Matrix Multiplication Beyond Auto-Tuning: Rewrite-based GPU Code Generation

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

ABSTRACT

Graphics Processing Units (GPUs) are used as general purpose parallel accelerators in a wide range of applications. They are found in most computing systems, and mobile devices are no exception. The recent availability of programming APIs such as OpenCL for mobile GPUs promises to open up new types of applications on these devices.

However, producing high performance GPU code is extremely difficult. Subtle differences in device characteristics can lead to large performance variations when different optimizations are applied. As we will see, this is especially true for a mobile GPU such as the ARM Mali GPU which has a very different architecture than desktop-class GPUs. Code optimized and tuned for one type of GPUs is unlikely to achieve the performance potential on another type of GPUs.

Auto-tuners have traditionally been an answer to this performance portability challenge. For instance, they have been successful on CPUs for matrix operations, which are used as building blocks in many high-performance applications. However, they are much harder to design for different classes of GPUs, given the wide variety of hardware characteristics.

In this paper, we take a different perspective and show how performance portability for matrix multiplication is achieved using a compiler approach. This approach is based on a recently developed generic technique that combines a high-level programming model with a system of rewrite rules. Programs are automatically rewritten in successive steps, where optimizations decision are made.This approach is truly performance portable, resulting in high-performance code for very different types of architectures such as desktop and mobile GPUs. In particular, we achieve a speedup of 1.7x over a state-of-the-art auto-tuner on the ARM Mali GPU.

1. INTRODUCTION

Graphics Processing Units (GPUs) have emerged as powerful general-purpose parallel accelerators. They have revolutionized the high-performance computing landscape and are about to bring big changes to mobile devices. Programming APIs such as OpenCL or RenderScript are now supported on most mobile GPUs and new types of mobile applications are emerging, such as real-time 3D scene reconstruction [17].

However, producing high-performance GPU code is notoriously hard. Low-level hardware features are directly exposed to programmers, requiring expert knowledge to achieve high performance. In addition, each type of devices comes with its own performance characteristics, requiring different optimizations. This problem is further exacerbated with mobile GPUs since optimizations benefitial for desktop GPUs (*e.g.*, AMD, Nvidia GPUs) can negatively impact performance on mobile GPUs, as we will see later in this paper.

Auto-tuners have been proposed to address performance portability issues on GPUs. They are generally based on a specialized parametric implementation of a computational kernel, such as matrix multiplication, and the tuning process explores the performance space on the targeted hardware. However, auto-tuners have two major drawbacks. First, writing the parametric implementation for a given kernel requires non-negligible effort from the programmer. Secondly, and more importantly, the implementation is limited by a finite set of parameters which might not be good at expressing complex composition of optimizations. As we will see, this can result in far from optimal performance when the parametric implementation is run on a device it was not originally designed for. In other words, auto-tuning alone is not sufficient to solve the performance portability challenge.

We argue that achieving true performance portability requires a more generic mechanism that expresses combinations of optimizations *beyond* a fixed parametric space. We advocate the use of a recently developed new high performance code generation technique based on rewrite rules [23]. Programs are expressed in a high-level functional programming model which shields the programmer from hardware peculiarities.The compiler is then free to automatically explore the optimization space using a system of rewrite rules. These rules encode algorithmic transformations as well as hardware-specific low-level optimizations. Recent work [22] has shown that this generic compiler approach leads to high performance for desktop-class GPUs from AMD and Nvidia.

In this paper, we demonstrate that this compiler-based technique is able to succeed where auto-tuners fail to deliver, using matrix multiplication as a use-case. Matrix multiplication is a well studied and useful primitive found at the heart of many numerical codes and algorithms in areas such as machine-learning. In addition, there exist high-performance reference implementations and specialized auto-tuners, which allow for meaningful comparison.

CASES '16, October 01-07 2016, Pittsburgh, PA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4482-1/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2968455.2968521

Using the ARM Mali GPU as an example, we show that an auto-tuner designed primarily for desktop-class GPUs is unable to achieve the full performance potential, resulting in a 40% performance loss. In contrast, our compiler-based approach delivers performance on par with the best hand-tuned version on each of the three platforms tested. This is possible due to the generic nature of the rewrite-based code generation technique, which allows us to encode generic optimizations that are combined during the exploration process. This includes vectorization and the use of built-in functions, which are highly beneficial for the Mali GPU.

To summarize, this paper makes the following contributions:

- We demonstrate the limitations of auto-tuning when applied on a different class of GPUs;
- We present how generic optimizations beneficial for the Mali GPU are expressed in a rewrite-based generator;
- Our experimental results show that a rewrite-based approach is performance portable and even outperforms hand-tuned code on Mali.

The rest of the paper is organized as follows. The next section shows that performance is far from portable between different classes of GPUs. Section 3 presents characteristics of the Mali GPU. Section 4 introduces the high-level functional language and the rewrite-based code generator we adopted. Section 5 discusses optimizations for matrix multiplication, how they are represented functionally and how they are encoded as rewrite rules. Sections 6 and 7 present our experimental setup and show results on how we automatically achieve high performance from a portable, high-level representation of matrix multiplication. Finally, sections 8–10 discuss our work, related work and conclude the paper.

2. MOTIVATION

Matrix multiplication is probably one of the most studied kernels in the high-performance community. Automatic tuning techniques have been applied quite successfully to this benchmark for over 20 years starting with PHiPAC [4] and ATLAS [25]. However, auto-tuners rely on a parametric implementation (or a parametric code generator) that is highly specialized to the target machine. This approach is well-suited in cases where little variation exists between different processing units but falls short when the target processing units exhibit significant variations. We illustrate this problem using the CLBlast library auto-tuned using CLTune, a state-of-the-art auto-tuner which has been shown [18] to achieve competitive performance on several GPUs.

Figure 1 shows the performance achieved by CLBlast on three different platforms; two desktop-class GPUs (Nvidia and AMD) and one mobile GPU (ARM Mali). For each platform, we compare the performance of the auto-tuner with the best open source reference implementation available: MAGMA [5] on Nvidia, clBLAS on AMD and code written and optimized by ARM's engineers [8] on the Mali GPU. The auto-tuner is able to achieve significant performance on both desktop GPUs, clearly beating the hand-written MAGMA and slightly outperforming AMD's clBLAS.

However, the auto-tuner is unable to achieve the full performance potential on the mobile GPU resulting in a 40% performance loss. This shortfall is explained by the fact that CLBlast has been primarily designed for desktop-class GPUs and includes optimizations that are beneficial on these machines but detrimental on the Mali GPU. While it is conceptually not difficult to realise what needs to be done to reach a higher-level of performance for some specific machine, it is extremely hard to write a parametric kernel which exposes these choices as a finite set of parameters. Especially given that a library enabled for auto-tuning, such as CLBlast, is already quite complex with more than 1500 lines of parametric OpenCL code just for matrix multiplication.

