Patterns and Rewrite Rules for Systematic Code Generation (From High-Level Functional Patterns to High-Performance OpenCL Code)

Patterns and Rewrite Rules for Systematic Code Generation

From High-Level Functional Patterns to High-Performance OpenCL Code

Michel Steuwer
The University of Edinburgh
University of Muenster
michel.steuwer@ed.ac.uk

Christian Fensch
Heriot-Watt University
c.fensch@ed.ac.uk

Christophe Dubach
The University of Edinburgh
christophe.dubach@ed.ac.uk

Abstract

Computing systems have become increasingly complex with the emergence of heterogeneous hardware combining multicore CPUs and GPUs. These parallel systems exhibit tremendous computational power at the cost of increased programming effort. This results in a tension between achieving performance and code portability. Code is either tuned using device-specific optimizations to achieve maximum performance or is written in a high-level language to achieve portability at the expense of performance.

We propose a novel approach that offers high-level programming, code portability and high-performance. It is based on algorithmic pattern composition coupled with a powerful, yet simple, set of rewrite rules. This enables systematic transformation and optimization of a high-level program into a low-level hardware specific representation which leads to high performance code.

We test our design in practice by describing a subset of the OpenCL programming model with low-level patterns and by implementing a compiler which generates high performance OpenCL code. Our experiments show that we can systematically derive high-performance device-specific implementations from simple high-level algorithmic expressions. The performance of the generated OpenCL code is on par with highly tuned implementations for multicore CPUs and GPUs written by experts.

Keywords Algorithmic Patterns, Rewrite Rules, Performance Portability, GPU, OpenCL, Code Generation

1. Introduction

Computing systems have become extremely complex and diversified implementing different forms of parallelism and memory hierarchies. Modern multicore CPUs and GPUs (Graphics Processing Units) are often used for general purpose computations. The drawback of such systems is the extreme difficulty of programming and extracting performance, requiring a deep understanding of the hardware. Software written and tuned for today's systems needs to be adapted frequently to keep pace with ever changing hardware.

Over the years, a wide range of languages, language extensions and frameworks have emerged for programming GPUs and other massively parallel devices. The two most common languages are CUDA and OpenCL, both directly exposing low-level hardware features. Directive based approaches such as OpenACC [33] and OpenMP [25], extensions to existing programming languages such as Cilk [3] or libraries like Intel TBB [32] have been proposed to reduce the complexity of developing code for multicore CPUs and GPUs. While these latter approaches simplify the development of applications, they all lead to an explosion of specialized implementations where the same algorithmic concept is tuned differently for each device. As a result, *performance portability* remains elusive; code optimized for one device might only achieve a fraction of the performance on a different device.

Several high-level programming models have been proposed to address this issue. Petabricks [29] allows the programmer to express different algorithm implementations and automatically picks the best one using auto-tuning. Higher-level dataflow programming language such as StreamIt [21] or LiquidMetal [15] have been designed with a similar goal in mind. Both languages use dedicated backend compiler for different hardware targets such as GPUs. Nvidia's NOVA [9] language takes a more functional programming approach where algorithmic patterns such as *map* or *reduce* are expressed as primitives recognized by the backend compiler. While definitively a step in the right direction, all these approaches rely on ad-hoc techniques such as hard-coded device-specific implementations or heuristics. When hardware changes occur, the backend compiler has to be re-tuned or re-engineered.

The root of the problem lies in a gap in the system stack between high-level algorithmic concepts on the one hand and low-level hardware paradigms on the other hand. In this work we propose to bridge this gap by defining a set of rewrite rules which systematically translates high-level algorithmic concepts into low-level hardware paradigms, both expressed as functional patterns. The rewrite rules are used to systematically derive semantically equivalent low-level expressions from high-level algorithm expressions written by the programmer. Once derived, we can automatically generate high performance code based on these expressions. Our approach is similar in spirit to Spiral [30], but relies on fine grain hardware patterns representing CPU and GPU hardware features. As a result, in our approach code generation becomes very simple since all optimization decisions are handled during the automatic rewriting process and no complex analysis is performed.

The power of our approach lies in the rewrite rules, written once by an expert system designer. These rules encode the different algorithmic choices and low-level hardware specific optimizations. The rewrite rules play the dual role of enabling the composition of algorithmic patterns and enabling the lowering of these patterns onto the low-level hardware paradigms. This results in a clear separation of concerns between high-level algorithmic patterns and low-level hardware paradigms. The rewrite rules define an implementation space that can be systematically searched to produce high performance code. We believe these principles pave the way to fully automated portable high performance code generation.

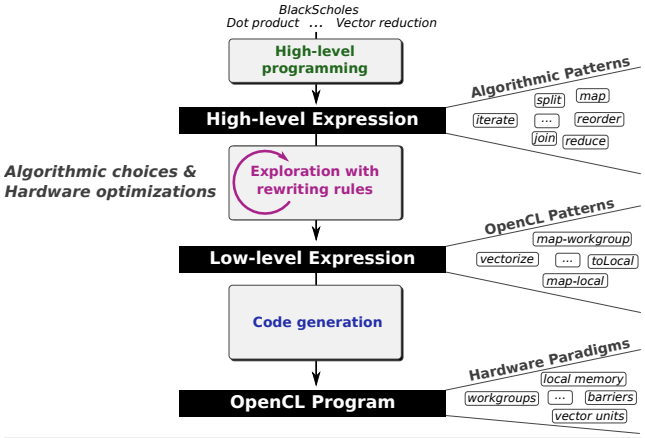


Figure 1: Overview of our system. The programmer expresses the problem with high-level algorithmic patterns. These are systematically transformed into low-level OpenCL patterns using a rule rewriting system. OpenCL code is generated by mapping the low-level patterns directly to the OpenCL programming model representing hardware paradigms.

This paper demonstrates the practicality of our approach using OpenCL as our target hardware platform. We compare our approach with highly-tuned linear algebra functions extracted from the state-of-the-art libraries and with larger benchmarks such as BlackScholes. We express them as compositions of high-level algorithmic patterns which are systematically lowered to low-level OpenCL patterns from which OpenCL code is generated. The performance of our generated code is competitive with highly-tuned BLAS linear algebra libraries such as Nvidia GPU CUBLAS, AMD GPU clBLAS and Intel MKL on the CPU.

Our paper makes the following key contributions:

- design of **high-level algorithmic patterns** used by the programmer and **low-level OpenCL patterns** representing the OpenCL programming model;
- develop a powerful set of **rewrite rules** that systematically expresses algorithmic and optimization choices;
- achieve **performance portability** by systematically applying rewrite rules to derive device-specific implementations, leading to performance on par with the best hand-tuned versions.

The paper is structured as follows. Section 2 provides a motivation. Sections 3 and 4 present our patterns and rewrite rules. Section 5 and 6 show our benchmarks and rules in action. Our experimental setup and performance results are shown in Sections 7 and 8. Finally, Section 9 discusses related work and Section 10 concludes.

2. Motivation

The overview of our approach is presented in Figure 1. The programmer writes a *high-level expression* composed of *algorithmic patterns*. Using a rewrite rule system, we systematically lower this high-level expression into a *low-level expression* consisting of *OpenCL patterns*. In this rewrite stage algorithmic and optimization choices in the high-level expression can be explored. The generated low-level expression is then fed into our code generator that emits an *OpenCL program*. This program is finally compiled to machine code by the vendor provided OpenCL compiler.

