# Performance Portable GPU Code Generation for Matrix Multiplication

Toomas Remmelg Thibaut Lutz Michel Steuwer Christophe Dubach
University of Edinburgh
{toomas.remmelg, thibaut.lutz, michel.steuwer, christophe.dubach}@ed.ac.uk

## Abstract

Parallel accelerators such as GPUs are notoriously hard to program; exploiting their full performance potential is a job best left for ninja programmers. High-level programming languages coupled with optimizing compilers have been proposed to attempt to address this issue. However, they rely on device-specific heuristics or hard-coded library implementations to achieve good performance resulting in non-portable solutions that need to be re-optimized for every new device.

Achieving performance portability is the holy grail of high-performance computing and has so far remained an open problem even for well studied applications like matrix multiplication. We argue that what is needed is a way to describe applications at a high-level without committing to particular implementations. To this end, we developed in a previous paper a functional data-parallel language which allows applications to be expressed in a device neutral way. We use a set of well-defined rewrite rules to automatically transform programs into semantically equivalent device-specific forms, from which OpenCL code is generated.

In this paper, we demonstrate how this approach produces high-performance OpenCL code for GPUs with a well-studied, well-understood application: matrix multiplication. Starting from a single high-level program, our compiler automatically generate highly optimized and specialized implementations. We group simple rewrite rules into more complex *macro-rules*, each describing a well-known optimization like tiling and register blocking in a composable way. Using an exploration strategy our compiler automatically generates 50,000 OpenCL kernels, each providing a differently optimized – but provably correct – implementation of matrix multiplication. The automatically generated code offers competitive performance compared to the manually tuned MAGMA library implementations of matrix multiplication on Nvidia and even outperforms AMD's clBLAS library.

***Categories and Subject Descriptors*** D.3.4 [*Processors*]: Code generation, Compilers, Optimization

***Keywords*** Performance Portability, GPU, Code Generation, Matrix Multiplication, High-level Parallel Programming



## 1. Introduction

GPUs (Graphic Processing Units) are the most successful parallel accelerators used in high-performance computing. However, producing high-performance code for such devices is extremely difficult as GPU programming languages (e. g., CUDA, OpenCL) expose many low-level hardware details. The memory hierarchy needs to be managed explicitly and memory accesses have to be carefully handled to avoid memory bank conflicts and ensure coalescing. The code also explicitly controls the mapping of parallelism at multiple levels: work-groups, threads, warps, and vector units. The resulting low-level device-tailored code is ultimately not performance portable.

There have been many efforts to address the challenge of programmability and performance portability. Projects such as Delite [19], Lime [6] or SkelCL [17] show promising results and greatly simplify GPU programming. They all exploit functional concepts such as composability, immutability and absence of side-effects to present a high-level interface to the programmer. However, their implementations still rely on manually optimized code or on device-specific compiler heuristics, which are not performance portable.

We argue that solving this problem requires exposing optimization choices in the compiler. Petabricks [2] is a good example of such an approach, where programmers define alternative implementations of their algorithms. The compiler and runtime then select the most appropriate choices. However, Petabricks is limited to a set of manually provided application specific options.

In [18], we propose to use a functional intermediate representation in the compiler and to express algorithmic and optimization choices in a unified rule-rewriting system. The functional representation provides an abstraction to reason about parallel programs at the algorithmic level and is used by programmers in a similar way to Delite [19]. In addition, an OpenCL-specific extension of the intermediate representation is used internally to functionally express OpenCL paradigms, such as vectorization, mapping of parallelism and data movement between the different address spaces. The rewrite rules define the optimization space in a formal way, transforming the program seamlessly between different algorithmic and low-level OpenCL-specific forms.

In this paper, we extend our original approach and show how it works in practice using matrix multiplication as a case study. Matrix multiplication is arguably one of the most studied applications in computer science. It is a fundamental building block of many scientific and high performance computing applications. Even though it has been studied extensively for many years, traditional compiler techniques still

do not deliver performance portability automatically. Naïve implementations of matrix multiplication deliver very poor performance on GPUs; programmers are forced to manually apply advanced optimizations to achieve high performance, as we will see in Section 2. These optimizations are not portable across different GPUs, making manual optimization costly and time-consuming.

To the best of our knowledge, we are the first to present a fully automated compilation technique which generates high performance GPU code for matrix multiplication for different GPUs from a single portable source program. Our approach achieves this by combining algorithmic and GPU specific optimizations to generate thousands of provably correct implementations. Using a pruning strategy, we generate and run 50,000 OpenCL kernels implementing matrix multiplication on three GPUs from AMD and Nvidia. The best implementations found match or exceed the performance of several high-performance GPU libraries on all platforms.

Our paper makes the following key contributions:

- An automated technique for generating high-performance code from a single portable high-level representation of matrix multiplication;
- Well-known optimizations for matrix multiplication are expressed as provably correct and composable *macro-rules*;
- An exploration strategy based on heuristics for generating 50,000 differently optimized OpenCL kernels;
- Experimental evidence that our approach matches the performance of highly tuned CUDA and OpenCL implementations on different GPUs.

The remainder of the paper is structured as follows. Section 2 provides a motivation. Section 3 gives an overview of our functional data-parallel language and compiler intermediate representation, while section 4 introduces our rewrite rules and how they are used to encode optimizations. Section 5 explains our exploration and compilation strategy. Sections 6 and 7 show our experimental setup and results. Finally, section 8 discusses related work and section 9 concludes.

## 2. Motivation

In this section we illustrate the shortcomings of existing GPU compilers to produce high-performance code from easy to write naïve implementations using matrix multiplication as an example. This results in a difficulty of writing high performing OpenCL programs requiring in-depth knowledge of various hardware characteristics.

The difficulty to achieve high performance motivates the need for new compilation techniques capable of automatically producing code close to manually optimized implementations from an easy to write high-level program.

***Easy to Write Version*** Figure 1 shows the OpenCL kernel of an easy to write naïve matrix multiplication implementation using a 2D thread space. The rows of matrix A and the columns of matrix B are mapped to the first and second dimension of the iteration space using the thread indices `gid0` and `gid1`. The for-loop performs the dot-product of a row of A and a column of B in line 6. The final statement stores the result into matrix C.