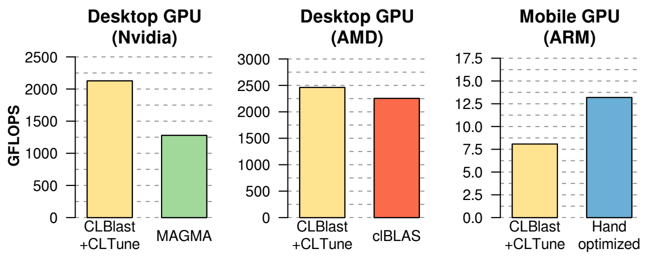

Figure 1: Performance comparison between auto-tuned (left bar) and hand-optimized (right bar) code. Higher is better.

What is needed is an approach that easily combines optimizations and produce a search space that includes the best performing implementations for different types of hardware. In this paper we propose to use a generic rewrite-based approach [23], which is not specific to matrix multiplication, and we show that it succeeds where the auto-tuner fails. This approach is also simpler to use since the compiler input is a high-level functional program. For instance, matrix multiplication is expressed in just five lines of code.

3. MALI GPU CHARACTERISTICS

ARM Mali-T628 GPU.

The Mali-T628 GPU is a mobile GPU implementing ARMs second generation Midgard micro-architecture. Each core has two arithmetic pipelines, each of which processes 128-bits of data at a time using SIMD operations. A single core can simultaneously manage up to 256 threads in hardware, depending on the amount of registers required by each thread. This large number of threads is used to hide memory latencies, as stalled threads waiting for memory can be overtaken by other threads.

Mali GPU Performance Characteristics.

Due to its hardware design, certain optimizations are crucial for achieving high performance on the Mali GPU. Optimizations techniques are discussed in the ARM documentation [2] as well as in [7] and [8]. These techniques are not aligned and quite often contradictory with advice given by AMD or Nvidia for their GPUs. This leads to a large performance portability gap when executing kernels optimized for a desktop GPU on the Mali GPU, or the other way around, as we will see in the evaluation (section 7).

Vectorization is one of the most important optimization given the SIMD architecture of the Mali GPU. OpenCL supports vectorization through the use of vector data types (*e.g.*, `float4`).Arithmetic operations performed on values of vector data types are performed by the hardware SIMD units. Special load operations exist in OpenCL for loading vector values from arrays of scalar values. Performing vectorized memory operations reduces the number of instructions issued and helps to better utilize the memory bandwidth.

While vectorization is a crucial optimization for the Mali GPU it is not recommended to be applied on Nvidia GPUs, as they do not have hardware vector units. Therefore, OpenCL code optimized for Nvidia GPUs will most likely make no use of vector data types and perform poorly on Mali GPUs.

Register pressure is an extremely important topic on the Mali GPU, as the number of registers influences the amount of threads managed by the hardware. Therefore, reducing the number of registers used increases the amount of active threads which helps to hide memory latencies and keeps the cores busy. If a kernel uses more than 4 128-bit registers, the number of threads drops from 256 to 128. If the kernel uses more than 8 registers, the amount of threads halves again.

While register pressure is also important on desktop GPUs, there are more registers available and the degree of thread level parallelism degrades more gracefully than on Mali.

There exist **optimizations for AMD and Nvidia GPUs** which are not beneficial on the Mali GPU. The local memory which is crucial for good performance on desktop GPUs is mapped to the same physical memory as the global memory on Mali. Its usage has, therefore, no performance benefits.

Memory accesses to the global memory are coalesced on AMD and Nvidia if all work items in the same execution batch access consecutive memory locations. Optimizing code for coalesced accesses is hugely beneficial on these architectures, where on Mali this might increase cache misses and memory accesses should be vectorized instead.

4. REWRITE-BASED CODE GENERATION

Traditionally, optimization decisions are encoded explicitly, and almost always manually, using a low-level programming model like OpenCL. This approach leads to code which is inherently not performance portable, because optimization choices for one architecture are often disadvantageous on other architectures, as hinted in the motivation.

To address the performance portability challenge we advocate the use of a high-level programming model coupled with automatic code generation. To this end we adopt a recently developed novel code generation technique based on rewrite rules [23]. It starts from a high-level functional program where implementation and optimization decisions are not explicitly specified. This empowers the compiler to automatically explore the implementation and optimization space and generate high performance OpenCL code. This has two main advantages: first, programming is simplified, as time consuming low-level programming is avoided; secondly, performance portability is achieved as code is optimized and specialized automatically by the compiler.

The following sections discuss how programs are expressed in a high-level functional language for data parallelism, how rewrite rules are used to transform high-level programs into functionally equivalent low-level programs and how OpenCL code is generated from them.

4.1 A High-Level Functional Language for Data Parallelism

We express programs in a functional high-level language specifically designed for data parallel applications. A set of primitives, shown in Table 1, is used to express the applications. The primitives do not determine *how* the computation is performed but only specify *what* has to be computed. This gives the compiler the freedom to choose different implementations for different GPUs enabling performance portability.

High-level data-parallel primitives	
map(f, xs)	Apply *f* to every element of array *xs*.
reduce(z, ⊕, xs)	Perform a reduction of the array *xs* using the binary operator ⊕ and its identity *z*.
zip(xs, ys)	Combine the arrays *xs* and *ys* pairwise.
split(n, xs)	Split the array *xs* into chunks of size *n*.
join(xs)	Opposite of split: merge 2D array *xs*.
Low-level OpenCL specific primitives	
mapGlb(f, xs)	Parallel *map* using global work items.
mapWrg(f, xs)	Parallel *map* using work groups.
mapLcl(f, xs)	Parallel *map* using local work items.
mapSeq(f, xs)	Sequential *map*.
reduceSeq(z,⊕,xs)	Sequential *reduction*.
asVector(n, xs)	Vectorize array *xs* with a width of *n*.
asScalar(xs)	Scalarize a vectorized array *xs*.
vectorize(f)	Vectorize the function *f*.
toGlobal(f)	Stores the result of *f* in global memory.
toLocal(f)	Stores the result of *f* in local memory.
toPrivate(f)	Stores the result of *f* in private memory.

Table 1: High-level and Low-level primitives.

```
1  λ (A : [[float]_M]_K, B : [[float]_K]_N) ↦
2    A >> map(λ rowOfA ↦
3     B >> transpose >> map(λ colOfB ↦
4       zip(rowOfA, colOfB) >>
5       map(mult) >> reduce(0.0f,add) ) )
```

Listing 1: Matrix multiplication expressed functionally. This is the input from which our compiler generates efficient OpenCL code targeted for the Mali GPU.