We now illustrate the advantages of our approach using a simple vector scaling example shown in Figure 2. The user expresses the computation by writing a high-level expression using our *map* algorithmic pattern as shown in Figure 2a. This coding style is similar to functional and dataflow programming.

```
1  def mul3(x) = x * 3     // user-defined function
2  def vectorScal = map(mul3)        // map pattern
```

(a) High-level expression written by the programmer.

rewrite rules

```
1  def mul3(x) = x * 3
2  def vectorScal = join ∘ map-workgroup(
3                    asScalar ∘ map-local(
4                      vectorize-4(mul3)
5                    ) ∘ asVector-4
6                  ) ∘ split-1024
```

(b) Low-level expression systematically derived using rewrite rules.

code generator

```
1  int4 mul3(int4 x) { return x * 3; }
2  kernel vectorScal(global int* in,out, int len){
3   for (int i=get_group_id; i < len/1024;
4        i+=get_num_groups) {
5    global int* grp_in  = in+(i*1024);
6    global int* grp_out = out+(i*1024);
7    for (int j=get_local_id; j < 1024/4;
8         j+=get_local_size) {
9      global int4* in_vec4 =(int4*)grp_in+(j*4);
10     global int4* out_vec4=(int4*)grp_out+(j*4);
11     *out_vec4 = mul3(*in_vec4);
12 } } }
```

(c) OpenCL program produced by our code generator.

Figure 2: Pseudo-code representing vector scaling. The user simply maps the `mul3` function over the elements of the input array (a). This high-level expression is systematically transformed into a low-level expression (b) using rewrite rules. Finally, our code generator turns the low-level expression into an OpenCL program (c).

Our technique first rewrites the user provided high-level expression into something closer to the OpenCL programming model. This is achieved by applying the rewrite rules presented later in Section 4. Figure 2b shows one possible derivation of the original high-level expression where the $\circ$ operator represents function composition, *i.e.*, $f \circ g(x) = f(g(x))$. Starting from the last line, we first split the input into chunks of 1024 elements. Each chunk is mapped onto a group of threads, called *workgroup*, with the *map-workgroup* low-level pattern (line 2). Within a workgroup (lines 3–5), we vectorize the elements (line 5), each mapped to a local thread inside a workgroup via the *map-local* low-level pattern (line 3). Each local thread now processes 4 elements, enclosed in a vector type. Finally, the *vectorize-4* pattern (line 4) implies that the user defined function `mul3` is vectorized. The exact meaning of our patterns will be given later in Section 3.

The last step consists of traversing the low-level expression and generating OpenCL code for each low-level pattern encountered (Figure 2c). The two map patterns generate the for loops (line 3–4 and 7–8) that iterate over the input array assigning work to the workgroups and local threads. The information of how many chunks each workgroup and thread processes comes from the corresponding *split*. In line 11 the vectorized version of the user defined `mul3` function (defined in line 1) is finally applied to the input array.

To summarize, our approach is able to generate OpenCL code starting from a high-level representation of a program. This is achieved by systematically lowering the high-level expression into a low-level form suitable for code generation. The next two sections present our high-level and low-level patterns, the code generation mechanism and the rewrite rules in more details.

Pattern	Type	Description
map(f)	$T[n] \to U[n], f : T \to U$	Apply function f to every element of the input array.
reduce(f, z)	$T[\,] \to T[1], f : (T, T) \to T, z : T$	Apply the reduction function f with initial value z to the input array.
zip(a,b)	$a : T[n], b : U[n] \to \langle T, U \rangle[n]$	Zip two arrays into an array of pairs.
split^n	$T[m][\,]^* \to T[m/n][n][\,]^*$	Splits the outer most dimension of an array in chunks of size n.
join	$T[m][n][\,]^* \to T[m * n][\,]^*$	Joins the two outer most dimensions of an array.
iterate^n*(f)*	$T[\,] \to T[\,], f : T[\,] \to T[\,]$	Iterate the function f over the input n times.
reorder	$T[n] \to T[n]$	Reorder the element of the input array.

Table 1: High-level algorithmic patterns used by the programmer. $T \to U$ means the function input type is T and output type U. We write $T[n]$ for an array of type T with size n and $[\,]^*$ denotes an arbitrary number of dimensions in an array.

Pattern	Type	Description
map-workgroup(f)	identical to *map(f)*	Each **workgroup** applies function f on a different element of the input array.
map-local(f)	identical to *map(f)*	Each **local thread** applies function f on a different element of the input array.
map-global(f)	identical to *map(f)*	Each **global thread** applies function f on a different element of the input array.
map-seq(f)	identical to *map(f)*	Apply function f to every element of the input array **sequentially**.
reduce-seq(f,z)	identical to *reduce(f,z)*	Apply reduction function f with initial value z to the input **sequentially**.
reorder-stride^s	identical to *reorder*	Access input array with a stride s to maintain **memory coalescing**.
toLocal(f)	$T[\,] \to U[\,], f : T[\,] \to U[\,]$	Change the storage location for the results of function f to **local memory**.
toGlobal(f)	$T[\,] \to U[\,], f : T[\,] \to U[\,]$	Change the storage location for the results of function f to **global memory**.
asVector^n	$T[\,]^*[m] \to Tn[\,]^*[m/n]$	Turns the elements of an array into **vector type**.
asScalar	$Tn[\,]^*[m] \to T[\,]^*[m * n]$	Turns the elements of an array into **scalar type**.
vect^n*(f)*	$T[n] \to U[n], f : T \to U$	**Vectorize** the function f by a factor n.

Table 2: Low-level OpenCL patterns used for code generation. The hardware paradigm used is highlighted in bold in the description.

3. Patterns Design and Implementation

One of the key ideas of this paper is to expose algorithmic choices and hardware-specific program optimizations as patterns that can be systematically derived using a rule rewriting system (discussed later in in Section 4). The high-level algorithmic patterns are designed to be used by the programmer directly. The low-level hardware patterns represent hardware specific concepts expressed by a low-level programming model such as OpenCL, the target chosen for this paper. Following the same approach, a different set of low-level hardware patterns could be designed to target other low-level programming models such as Pthreads or MPI.

This section discusses the design of our patterns and how we generate OpenCL code for them. We define our patterns as functions which are implicitly applied to exactly one input array and produces one output array. To simplify our implementation we decided to encode all types as arrays with primitives represented with arrays of length 1. The only exceptions are the user-provided functions such as the `mul3` function in Figure 2a that operates on a primitive type.

3.1 Algorithmic Patterns

Table 1 presents our high-level algorithmic patterns. These patterns are not tied to any specific hardware feature and are used to define the program at the algorithmic level by the programmer.

Map The *map* pattern is well known in functional programming and applies a given function f to all elements of its input array.

Reduce The *reduce* pattern (a.k.a. fold or accumulate) uses a given binary function f to combine all elements of the input array. We require the function f to be associative and commutative which allows for an efficient parallel implementation.

Zip and Split/Join These patterns transform the shape of the data and we store this information, *i.e.*, number of dimensions and size of each dimension, in the type system. The *zip* pattern fuses two arrays into an array of pairs. The *split* pattern, which is most often combined with a *join*, partitions an array into chunks of specific size resulting in an extra dimension. The corresponding *join* pattern does the opposite; it reassembles arrays of arrays by merging dimensions. These two patterns used together are similar to the split-join concept from data flow languages such as StreamIt [39].

