Generating Performance Portable Code using Rewrite Rules
From High-Level Functional Expressions to High-Performance OpenCL Code

Michel Steuwer
The University of Edinburgh (UK)
University of Münster (Germany)
michel.steuwer@ed.ac.uk

Christian Fensch
Heriot-Watt University (UK)
c.fensch@hw.ac.uk

Sam Lindley
The University of Edinburgh (UK)
sam.lindley@ed.ac.uk

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

Abstract
Computers 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 resulting in a tension between performance and code portability. Typically, code is either tuned in a low-level imperative language using hardware-specific optimizations to achieve maximum performance or is written in a high-level, possibly functional, language to achieve portability at the expense of performance.

We propose a novel approach aiming to combine high-level programming, code portability, and high-performance. Starting from a high-level functional expression we apply a simple set of rewrite rules to transform it into a low-level functional representation, close to the OpenCL programming model, from which OpenCL code is generated. Our rewrite rules define a space of possible implementations which we automatically explore to generate hardware-specific OpenCL implementations. We formalize our system with a core dependently-typed λ-calculus along with a denotational semantics which we use to prove the correctness of the rewrite rules.

We test our design in practice by implementing a compiler which generates high performance imperative OpenCL code. Our experiments show that we can automatically derive hardware-specific implementations from simple functional high-level algorithmic expressions offering performance on a par with highly tuned code for multicore CPUs and GPUs written by experts.

Categories and Subject Descriptors D3.2 [Programming Languages]: Language Classification – Applicative (functional) languages; Concurrent, distributed, and parallel languages; D3.4 [Processors]: Code generation, Compilers, Optimization

Keywords Algorithmic patterns, rewrite rules, performance portability, GPU, OpenCL, code generation

1. Introduction
In recent years, graphics processing units (GPUs) have emerged as the power horse of high-performance computing. These devices offer enormous raw performance but require programmers to have a deep understanding of the hardware in order to maximize performance. This means software is written and tuned on a per-device basis and needs to be adapted frequently to keep pace with ever changing hardware.

Programming models such as OpenCL offer the promise of functional portability of code across different parallel processors. However, performance portability often remains elusive; code achieving high performance for one device might only achieve a fraction of the available performance on a different device. Figure 1 illustrates this problem by showing how a parallel reduce (a.k.a. fold) implementation, written and optimized for one particular device, performs on other devices. Three implementations have been tuned to maximize performance on each device: the Nvidia_opt and AMD_opt implementations are tuned for the Nvidia and AMD GPU respectively, implementing tree-based reduce using an iterative approach with carefully specified synchronization primitives. The Nvidia_opt version utilizes the local (a.k.a. shared) memory to store intermediate results and exploits a hardware feature of Nvidia GPUs to avoid certain synchronization barriers. The AMD_opt version does not perform these two optimizations but instead uses vectorized operations. The Intel_opt parallel implementation, tuned for an Intel CPU, also relies on vectorized operations. However, it uses a much coarser form of parallelism with fewer threads, in which each thread performs more work.

![Figure 1: Performance is not portable across devices. Each bar represents the device-specific optimized implementation of a parallel reduce implemented in OpenCL and tuned for an Nvidia GPU, AMD GPU, and Intel CPU respectively. Performance is normalized with respect to the best implementation on each device.](image-url)
Figure 1 shows the performance achieved by each implementation on three different devices. Running an implementation which has been optimized on a different device leads to suboptimal performance in all cases. Consider the AMD_opt implementation, for instance, where we see that the performance loss is 20% when running on the Nvidia GPU and 90% (i.e., 10 x slower) when running on the CPU. The CPU optimized version, Intel_opt, achieves less than 20% (i.e., 5 x slower) when run on a GPU. Finally, it is worth noting that the Nvidia_opt version, which performs quite badly on the AMD GPU, actually fails to execute correctly on the CPU. This is due to a low-level optimization which removes synchronization barriers which can be avoided on the GPU, but are required on the CPU for correctness.

This lack of performance portability is mainly due to the low-level nature of the programming model; the dominant programming interfaces for parallel devices such as GPUs exposes programmers to many hardware-specific details. As a result, programming becomes complex, time-consuming, and error prone.