The primitives are customized with application specific functions and can be freely composed as they are functions themselves. For example, to sum up the absolute values of an array we write: `reduce(0, plus, map(abs, xs))`. In this paper we use a notation similar to dataflow programming where, expressions are read from left to right. This example is written as: `xs >> map(abs) >> reduce(0, plus)`. We will call programs written in this language *(functional) expressions* to distinguish them from the generated OpenCL code.

Listing 1 shows the functional matrix multiplication expression. The first `map` in line 2 performs a computation for every row of A. In line 3 we map over the transposed matrix B to obtain a single column of B. In lines 4 and 5 the dot product for every pair of row and column is computed.

4.2 Lowering to OpenCL using Rewrite Rules

The key idea of the rewrite-based compilation technique [23] is to expose implementation and optimization choices in the compiler in the form of rewrite rules. A rewrite rule is a syntactic program transformation which is proven to preserve the semantic of the program. These rewrites can, therefore, be safely and automatically applied to explore different implementation choices. This is in sharp contrast to traditional optimizing compilers where complicated static analysis is required to prove that transformations are valid.

The rewrite rules are coupled with a set of low-level primitives which resemble the OpenCL programming model (Table 1). The rewrite rules bridge the gap between the high-level algorithmic primitives and the low-level OpenCL specific primitives. Implementation choices are made by choosing between different rewrite rules. For example, we might decide to perform a *map* operation sequentially by rewriting it into the *mapSeq* primitive. Alternatively the high-level *map* can be rewritten into a different low-level *map* to exploit parallelism using global threads (*mapGlb*) or local threads (*mapLcl*) grouped in work-groups (*mapWrg*). Once a high-level program is rewritten into a low-level program, all decisions about *how* the computation should be executed in OpenCL have been made and explicitly encoded in the program. We will explain the rewrite process and how it is guided to achieve high performance in section 5.4.

4.3 OpenCL Code Generation

The last stage consists of generating OpenCL code from a low-level program. No implementation or optimization decisions are made at this point, as these have already been made using rewrite rules. Every low-level primitive from Table 1 directly corresponds to a piece of OpenCL code. Therefore, the code generation process is straightforward and consists of traversing the low-level program and emitting the corresponding OpenCL code fragment. The code generation process relies on information about the length of arrays which are stored in their types. This information is used for the allocation of memory as well as for emitting indices when accessing data, as we will see in more detail in the next section.

5. OPTIMIZING MATRIX MULTIPLICATION FOR MALI

This section discusses how to optimize matrix multiplication for Mali. It first investigates a hand-optimized OpenCL kernel and how it is expressible in the functional language. Then it shows that the functional representation is suitable for expressing optimizations structurally as rewrite rules.

5.1 Manually Optimized OpenCL Kernel

ARM recently published a paper where they discuss optimization techniques for their Mali GPU [8]. One of the applications investigated is the general matrix multiplication for which multiple optimized OpenCL kernels are presented. Listing 2 shows the best performing version developed by ARM's engineers [8]. To keep the discussion simple we show a slightly simpler version, which concentrates on the actual matrix multiplication and omits the scalar values α and β used in the BLAS formulation of GEMM.

OpenCL kernel analysis.

The OpenCL kernel shown in Listing 2 applies **vectorization** and **blocking** as its two main optimizations. The for loop in line 8 iterates over blocks (or *tiles*) comprising of 2 `float4` elements from matrix `A` and `B`. These elements are loaded into private variables in lines 9–12. The dot products of all four combinations of `float4` elements from matrix `A` and `B` are computed using the OpenCL built-in `dot` function (lines 13 and 14) resulting in four separate intermediate results. These are combined into a single `float4` value (line 13) which is added to the accumulation variable `ab` (declared in line 7).

The vectorization of the addition operation in line 13 is

```
 1  kernel void mm(global float4* const A,
 2                 global float4* const B,
 3                 global float2* C, uint n) {
 4    uint i = get_global_id(0);
 5    uint j = get_global_id(1);
 6    uint nv4 = n >> 2;
 7    float4 ab = (float4)(0.0f);
 8    for (uint k = 0; k < nv4; ++k) {
 9      float4 a0 = A[ 2*i   *nv4+k];
10      float4 a1 = A[(2*i+1)*nv4+k];
11      float4 b0 = B[ 2*j   *nv4+k];
12      float4 b1 = B[(2*j+1)*nv4+k];
13      ab += (float4)(dot(a0, b0), dot(a0, b1),
14                     dot(a1, b0), dot(a1, b1)); }
15    uint ix = 2*i*(n>>1) + j;
16    C[ix]          = ab.s01;
17    C[ix + (n>>1)] = ab.s23; }
```

Listing 2: Optimized OpenCL matrix multiplication kernel. This listing shows the `blockedNT` version from [8].

independent of the use of vector data types for the elements of matrix `A` and `B`. Instead, the blocking of 2 values from `A` and 2 values from `B` leads to 4 intermediate results which are added to the accumulation variable using a vector addition. After the loop, the results are written to global memory in two instructions (lines 16 and 17) using a vector width of 2.

Optimized matrix multiplication expressed functionally.

Listing 3 shows a functional expression resembling the optimized implementation shown in Listing 2. Starting from the top, the blocking optimization is expressed by splitting matrices `A` (line 2) and `B` (line 3) by a factor of 2. This groups 2 rows of `A` and 2 columns of `B` together. The `mapGlb` primitives used in lines 2 and 3 express the mapping of parallelism to global threads in OpenCL: every global thread processes a pair of *2 rows of* `A` and *2 columns of* `B`.

To complete the **blocking** of `A`, we first transpose a block of 2 rows of `A` (line 4), split each row into chunks of 4 elements and then transpose back to obtain tiles with 2×4 `float` values. The same process is applied to `B` in lines 6 and 7. The `zip` (line 4) combines the tiles of `A` and `B` together. These pairs of tiles are then processed by the `reduceSeq` in line 8 which corresponds to the for loop in the OpenCL kernel.

When processing a single pair of a *tile of* `A` and a *tile of* `B` inside of the reduction, the pairs are copied into the private memory in lines 10–13. The `asVector(4)` primitive (used in lines 11 and 13) **vectorizes the data** by turning 4 individual `float` values of a tile into a single `float4` value. This section corresponds to the lines 9–12 in Listing 2 where values from matrices `A` and `B` are loaded into private variables.