Iterate The *iterate* pattern corresponds to the mathematical definition of iteratively applying a function. It is defined as: $f^0 = id$ and $f^{n+1} = f^n \circ f$. In terms of implementation, our code generator emits a for-loop to perform the iteration, and two pointers for input and output. After each iteration, we swap the pointers, so that the output of the last iteration becomes the input for the next one.

Reorder The *reorder* pattern is used to specify that the ordering of the elements of an array does not matter. This allows our system to reorder arbitrarily the elements of an array and might enable optimizations, as we will see later.

3.2 OpenCL-specific Patterns

It is well known, that programming parallel hardware such as manycore CPUs and GPUs is quite complex. In order to achieve the highest performance, programmers often use a set of rules of thumb to drive the optimization of their application for the specific devices. In fact each hardware vendor provides optimization guides [1, 27] that extensively cover hardware particularities and how to optimize code for them.

In this paper we focus on the OpenCL programming model, which is a popular low-level programming model used to program manycore CPUs and GPUs. Programming these devices consists of writing a compute *kernel* in OpenCL C that executes on the device and writing the host code that orchestrates data movement, allocates memory and manages the execution on the device.

We encode the OpenCL programming model by formalizing hardware paradigms expressed as patterns. Table 2 gives an overview of the OpenCL-specific patterns we have identified.

Parallel Maps The different *map* patterns represent possible ways of mapping computations to the hardware and exploit parallelism in OpenCL. The *map-workgroup(f)* pattern assigns work to a group of threads, called *workgroup* in OpenCL, by letting every workgroup apply the function f on a different element of the input array. Similarly, the *map-local(f)* pattern assigns work to a local thread inside a workgroup. As workgroups are optional in OpenCL the *map-global(f)* pattern assigns work to a global thread, *i.e.*, a thread not organized in a workgroup. This allows us to map computations in different ways to the thread hierarchy of OpenCL.

The code generation for all these map patterns is similar, we describe it using *map-workgroup(f)* as an example. A loop is generated, where the iteration variable is determined by the *workgroup-id*, which is provided by OpenCL. Inside of the loop, a pointer is generated to partition the input array, so that every workgroup processes a different chunk of data. We use this pointer as the input for the function f being bound to the map. Similarly, we generate and use a pointer for the output. After emitting the code for the loop, we continue with the body of the loop by generating the code for the function f. When the generation of the body is finished, an appropriate synchronization mechanism for the given map pattern is added. For instance after a *map-local* we add a barrier synchronization to synchronize the threads inside of the workgroup.

Sequential Map and Reduce The *map-seq(f)* and *reduce-seq(f, z)* patterns perform a sequential map and reduction, respectively, within a single thread. In both cases the generated code consists of a simple for loop iterating over the array and calling the function f. In case of the reduction an accumulation variable is initialized with z. The variable is then passed to the function f in each iteration and the computed result is stored in the accumulation variable and finally written to the output of the reduction.

Reorder-stride Using this pattern the elements of an array are reordered with a stride s. In effect, this generates an access to an array such that $out[i] = in[i/n + s \cdot (i \bmod n)]$, where $n \cdot s$ is the size of the array. This hardware pattern ensures that after splitting the workload, consecutive threads access consecutive memory elements (known as a *coalesce memory access*) which is beneficial on modern GPUs as it maximizes the memory bandwidth.

Our implementation of this pattern does not produce code directly, but generates instead an index function, which is used when accessing the array the next time. While not discussed here, our design allows us to support user-defined index functions as well.

Local/Global The *toLocal(f)* and *toGlobal(f)* patterns are used to determine where the result of function f should be stored. OpenCL defines two distinct address spaces: global and local. Global memory is the commonly used large but slow memory. On GPUs, the small local memory has a high bandwidth with low latency and is used to store frequently accessed data. With these two patterns, we can in effect exploit the memory hierarchy defined in OpenCL. These patterns act similarly to a typecast and are in fact implemented as such so that no code is emitted directly.

In our design, every function reads its input and writes its output using pointers provided by the callee function. As a result, we can simply force a store to local memory by wrapping any function with our *toLocal* pattern. In the code generator, this will simply change the output pointer of function f to an area in local memory.

Vectorize and asVector/asScalar The OpenCL programming model supports vectorization with special data types such as `int4` where any operations on this type will be executed in the hardware vector units. In the absence of vector units in the hardware, the OpenCL compiler scalarizes the code automatically.

The *asVector* and *asScalar* patterns change the data type into vector elements and scalar elements respectively. For instance, in OpenCL an array of `int` is transformed into an array of `int4` as seen in the motivation example (Figure 2). The *vect^n(f)* pattern vectorizes a function by simply converting all the operations in f that apply to vector types into vectorized operations. Our current implementation can only vectorize functions containing simple arithmetic operations such as $+$ or $-$. In case of more complex functions, we rely on external tools [24] for vectorizing the operations.

4. Rewrite Rules

This section introduces our set of rewrite rules that transform high-level expressions written using our algorithmic patterns into semantically equivalent expressions. One goal of our approach is to keep each rule as simple as possible and only express one fundamental concept at a time. For instance the vectorization rule, as we will see, is the only place where we express the vectorization concept. This is different from most prior approaches that would produce a special vectorized version of different algorithmic patterns such as map or reduce. The superiority of our approach lies in the power of composition; many rules can be applied successively to produce expressions that compose hardware concepts or optimizations and that are provably correct by construction.

Similarly to our patterns, we distinguish between algorithmic and lowering rules. Algorithmic rules produces derivations that represent the different algorithmic choices and are shown in Figure 3. Figure 4 shows our OpenCL-specific rules which map expressions to OpenCL patterns. Once the expression is in its lowest form, it is possible to produce OpenCL code for each single pattern easily with our code generator as described in the previous section.

4.1 Algorithmic Rules

Iterate decomposition The rule in Figure 3a expresses the fact an iteration can be decomposed into several iterations.

Reorder commutativity Figure 3b shows a rule stating that if the data can be reordered arbitrarily it does not matter if we apply a function f to each element before or after the reordering.

Split-join The split-join rule in Figure 3c partitions a map into two maps. This allows us to nest map patterns in each other and, thus, *map* the computation to the thread hierarchy of the OpenCL programming model such as *map-workgroup(map-local(f))* as seen in our motivation example (Figure 2).

Reduction The reduction (and associated partial reduction) in Figure 3d is currently our most complex rule but also the most powerful one. It expresses the reduction function as a composition of other primitive functions, which is a fundamental aspect of our work. From the algorithmic point of view we first define a partial reduction pattern *part-red*. This partial reduction reduces an array of n elements to an array of m elements where $1 \leq m < n$. The reduction can be derived in a partial reduction combined with a full reduction which ensures we end up with one unique element.