Several high-level programming models have been proposed to tackle the programmability issue and shield programmers from low-level hardware details. High-level dataflow programming language such as StreamIt [25] and LiquidMetal [19] allow the programmer to easily express different implementations at the algorithm level. Nvidia’s NOV A [12] language takes a more functional approach in which higher-order functions such as map and reduce are expressed as primitives recognized by the backend compiler. Similarly, Accelerate [9] allows the programmer to write high-level functional code in a DSL embedded in Haskell, and automatically generate CUDA code for the GPU. For instance, the parallel reduce discussed earlier would be written in Accelerate as:

```
sum xs = fold (+) 0 (use xs)
```

These kind of approaches hide the complexity of parallelism and low-level optimizations from the user. However, they rely on hard-coded device-specific implementations or heuristics to drive the optimization process. When targeting different devices, the library implementation or backend compiler has to be re-tuned or even worst re-engineered. In order to address the performance portability issue, we aim to develop mechanisms that can effectively explore device-specific optimizations. The core idea is not to commit to a specific implementation or set of optimizations but instead to let a tool automate the process.

In this paper we present an approach which compiles a high-level functional expression – similar to the one written in Accelerate – into highly optimized device-specific OpenCL code. We show that we achieve performance on a par with expert-written implementations on an Intel multicore CPU, an AMD GPU, and an Nvidia GPU. Central to our approach is a set of rewrite rules that systematically translate high-level algorithmic concepts into low-level hardware paradigms, both expressed in a functional style. The rewrite rules are based on the kind of algebraic reasoning well-known to functional programmers, and pioneered by Bird [5] and others in the 1980s. They are used to systematically transform programs into a low-level representation, from which high-performance code is generated automatically.

The power of our technique 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 high-level algorithmic concepts and enabling the mapping of these onto hardware paradigms, but also critically provide correctness preserving exploration of the implementation space. The rules enable a clear separation of concerns between high-level algorithmic concepts and low-level hardware paradigms while using a unified framework. The defined implementation space is automatically searched to produce high performance code.

Figure 2: The overview of our approach is presented in Figure 2. The programmer writes a high-level expression composed of algorithmic primitives. Using rewriting rules, we map this high-level expression into a low-level expression consisting of OpenCL primitives. In the rewriting stage, different algorithmic and optimization choices can be explored. The generated low-level expression is then fed into our code generator that emits an OpenCL program compiled to machine code by the vendor provided OpenCL compiler.

The primary contributions of our paper are as follows:

- a collection of high-level functional algorithmic primitives for the programmer and low-level functional OpenCL primitives representing the OpenCL programming model;
- a core dependently-typed calculus and denotational semantics;
- a set of rewrite rules that systematically express algorithmic and optimization choices, bridging the gap between high-level functional programs and OpenCL;
- proofs of the soundness of the rewrite rules with respect to the denotational semantics;
- achieving performance portability by systematically applying rewrite rules to yield device-specific implementations, with performance on a par with the best hand-tuned versions.

The remainder of the paper is structured as follows. Section 2 provides an overview of our technique. Sections 3 and 4 present our functional primitives and rewrite rules. Section 5 presents a core language and denotational semantics, which we use to justify the rewrite rules. Section 6 explains our automatic search strategy, while Section 7 introduces our benchmarks. Our experimental setup and performance results are shown in Sections 8 and 9. Finally, Section 10 discusses related work and Section 11 concludes.

2. Overview

```latex
\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2.png}
\caption{The overview of our approach is presented in Figure 2. The programmer writes a high-level expression composed of algorithmic primitives. Using rewriting rules, we map this high-level expression into a low-level expression consisting of OpenCL primitives. In the rewriting stage, different algorithmic and optimization choices can be explored. The generated low-level expression is then fed into our code generator that emits an OpenCL program compiled to machine code by the vendor provided OpenCL compiler.}
\end{figure}
```
We illustrate the mechanisms of our approach using a simple vector scaling example shown in Figure 3. The user expresses the computation by writing a high-level expression using the map primitive as shown in Figure 3a. Our expressions are glued together using rewrite rules in a search process. Finally, our code generator turns the high-level expression into an OpenCL program (c).