For each combination of a *row of a tile of* `A` and a *column of a tile of* `B`, each represented by a `float4` value, we perform the dot product computation in lines 17–19. The dot product is expressed as a combination of the `zip`, `mapSeq` and `reduceSeq` primitives. The `zip` (line 17) combines the two `float4` values from the tiles of `A` and `B`, before the `mapSeq(mult4)` (line 18) performs the **vectorized multiplication** of the two values. To finish the dot product computation, `reduceSeq(0.0f, add)` (line 19) adds up the multiplied values after they have been turned back into scalar values using the `asScalar` primitive (line 18). This section corresponds to the four occurrences of the `dot` function in lines 13 and 14 in Listing 2.

```
 1 λ (A, B) ↦
 2  A >> split(2) >> mapGlb0(λ 2RowsOfA ↦
 3   B >> split(2) >> mapGlb1(λ 2ColsOfB ↦
 4    zip( 2RowsOfA >> transpose >>
 5           split(4) >> transpose,
 6         2ColsOfB >> transpose >>
 7           split(4) >> transpose ) >>
 8    reduceSeq(init = (float4)0.0f,
 9     λ (acc, (tileOfA, tileOfB)) ↦
10      ⟨tileOfA >> mapSeq(
11        asVector(4) >> toPrivate(id4)),
12       tileOfB >> mapSeq(
13        asVector(4) >> toPrivate(id4))⟩ >>
14      (λ (tileOfAp, tileOfBp) ↦
15       tileOfAp >> mapSeq(λ rowOfTileOfA ↦
16        tileOfBp >> mapSeq(λ colOfTileOfB ↦
17         zip(rowOfTileOfA, colOfTileOfB) >>
18         mapSeq(mult4) >> asScalar >>
19         reduceSeq(0.0f, add) ) ) ) >>
20      (λ 2x2DotProducts ↦
21       2x2DotProducts >> join >> asVector(4) >>
22       mapSeq(add4(acc)) ) ) >>
23    asScalar >> asVector(2) >>
24    toGlobal(mapSeq(id2)) ) )
```

Listing 3: Low-level functional expression resembling the OpenCL kernel presented in Listing 2.

To complete the reduction over multiple tiles, we must add the computed intermediate result to an accumulation variable. To achieve this, we flatten the computed 2×2 *dot products* into a one dimensional array using the `join` primitive (line 21). The resulting array of 4 `float` values is vectorized, using the `asVector(4)` primitve and added to the accumulation variable `acc` in line 22. This section corresponds to the **vectorized += operation** in Listing 2 (line 13).

Finally, to write the computed results back to the global memory the vector width is changed using `asScalar` and `asVector(2)` before the actual copy operation in line 24. This last section corresponds to the lines 16 and 17 from Listing 2.

This example should give some intuition on how optimized programs are expressed functionally. This representation enables the automatic transformation of the high-level program in Listing 1 into low-level expressions such as Listing 3 using rewrite rules, as the rest of the paper shows.

5.2 Optimizations Expressed Structurally

This section investigates individual optimizations, shows how they are expressed functionally and presents the corresponding generated OpenCL code.

Mapping of parallelism.

In OpenCL, programmers have different choices on how to map the computation to the hardware, which directly affects performance. The programmer might decide to group threads (*work items*) into work groups and use their associated *local ids* together with their *work group ids* to distribute the work. Sometimes it is possible to use the *global ids* of work items independently of their work group ids.

In our approach, using the different layers of this hierarchy is expressed by using different low-level variations of the *map* pattern. All variations share the same high level semantics: applying a function to each element of the input array to produce the output array. The low-level variations differ in their

```
A >> mapGlb0(λ rowOfA ↦
  B >> mapGlb1(λ colOfB ↦ ... ) )
```

(a) Functional expression using the `mapGlb` primitive.

```
kernel void KERNEL(...) {
 for (int g_id_0 = get_global_id(0); g_id_0<N;
      g_id_0 += get_global_size(0))
  for (int g_id_1 = get_global_id(1); g_id_1<N;
       g_id_1 += get_global_size(1))
    ... }
```

(b) Generated OpenCL code for an arbitrary global size.

```
kernel void KERNEL(...) {
  int g_id_0 = get_global_id(0);
  int g_id_1 = get_global_id(1);
  ... }
```

(c) Generated OpenCL code for fixed global size.

Figure 2: Exploiting parallelism using global work items.

OpenCL implementations, where the computation might be performed sequentially (*mapSeq*), or in parallel, distributing the workload across work groups (*mapWrg*), local work items (*mapLcl*) or global work items (*mapGlb*).

Figure 2 shows one possible mapping of parallelism for matrix multiplication. In Figure 2a, the $\texttt{mapGlb}_0$ primitive is used to perform a computation for every row of A. Nested inside is the $\texttt{mapGlb}_1$ primitive which maps over the columns of B. As we use the `mapGlb` primitives we indicate, that a work item with the global ids `g_id_0` and `g_id_1` will process a combination of a row of A and a column of B.

Figure 2b shows the corresponding OpenCL code generated for this expression. The two for loops correspond to the map primitives. In the generic case it is unclear how many global work items will be launched at execution time, therefore, `for` loops are emitted and a single work item might process multiple data elements. For matrix multiplication (and many other applications) it is common to specialize the OpenCL kernel so that it only works if a matching global size is selected at execution time. To support this, we use array length information to statically prove that each work item executes the loop exactly once and avoid generating the loop altogether. The resulting OpenCL code is shown in Figure 2c.

Vectorized memory operations.

Vectorizing load and store instructions helps to better utilize the memory bandwidth by issuing larger memory transfers with a single instruction. OpenCL provides specific `vload` and `vstore` instructions for loading or storing vector values from arrays of scalar values.

We decompose vectorized memory operations into two parts, as shown in Figure 3a: first, interpreting the initially scalar array as a vectorized array using `asVector`; secondly, copy the data by applying the vectorized identity function `id4` to every element of the vectorized array. In the example `toPrivate` indicates a copy into the private memory. We keep track of the length of arrays in their types. Let us assume that A in the example is an array of N `float` values. Therefore, we write its type as: $[\texttt{float}]_N$. After applying `asVector(4)` to it we obtain an array with type: $[\texttt{float4}]_{N/4}$. We use this length information when generating indices in OpenCL.

The generated OpenCL code is shown in Figure 3b. The `id4` function is declared in the first line and models a copy

```
... A >> asVector(4)
      >> toPrivate(mapSeq(id4)) >> ...
```

(a) Functional expression using the `asVector` primitive.

```
1  float4 id4(float4 x) { return x; }
2  kernel void KERNEL(const global float* A) {
3    ...
4    float4 elemsOfA_0 = id4(vload4(index_0, A));
5    float4 elemsOfA_1 = id4(vload4(index_1, A));
6    ... }
```

(b) Generated OpenCL code using `vload` instructions.