While this version is easy to write, no existing compiler can generate efficient code from it, despite many years of fruitful research on automatic compiler optimizations. Advanced optimizations like the usage of local memory, tiling, or register blocking are not applied automatically by compilers.

```
1 kernel mm(global float* A, B, C, int N, K, M) {
2   int gid0 = global_id(0);
3   int gid1 = global_id(1);
4   float acc = 0.0f;
5   for (int i=0; i<K; i++)
6     acc += A[gid1*K+i]*B[i*M+gid0];
7   C[gid1*M+gid0] = acc;
8 }
```

**Figure 1:** Naïve OpenCL kernel for matrix multiplication.

```
1  kernel mm_amd_opt(global float * A, B, C,
2                    int K, M, N) {
3  local float tileA[512]; tileB[512];
4
5  private float acc_0;      ...; acc_31;
6  private float blockOfB_0; ...; blockOfB_3;
7  private float blockOfA_0; ...; blockOfA_7;
8
9  int lid0 = local_id(0); lid1 = local_id(1);
10 int wid0 = group_id(0); wid1 = group_id(1);
11
12 for (int w1=wid1; w1<M/64; w1+=num_grps(1)) {
13  for (int w0=wid0; w0<N/64; w0+=num_grps(0)) {
14
15   acc_0 = 0.0f; ...; acc_31 = 0.0f;
16   for (int i=0; i<K/8; i++) {
17    vstore4(vload4(lid1*M/4+2*i*M+16*w1+lid0,A)
18            ,16*lid1+lid0, tileA);
19    vstore4(vload4(lid1*N/4+2*i*N+16*w0+lid0,B)
20            ,16*lid1+lid0, tileB);
21    barrier(...);
22
23    for (int j = 0; j<8; j++) {
24     blockOfA_0 = tileA[0+64*j+lid1*8];
25     ... 6 more statements
26     blockOfA_7 = tileA[7+64*j+lid1*8];
27     blockOfB_0 = tileB[0 +64*j+lid0];
28     ... 2 more statements
29     blockOfB_3 = tileB[48+64*j+lid0];
30
31     acc_0  += blockOfA_0 * blockOfB_0;
32     acc_1  += blockOfA_0 * blockOfB_1;
33     acc_2  += blockOfA_0 * blockOfB_2;
34     acc_3  += blockOfA_0 * blockOfB_3;
35     ... 24 more statements
36     acc_28 += blockOfA_7 * blockOfB_0;
37     acc_29 += blockOfA_7 * blockOfB_1;
38     acc_30 += blockOfA_7 * blockOfB_2;
39     acc_31 += blockOfA_7 * blockOfB_3;
40    }
41    barrier(...);
42
43   }
44
45   C[ 0+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_0;
46   C[16+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_1;
47   C[32+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_2;
48   C[48+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_3;
49   ... 24 more statements
50   C[ 0+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_28;
51   C[16+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_29;
52   C[32+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_30;
53   C[48+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_31;
54 } } }
```

**Figure 2:** Hand-optimized OpenCL kernel for fast matrix multiplication on an AMD GPU.

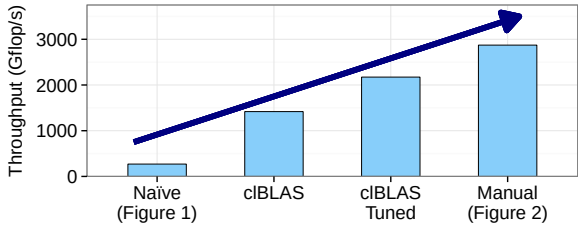



**Figure 3:** Performance comparison of matrix multiplication implementations on an AMD GPU.

***Manually Optimized Version*** Figure 2 shows a manually optimized version of matrix multiplication tuned for an AMD GPU. This version performs a tiled matrix multiplication [12, 14] using local memory. Register blocking [14] is used where each tile is further partitioned into smaller blocks stored in registers. Please notice that figure 2 shows a shortened version omitting similar declarations (e. g., see line 5) and statements (e. g., see line 25). The original source code is 268 lines long.

The implementation in figure 2 takes advantage of many hardware features such as vectorized loads and local memory, which involves the use of synchronization primitives. The parallelism is decomposed and mapped in a very specific way, taking advantage of the thread hierarchy and increasing registers usage using register blocking. In more detail, copying the tiles into local memory is performed in lines 17–21. Lines 24–26 and lines 27–29 perform register blocking for tile of A and B respectively. Lines 31–39 perform a partial dot-product between a block of tile A and B and accumulate temporary results in private memory. Once all the partial dot-products have been computed and accumulated, lines 45–53 store the final result of the dot-product into global memory.

***Performance Comparison*** Figure 3 shows the performance comparison of the two versions of matrix multiplication shown in figure 1 and figure 2 together with two versions from the AMD clBLAS library. The clBLAS library provides an expert written implementation of matrix multiplication. In addition, a tuning script is provided for automatically choosing implementation parameters for specific GPUs.

We can see from figure 3 that the clBLAS library version performs 5× better than the naïve version, the tuned library version 8× better and the hand-optimized version even 10× better. It is obvious from this data – and maybe not very surprising – that current OpenCL compilers fail to automatically reach the performance of optimized libraries or hand-tuned kernels staring from a naïve version. Manual optimizations are still crucial in OpenCL to achieve high performance and it is often possible to beat highly optimized library implementations with manual optimizations and specializations.

Ideally, programmers should write simple programs like the naïve version and automatically obtain the performance of the hand-tuned one.

***Towards High-Performance Code from High-Level Programs*** We argue that automatically producing high-performance code is possible if we start from a high-level functional program representation and keep it in the compiler pipeline for as long as possible. To this end we define a functional intermediate representation [18] and encode compiler optimizations as rewrite rules which transform the program into semantically equivalent optimized forms. The rewrite rules express choices available to the compiler such as how parallelism is exploited, where data is stored, or if vectorization is applied.

This design offers two main advantages: first, a functional representation ensures that high-level semantic information is available to the compiler, reducing the need for complicated static analysis; secondly, the transformations expressed by the rewrite rules are composable and provably correct, guaranteeing correctness of the generated specialized code. As we will see, this design based on a functional representation of programs leads to a compiler that produces high-performance code like that shown in figure 2 from a high-level program comparable to the one shown in figure 1.

## 3. A Functional Language for Data Parallelism

This section presents our high-level language and how it is used for matrix multiplication. Programmers use a set of composable data-parallel primitives to express their programs.

We use a functional intermediate representation to preserve the primitives' high-level parallel semantics which are exploited by the compiler to perform complex optimizations.

We discuss how the compiler performs optimizations on its internal representation in section 4.

### 3.1 Language Tour by Example

Figure 4 shows matrix multiplication expressed in our domain specific language embedded in *Scala*. The code shown is the entire input from which our compiler generates high-performance OpenCL code. This representation is purely algorithmic and does not encode any optimization decisions, which will be made automatically by our compiler later on.