Partial Reduction The first possible derivation for partial reduction, in Figure 3d, leads to the full reduction which means $m = 1$. The next possible derivation expresses the fact that it is possible to reorder the elements to be reduced, expressing the commutativity property of our definition of reduction. The third derivation is actually the only place where parallelism is expressed in the definition of our reduction pattern. This rule expressed the fact that it is valid to partition the input elements first and then reduce them independently. Finally, the last possible derivation expresses the notion that it is possible to perform a partial reduction with an iterative process by repetitively applying the same partial reduction function. This concept is very important when considering how the reduction function is typically implemented on a GPU (iteratively reducing within a workgroup using the local memory).

$$\textbf{\textit{iterate}}^{m+n}(f) \rightarrow \textbf{\textit{iterate}}^{m}(f) \circ \textbf{\textit{iterate}}^{n}(f)$$

(a) Iterate decomposition

$$\begin{aligned} \textit{map}(f) \circ \textit{reorder} &\rightarrow \textit{reorder} \circ \textit{map}(f) \\ \textit{reorder} \circ \textit{map}(f) &\rightarrow \textit{map}(f) \circ \textit{reorder} \end{aligned}$$

(b) Reorder commutativity

$$\textit{map}(f) \rightarrow \textbf{\textit{join}} \circ \textit{map}(\textit{map}(f)) \circ \textbf{\textit{split}}^{n}$$

(c) Split-join

$$\begin{aligned} \textit{reduce}(f,z) &\rightarrow \textit{reduce}(f,z) \circ \textit{part-red}(f,z) \\ \textit{part-red}(f,z) &\rightarrow \textit{reduce}(f,z) \\ &\mid \textit{part-red}(f,z) \circ \textit{reorder} \\ &\mid \textbf{\textit{join}} \circ \textit{map}(\textit{part-red}(f,z)) \circ \textbf{\textit{split}}^{n} \\ &\mid \textbf{\textit{iterate}}^{n}(\textit{part-red}(f,z)) \end{aligned}$$

(d) Reduction

$$\begin{aligned} \textbf{\textit{split}}^{n} \circ \textbf{\textit{join}}^{n} \mid \textbf{\textit{join}}^{n} \circ \textbf{\textit{split}}^{n} &\rightarrow \textit{id} \\ \textbf{\textit{asVector}}^{n} \circ \textbf{\textit{asScalar}}^{n} \mid \textbf{\textit{asScalar}}^{n} \circ \textbf{\textit{asVector}}^{n} &\rightarrow \textit{id} \end{aligned}$$

(e) Simplification rules

$$\begin{aligned} &\textit{map}(f) \circ \textit{map}(g) \rightarrow \textit{map}(f \circ g) \\ &\textbf{\textit{reduce-seq}}(f,z) \circ \textbf{\textit{map-seq}}(g) \rightarrow \\ &\quad \textbf{\textit{reduce-seq}}(\lambda\, acc, x : f(acc, g(x)), z) \end{aligned}$$

(f) Fusion rules

Figure 3: Algorithmic rules. Bold patterns are known to the code generator.

Simplification Rules Figure 3e shows our simplification rules. They express the fact that consecutive *split-join* pairs and *asVector-asScalar* pairs are equivalent to the identity function *id*.

Fusion Rules Finally, our fusion rules are shown in Figure 3f. The first rule fuses the functions applied by two consecutive maps. The second rule fuses the map-reduce pattern by creating a lambda function that is the results of merging function f and g from the original reduction and map respectively. This rule only applies to the sequential version since this is the only implementation not requiring the associativity property required by the more generic *reduce* pattern. When generating code, these rules in effect allow us to fuse the implementation of the different functions and avoid having to store temporary results. More generic rules for fusion have been studies by the functional programming community [10, 23]. However, as we currently focus on a restricted set of patterns our simpler fusion rules have, so far, proven to be sufficient.

4.2 OpenCL-Specific Rules

Figure 4 shows our OpenCL-specific rules that are used to apply OpenCL optimizations and to lower high-level concepts down to OpenCL-specific ones. Patterns that are known to the code generator are shown in bold in both Figure 3 and 4.

Maps The rule in Figure 4a is used to produce OpenCL-specific map implementations that match the thread hierarchy of the OpenCL programming model. Our implementation maintains context information to ensure the thread hierarchy is respected. For instance, it is only legal to nest a *map-local* inside a *map-workgroup*.

Reduction There is only one lowering rule for reduction (Figure 4b), which expresses the fact that the only OpenCL implementation known to the code generator is a sequential reduction. Possible parallel implementations of the reduction pattern are defined at a higher level by composition of other algorithmic patterns. To the best of our knowledge, all other existing high performance compilers treat the reduction directly as an irreducible primitive operation. The power of our approach is that the code generator implementation only needs to know about the simple sequential reduction. As a result, it is possible to explore different implementation for the reduction by simply applying different rules.

$$\begin{aligned} \textit{map}(f) \rightarrow \textbf{\textit{map-workgroup}}(f) \mid \textbf{\textit{map-local}}(f) \\ \mid \textbf{\textit{map-global}}(f) \mid \textbf{\textit{map-seq}}(f) \end{aligned}$$

(a) Map

$$\textit{reduce}(f,z) \rightarrow \textbf{\textit{reduce-seq}}(f,z)$$

(b) Reduction

$$\textit{reorder} \rightarrow \textbf{\textit{reorder-stride}}^{s} \mid \textit{id}$$

(c) Stride accesses or normal accesses

$$\begin{aligned} \textbf{\textit{map-local}}(f) &\rightarrow \textbf{\textit{toGlobal}}(\textbf{\textit{map-local}}(f)) \\ \textbf{\textit{map-local}}(f) &\rightarrow \textbf{\textit{toLocal}}(\textbf{\textit{map-local}}(f)) \end{aligned}$$

(d) Local/Global memory

$$\textit{map}(f) \rightarrow \textbf{\textit{asScalar}} \circ \textit{map}(\textbf{\textit{vect}}^{n}(f)) \circ \textbf{\textit{asVector}}^{n}$$

(e) Vectorization

Figure 4: OpenCL-specific rules. Bold patterns are known to the code generator.

Reorder Figure 4c presents the rule that reorders elements of an array. In our current implementation, we support two types of reordering: no reordering, represented by the *id* identify function, and *reorder-stride* which reorders elements with a certain stride s. As described earlier, the major use case for the stride reorder is to enable coalesced memory accesses.

Local/Global Figure 4d shows two rules that enable GPU local memory usage. They express the fact that the result of a *map-local* can always be stored in local memory or back in global memory. This holds since a *map-local* always exists within a *map-workgroup* for which the local memory is defined. These rules allow us to determine how the data is mapped to the GPU memory hierarchy.

Vectorization Finally, Figure 4e shows the vectorization rule. Vectorization is achieved by using the *asVector* and corresponding *asScalar* which changes the element type of an array and adjust the length accordingly. This rule is only allowed to be applied once to a given *map(f)* pattern. This constrain can easily be checked by looking at the function's type; if it is a vector type, the rule cannot be applied. Another set of rules, not shown here for space reason, are used to propagate the $vect^n$ function recursively within f.