Figure 3: Vectorized memory operations.

```
... >> zip(elemsOfA, elemsOfB)
    >> mapSeq(vectorize(4, mult))
    >> asScalar >> reduceSeq(0.0f, add) >> ...
```

(a) Functional expression performing a vectorized dot product.

```
1   float4 mult4(float4 l,float4 r){ return l*r;}
2   float  add(float l,float r){ return l+r;}
3   kernel void KERNEL(const global float* A,
4                      const global float* B) {
5    ...
6    float4 tmp = mult4(elemsOfA, elemsOfB);
7    float acc = 0.0f;
8    acc= add(acc,tmp.s0); acc= add(acc,tmp.s1);
9    acc= add(acc,tmp.s2); acc= add(acc,tmp.s3);
10   ... }
```

(b) Generated OpenCL code using vector instructions.

Figure 4: Vectorized arithmetic operations.

operation in the functional expression. It will be inlined and, therefore, optimized away by the OpenCL compiler. After vectorizing the array its `float4` values are loaded using `vload4` instructions. As arrays in private memory are not stored in registers we unroll the array into private variables. We show the first two variables in lines 4 and 5. To unroll the array, its size has to be statically known, which is the case for arrays obtained through fixed size tiling. We use symbolic computations to compute indices like $index_0$ using the length information stored in the array's type.

Vectorized arithmetic operations.
Vectorizing arithmetic operations is one of the most important optimizations on Mali GPUs due to its SIMD architecture. We discuss the vectorization of the dot product computation as an example, which is used as a building block in matrix multiplication as seen in Listing 3 lines 17–19.

The dot product is represented functionally by combining two arrays using the `zip` primitive. It is followed by `map(mult)` which performs a pairwise multiplication before `reduce(0, add)` adds up all the intermediate results. Figure 4a shows a vectorized version of the dot product. The `vectorize(4, mult)` primitive is used to vectorize the multiplication with a vector width of 4. Currently we support the vectorization of simple functions but we intend to incorporate existing research on vectorizing more complex functions [9] in the future. After performing the vectorized pairwise multiplication, all values are added up to compute the scalar result by first interpreting the vectorized data as scalar, and then by performing a reduction using scalar addition.

The generated OpenCL code is shown in Figure 4b. The vectorized function `mult4` performs the multiplication operation on two `float4` values. The `add` function in line 2 is not vectorized and operates on scalar `float` values. This example OpenCL code assumes that only two `float4` values are combined and multiplied producing a temporary `tmp` in line 6. The following two lines reduce the vector by accessing its individual components to produce the final result.

5.3 Optimizations Expressed as Rewrite Rules

The optimizations discussed above are easily expressed as rewrite rules. Let us take the vectorization of the dot product (Figure 4) as an example. The following rewrite rule describes this transformation independently of the concrete operation performed by the function f or the concrete vector width n:

```
  zip(a, b) >> map(f)
⟹
  zip(asVector(n, a), asVector(n, b)) >>
    map(vectorize(n, f)) >> asScalar
```

It is easy to see that this rule is correct, since the result of both expressions is an array of scalar values computed by applying the function f to pairs of elements from `a` and `b`.

Similarly we define a rule for vectorizing a reduction:

```
  a >> reduce(z, ⊕)
⟹
  a >> asVector(n)
    >> reduce(asVector(n, z), vectorize(n, ⊕))
    >> toScalar >> reduce(z, ⊕)
```

The rewritten expression performs a reduction on the vectorized data using the vectorized operator $\oplus$ before the final result is computed by a scalar reduction of the components of the vectorized result of the first reduction. For this rewrite to be correct, we require the reduction operator $\oplus$ to be commutative, as we change the order in which elements are processed.

In Listing 2 the OpenCL build-in function *dot* is used to perform a dot product of two `float4` values. This function can be implemented more efficiently by the compiler, *e.g.*, by using specialized hardware instructions. As we will see, this is highly beneficial on Mali. We can easily define a rule to detect a sequence of patterns computing a dot product and rewrite it into a function call of the `dot` built-in function:

```
  zip(x, y) >> mapSeq(mult4) >> asScalar
            >> reduceSeq(z, add)
⟹
  dot(x, y) >> reduceSeq(z, add)
```

For this rule to fire x and y must be of type `float4`. The additional `reduceSeq` after applying the `dot` adds the computed result to the accumulation variable which is initialized with z. This shows how a very specialized optimization can be implemented as a simple generic rewrite rule.

5.4 Automatic Exploration

Having defined optimizations as rewrite rules, it is now possible to explore the space automatically by applying a combination of rules to the input program. However, the resulting space is extremely large, even potentially unbounded, which opens up a new research challenge. This is in stark contrast to classical auto-tuners which have a much smaller space to explore due to their parametric nature. However, this is also the main reason why auto-tuners sometimes fail to achieve high-performance as seen in the motivation; they are bound by the fixed set of parameter chosen by the implementer and cannot search beyond these. In contrast, our rewrite-based approach is able to combine the various optimizations expressed as rules in any way and can, therefore, explore a far larger amount of implementations unreachable with classic auto-tuning.

We present here a first, simple and heuristic-based pruning strategy to tackle the space complexity problem. Future research will investigate more advanced techniques to fully automate the pruning process, *e.g.*, using combinations of micro-benchmarking and machine learning.

For matrix multiplication, we start exploring from the high level expression shown in Listing 1. Rewrite rules are automatically applied until a low-level expression is produced such as Listing 3 from which OpenCL code is generated.

To **explore different algorithmic optimization choices**, we encode the optimizations discussed in section 5.3 plus 1D and 2D register blocking, and tiling presented by others [22]. Starting from the *high-level expression* in Listing 1, we apply these rewrite rules at all valid locations in an arbitrary order. This leads to a large number of expressions which we filter based on simple heuristics to limit the depth of nesting of maps and keep the distance between multiplication and addition in the dot product small. Expressions with deep nesting or with temporaries between the two crucial arithmetic operations are unlikely to deliver great performance.

To further rewrite into a *low-level expression* containing only OpenCL-specific primitives, we restrict ourself to a couple of fixed parallelism mapping. For instance, the two outermost *map*s are mapped to the global work items using *mapGlb* when the local memory is not used in the expression since using work-group is usually not beneficial otherwise. Using these heuristics, our system produces 31 differently optimized low-level expressions automatically derived from the single, high-level expression of matrix multiplication.