![Figure 3: Pseudo-code representing vector scaling.](image)

We let $\lambda x.s$ represent a function mapping an argument $x$ to a value $s$. Thus, $\lambda x . \text{map} (\lambda x . x * 3) x$s represents a high-level expression which multiplies an array by 3. Using rewrite rules, the high-level expression is transformed into a low-level expression by writing

$$\lambda x . (\text{join} \circ \text{mapWorkgroup} (\text{joinVec} \circ \text{mapLocal} (\lambda x . x * 3)) \circ \text{splitVec} 4) \circ \text{split} 1024) x$$

This is the low-level, search-based expression derived using rewrite rules and search.

1. int4 mul3(int4 x) { return x * 3; }
2. kernel vectorScaI(global int* in, out, int len){
3. for (int i=get_group_id(); i < len/1024; i+=get_num_groups)
4. global int* grp_in = in+(i*1024);
5. global int* grp_out = out+(i*1024);
6. for (int j=get_local_id(); j < 1024/4; j+=get_local_size) {
7. global int4 in_vec4 = as4(grp_in+(j*4));
8. global int4 out_vec4 = as4(grp_out+(j*4));
9. out_vec4 = mul3(in_vec4);
10. }
11. }

We can transform this expression to an OpenCL program which we write using the rewrite rules of Figure 4.

![Figure 4: High-level algorithmic primitives.](image)

We let $\lambda x.s$ represent a function mapping an argument $x$ to a value $s$. Thus, $\lambda x . \text{map} (\lambda x . x * 3) x$s represents a high-level expression which multiplies an array by 3. Using rewrite rules, the high-level expression is transformed into a low-level expression by writing

$$\lambda x . (\text{join} \circ \text{mapWorkgroup} (\text{joinVec} \circ \text{mapLocal} (\lambda x . x * 3)) \circ \text{splitVec} 4) \circ \text{split} 1024) x$$

This is the low-level, search-based expression derived using rewrite rules and search.

1. int4 mul3(int4 x) { return x * 3; }
2. kernel vectorScaI(global int* in, out, int len){
3. for (int i=get_group_id(); i < len/1024; i+=get_num_groups)
4. global int* grp_in = in+(i*1024);
5. global int* grp_out = out+(i*1024);
6. for (int j=get_local_id(); j < 1024/4; j+=get_local_size) {
7. global int4 in_vec4 = as4(grp_in+(j*4));
8. global int4 out_vec4 = as4(grp_out+(j*4));
9. out_vec4 = mul3(in_vec4);
10. }
11. }

We can transform this expression to an OpenCL program which we write using the rewrite rules of Figure 4.

We let $\lambda x.s$ represent a function mapping an argument $x$ to a value $s$. Thus, $\lambda x . \text{map} (\lambda x . x * 3) x$s represents a high-level expression which multiplies an array by 3. Using rewrite rules, the high-level expression is transformed into a low-level expression by writing

$$\lambda x . (\text{join} \circ \text{mapWorkgroup} (\text{joinVec} \circ \text{mapLocal} (\lambda x . x * 3)) \circ \text{splitVec} 4) \circ \text{split} 1024) x$$

This is the low-level, search-based expression derived using rewrite rules and search.

1. int4 mul3(int4 x) { return x * 3; }
2. kernel vectorScaI(global int* in, out, int len){
3. for (int i=get_group_id(); i < len/1024; i+=get_num_groups)
4. global int* grp_in = in+(i*1024);
5. global int* grp_out = out+(i*1024);
6. for (int j=get_local_id(); j < 1024/4; j+=get_local_size) {
7. global int4 in_vec4 = as4(grp_in+(j*4));
8. global int4 out_vec4 = as4(grp_out+(j*4));
9. out_vec4 = mul3(in_vec4);
10. }
11. }

We can transform this expression to an OpenCL program which we write using the rewrite rules of Figure 4.

There is a high-level account of the primitives; Section 5 gives a more formal account. Figure 4 and 5 present our algorithmic and OpenCL primitives. The type system we present here is monomorphic (largely to keep the formal presentation in Section 5 simple), however, we do rely on a restricted form of dependent types. The only kind of type-dependency we allow is for array types, whose size may depend on a run-time value. Type inference is beyond the scope of this paper, but in the future we intend to apply ideas from systems such as DML [45] to our setting.