4.3 Summary

The power of our approach lies in the composition of our rules that produce complex low-level expressions from simple high-level expressions. Looking back at our motivation example in Figure 2, we see how a simple algorithmic pattern such as *map* can effectively be derived into a low-level expression by applying the rules. This expression matches various hardware concepts expressible with the OpenCL programming model such as mapping computation and data to the GPU thread and memory hierarchy and vectorization. Each single rule encodes a simple, easy to understand, provable fact. By composition of the rules we systematically derive low-level expressions which are semantically equivalent to the high-level expressions by construction. This results in a powerful mechanism to safely explore the space of possible implementations.

```
1  def add(x, y)  = x + y
2  def mult(x, y) = x * y
3  def abs(x)     = if (x < 0) -x else x
4
5  def scal(a, x⃗) = map(mult(a), x⃗)
6  def asum(x⃗)    = reduce(add, 0) ∘ map(abs, x⃗)
7  def dot(x⃗, y⃗)  = reduce(add, 0) ∘ map(mult) ∘ zip(x⃗, y⃗)
8  def gemv(A, x⃗, y⃗, a, b) =
9     z⃗ = map(scal(a) ∘ dot(x⃗), A)
10    map(add) ∘ zip( z⃗, scal(b, y⃗) )
```

Figure 5: Linear algebra kernels from the BLAS library expressed using our high-level algorithmic patterns.

5. Benchmarks

We now discuss how applications from linear algebra, mathematical finance and physics can be represented as expressions composed of our high-level algorithmic patterns. We use the following conventions to simplify the syntax: non-capitalized letters (*e.g.*,x) denote scalar variables, letters with an arrow on top (*e.g.*, $\vec{x}$) denote 1D vectors, and capitalized letters (*e.g.*, A) denote 2D matrices.

5.1 Linear Algebra Kernels

We choose linear algebra kernels as our first set of benchmarks, because they are well known, easy to understand, and used as building blocks in many other applications. Figure 5 shows how we express vector scaling (line 5), sum of absolute values (line 6), dot product of two vectors (line 7) and matrix vector multiplication (line 8–10) using our high-level patterns. While the first three benchmarks perform computations on vectors, matrix vector multiplication was chosen to illustrate a computation using a 2D data structures.

For scaling (line 5), the *map* pattern applies a function to each element which multiplies it with a constant. This function is expressed by partially applying the `mult` function, *i.e.*, binding `a` to the first argument of `mult`. The sum of absolute values (line 6) and the dot product (line 7) applications both produce scalar results by performing a summation, which we express using the *reduce* pattern combined with the addition. For dot product, a pair-wise multiplication of its two input vectors is performed before applying the reduction. This is expressed using the *zip* and *map* patterns.

Line 8–10 shows the implementation of matrix vector multiplication as defined by the BLAS library: $\vec{y} = \alpha A\vec{x} + \beta\vec{y}$. To multiply matrix A with vector $\vec{x}$, the *map* pattern maps the computation of the dot-product with the input vector $\vec{x}$ to each row of the matrix A (line 9). Notice how we are reusing the high-level expressions for dot-product and scaling as building blocks for the more complex matrix-vector multiplication. This shows the power of our system: expressions describing algorithmic concepts can be reused, without committing to a particular low-level implementation; The dot-product from gemv (line 9) might be implemented in a totally different way from the stand-alone dot-product kernel (line 7).

5.2 Mathematical Finance Application

The BlackScholes application uses a Monte-Carlo method for option pricing and computes for each stock price `s` a pair of call and put options `{c, p}`. Figure 6 shows the BlackScholes implementation, where the function defined in line 1 computes the call and put option for a single stock price `s`. Two intermediate results `d1` and `d2` are computed and used to compute the call and put options which are returned as a single pair. The `compD1`, `compD2`, `compCall` and `compPut` functions are not shown here since they only contain purely sequential code implementing the BlackScholes model. This `BSComputation` function is applied to all stock prices, stored in a vector $\vec{s}$, using the *map* pattern in line 4.

```
1  def BSComputation(s) =
2    d1 = compD1(s); d2 = compD2(d1,s)
3    return { compCall(d1,d2,s), compPut(d1,d2,s) }
4  def blackScholes(s⃗) = map(BSComputation, s⃗)
```

Figure 6: BlackScholes mathematical finance application expressed using our high-level algorithmic patterns.

```
1  def updateF(f, nId, p, p⃗, t) =
2    n = p⃗[nId];  d = calculateDistrance(p, n)
3    if (d < t) f += calculateForce(d)
4    return f
5  def md(p⃗, N, t) = map(
6    λ p,n⃗: reduce(λ f,nId: updateF(f,nId,p,p⃗,t), 0, n⃗)
7   ) ∘ zip(p⃗, N)
```

Figure 7: Molecular dynamics physics application expressed using our high-level algorithmic patterns.

5.3 Physics Application

Another application we consider is the the molecular dynamics (MD) application from the SHOC [12] benchmark suite. It calculates the sum of all forces acting on a particle from its neighbors. Figure 7 shows the implementation using our high-level patterns.

The function `updateF` is defined in line 1 and updates the force `f` of particle `p` by computing and adding the local force between a single particle and one of its neighbors. `updateF` takes an index of a neighbor `nId`, the vector storing all particles $\vec{p}$, and a threshold `t` as additional parameters. Using `nId` and $\vec{p}$ the neighboring particle is accessed in line 2 and the distance between the neighboring particle and the particle `p` ist computed. If the distance is below the given threshold `t` the local force between the two particles is calculated based on the distance and added to the overall force `f` (line 3) which is finally returned in line 4. Otherwise the particle is ignored in the summation.

For computing the force for all particles $\vec{p}$, we use the *zip* pattern (line 7) to build a vector of pairs. Each pair combines a single particle with the indices of all of its neighboring particles. The function which is applied to each pair by the *map* pattern (line 5) is expressed as an lambda expression (line 6). Computing the resulting force exerted by all the neighbors on one particle is done by applying the *reduce* pattern on vector $\vec{n}$ which stores the indices of the neighboring particles. We use the previously defined function `updateF` inside the reduction to compute the force each particle with index `nId` add to the overall force on `p`. At this point we fix all but the first two arguments as the other arguments remain constant for particle `p`. The usage of lambda expressions in our system allows for easy binding of additional information as arguments to functions. This application example should give some evidence that our patterns are flexible enough to implement real world applications.

6. Deriving Specialized Implementations

This section shows how our rules can be applied to derive different implementations starting from the same high-level expression. We illustrate this process using the *asum* benchmark from the previous section as a simple example. The computation can easily be expressed using two of our high-level algorithmic patterns, as shown in Figure 8 (1). The *abs* function is applied to every element of the input vector $\vec{x}$ and then the intermediate result is summed up using the *reduce* pattern which is customized with the addition operator.

$$
\begin{aligned}
\mathit{asum}(\vec{x}) &= \mathit{reduce}(+,0) \circ \mathit{map}(\mathit{abs}, \vec{x}) && (1)\\
&\overset{3d}{=} \mathit{reduce}(+,0) \circ \mathit{join} \circ \mathit{map}(\mathit{part\text{-}red}(+,0)) \circ \mathit{split}^n \circ \mathit{map}(\mathit{abs}, \vec{x}) && (2)\\
&\overset{3c}{=} \mathit{reduce}(+,0) \circ \mathit{join} \circ \mathit{map}(\mathit{part\text{-}red}(+,0)) \circ \mathit{split}^n \circ \mathit{join} \circ \mathit{map}(\mathit{map}(\mathit{abs})) \circ \mathit{split}^n(\vec{x}) && (3)\\
&\overset{3e}{=} \mathit{reduce}(+,0) \circ \mathit{join} \circ \mathit{map}(\mathit{part\text{-}red}(+,0)) \circ \mathit{map}(\mathit{map}(\mathit{abs})) \circ \mathit{split}^n(\vec{x}) && (4)\\
&\overset{3f}{=} \mathit{reduce}(+,0) \circ \mathit{join} \circ \mathit{map}(\mathit{part\text{-}red}(+,0) \circ \mathit{map}(\mathit{abs})) \circ \mathit{split}^n(\vec{x}) && (5)\\
&\overset{4a}{=} \mathit{reduce}(+,0) \circ \mathit{join} \circ \mathit{map}(\mathit{part\text{-}red}(+,0) \circ \mathit{map\text{-}seq}(\mathit{abs})) \circ \mathit{split}^n(\vec{x}) && (6)\\
&\overset{3d\&4b}{=} \mathit{reduce}(+,0) \circ \mathit{join} \circ \mathit{map}(\mathit{reduce\text{-}seq}(+,0) \circ \mathit{map\text{-}seq}(\mathit{abs})) \circ \mathit{split}^n(\vec{x}) && (7)\\
&\overset{3f}{=} \mathit{reduce}(+,0) \circ \mathit{join} \circ \mathit{map}(\mathit{reduce\text{-}seq}(\lambda\ \mathit{acc}, a : \mathit{acc} + \mathit{abs}(a), 0) \circ \mathit{split}^n(\vec{x}) && (8)
\end{aligned}
$$