The computation of the dot-product is first defined on line 1. The *zip* function combines two arrays into a single array of pairs. The *map* function applies a given function, in this case multiplication (`mult`), to each element of the input array. In the example, this results in performing a pairwise multiplication of arrays `a` and `b`. Finally, the *reduce* function performs a reduction using the given function (`add`).

In the definition of matrix multiplication starting on line 6 the two input matrices `A` and `B` are applied to the *map* function on line 7 and 8. Each matrix is represented as a two-dimensional array, where the outer dimension contains the rows and each row is represented as an array of elements. To access the columns of matrix *B* we transpose it, i. e., switch rows and columns, on line 9. Therefore, one can also read these lines as: "For each row of *A* and each column of *B* compute the dot-product".

The types for the matrices `A` and `B` are given on lines 4 and 5 as nested two-dimensional arrays. We encode the array length as part of the type using the variables `N`, `M`, and `K`. This allows to specify that the length of a row of matrix `A` must match the length of a column of matrix `B`.

```
1 val dotProduct = fun( (a, b) =>
2   reduce(add, 0.0f, map(mult, zip(a, b))) )
3
4 val mm = fun( Array(Array(Float, M), K),
5               Array(Array(Float, K), N),
6  (A, B) =>
7   map(fun(aRow =>
8    map(fun(bCol =>
9     dotProduct(aRow,bCol)), transpose(B))), A))
```

**Figure 4:** Source code of matrix multiplication in our functional language embedded in Scala. This is the entire input from which our compiler generates efficient GPU code.

```
1 dotProduct(a, b) = reduce(+, 0, map(×, zip(a, b)))
2 matrixMult(A, B) =
3   map(λ rowA .
4     map(λ colB .
5       dotProduct(rowA, colB), split(n, reorder(idxF, join(B))), A)
```

**Figure 5:** Functional compiler intermediate representation of matrix multiplication.

### 3.2 Compiler intermediate representation

Internally, our compiler uses a functional intermediate representation (IR), introduced in [18], which consists of variables, like $A$ or $b$, and three types of functions:

1. predefined scalar functions(+, ×);
2. data-parallel primitives, like *map* or *reduce* (see Table 1);
3. *lambda expressions*, anonymous functions which can nowadays be found in all popular programming languages, written as $\lambda\, a\, .\, body$ in this paper.

One important note is that these functional expressions are read from right to left instead of the familiar left-to-right direction in imperative programming.

Table 1 shows a summary of our data-parallel primitives. The *algorithmic primitives* correspond directly to building blocks that are exposed in our Scala front-end. The *OpenCL primitives* represent OpenCL paradigms and are directly mapped to OpenCL code in the back-end.

***Matrix Multiplication*** For the rest of this paper, we will use the compiler intermediate representation of matrix multiplication as shown in figure 5. This is a straight forward translation from the matrix multiplication program written in our domain specific language shown in figure 4.

The *map*, *zip*, and *reduce* functions used in the original code are represented with the corresponding primitives in our IR to preserve their high-level semantics. The *transpose* function used in the original program to transpose matrix $B$ is represented with three primitives in our IR:

$$transpose = split(n) \circ reorder(idxF) \circ join$$

The circle operator ($\circ$) denotes function composition, i. e., $f(g(x)) = (f \circ g)(x)$. For transposition, first the *join* primitive concatenates all rows of a two-dimensional matrix; then the *reorder* primitive reorders the flattened array based on an index function *idxF*, which implements the transposition by remapping the array indices appropriately; finally, the *split(n)* primitive does the opposite of *join* and turns a one-dimensional array into a two-dimensional matrix with rows of a given length $n$. These three operations are not actually performed in the generated code. Instead, each primitive produces a compiler internal data structure which influences how the following primitive accesses its input data. We will discuss this in more detail in section 4.

### 3.3 GPU Specific Extensions

The representation of matrix multiplication in figure 5 captures the algorithmic perspective, but it is far from obvious how efficient GPU code is generated from this program. To address this issue, we designed an extension to our IR specifically targeted for GPU programming and tailored for the hardware characteristics of modern GPUs. Each OpenCL primitive from Table 1 is used to express a specific feature exposed in the OpenCL programming model.

| **Algorithmic Primitives** | | **OpenCL Primitives** | |
|---|---|---|---|
| *map* | *zip* | *mapWorkgroup* | *toGlobal* |
| *reduce* | *split* | *mapLocal* | *toLocal* |
| *reorder* | *join* | *mapSeq* | *toPrivate* |
| | | *vectorize* | *gather* |
| | | *toVector* | *scatter* |
| | | *toScalar* | |

**Table 1:** Overview of our functional primitives.

***GPU Thread Hierarchy*** GPUs are organized as multicore processors with each core executing multiple threads concurrently. In OpenCL, this concept is represented by *workgroups*, which contain multiple *local threads*, each of which performs *sequential* work.

Similarly, primitives exist in our language to exploit this hierarchy in the form of the *mapWorkgroup*, *mapLocal* and *mapSeq* primitives. These variations of the algorithmic *map* have the same high-level meaning: they apply the given function to its input array. At code generation time, the *mapWorkgroup* will produce OpenCL code which distributes the work among workgroups for instance.

***Scalar and Vector Units*** OpenCL exposes the possibility to use vector units as data types. Declaring a variable of type `float4` for instance, implies that the operations performed on this variable are executed by vector units. If the hardware does not support vector operations, the code is scalarized automatically.

In our IR, we support vectorization of data and, to a limited form, functions. Vectorization of data is indicated by the use of the *toVector* and *toScalar* primitives. To vectorize a function $f$ it can be wrapped in the *vectorize(f)* primitive. Currently only simple functions, like + or ×, are supported. We plan to incorporate work on whole-function vectorization [7] in the future.

***GPU Memory Hierarchy*** OpenCL defines *memory address spaces* to reflect the memory hierarchy found in modern GPUs. Most data resides in *global memory* but programmers can explicitly move data into the faster but smaller *local memory*. Variables in *private memory* are stored in registers.

In our IR, copies between memory spaces are explicitly encoded using the *toGlobal*, *toLocal*, and *toPrivate* primitives. These primitives accept a function $f$ as its argument and indicate that $f$ should write its result in the specified memory space.

***Memory Coalescing*** GPUs are very sensitive to how data in the global memory is accessed. It is generally beneficial for concurrently executing threads to access data in small blocks of a few hundred bytes. These memory accesses are *coalesced* by the hardware into a single access instead of issuing multiple requests.