We let $I$ range over sizes. A size can be a size variable $n$, or a product $I \times J$ of sizes $I$ and $J$. We let $A, B$ range over types. We write $A 	o B$ for a function from type $A$ to type $B$ and $(n : size)$ to $A^n$ for a dependent function from size $n$ to type $B$ (where $B$ may include array types whose sizes depend on $n$). We write $A \times B$ for the product of types $A$ and $B$ and $1$ for the unit type. We write $[A]_I$ for an array of size $I$ with elements of type $A$. The primitives are annotated with type and size subscripts. Thus, formally each one actually represents a type-indexed family of primitives. We often omit subscripts when they are not relevant or can be trivially inferred.

3.1 Algorithmic Primitives

As in Accelerate [9, 30], we deliberately restrict ourselves to a set of primitives for which we know that high performance CPU and GPU implementations exist. In contrast to Accelerate, we allow nesting of primitives to express nested parallelism. Nesting of arrays is used to represent multi-dimensional data structures like matrices. Figure 4 presents the high-level primitives used to define programs at the algorithmic level. The map and zip primitives are standard.

The reduce primitive is a special case of a fold returning a single reduced element in an array of size 1. We assume the supplied function is associative and commutative in order to admit efficient parallel implementations. Returning the result as an array with a single element allows for a more compositional design, in which our primitives operate on arrays rather than scalar values.
The mapGlobal primitive performs a sequential map within a single thread. We describe this using mapGlobal as an example. A loop is generated, where the iteration variable is determined by the workgroup-id function from the OpenCL API. Inside the loop, a pointer is generated to partition the input array, so that every workgroup calls the given function \( f \) on a different chunk of data. An output pointer is generated similarly. We continue with the body of the loop by generating the code for the function \( f \) recursively. Finally, an appropriate synchronization mechanism is added for the given map primitive. For instance, after a mapLocal we add a barrier synchronization for the threads inside the workgroup.

### Local/Global

The toLocal and toGlobal primitives are used to determine where the result of the given 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 or for efficient communication between local threads (shared memory). With these two primitives, we can in effect exploit the memory hierarchy defined in OpenCL. These primitives act similarly to a typecast (their high-level semantics is that of the identity function) and are in fact implemented as such, so that no code is emitted directly. We check for incorrect use of these primitives in our implementation. For example, the implementation checks that a toLocal primitive is eventually followed by a toGlobal primitive to ensure that the final result is copied back into global memory, as required by OpenCL. We plan to extend our type system in the future to track the memory location of arrays using an effect system.

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

### Sequential Reduce

The reduceSeq primitive performs a sequential reduce within a single thread. The generated code consists of an accumulation variable which is initialized with the given initial value. A loop is generated iterating over the array and calling the given function which stores its intermediate result in the accumulation variable. Note, that we require the function passed to reduce to be associative and commutative in order to enable an efficient parallel implementation. We do not impose the same restriction for the reduceSeq function, as here we guarantee a sequential order of execution; thus reduceSeq has a more general type.

### Partial Reduce

The reducePart primitive performs a partial reduce, i.e., an array of \( n \) elements is reduced to an array of \( m \) elements where \( 1 \leq m \leq n \). While not directly used to generate OpenCL code, reducePart is useful as an intermediate representation for deriving different implementations of reduce as we will see in the next section.

### Reorder Stride

The high-level semantics of reorderStride,\( A,J \ n \) is just like reorder,\( A,J \). The low-level implementation actually performs a specific reordering in which the array is reordered with a stride \( n \), that is, element \( i \) is mapped to element \( i / n + n \times (i \% n) \). In the generated OpenCL code this primitive ensures that after splitting the workload, consecutive threads access consecutive memory elements (i.e., coalesce memory access), which is beneficial on modern GPUs as it maximizes memory bandwidth.

Our implementation does not produce code directly, but generates instead an index function, which is used when accessing the array the next time. While beyond the scope of this paper, our design supports user-defined index functions as well.