Figure 8: Derivation for $\mathit{asum}(\vec{x})$ to a fused version. The numbers above the equality sign refer to the rules from Figure 3 and Figure 4.

(a)

```
def asum(x⃗) = reduce-seq o join o map-workgroup(
     join o toGlobal( map-local( map-seq(id) ) ) o split-1
   o iterate-7( join o map-local( reduce-seq(plus, 0) ) o split-2 )
   o join o toLocal( map-local( reduce-seq(absAndPlus, 0) ) ) o split-2048 o reorder-stride
 ) o split-262144(x⃗)
```

(b)

```
def asum(x⃗) = reduce-seq o join o asScalar o map-workgroup(
     join o toGlobal( map-local( map-seq(vectorize-4(id) ) ) ) o split-1
   o iterate-8( join o map-local( reduce-seq(vectorize-4(plus), vectorize-4(0)) ) o split-2 )
   o join o toLocal( map-local( reduce-seq(vectorize-4(absAndPlus), vectorize-4(0)))) o split-2 o reorder-stride
 ) o asVector-4 o split-2048(x⃗)
```

(c)

```
def asum(x⃗) = reduce-seq o join ∘ asScalar ∘ map-workgroup(
   join o map-local( reduce-seq(vectorize-4(absAnd+), vectorize-4(0)) ) o split-8192
 ) o asVector-4 o split-32768(x⃗)
```

Figure 9: Low-level expressions performing the sum of absolute values specialized for Nvidia (a), AMD (b), and Intel (c). These expressions are systematically derived by our system from the high-level expression $\mathit{reduce}(+,0) \circ \mathit{map}(\mathit{abs}, \vec{x})$.

6.1 Deriving a Fused Implementation

To achieve good performance it is in general beneficial to avoid storing intermediate results. Rule 3f allows us to apply this principle and fuse two patterns into one, thus, avoiding an intermediate result. Figure 8 shows how we can systematically derive a fused version of the *asum* application from the high-level expression written by the programmer. We write the derivation as a sequence of equations using a slightly more mathematical notation, where the numbers above the equality sign refer to the rules applied.

To obtain expression (2) we apply the reduction rule 3d twice: first to replace *reduce* with *reduce* $\circ$ *part-red* and then a second time to expand *par-red*. Afterwards, we expand *map* to get (3), which can be simplified by removing the two corresponding *join* and *split* patterns. In the step from (4) to (5) two *map* patterns are fused and in the next step the nested *map* is lowered into the *map-seq* pattern to obtain (6). By first transforming *part-red* back into *reduce* (using rule 3d) and then applying the lowering rule 4b we get (7). Finally, we apply rule 3f to fuse the *map-seq* and *reduce-seq* into a single *reduce-seq*. This sequence of transformations results in expression (8) which allows for a more optimal implementation since no temporary storage is required for the intermediate result.

6.2 Deriving Device Specific Implementations

The previous section showed how an optimization can be systematically applied which is generally beneficial on every hardware platform. However, there exist many optimizations which are highly specific to a particular hardware architecture. For instance it is often beneficial to apply vectorization on an Intel CPU but not on an Nvidia GPU, on the contrary using local memory is usually beneficial on GPUs but not on CPUs. Figure 9 shows three different implementations of the *asum* benchmark which have been derived using the same systematic approach of applying rules as seen in Figure 8. These implementations have been in fact inspired by hand-tuned OpenCL and CUDA kernels from the different vendors. This demonstrates the expressive power of our OpenCL patterns. Each implementation is optimized to take advantage of the features of a particular hardware architecture. The integer parameters deciding how to split the data and the width of vectorization where chosen by exploring different values empirically.

The first implementations, shown in Figure 9a, is optimized for an Nvidia GPU. The input vector $\vec{x}$ is split in large chunks which are processed in parallel by different workgroups. Inside each workgroup the `reorder-stride` pattern ensures fast coalesced memory accesses when loading the data from global memory. Each local thread reduces 2048 elements and stores the intermediate result in local memory. Afterwards, the entire workgroup performs an iterative computation to reduce the intermediate results down to a single result before this is copied back to global memory. Figure 9b shows the AMD optimized implementation which is similar to the previous one. The same set of optimizations have been applied to take advantage of the local memory and to ensure coalesced memory accesses. In addition, since the AMD GPU has vector units, the reduction has been vectorized by a width of four.

The third implementation as seen in Figure 9c, is targeted at an Intel CPU and is very different. It neither uses local memory nor the `reorder-stride` pattern. Vectorization is applied, similar to the AMD version, but in contrast, the implementation uses different numbers for partitioning the data for workgroup and local threads; only a single thread is active inside each workgroup This corresponds to the fact, that there is less parallelism available on a CPU compared to GPUs. These three different implementations derived from the same high-level expression should give some evidence of the power of our approach which is able to systematically derive highly hardware-specific implementations.

6.3 Towards Automatic Derivation

We have shown how our system can systematically transform and optimize programs at an algorithmic level and at a hardware level without performing complex compiler analysis. While manually deriving the expression is a tedious process, our vision is for this to take place in a fully automated way following the principles introduced in our work. Our rules and OpenCL-specific patterns pave the way to a fully automatic search strategy starting from the high-level expression. There has been other work in this area ranging from random search to sophisticated search strategies based on performance models or machine learning techniques [13, 29]. We see this work as completely orthogonal to this paper, since our first focus is to develop the systematic foundations necessary to apply such techniques. Nonetheless, we have implemented a prototype automatic search technique that is actually able to find expressions with similar performance to those presented in Figure 9. This suggest it is possible to automatically derive highly tuned implementations from high-level expressions but this is left for future work.

7. Experimental Setup

This section describes some implementation details of our code generator and our experimental setup used for the experiments.

7.1 Implementation Details

Our system is implemented in C++11, using the template system and support for lambda functions. When generating code for a derived expression two basic steps are performed. First, we use the Clang/LLVM compiler library to parse the input expression and produce an abstract syntax tree for it. Second, we traverse the tree and emit code for every function call representing one of our low-level hardware patterns.

As part of the first step, we have developed our own type system which plays a dual role. First, it prevents the user to produce incorrect expressions. Secondly, the type system encodes information necessary for code generation, such as memory address space and array size information, which are used to allocate memory.