Once a low-level expression has been produced, the next step consists of **selecting kernel parameters** which is similar to classical auto-tuning techniques. Every low-level expression explicitly encodes which optimizations are applied but not which numerical parameters are picked for them. We automatically explore these parameters, for example the vector width or the size of a tile, for all 31 low-level expression, resulting in 677 specialized OpenCL kernels. Because we focus on Mali, we fixed the vector width to 4 to prune the space. We explored the tiles sizes in a reasonable range ensuring that the generated kernel does not require too much memory.

Once an OpenCL kernel has been generated, we still need to decide on the work group size **runtime parameter**,*i.e.*, the number of threads in a work-group. For matrix multiplication, work groups are two-dimensional and we have to select the number of threads in both dimensions. We conducted a search of the work-group size within the range allowed by OpenCL.This resulted in $11,628$ unique combinations of runtime parameters and optimizations on Mali.

5.5 Summary

In this section we have looked at optimizations for the Mali GPU, how they are implemented in OpenCL and in a generic rewrite-based code generator. These rewrites enable our compiler to automatically combine various optimizations and transform high-level programs into optimized functional low-level expressions from which OpenCL code is generated.

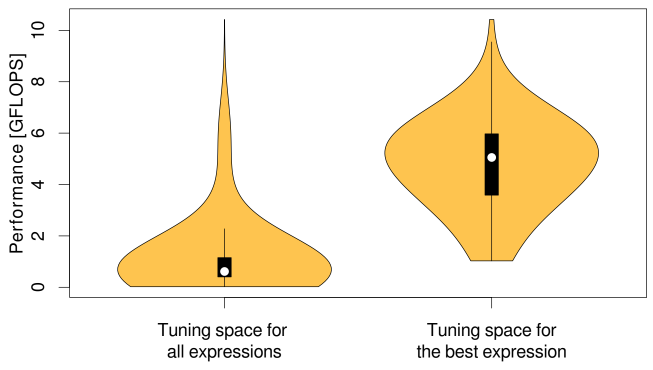

Figure 5: Performance distribution of matrix multiplication kernels generated from functional expressions. The white dot is the median. The black box delimits the 25%–75% quantiles.

6. EXPERIMENTAL SETUP

We evaluated our code generation technique using matrix multiplication with differently sized square and rectangular matrices ($512^2 * 512^2$, $1024^2 * 1024^2$, $2048 \times 512 * 512 \times 2048$, $512 \times 2048 * 2048 \times 512$) of single precision floating point values. We report the median performance in GFLOPS of at least five runs.

We use an ODROID XU3 board with a Samsung Exynos5422 system on a chip. The mobile Mali-T628 MP6 GPU is separated into two OpenCL devices. We used the first device with 4 cores and the Mali SDK 1.1 as OpenCL implementation. We disabled DVFS and locked the clock frequency at 600 MHz.

To investigate performance portability we also performed experiments on two desktop GPUs: a Nvidia GTX Titan Black using CUDA 6.0 and driver 331.79, and an AMD Radeon HD 7970 using AMD APP SDK 2.9.214.1 and driver 1526.3.

7. EVALUATION

This section evaluates our approach using matrix multiplication as a case study. We start by analyzing the performance distribution of the entire exploration space and compare it against the performance of the OpenCL kernels generated from the best expression in the space. We then analyze the performance impact of particular optimizations and compare performance against manually optimized implementations as well as the auto-tuned CLBlast library. Unless specified otherwise, we use $1024^2 * 1024^2$ as default input size.

7.1 Space Exploration

Following the strategy described in section 5.4, the generation of the 677 OpenCL kernels from 31 functional expressions took less than half an hour while performing the $11,628$ executions took about a day. Figure 5 shows the performance distribution of executions of the kernels generated. The violin plot represents a smooth histogram (turned by 90 degrees) of the performance measured in GFLOPS. The left-most violin corresponds to the entire space (*i.e.*, all the expressions + parameter tuning) while the right-most violin represent the distribution of tuning the parameter of the kernel resulting from the single best expression (Listing 4).

As can be seen on the left, most executions in the entire space result in very low performance with a median of less than 1 GFLOP. When focusing on the executions of kernels generated from the best expression (right-side), performance increases significantly with a median of ~5 GFLOPS.

```
 1 λ (A, B) ↦
 2  A >> split(n) >> mapGlb0(λ nRowsOfA ↦
 3   B >> split(m) >> mapGlb1(λ mColsOfB ↦
 4     zip( transpose(nRowsOfA) >> split(k),
 5          transpose(mColsOfB) >> split(k) ) >>
 6     reduceSeq(init = make2DArray(n,m, 0.0f),
 7      λ (accTile, (tileOfA, tileOfB)) ↦
 8       zip(accTile, transpose(tileOfA)) >>
 9       mapSeq(λ (accRow, rowOfTileOfA) ↦
10        zip(accRow, transpose(tileOfB)) >>
11        mapSeq(λ (acc, colOfTileOfB) ↦
12         dot(rowOfTileOfA >> asVector(k),
13             colOfTileOfB >> asVector(k)) >>
14         reduceSeq(acc, add)
15        ) >> join ) ) >>
16     toGlobal(mapSeq(mapSeq(mapSeq(id)))) >>
17     transpose() >> map(transpose) >> transpose
18   ) >> join >> transpose ) >> join
```

Listing 4: The best performing low-level expression automatically derived from the high-level expression in Listing 1 using rewrite rules.

The results show that picking the right combination of optimizations, explicitly encoded in the low-level expression after rewriting, is crucial for avoiding bad performing code. Nevertheless, it is still important to tune the kernel and runtime parameters for achieving high-performance.

Listing 4 shows the best performing expression found automatically for the 1024^2x1024^2 input size. By investigating the expression we can see that vectorization has been applied (lines 12–13), the *dot* built-in function is used (line 12), and tiling is performed with the `split` and `transpose` in lines 2–5. The vector width (k) and tile sizes ($n \times k$ and $m \times k$) are still parameters that are picked prior to OpenCL code generation. After conducting the parameter exploration, the values leading to the fastest kernel are $k = 4$ and $n = m = 2$.

The expression is similar to the one resembling the manually optimized OpenCL kernel (Listing 3). Nevertheless, there are a few notable differences. For example the tiles are not explicitly copied to private memory and, therefore, more loads to the global memory are issued. However, as we will see in the next subsection, this does not affect performance negatively as these accesses are probably cached. In fact, the generated OpenCL kernel is *faster* than the kernel which explicitly copies the data into private memory.

7.2 Performance Comparison Against Manually Optimized Kernel

Figure 6 shows a performance comparison of three matrix multiplication implementations. The first bar shows the performance of the generated OpenCL kernel from the expression resembling the manually optimized kernel (Listing 3) whose performance is shown as the last bar. The second bar shows the best OpenCL kernel generated by automatically deriving the expression shown in Listing 4 from the five-line long high-level expression of matrix multiplication (Listing 1). For the two automatically generated OpenCL kernels we indicate the performance benefit measured when including the rewrite rule for introducing the *dot* built-in function.