### Vectorization

The OpenCL programming model supports SIMD vector 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.
Of course compiled differently. Concretely, the mapVec function is
sent the different algorithmic choices and are shown in Figure 6.
Figure 6: Algorithmic rules. Bold functions are known to the code
low-level rules. Algorithmic rules produce derivations that repre-
respect the types involved. That the rules are sound. The rules are only valid given that they re-
compose hardware concepts or optimizations. In Section 5 we show
contrasts with many prior approaches that provide special vectorized
see, is the only place where we express vectorization. This con-
concept at a time. For instance the vectorization rule, as we will
express the fact that consecutive split-join pairs and splitVec-
joinVec pairs are equivalent to the identity.

\[
\text{iterate} \ (I + J) \ M \rightarrow \ \text{iterate} \ I \ M \circ \ \text{iterate} \ J \ M
\]

(a) Iterate decomposition rule

\[
\text{map} \ M \circ \ \text{reorder} \rightarrow \ \text{reorder} \circ \ \text{map} \ M
\]

\[
\text{reorder} \circ \ \text{map} \ M \rightarrow \ \text{map} \ M \circ \ \text{reorder}
\]

(b) Reorder commutativity rules

\[
\text{map}_{A,B,I \times J} \ M \rightarrow \ \text{join}_{B,I \times J} \ M \circ \ \text{map} \ (\text{map} \ M) \circ \ \text{split}_{A,J} I
\]

(c) Split-join rule

\[
\text{reduce}_{A,I \times J} \ M N \rightarrow \ \text{reduce}_{A,J} \ M N \circ \ \text{reducePart}_{A,I} \ M N J
\]

\[
\text{reducePart}_{A,I} \ M N J \rightarrow \ \text{reduce}_{A,I} \ M N \circ \ \text{reorder}
\]

\[
\text{reducePart}_{A,K} \ M N J \rightarrow \ \text{iterate}_{A,K} \ (\text{reducePart}_{A,I} \ M N J)
\]

\[
\text{reducePart}_{A,J} \ M N (J \times K) \rightarrow \ \text{join} \circ \ \text{map} \ (\text{reducePart}_{A,M N J}) \circ \ \text{split}_{A,K} \ (I \times J)
\]

(d) Reduce rules

\[
\text{join} \circ \ \text{split} I \ | \ \text{split}_{A,J} I \circ \ \text{join}_{B,J} I \rightarrow \ \text{id}
\]

\[
\text{joinVec} \circ \ \text{splitVec} I \ | \ \text{splitVec}_{A,J} I \circ \ \text{joinVec}_{A,I,J} \rightarrow \ \text{id}
\]

(e) Cancellation rules

\[
\text{map} \ M \circ \ \text{map} N \rightarrow \ \text{map} \ (M \circ N)
\]

\[
\text{reduceSeq} \ M N \circ \ \text{mapSeq} \ P \rightarrow \ \text{reduceSeq} \ (\lambda (\text{acc}, x).M (\langle \text{acc}, P x \rangle)) \ N
\]

(f) Fusion rules

Figure 6: Algorithmic rules. Bold functions are known to the code
generator.

At a high-level, vectors are just a special case of arrays. We write \((A)^I\) for the type of a vector of size \(I\) with elements of type
\(A\). The mapVec, splitVec, and joinVec primitives behave just like the corresponding operations on arrays, though at a low-level they are
of course compiled differently. Concretely, the mapVec primitive vectorizes a function by simply converting all of its operations that apply to vector types into vectorized operations. Our current implement-
aption can only vectorize functions containing simple arithmetic operations such as \(+\) or \(-\). For more complex functions, we rely on external tools [27] for vectorizing the operations, without performing further analysis.

4. Rewrite Rules

This section presents our rewrite rules, which transform high-level expressions written using the algorithmic primitives into semanti-
cally 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 vectorization. This con-
trasts with many prior approaches that provide special vectorized
versions of different algorithmic primitives such as map and reduce.
Many rules can be applied successively to produce expressions that
compose hardware concepts or optimizations. In Section 5 we show
that the rules are sound. The rules are only valid given that they re-
spect the types involved.