The design of our code generator is straightforward since no optimization decisions are made at this stage. We avoid performing complex analysis of the code which makes our design very different compared to traditional optimizing compilers.

7.2 Hardware Platforms and Evaluation Methodology

We used three hardware platforms to perform the runtime experiments: an Nvidia GeForce GTX 480 GPU, an AMD Radeon HD 7970 GPU and a dual socket Intel Xeon E5530 server, with 8 cores in total and hyper-threading enabled. We used the latest OpenCL runtime from Nvidia (CUDA-SDK 5.5), AMD (AMD-APP 2.8.1) and Intel (XE 2013 R3 3.2.1.16712). The GPU drivers installed on our Linux system were 310.44 for Nvidia and 13.1 for AMD.

We use the profiling APIs from OpenCL and CUDA to measure kernel execution time and the *gettimeofday* function for the CPU implementation. Following the Nvidia benchmarking methodology [20], the data transfer time to and from the GPU is excluded. We repeat each experiment 1000 times and report median runtimes.

For our linear algebra benchmarks, we have performed experiments with two input sizes. For *scal*, *asum* and *dot*, the small input size corresponds to a vector size of 16M elements (64MB). The large input size uses 128M elements (512MB, the maximum OpenCL buffer size for our platforms). For *gemv*, we use an input matrix of 4096×4096 elements (64MB) and a vector size of 4096 elements (16KB) for the small input size. For the large input size, the matrix size is 8192×16384 elements (512MB) and the vector size 8192 elements (32KB). For *BlackScholes*, the problem size is fixed to 4 million elements and for *MD* it is 12288 particles.

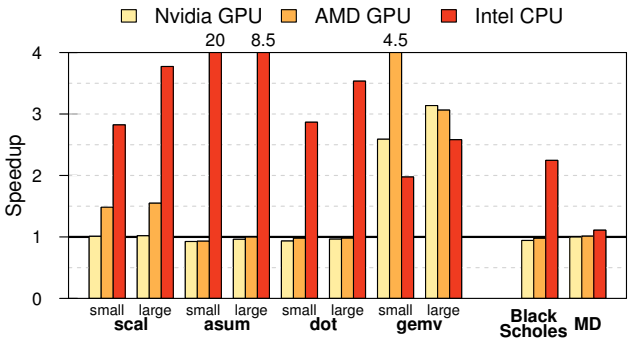


Figure 10: Performance of our approach relative to a portable OpenCL reference implementation. We outperform the clBLAS implementation on most benchmarks and platforms.

8. Results

We now evaluate our approach compared to reference OpenCL implementations of our benchmarks on all platforms. Furthermore, we compare the BLAS routines against platform-specific tuned implementations.

8.1 Comparison vs. Portable Implementation

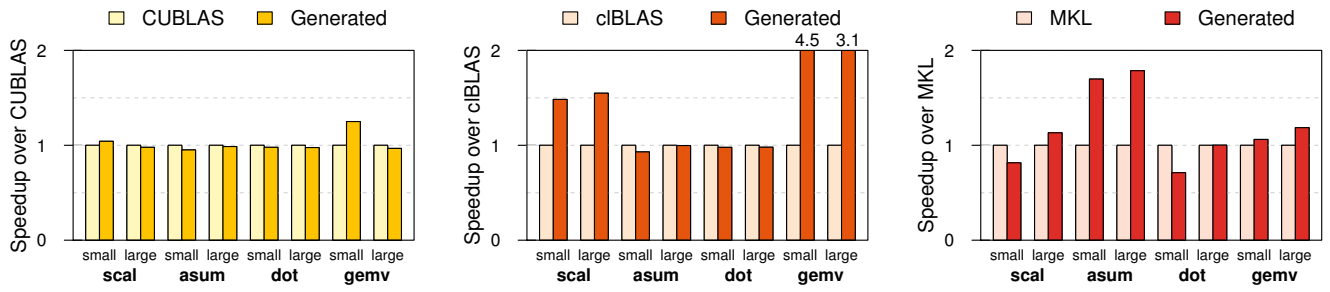
We want to show how our approach performs across three platforms. We use the BLAS OpenCL implementations written by AMD as our baseline for this evaluation since it is inherently portable across our different platforms. Figure 10 shows the performance of our approach relative to clBLAS for the BLAS routines. As can be seen, we achieve better performance than clBLAS on most platforms and benchmarks. The speedups are the highest on the CPU, with up to 20× for the *asum* benchmark with a small input size. The reason is that clBLAS was written and tuned specifically for an AMD GPU which usually exhibit a larger number of parallel processing units. As we saw in Section 6.2, our systematically derived expression for this benchmark is specifically tuned for the CPU by avoiding creating too much parallelism, which is what gives us such large speedup.

Figure 10 also shows the results we obtain relative to the Nvidia SDK *BlackScholes* and SHOC molecular dynamics *MD* benchmark. For *BlackScholes*, we see that our approach is on par with the performance of the Nvidia implementation on both GPUs. On the CPU, we actually achieve a 2.2× speedup due to the fact that the Nvidia implementation is tuned for GPUs while our implementation generates different code for the CPU. For *MD*, we are actually on par with the OpenCL implementation on all platforms.

8.2 Comparison vs. Highly-tuned Implementations

We now compare our approach with a highly-tuned implementation for each platform. For Nvidia, we pick the highly tuned CUBLAS CUDA-specific implementation of BLAS written by Nvidia. For the AMD GPU, we use the same clBLAS implementation as before given that it has been written and tuned specifically for AMD GPUs. Finally, for the CPU we use the Math Kernel Library (MKL) implementation of BLAS written by Intel which is known for its high performance.

Figure 11a shows that we actually match the performance of CUBLAS for *scal*, *asum* and *dot* on the Nvidia GPU. For *gemv* we outperform CUBLAS on the small size by 20% while we are within 5% for the large input size. Given that CUBLAS is a proprietary library highly tuned for Nvidia GPUs, these results should offer some confidence that our technique is able to achieve high performance.



(a) Nvidia GPU **(b)** AMD GPU **(c)** Intel CPU

Figure 11: Performance comparison of our approach relative to a highly-tuned platform-specific library; CUBLAS for Nvidia, clBLAS for AMD and MKL for the CPU. Our approach matches the performance of CUBLAS and MKL, and outperforms clBLAS on some routines.

On the AMD GPU, we are surprisingly up to 4.5× faster than the clBLAS implementation on *gemv* small input size as shown in Figure 11b. The reason for this is found in the way clBLAS is implemented. For the *gemv* benchmark, clBLAS performs automatic code generation using fixed templates. In contrast to our approach they only generate one implementation since they do not explore different pattern compositions.

For the Intel CPU (Figure 11c), we see that our approach beats MKL for one benchmarks and match the performance of MKL on most of the other three benchmarks. For the small input sizes for the *scal* and *dot* benchmarks we are within 13% and 30% respectively. For the larger input sizes we are on par with MKL for both benchmarks. The *asum* implementation in the MKL does not use thread level parallelism, where our implementation does and, thus, achieves a speedup of up to 1.78 on the larger input size.

This section has shown how our approach can generate *performance portable* code that is competitive with highly-tuned platform specific implementations.