Our compiler can influence the order in which threads access memory using the *reorder* primitive. This primitives can be either lowered into a *gather* or *scatter* primitive. The *gather* will reorder the memory reads of the following primitive while *scatter* will reorder the writes of the preceding primitive. We will discuss in section 4 when it is legal to automatically use this primitive to ensure coalesced memory accesses.

### 3.4 Summary

Our functional IR together with its GPU extensions allows for a very precise description of which GPU features should be exploited, e. g., how the computation should be mapped to the thread hierarchy using the different variants of *map*. This enables the compiler to represent different variations of the matrix multiplication example using a unified IR, where each variant exploits a different set of GPU features.

It is easy to generate code using the OpenCL specific patterns, as optimization decisions are encoded explicitly and each primitive directly corresponds to a templated OpenCL code. In the next section, we discuss how optimizations are expressed as sequences of provably correct rewrite rules.

## 4. Optimizing by Rewriting

This section discusses how we optimize programs represented in our IR and transform them into forms exploiting GPU features explicitly. Furthermore, we show how our encoding of optimizations as sequences of rewrite rules enables us to freely combine optimizations and apply them automatically in an exploration process described in section 5.

### 4.1 Rewrite Rules

A *rewrite rule* is a well-defined transformation of an expression represented in our IR. Each rule encodes a simple – and provably correct – rewrite. For instance, the *fusion rule* combines two successive *map* primitives into a single one:

$$map(f) \circ map(g) \rightarrow map(f \circ g)$$

We currently have a few dozens of rewrite rules encoded in our system. For space reasons, we will only present a few here in detail, but all rewrite rules are very similar in style. The correctness of these rules has been proven following the same methodology as presented in [18].

In addition to the rules describing purely algorithmic transformations, there are also rules lowering the algorithmic primitives to OpenCL specific primitives. For example, the algorithmic *map* primitive can be mapped to any of the OpenCL specific *map* primitives, i. e., *mapWorkgroup*, *mapLocal*, or *mapSeq*, as long as the OpenCL thread hierarchy is respected.

Interesting interactions exist between the algorithmic and OpenCL specific rules. For example, the algorithmic *split-join* rule transforms a *map* primitive following a divide-and-conquer style:

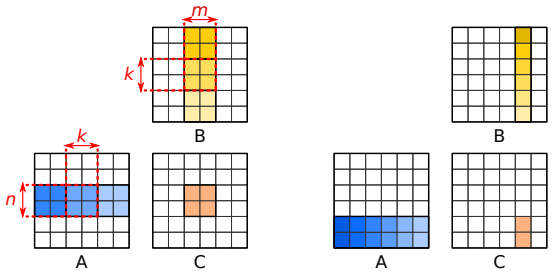
$$map(f) \rightarrow join \circ map(map(f)) \circ split(n)$$

Here the *split(n)* primitive divides the input into chunks of size $n$, which are processed by the outer *map* and each single chunk is processed by the inner *map*. Finally, the *join* primitive collects and appends all results. This rule transforms a flat one-dimensional *map* primitive into a nested expression which can easily be mapped to the OpenCL thread hierarchy, e. g.:

$$join \circ mapWorkgroup(mapLocal(f)) \circ split(n)$$

This interaction allows our compiler to explore different strategies of mapping algorithmic expressions to the GPU hardware. In the example above, the parameter $n$ directly controls the amount of work performed by the workgroups and local threads which is an important tuning factor.

In the following section, we discuss how these simple rewrite rules are combined to express rich optimizations which are crucial for applications like matrix multiplication.




**(a)** Tiling **(b)** Register Blocking

**Figure 6:** Example of two classical optimizations for matrix multiplication.

### 4.2 Tiling

Tiling is a common optimization used on CPUs and GPUs [5, 12, 14] and is highly beneficial for our matrix-matrix multiplication use case application. The idea behind tiling is to increase data locality by fitting small portions of data into local memory. Then the computations are performed while the data is in the fast memory region.

In the context of matrix multiplication, we want to create 2D tiles for the output matrix C. Our compiler achieves this by splitting each input matrix along both dimensions, so that they are decomposed into multiple *tiles*, which are multiplied in local memory. Figure 6a visualizes this situation. The highlighted tiles of matrices A and B are multiplied in local memory and summed up to compute a tile of matrix C.

***Building the tiles*** Interestingly, we do not need to create new constructs to build the tiles and can reuse the exact same primitives described in [18]. We reuse *map* and *split*, and the high-level function *transpose* presented earlier, to produce a tiled representation of matrix $A$ (or $B$):

$$tile(n, k, A) = map(map(transpose) \circ split(k) \circ transpose) \circ split(n, A)$$

The first *split(n)* divides $A$ into chunks of $n$ rows. The second *split(k)* after the *transpose* divides each chunk into 2D tiles of size $n \times k$. To get the original orientation back, we apply *transpose* to each tile. The reuse of our unmodified primitives illustrate the power of composition and shows that larger building block can be build on top of a very small set of primitives. This makes the design of the compiler easier since the compiler only need to handle a very small set of primitives and does not need to know about higher-level building blocks such as *transpose* or *tile*.

***Copying to local memory*** Once the tiles built, we need to copy them to local memory. Scalar values can be copied using the identity function ($id(x) = x$). From this, an array is copied by composing *map* and *id*. To copy a two dimensional tile, we use *id* nested inside two *map*s:

$$copy2D = map(map(id))$$

Composing this function with the *toLocal* primitive, an array is copied into local memory:

$$toLocal(copy2D)$$

***Combining everything*** If we apply these basic principles to both matrices and use *zip* to combine them, we obtain the second expression shown in figure 7. The *tile* function is

used twice to tile both matrices in the last two lines. Line 11 combines a row of tiles of matrix $A$ with a column of tiles of matrix $B$ using the *zip* primitive. Lines 9 and 10 copy individual tiles into local memory. The computation of dot-product remains unchanged (line 8) and is nested in two *map* primitives, now operating on pairs of tiles instead on entire matrices. To compute matrix multiplication in a tiled fashion, we have to add up the intermediate results computed by multiplying the pairs of tiles. This is done using the *reduce* primitive introduced in line 4 combined with the + operation used in line 5 to add up two tiles. Finally, the computed results are copied back into global memory in line 3.

This complex transformation is achieved by applying simple rewrite rules like the ones presented earlier in this section. As each of these simple rules is provably correct, by composition the bigger transformations are automatically valid as well. This is a major advantage compared to traditional compiler techniques, where complex analysis is required to apply such big optimization steps.

