel Processing, 2014, pp. 800–811.

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

[8] J. Kurzak, S. Tomov, J. Dongarra, Autotuning GEMM kernels for the Fermi GPU, IEEE Transactions on Parallel and Distributed Systems 23 (11) (2012) 2045–2057.

[9] K. Matsumoto, N. Nakasato, S. Sedukhin, Implementing a code generator for fast matrix multiplication in OpenCL on the GPU, in: 2012 IEEE 6th Intl. Symp. on Embedded Multicore Socs (MCSoC), 2012, pp. 198–204.

[10] P. Tillet, K. Rupp, S. Selberherr, C.-T. Lin, Towards performance-portable, scalable, and convenient linear algebra, in: 5th USENIX Workshop on Hot Topics in Parallelism, USENIX, Berkeley, CA, 2013.

[11] J. Shen, J. Fang, H. Sips, A. L. Varbanescu, An application-centric evaluation of OpenCL on multi-core CPUs, Parallel Computing 39 (12) (2013) 834 – 850.

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

[13] M. Wall, GAlib: A C++ Library of Genetic Algorithm Components, 1996.

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

[15] P. D. S. Tomov, R. Nath, MAGMA: Matrix algebra on GPU and multicore architectures (2011).

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

[17] K. Matsumoto, N. Nakasato, S. Sedukhin, Performance tuning of matrix multiplication in OpenCL on different GPUs and CPUs, in: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:, 2012, pp. 396–405.

[18] J. Fang, A. Varbanescu, H. Sips, An auto-tuning solution to data streams clustering in OpenCL, in: Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on, 2011.

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

[20] Y. Oleynik, R. Mijaković, I. A. C. Ureña, M. Firbach, M. Gerndt, Recent advances in Periscope for performance analysis and tuning, in: Tools for High Performance Computing 2013, Springer, 2014, pp. 39–51.

[21] A. Hartono, B. Norris, P. Sadayappan, Annotation-based empirical performance tuning using Orio, in: Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, 2009, pp. 1–11.

[22] A. Mametjanov, D. Lowell, C.-C. Ma, B. Norris, Autotuning stencil-based computations on GPUs, in: Proc. 2012 IEEE Intl. Conf. on Cluster Computing, 2012, pp. 266–274.

[23] N. Chaimov, B. Norris, A. Malony, Toward multi-target autotuning for accelerators, in: Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on, 2014, pp. 534–541.

[24] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, B. Catanzaro, Nitro: A framework for adaptive code variant tuning, in: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, 2014, pp. 501–512.

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