We can see that the performance of the OpenCL kernel generated via our automatic exploration even slightly outperforms the manually optimized kernel. It is important to note that this is achieved completely automatically by systematically applying rewrite rules starting from a high-level

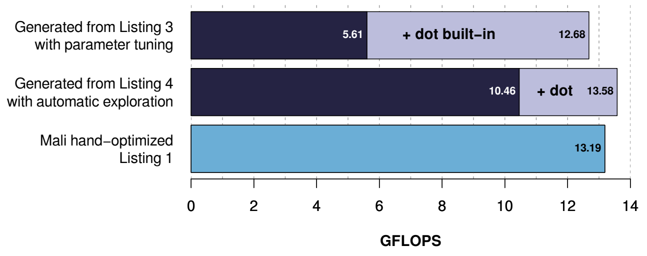

Figure 6: Performance of different matrix multiplication kernels. Our fully automated exploration technique produces a kernel that outperforms the manually optimized kernel.

representation of matrix multiplication. The rewrite rule that introduces the *dot* built-in turns out to be crucial, giving an extra 30% of performance as can be seen.

The kernel generated from the expression resembling the manually optimized implementation (the first bar) achieves 96% of the performance of the manually written kernel. Again, the usage of the *dot* built-in is crucial for achieving high performance for matrix multiplication on Mali.

7.3 Performance Portability and Performance Comparison Against Auto-tuning

To investigate portability of performance across different classes of GPUs we compare our approach against the CLBlast[1] library auto-tuned with the state-of-the-art CLTune[2] [18] on three GPUs from AMD, ARM, and Nvidia. As reference points, we use the clBLAS[3] library developed by AMD using OpenCL, as well as an implementation particularly tuned for each architecture: the hand tuned version shown in Listing 3 on Mali, cuBLAS on Nvidia, and clBLAS on AMD.

Figure 7 shows the performance comparison of all implementations on four different input sizes and shapes. The auto-tuned CLBlast library delivers high performance on the two desktop GPUs, achieving performance higher than clBLAS on the AMD GPU. On Nvidia, CLBlast achieves about 80% of the performance of cuBLAS for three inputs sizes. That is a very good number, as the proprietary cuBLAS relies on advanced assembly-level optimizations which cannot be implemented using CUDA or OpenCL [11]. However, on the mobile Mali GPU the auto-tuning approach is less successful, achieving only about 60% of the performance of the hand optimized implementation on three inputs and 25% slower than our own results on the other input.

This shows that performance portability is not achieved purely using auto-tuning. By investigating the tuned OpenCL kernel used by CLBlast, we could see that the built-in *dot* function or vectorized operations are not used which – as we have seen – is crucial for achieving high performance on Mali. On the desktop GPUs these optimizations are not required as there is no hardware support for vectorization. Furthermore, the overall structure of the kernel is similar to the one used for the desktop GPUs, clearly showing that CLBlast was developed for these GPUs and applied to Mali as an afterthought.

Our rewrite-based approach delivers high performance on the desktop GPUs *and* on the mobile GPU. Performance on the desktop GPUs is very close (Nvidia) or even slightly better (AMD) compared to CLBlast on all input sizes. Crucially, the

[1]d190bec from https://github.com/CNugteren/CLBlast
[2]ad94a3d from https://github.com/CNugteren/CLTune
[3]d16f7b3 from https://github.com/clMathLibraries/clBLAS

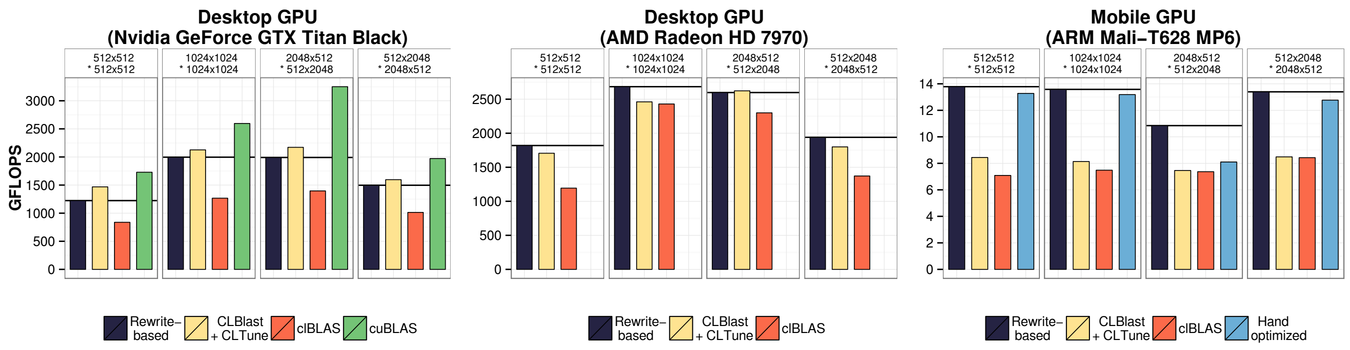

Figure 7: Performance of matrix multiplication on two desktop GPUs and one mobile GPU for different input sizes. The *rewrite-based* approach is the only one that achieves performance portability across desktop-class *and* mobile GPUs.

rewrite-based approach consistently achieves a large performance improvement on the Mali GPU compared to CLBlast (up to 1.7× better). It is able to outperform any other implementation on Mali, especially for the third input size where choosing a larger tile size increases the amount of work per thread which is beneficial for this type of matrix shape.

The key for achieving high performance is the support for architecture specific optimizations expressed as generic rewrite rules and the ability to generate structurally-different OpenCL kernels. In fact, when running the best OpenCL kernel generated for Mali on the Nvidia GPU we obtain only 4% of the performance compared to running the kernel optimized for this GPU (*i.e.*, 25x slower) as seen in Table 2. Conversely, running the kernel optimized for the desktop class AMD GPU on Mali results in only 11% of the performance achieved with the best kernel we generate for the embedded GPU (*i.e.*, 9x slowdown). The Nvidia kernel does not even run on Mali due to insufficient hardware resources.

On the desktop GPUs our approach generates kernels exploiting the hierarchical organization of threads, local memory, tiling, and the fused multiply-add instruction, whereas on the mobile GPU, a flat organization of threads, vectorization, and the *dot* built-in are crucial. These very different OpenCL kernels are derived from a single high-level expression of matrix multiplication using rewrites.