### 4.3 Register Blocking

Register blocking is another traditional optimization technique [14]. The idea is to swap nested loops such that a data item is loaded into a register and during the execution of the inner loop, this item is reused while iterating over a block of data from the other matrix. Figure 6b shows register blocking for matrix multiplication. Here each element of the highlighted column of B is reused while iterating over a single column of the highlighted block of A.

We represent this optimization by swapping nested *map* primitives as shown in the third expression in figure 7. We start by *split*ing *tileA* on line 14 to form multiple blocks of rows. For combining multiple rows of *tileA* with a single column of *tileB* we transpose the resulting blocks of rows of A (*aBlocks*) before using *zip* on line 12. Then we apply *map* (line 9) to obtain a pair of elements of *tileA* (*aBlock*) together with a single element of *tileB* (*b*). We copy *b* into private memory (*bp*) and reuse it on line 10 while iterating over *aBlock* using the *map* primitive.

### 4.4 General Optimizations

Numerous rewrite rules encoding small and generic optimizations exist in our system. These rules are applied to simplify the expression, to avoid unnecessary intermediate results, vectorize functions, or to ensure memory coalescing when accessing global GPU memory. We discuss a few examples in this section.

***Simplification Rules*** Earlier we already saw a fusion rule combining two *map* primitives. A similar rule to fuse a combination of *map* and *reduceSeq* also exists:

$$reduceSeq(f, id) \circ map(g) \rightarrow reduceSeq(\lambda(acc, x)\ .\ f(acc, g(x)), id)$$

This rule avoids an intermediate array produced by the *map (g)* primitive, as the function $g$ is applied to all elements of the input array inside the reduction immediately before the element is combined with the reduction operator $f$.

***Vectorization*** The following rule is applied to make use of the vector units in the hardware. It rewrites a *map* primitive into a vectorized version:

$$map(f) \rightarrow toScalar \circ map(\ vectorize(f)\ ) \circ toVector(n)$$

For example, this rule can be applied to vectorize copying of a tile into local memory which is a technique advocated by AMD in their example OpenCL codes [1].

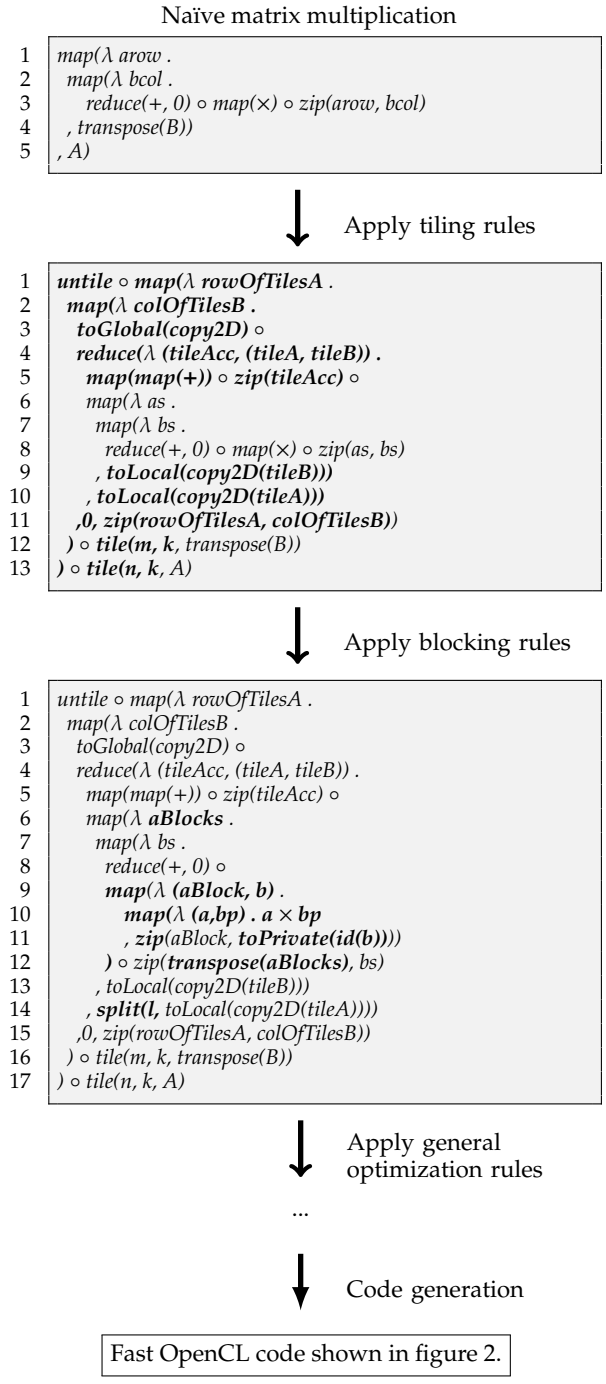



**Figure 7:** Transforming matrix multiplication by combining optimizations.

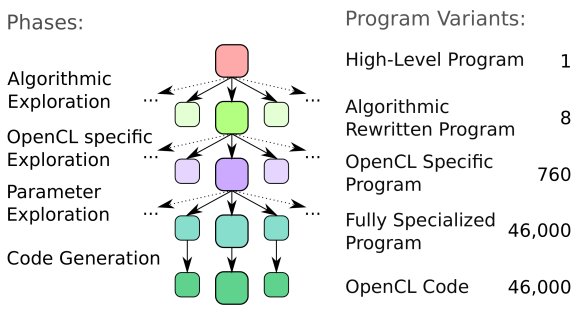



**Figure 8:** Exploration and compilation strategy

***Memory Coalescing*** In section 3 we introduced the *reorder* primitive, which allows us to specify an index function to reorder an array. It is important to point out, that this reordering is not performed in the generated code by producing a reordered array. Instead, the index computation required to perform the reordering is delayed until the next primitive accesses the input array. This is similar to lazy evaluation. Therefore, a *reorder* primitive effectively controls how the following primitive will access its input array.

We can take advantage of this design by applying the following rewrite rule:

$$map(f) \rightarrow reorder(stride^{-1}) \circ map(f) \circ reorder(stride)$$

This rule rewrites an arbitrary *map* primitive to access its input array in a strided fashion, enabling memory coalescing. To ensure correctness, the reordering has to be undone, by reordering the computed array with the inverse index function as used before. In situation where each thread processes multiple data elements in *f*, this transformation ensures that these elements are accessed in a coalesced way.

### 4.5 Summary

In this section, we discussed examples of rewrite rules and how they are used to implement complex optimizations. Furthermore, we have seen in figure 7 how these optimizations are combined to transform a simple program into a more optimized and specialized form. We eventually reach a program from which our compiler generates OpenCL code similar to the highly optimized code shown in figure 2 in the motivation section. Because the rewrite rules are well-defined and proven to be correct, we can automate their application and explore different optimizations for a single program, as we will discuss in the next section.