9. Related Work

Algorithmic Patterns Algorithmic patterns or skeletons [8] have been around for more than two decades. Pattern-based libraries for platforms ranging from cluster systems [34] to GPUs [37] have been proposed with recent extension to irregular algorithms [17]. This includes popular framework such as Map-Reduce [14] from Google. Many researchers have looked at the problem of optimizing map-reduce operations for different type of hardware. Paraprox [35] for instance uses automatic detection of algorithmic patterns to apply optimization at the expense of accuracy. Compared to our approach, most prior works rely on hardware-specific implementations to achieve high performance. Conversely, we systematically generate implementations using fine-grain OpenCL patterns combined with our rule rewriting system.

Functional Approaches for GPU Code Generation Accelerate is a functional domain specific language built within Haskell to support GPU acceleration [7, 26]. Recently, Nvidia has presented NOVA [9], a new functional language target at code generation for GPUs, and Copperhead [5], a data parallel language embedded in Python. NOVA shares many concepts from Accelerate and Copperhead and offers familiar data parallel patterns. HiDP [43] is a hierarchical data parallel language which maps computations to the OpenCL programming model similar to our approach. All these projects rely on analysis of user code or hand-tuned versions of high-level algorithmic patterns. In contrast, our approach uses rewrite rules and low-level hardware patterns to produce high-performance code in a portable way.

Halide [31] is a domain specific approach targeting image processing pipelines. It separates a programs functional algorithmic description from optimization decisions and applies autotuning to find the best optimization on different hardware platforms. Our work is domain agnostic and takes a different approach to achieve high performance. We systematically describe hardware paradigms as functional patterns instead of encoding specific optimizations which might not apply to future hardware generations.

Rewrite-rules for Optimizations Rewrite rules have been used very early as a way to automate the optimization process of functional programs [23]. Recently, rewriting has been applied to HPC applications [28] as well, where the user annotates imperative code providing information necessary for the rewrite process. Similar to us, Spiral [30] also uses rewrite rules to optimize signal processing programs and was more recently adapted to linear algebra [36] and other mathematical domains [16]. In contrast our rules and OpenCL hardware patterns are expressed at a much finer level, allowing for highly specialized and optimized code generation.

High-level Code Generation for GPUs A large body of work has explored how to automatically generate high performance code for GPUs. Dataflow programming models such as IBM's LiquidMetal [15] or StreamIt [39] have been used to automatically produce GPU code with OpenCL or CUDA [21, 22, 40]. Directive based approach have also been used such as OpenMP to CUDA [25], OpenACC to OpenCL [33], or hiCUDA [19] which translates sequential C code to CUDA. Optimized implementations for directive based reductions on GPUs has been presented [42] as well. X10 [38], a language for high performance computing, can also be used to program GPUs [11]. However, the programming style remains low-level since the programmer has to express the same low-level operations found in CUDA or OpenCL. Recently, researchers have looked at generating efficient GPU code for loops using the polyhedral framework [18, 41]. Delite [4, 6], a system that enables the creation of domain-specific languages, can also target multicore CPUs or GPUs. Unfortunately, all these approaches do not provide full performance portability since the mapping of the application assumes a fixed platform and the optimizations and implementations are targeted at a specific device.

Finally, Petabricks [2] takes a different approach by letting the programmer specify different algorithms implementations. The compiler and runtime then choose the most suitable one based on an adaptive mechanism and can produce OpenCL code [29]. Compared to our work, they generate optimized code by relying on static analysis. Our code generator does not make any decisions nor perform any analysis since the optimization process happens at a higher level within our rewrite rules.

10. Conclusion

In this paper, we have presented a novel approach based on rewrite rules to represent algorithmic principles as well as low-level hardware-specific optimization. We have shown how these rules can be systematically applied to transform a high-level expression into a device-specific implementation. This results in a clear separation of concern between high-level algorithmic concepts and low-level hardware optimizations which pave the way for fully automated high performance code generation.

To demonstrate the power of our approach in practice, we have developed OpenCL-specific rules and patterns together with an OpenCL code generator. The design of the code generator is straight-forward given that all optimizations decisions are made with the rules and no complicated analysis passes are needed. We achieve performance on par with highly tuned platform-specific BLAS libraries on three different devices; AMD GPU, Nvidia GPU and Intel CPU. For benchmarks such as matrix vector multiplication we even reach a speedup of up to 4.5. We also show that our technique achieves portable performance for more complex applications such as the BlackScholes benchmark or for molecular dynamics simulation.

References

[1] *AMD Accelerated Parallel Processing OpenCL Programming Guide*. AMD, 2013.

[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. In *PLDI*, 2009.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPoPP*, 1995.

[4] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, 2011.

[5] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *PPoPP*, 2011.

[6] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPoPP*, 2011.

[7] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP' 11*.

[8] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, 1989.

[9] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. NOVA: A functional language for data parallelism. In *ARRAY*, 2014.

[10] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *ICFP '07*. ACM, 2007.

[11] D. Cunningham, R. Bordawekar, and V. Saraswat. GPU programming in a high level language: Compiling X10 to CUDA. In *X10*, 2011.

[12] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU*, 2010.

[13] F. de Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. Bandit-based optimization on graphs with application to library performance tuning. In *ICML*, 2009.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *CACM*, 51, 2008.

[15] 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). In *PLDI*, 2012.

[16] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel. Operator language: A program generation framework for fast kernels. In *DSL WC*, 2009.

[17] C. González and B. Fraguela. An algorithm template for domain-based parallel irregular algorithms. *IJPP*, 42(6), 2014.

[18] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *CGO '14*.

[19] T. D. Han and T. S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE TPDS*, 22(1), Jan. 2011.

[20] M. Harris. *Optimizing Parallel Reduction in CUDA*. Nvidia, 2007.

[21] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: Portable stream programming on graphics engines. In *ASPLOS*, 2011.

[22] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh. Scalable framework for mapping streaming applications onto multi-GPU systems. In *PPoPP*, 2012.

[23] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop'01*, 2001.

[24] R. Karrenberg and S. Hack. Whole-function vectorization. In *CGO'11*.

[25] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP'99*.

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

[27] *Nvidia OpenCL Best Practices Guide*. Nvidia, 2011.

[28] A. Panyala, D. Chavarria-Miranda, and S. Krishnamoorthy. On the use of term rewriting for performance optimization of legacy HPC applications. In *ICPP*, 2012.

[29] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. In *ASPLOS '13*.

[30] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005.

[31] 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. In *PLDI*, 2013.

[32] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[33] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande. accULL: An OpenACC implementation with CUDA and OpenCL support. In *Euro-Par*, 2012.

[34] C. Rodrigues, T. Jablin, A. Dakkak, and W.-M. Hwu. Triolet: A programming system that unifies algorithmic skeleton interfaces for high-performance cluster computing. In *PPoPP*, 2014.

[35] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS*, 2014.

[36] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. In *CGO*, 2014.

[37] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A portable skeleton library for high-level GPU programming. In *HIPS Workshop*, 2011.

[38] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 and APGAS at petascale. In *PPoPP*, 2014.

[39] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *CC*, 2002.

[40] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *CGO*, 2009.

[41] 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.

[42] R. Xu, X. Tian, Y. Yan, S. Chandrasekaran, and B. Chapman. Reduction operations in parallel loops for GPGPUs. In *PMAM*, 2014.

[43] Y. Zhang and F. Mueller. HiDP: A hierarchical data parallel language. In *CGO*, 2013.