7.4 Summary

This section has shown that a rewrite-based approach achieves high performance on two desktop GPUs and the mobile Mali GPU starting from a single portable high-level expression.The comparison against the state-of-the-art auto-tuner, CLBlast, shows that tuning a fixed parameter space does not achieve performance portability across different classes of GPUs.

8. DISCUSSION

While this paper has focused on matrix multiplication, the proposed approach is in fact more generic. The high-level functional language introduced in section 4 has been deliberately designed to be more restrictive than general purpose languages to enable efficient parallel code generation. However, the language and the rewrite rules are fully extensible and can be used for expressing a larger class of data-parallel applications. For instance, we are currently working on extensions to support sparse linear algebra and stencil applications using the exact same methodology presented.

		Run on		
		Nvidia	AMD	Mali
Tuned for	Nvidia	100.0 %	27.5 %	N/A
	AMD	20.5 %	100.0 %	11.6 %
	Mali	4.2 %	14.4 %	100.0 %

Table 2: Performance portability of kernels ($1024^2 * 1024^2$)

9. RELATED WORK

Auto-tuning approaches.

There exist a large number of auto-tuning projects in the literature. We highlight two recent examples. OpenTuner [1] is a recent generic framework for creating domain-specific multi-objective auto-tuners. It supports a variety of search techniques, as well as user-defined ones providing domain specific knowledge. CLTune [18] is an auto-tuner for optimizing OpenCL kernels. Using CLTune requires the kernel to be written in a auto-tuning friendly style and might require providing alternative implementations to achieve good performance on a variety of devices. It supports a broad range of strategies to efficiently search the space of parameters.

Auto-tuning has been successfully applied to matrix multiplication. Previous work includes templates for different implementations with auto-tuning targeting Nvidia GPUs [10]. Other work [13] has auto-tuned pre-written kernels.

As we have demonstrated in this paper, our rewrite-based approach is more fundamental than classical parameter based auto-tuning, as the rewrite rules allow to drastically change the structure of generated OpenCL kernels achieving true performance portability across desktop and mobile GPUs.

Mali GPU optimizations.

There is extensive literature on optimizations for desktop-class GPUs but significantly less work on mobile GPUs. For instance, prior work [7] has shown how manual optimizations for Mali GPUs can be used for HPC-style workloads. The paper [15] discusses code generation for an mobile GPU from a domain specific language for image processing.

In contrast, this paper presents a novel technique based on rewrite rules to *automatically* generate optimized code for data parallel applications targeting the Mali GPU.

Polyhedral compilation [3] has been applied to optimize OpenCL code multiple GPUs, including Mali. Unfortunately, the Mali GPU was excluded from the matrix multiplication benchmark. Polyhedral compilation requires complex static analysis for applying transformations on loops with affine memory accesses. In contrast, our rewrite-based approach avoids any static analysis and instead starts out from a simpler functional language without any explicit loops.

High-level approaches for GPU programming.
There exists a rich literature on high-level approaches for GPU programming inspired by functional programming.

Delite [12], Halide [21], Lime [6] all address programmability issues showing that it is possible to simplify GPGPU programming. They all exploit functional concepts such as composability, immutability and absence of side-effects to increase programmability. However, these approaches still rely on hard-coded optimizations which are not performance portable across different types of GPUs.

SkelCL [24] and Accelerate [14] are examples of domain specific languages (DSLs) using data parallel patterns to simplify GPU programming. Bones [19] is a pattern based GPU compiler automatically detecting algorithm species and mapping them to patterns. All three projects rely on pre-written implementations of patterns to generate GPU code. HiDP [16] is a hierarchical data parallel language for GPU programming generating CUDA code. Petabricks [20] allows the user to specify algorithmic choices of implementations which are automatically selected by the compiler and runtime.

All of these high-level projects rely on manually optimized GPU code or device-specific compiler optimizations making the generated code not performance portable. Re-targeting any of these projects to generate efficient code for Mali requires considerable effort, while the approach advocated in this paper based on rewrite rules is capable of generating efficient code for mobile as well as for desktop-class GPUs.

10. CONCLUSION

This paper has shown that the classical auto-tuning technique for matrix multiplication is not performance portable across different types of GPUs. Auto-tuners are built around complex parametric implementations which ultimately end up being specialized for a certain class of devices.

In this paper, we have taken a different approach based on a generic rewrite-based GPU code generator that encodes optimization choices as rewrite rules. This approach can easily apply composition of optimizations which leads to very high-performance code, even on a mobile GPU. The rewrite-based technique makes it easy to add optimizations such as expressing the OpenCL dot-product built-in function, which makes a large performance difference on the Mali GPU. Performing the same optimizations in an auto-tuner parametric implementation would require a significant effort and might result in a highly specialized device-specific version.

Overall, we have shown that the rewrite-based code generator offers true performance portability across desktop GPUs and the Mali mobile GPU. It achieves 1.7x speedup over a state-of-the-art auto-tuner on the Mali GPU and even outperforms the human-expert hand-written version on this device.

Acknoledgements

This work was partially funded by Google.

11. REFERENCES

[1] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. PACT. ACM, 2014.

[2] ARM. Mali-T600 Series GPU OpenCL – Dev. Guide.

[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] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. ICS. ACM, 1997.

[5] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki. *Numerical Computations with GPUs*, chapter Accelerating Numerical Dense Linear Algebra Calculations with GPUs. Springer Intl. Publishing, 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] I. Grasso, P. Radojkovic, N. Rajovic, I. Gelado, and A. Ramírez. Energy efficient HPC on embedded SoCs: Optimization techniques for Mali GPU. IPDPS. IEEE, 2014.

[8] J. Gronqvist and A. Lokhmotov. Optimising OpenCL kernels for the ARM Mali-T600 GPUs. In *GPU Pro 5: Advanced Rendering Techniques*. 2014.

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

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

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

[12] 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.

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

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

[15] R. Membarth, O. Reiche, F. Hannig, and J. Teich. Code generation for embedded heterogeneous architectures on Android. DATE. European Design and Automation Association, 2014.

[16] F. Mueller and Y. Zhang. Hidp: A hierarchical data parallel language. CGO. IEEE, 2013.

[17] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. J. Kelly, A. J. Davison, M. Luján, M. F. P. O'Boyle, G. Riley, N. Topham, and S. Furber. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. ICRA, 2015.

[18] C. Nugteren and V. Codreanu. Cltune: A generic auto-tuner for opencl kernels. MCSoC. IEEE, 2015.

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

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

[21] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. PLDI. ACM, 2013.

[22] T. Remmelg, T. Lutz, M. Steuwer, and C. Dubach. Performance portable GPU code generation for matrix multiplication. GPGPU. ACM, 2016.

[23] 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.

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

[25] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. SC. IEEE, 1998.