## 5. Exploration and Compilation Strategy

This section describes how we compile a single high-level program, as seen in figure 4, to OpenCL code by applying rewrite rules automatically to explore different optimization choices. Figure 8 gives an overview of our exploration and compilation strategy. For matrix multiplication, we start from a single high-level program to generate 46,000 OpenCL kernel in four phases, which we discuss in the following: algorithmic exploration, OpenCL specific exploration, parameter exploration, and code generation.

### 5.1 Algorithmic Exploration Using Macro Rules

By design, each rewrite rule encodes a simple transformation. As discussed in the previous section, more complex optimizations are achieved by composition.

We decided to guide the automatic rewrite process by grouping rewrite rules together into *macro rules* which encode bigger transformations. A macro rule aims to achieve a particular optimization goal, such as apply tiling or blocking. These macro rules are more flexible than the simple rules. They try to apply different sequences of rewrites to achieve their optimization goal, whereas a simple rewrite rule always performs exactly the same transformation. For example, it might be required to first rewrite the source expression into a form where the rewrites performing the actual optimization (e. g., tiling) can be applied.

To explore different algorithmic optimization choices, we encoded 4 macro rules: 1D blocking, 2D blocking, tiling, and a tiling optimization applied to the innermost loop. Starting from the high-level matrix multiplication program in figure 5, we apply these macro rules at all valid locations in an arbitrary order leading to approximately 20,000 different variations.

In order to reduce the search space, we discard programs which are unlikely to deliver good performance on the GPU using two heuristics. The first heuristic limits the depth of the nesting in the program: some rules are always applicable, however they are unlikely to improve performance after exploiting all levels and dimensions of the OpenCL thread hierarchy. Using the first heuristic we decided to focus on around one hundred rewritten programs. The second heuristic looks at the distance between the addition and multiplication operations. A small distance increases the likelihood of fusing these two instructions together and avoiding intermediate results. The number of expressions after applying the second heuristic is reduced to 8, which are then passed to the next phase.

### 5.2 OpenCL Specific Exploration

For each algorithmically rewritten program, we explore different mapping strategies to the GPU. We chose a fixed mapping strategy for the OpenCL thread hierarchy: the two outermost *map* primitives are turned into *mapWorkgroup* primitives to perform these computations across a two-dimensional grid of workgroups. The next two *map*s are rewritten into *mapLocal* primitives to exploit the parallelism inside of a two-dimensional workgroup. Finally, all further nested *map* primitives will be executed sequentially. This strategy is common in GPU programming.

For the memory hierarchy, we explored the usage of local and private memory. We limited the number of copies into each memory space to two, to avoid expressions which perform many meaningless copies.

Starting from the 8 algorithmically rewritten programs, we automatically generate 760 OpenCL specific programs with a particular mapping decision encoded.

### 5.3 Parameter Exploration

Every OpenCL specific program contains parameters, e. g., the argument to *split(n)* controlling the size of a tile or a block. We performed an automatic exploration of these parameters by exhaustively picking all possible parameter values in a reasonable range. Furthermore, we make sure that the parameters picked will not generate an OpenCL kernel requiring too much private, local, or global memory. We also discard parameter combinations leading to an unreasonably small or high number of workgroups or local threads.

For the 760 OpenCL specific programs we generate around 46,000 fully specialized programs.

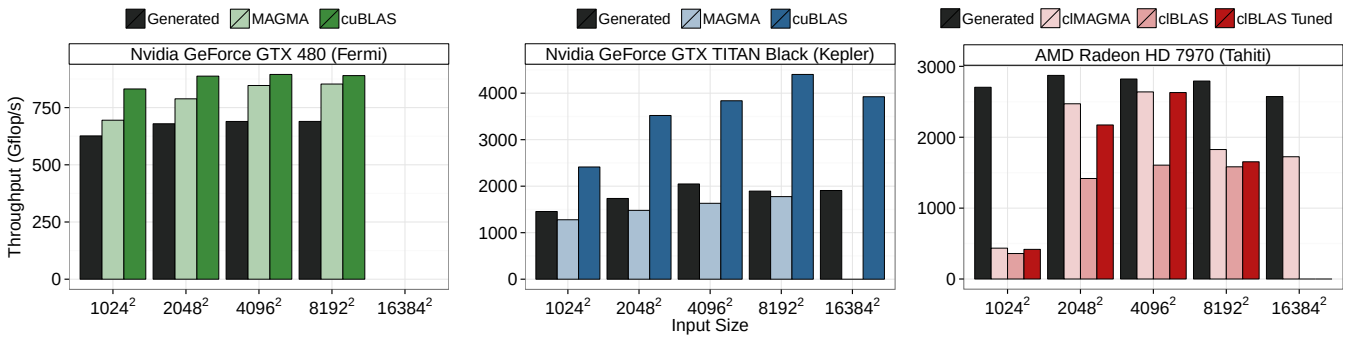



**Figure 9:** Performance results for matrix multiplication compared against high-performance BLAS libraries.

### 5.4 Code Generation

For every fully specialized program we generate an OpenCL kernel. Each fully specialized program encodes a different implementation of matrix multiplication with a different set of optimizations applied.

Code generation is straightforward, as all optimization and implementation decisions have been made by the previous exploration phases and the program only contains OpenCL specific primitives (as introduced in section 3, table 1). For each primitive a corresponding OpenCL code snippet is emitted to generate the final OpenCL kernel.

### 5.5 Summary

By defining rewrite rules and expressing larger optimizations with them, we are able generate tens of thousands of OpenCL kernel which are all correct by construction. This enables us to explore combinations of the tiling and register blocking optimizations combined with strategies for mapping expressions to GPUs and numerical parameters. In section 7 we will discuss the performance results, but first we briefly discuss our experimental setup in the next section.

## 6. Experimental Setup

### 6.1 Implementation Details

We implemented the functional language in Scala, taking advantage of Scala's flexible syntax. The rewrite rules are encoded using pattern matching. For applying a rewrite rule, we check if the specified pattern matches and if that is the case we perform the rewrite. Our internal design is similar to the higher-order IR presented in [11] and we encode information about array sizes in a custom type system.

### 6.2 Hardware Platforms and Input Sizes

We used three platforms: 1) a Nvidia GTX Titan Black (Kepler architecture) using CUDA 6.0 and driver 331.79; 2) a Nvidia GTX 480 (Fermi architecture) using CUDA 6.5 and driver 340.65; 3) a AMD HD 7970 (Tahiti architecture) using AMD APP SDK 2.9.214.1 and driver 1526.3.