As with the primitives, we distinguish between algorithmic and
low-level rules. Algorithmic rules produce derivations that repre-
sent the different algorithmic choices and are shown in Figure 6.

\[
\begin{array}{c}
\text{map} \ M \rightarrow \ \text{mapWorkgroup} \ M \mid \text{mapLocal} \ M \\
\text{mapGlobal} \ M \mid \text{mapSeq} \ M
\end{array}
\]

(a) Map rules

\[
\begin{array}{c}
\text{reduce}_{A,I} \ M N \rightarrow \ \text{reduceSeq}_{A,A,I} \ M N
\end{array}
\]

(b) Reduce rule

\[
\begin{array}{c}
\text{reorder}_{A,I \times J} \rightarrow \ \text{reorderStride}_{A,J} I \ | \ \text{id}
\end{array}
\]

(c) Stride accesses or normal accesses rules

\[
\begin{array}{c}
\text{mapLocal} \ M \rightarrow \ \text{toGlobal} \ (\text{mapLocal} \ M)
\end{array}
\]

mapLocal \ M \rightarrow \ \text{toLocal} \ (\text{mapLocal} \ M)

(d) Local/Global memory rules

\[
\begin{array}{c}
\text{map}_{A,B,I \times J} \ M \rightarrow \ \text{joinVec}_{B,I \times J} \circ \ \text{map}_{A,B,J} (\text{mapVec}_{A,B,I} \ M) \circ \ \text{splitVec}_{A,J} I
\end{array}
\]

(e) Vectorization rule

Figure 7 shows our OpenCL-specific rules which map expressions
to OpenCL primitives. Once an expression is in its lowest-level
form, it is possible to produce OpenCL code for each single primitive
easily with our code generator as described in the previous section.

4.1 Algorithmic Rules

Iterate Decomposition Rule The rule 6a expresses the fact that
an iteration can be decomposed into several iterations.

Reorder Commutativity Rule Figure 6b shows 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 Rule The split-join rule in Figure 6c partitions a map
into two maps. This allows us to nest map primitives in each other
and, thus, maps the computation to the thread hierarchy of the
OpenCL programming model.

Reduce Rules The reduce rules of Figure 6d decompose applica-
tions of the reduce function and the partial reduce function.

- A reduce can be decomposed into a partial reduce combined
  with a full reduce.
- A partial reduce can be turned back into a full reduce if it yields
  a single element.
- A partial reduce can be reordered, exploiting the restriction of
  reducePart to commutative functions.
- A partial reduce can be decomposed into an iteration of a
  smaller instance of the same partial reduce. This idea is impor-
tant when considering how the reduce function is commonly
implemented on a GPU (iteratively reducing within a work-
  group using the local memory).
- A partial reduce can split the input elements, reduce them inde-
  pendently, and then join them back together. This final case is
  actually the only place where parallelism is made explicit in the
  reduce rules. It exploits the restriction of reducePart to associa-
  tive functions.

Cancellation Rules Figure 6e shows our cancellation rules. They
express the fact that consecutive split-join pairs and splitVec-
joinVec pairs are equivalent to the identity.
Fusion Rules Finally, our fusion rules are shown in Figure 6f. The first rule fuses the functions applied by two consecutive maps. The second rule fuses the map-reduce pattern by creating a lambda abstraction that is the result of merging functions \( f \) and \( g \) from the original reduce 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 primitive. When generating code, these rules in effect allow us to fuse the implementation of different functions and avoid having to store temporary results. The functional programming community has studied more sophisticated and generic rules for fusion \([13, 26, 30]\). However, for our current restricted set of benchmarks our simpler fusion rules have proven to be sufficient. We intend to incorporate related work into our approach in the future.

4.2 OpenCL-Specific Rules

Figure 7 shows our OpenCL-specific rules that are used to apply OpenCL optimizations and to lower high-level concepts down to OpenCL-specific ones. Primitives that are known to the code generator are shown in bold.

Map Rules The rule in Figure 7a is used to produce OpenCL-specific map implementations that match the OpenCL thread hierarchy. Our implementation maintains context information to ensure the OpenCL thread hierarchy is respected. For instance, it is only legal to nest a \( \text{mapLocal} \) inside a \( \text{mapWorkgroup} \) and it is not legal to nest two \( \text{mapLocal} \) in each other.

Reduce Rule There is only one low-level rule for reduce (Figure 7b), which expresses the fact that the only implementation known to the code generator is a sequential reduce. Parallel implementations are defined at a higher level by composition of other algorithmic primitives. Most existing approaches treat the reduce primitive. When generating code, these rules in effect allow us to fuse the implementation of different functions and avoid having to store temporary results. The functional programming community has studied more sophisticated and generic rules for fusion \([13, 26, 30]\). However, for our current restricted set of benchmarks our simpler fusion rules have proven to be sufficient. We intend to incorporate related work into our approach in the future.