We compare against the following high-performance BLAS libraries: clMAGMA 1.3.0, MAGMA 1.6.1, clBLAS 2.4.0, and the cuBLAS version bundled with CUDA.

We perform experiments using single precision floating point numbers with matrix sizes from $1024^2$ to $16,384^2$. It is not possible to run the largest data size on the GTX 480 because of space constraints.

We report the median runtime for 10 executions for each kernel measured using the device high resolution timers.

## 7. Results

This section investigates the results obtained by executing the automatically generated OpenCL kernels on three different GPUs. We are interested in the highest performance, the time required to find high-performance kernels, and if there exists a universally good kernel which can be used for all input sizes and GPUs.

### 7.1 Comparison with High-Performance Libraries

We exhaustively executed all generated OpenCL kernel on all platforms with different data sizes. Figure 9 shows a comparison of the fastest automatically generated kernel compared against manually implemented high-performance BLAS libraries. The left graph shows the performance achieved on Nvidia Fermi, the middle and right graphs show the performance on Nvidia Kepler and AMD Tahiti respectively.

On Fermi we can observe that we achieve 80–90% of the performance of the highly tuned MAGMA library [8] and 75–77% of the Nvidia provided cuBLAS library. On the more modern Kepler architecture we outperform MAGMA, implemented in CUDA, by 7–25%, even though the Nvidia OpenCL tool-chain currently does not support Kepler architecture specific optimizations. The extremely good results of cuBLAS compared to any other implementation are due to hand-optimized assembly level code which overcomes shortcomings of the CUDA compiler [9].

The results for the AMD Tahiti architecture show that we are able to find OpenCL kernels with faster execution time as the MAGMA and clBLAS libraries. The performance of clBLAS is slightly improved using a provided tuning script which chooses device-specific implementation parameters.

Overall, the results show, that we can automatically generate code of same or better quality as the state-of-the-art hand tuned open source BLAS library MAGMA. Furthermore, we clearly achieve higher performance as clBLAS and show similar performance to cuBLAS on the Nvidia Fermi GPU.

### 7.2 Finding Good Generated Kernels

Figure 10 shows the distribution of performance as well as the median and quartiles for all 46,000 generated kernels on the three test platforms for the smallest input size. All three graphs show a similar shape with poor performance for most kernels. The maximal performance is only reached by a few generated kernels. This highlights the difficulty of optimizing matrix multiplication kernels: only a few kernels find the right balance for applying the tiling and blocking optimizations, make good use of the local memory, and chose well suited implementation parameters.

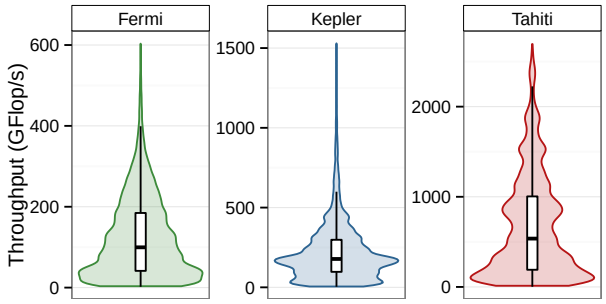



**Figure 10:** Distribution of performance for generated kernels.

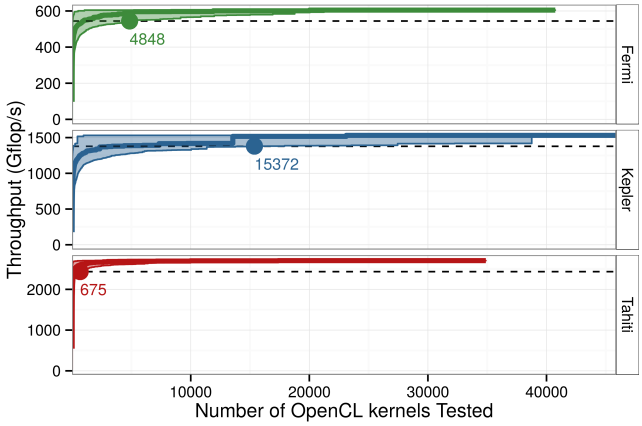



**Figure 11:** Performance evolution for randomized search of automatically generated OpenCL kernels. The dotted line represents 90% of the performance achieved by the best kernel. The dots mark how many kernels have to be tested to reach this performance with a confidence of 95%.

However, despite the overall shape of the optimization space we can find good performing kernels even when using a simple random search strategy. Figure 11 shows the evolution of performance when testing the generated OpenCL kernels in random order. The y-axis shows performance in GFlops and the x-axis shows the number of OpenCL kernels tested.

Performance improves quickly for all three platforms. On the AMD Tahiti architecture after testing only 675 kernels (2% of the search space), one can expect to find an OpenCL kernel reaching 90% of the maximum performance with a confidence of 95%. To reach 90% of the maximum performance with the same confidence on Fermi, 4,848 kernels (12%) have to be tested and 15,372 kernels (33%) on Kepler.

For a matrix of size $1024^2$, a single kernel execution takes on average 10ms (Tahiti), 26ms (Kepler), and 60ms (Fermi). Even the execution of 15,000 OpenCL kernels can, therefore, be performed in a reasonable time frame. Overall the exhaustive execution of all 46,000 OpenCL kernel took less than an hour on Tahiti and Kepler and about three hours on Fermi, including the overheads of data transfers and validation. The generation of all kernels with our prototype compiler implemented in Scala took about 2 hours and 40 minutes. The compilation of all generated OpenCL kernels to binaries took 20 minutes for Nvidia and 1 hour for AMD.

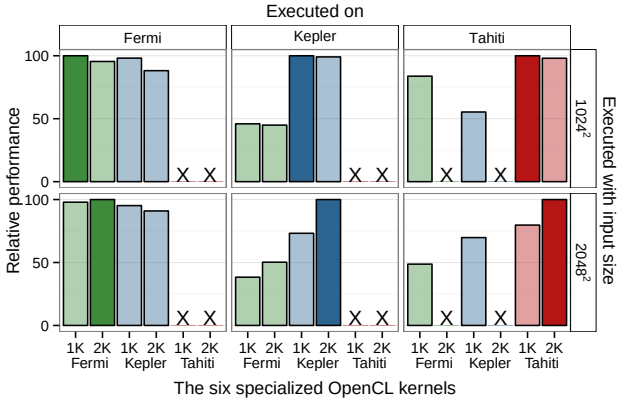



**Figure 12:** Performance of the best OpenCL kernels across platforms and data sizes.

### 7.3 Cross Platform Performance

Figure 12 shows the performance of six OpenCL kernels optimized and specialized for a particular configuration of device and input size executed across all devices and two input sizes. For example, the graph on the top in the middle shows the performance of the six optimized kernels executed on the Kepler GPU with an input size of $1024^2$. We can see that the kernels optimized for the Fermi GPU (the first two bars) only achieve 45% of the optimal performance. The kernel optimized for the AMD GPU (the last two bars) do not even run on the Kepler GPU, due to resource restrictions (marked with a cross in the figure).

The performance across input sizes on the same GPU is usually quite good indicated by a pair of similar graphs: one on the top for the smaller input size and the other one below for the larger input size. The performance across GPUs for a fixed input size varies more widely as one can see by comparing the graphs horizontally.

Overall, this shows that none of the kernels selected as the best on each GPU is portable offering good performance on the other GPUs as well.

### 7.4 Summary

The results show that our methodology generates high-performance OpenCL kernels for matrix multiplication with performance on par or faster than state-of-the-art BLAS library implementations. The analysis of our data shows, that is it sufficient to execute a small portion of the search space to find an high-performance OpenCL kernel. Finally, we show that no single portable kernel performing well on all GPUs and input sizes exists. This shows that traditional approaches, writing specialized kernels manually are non-portable by design.

## 8. Related Work

#### Matrix Multiplication on GPUs

MAGMA [8] develops general templates for different versions of matrix multiplication with an auto-tuning framework targeting NVIDIA GPUs. The clMAGMA [5] extension targets AMD GPUs with OpenCL. Other work [12] has used similar techniques with pre-written kernels and auto-tuning techniques.

Other researchers [9] have explored the limitations of GPU programming languages when using matrix multiplication as a case study. They conclude that GPU compilers are not able to perform well with register allocation or instruction reordering, leading to decreased levels of performance.

All these projects rely on manually written code which has to be tuned with huge human effort. Our methodology completely automates the optimization process and generates high-performance code on par with the best hand-written OpenCL code.

**Polyhedral GPU Compilation**

C-to-CUDA [4] and PPCG [20] are both polyhedral compilers. They create a polyhedral model of the parallel nested loops in the source code and perform advanced loop optimizations such as tiling and blocking to static loop nests with affine loop bounds and subscripts.

Pencil [3] is an intermediate language defined as a restricted subset of C99. It is intended as an implementation language for libraries and a compilation target by DSLs. It also relies on the use of the polyhedral model to optimize code and is combined with an auto-tuning framework.

Our approach is different since we do not rely on heavy static analysis but instead exploit high-level semantic information of our functional patterns such as map and reduce. In addition, in our approach compiler optimizations are simply implemented as provably correct rewrite rules, greatly simplifying the process of writing compiler optimizations.

**High-Level GPU Programming Approaches**

Many high-level approaches for GPU programming are inspired by functional programming.

SkelCL [17] is a high-level pattern-based library implemented in C++. Data parallel patterns are implemented as fixed OpenCL kernels lacking portability. Accelerate [13] is a domain specific language embedded in Hasekll for GPU programming. The implementation relies on templates of manually written CUDA kernels. Bones [15] is a pattern based GPU compiler automatically detecting algorithm species and mapping them to patterns. The pattern implementations are pre-written and not portable.

Delite [19] is a compiler framework for creating DSLs. Recently, they explored different strategies for mapping data-parallel computations onto GPU hardware [10]. Petabricks [2] allows the user to provide several algorithmic choices or implementations of the same algorithm. Petabricks was recently extended to generate OpenCL code [16].

These high-level projects rely on manually optimized GPU code or hard-coded device-specific compiler optimizations making the generated code not performance portable.

## 9. Conclusion

This paper has presented a novel technique to automatically generate performance portable GPU code. Programs are expressed in a high-level style using a small functional intermediate language. A system of rewrite rules encodes algorithmic and low-level transformations and is used to generate different implementations of the same program. By applying the rewrite rules automatically, the system generates thousands of semantically equivalent OpenCL kernels.

Using matrix multiplication as a case study, we have shown how we are able to generate 46,000 differently optimized OpenCL kernels. Out of these, the best generated kernels provide performance on par or even better than state-of-the-art high-performance OpenCL library implementations on Nvidia and AMD GPUs. By investigating the performance among kernels, we have shown that sampling a small fraction of the generated kernels is sufficient to achieve high performance. Furthermore, among all generated OpenCL kernels there was not a single portable kernel providing good performance across all investigated GPUs.

## Acknowledgments

This work was partially funded by Google and Oracle Labs.

## References

[1] AMD. APP OpenCL programming guide.

[2] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. PLDI. ACM, 2009.

[3] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, J. Absar, S. van Haastregt, A. Kravets, et al. Pencil: A platform-neutral compute intermediate language for accelerator programming. PACT. ACM, 2015.

[4] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. CC/ETAPS. Springer-Verlag, 2010.

[5] C. Cao, J. Dongarra, P. Du, M. Gates, P. Luszczek, and S. Tomov. clMAGMA: High performance dense linear algebra with OpenCL. IWOCL. ACM, 2014.

[6] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). PLDI. ACM, 2012.

[7] R. Karrenberg and S. Hack. Whole-function vectorization. CGO. IEEE, 2011.

[8] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning GEMM kernels for the fermi GPU. *IEEE TPDS*, 23(11), 2012.

[9] J. Lai and A. Seznec. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. CGO. IEEE, 2013.

[10] H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun. Locality-aware mapping of nested parallel patterns on GPUs. MICRO. IEEE, 2014.

[11] R. Leißa, M. Köster, and S. Hack. A graph-based higher-order intermediate representation. CGO. IEEE, 2015.

[12] K. Matsumoto, N. Nakasato, and S. G. Sedukhin. Performance tuning of matrix multiplication in OpenCL on different GPUs and CPUs. SCC. IEEE, 2012.

[13] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. ICFP. ACM, 2013.

[14] A. C. McKellar and E. G. Coffman, Jr. Organizing matrices and matrix operations for paged memory systems. *Commun. ACM*, 12(3), 1969.

[15] C. Nugteren and H. Corporaal. Bones: An automatic skeleton-based C-to-CUDA compiler for GPUs. *ACM TACO*, 11(4), 2014.

[16] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. ASPLOS. ACM, 2013.

[17] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A portable skeleton library for high-level GPU programming. IPDPSW. IEEE, 2011.

[18] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. ICFP. ACM, 2015.

[19] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM TECS*, 13(4s), 2014.

[20] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM TACO*, 9(4), 2013.