4.3 Summary

In our approach the power of composition allows our rules to produce complex low-level expressions from simple high-level expressions. Looking back at our example in Figure 3, we see how a simple algorithmic pattern can effectively be derived into a low-level expression by applying the rules. This expression matches hardware concepts expressible with OpenCL such as mapping computation and data to the thread and memory hierarchy. Each single rule encodes a simple, easy to understand, and 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.

5. Core Language

In this section we formalize a core language for programming with the primitives of Section 3. We specify a type system and a denotational semantics for the core language, which we use to justify the correctness of the rewrite rules of Section 4.

5.1 Typing Rules

Figure 8 shows the typing rules for the core language. The type schemas for constants are given in Figure 4 in Section 3. A size environment \( \Delta \) is a set of size variables. A type environment \( \Gamma \) is a map from term variables to types. The judgement \( \Delta \vdash I \) states that in size environment \( \Delta \) the size \( I \) is well-formed. The judgement \( \Delta \vdash A \) states that in size environment \( \Delta \) the type \( A \) is well-formed. The typing judgement \( \Delta; \Gamma \vdash M : A \) states that in size environment \( \Delta \) and type environment \( \Gamma \), the term \( M \) has type \( A \). The typing rules are straightforward.

5.2 Semantics

We give a set-theoretic denotational semantics for the core language. It is presented in Figure 9. Sizes are interpreted straightforwardly as natural numbers. Types are interpreted as sets. We write \( F \) for the set of floating point numbers in the meta language. We overload some of the type constructors in the object language as the corresponding set constructors in the meta language, for instance, \( X \rightarrow Y \) denotes the set of functions from the set \( X \) to the set \( Y \). Size-dependent functions are interpreted as size-dependent functions in the meta language. Arrays are interpreted in the obvious way as functions from sizes to elements.

Size environments are interpreted as \( \text{size maps} \), partial maps from size variables to natural numbers. Type environments are interpreted as \( \text{type maps} \), partial maps from term variables to sets.

Sizes, types, type environments, terms and primitives are all interpreted with respect to a partial map \( \rho \) from size variables to natural numbers (that is, the interpretation of a size environment). Similarly, terms are interpreted with respect to a partial map \( \rho \) from term variables to values. We overload \( \lambda \)-abstraction, pairing, and unit in the obvious way in the meta language.

The interpretation of terms is standard. The interpretations of the primitives are also quite straightforward. Note that for simplicity we here ascribe a fixed evaluation order to the operation of reduce, but when we actually apply the rewrite rules we ensure that the operation is associative and commutative, allowing it to be reordered. The \( \text{iterate} \) operation supplies a successively smaller size for each iteration.

We define function composition in the standard way, both in the object and meta language:

\[
M \circ N \equiv \lambda x. M (N x) \quad f \circ g \equiv \lambda v. f (g v)
\]

**Theorem 1** (Type soundness).

\[
\Delta; \Gamma \vdash M : A \Rightarrow [M]_{[\Delta]}; ([\Gamma]_{[\Delta]}) \in [A]_{[\Delta]}
\]

**Proof.** By induction on the derivation \( \Delta; \Gamma \vdash M : A \).

Our core language can be naturally extended to include all of the primitives of Figures 4 and 5. One can model \( \text{reorder} \) by lifting the entire semantics to model non-determinism by returning sets of
values rather a single value. Many of the low-level primitives have
the same denotation as the corresponding high-level primitives:

\[
\begin{align*}
\Delta \vdash \text{size} \quad &\quad \Delta \vdash \text{join} \\
\Delta \vdash \text{split} \quad &\quad \Delta \vdash \text{reducePart}
\end{align*}
\]

The semantics of the remaining two primitives is as follows:

\[
\begin{align*}
\text{reduceSeq} \quad &\quad \text{reducePart}_{A, I} \\
\text{mapSeq} \quad &\quad \text{mapWorkgroup}
\end{align*}
\]

The semantics of the remaining two primitives is as follows:

\[
\begin{align*}
\text{mapSeq} \quad &\quad \text{mapLocal} \\
\text{mapWorkgroup} \quad &\quad \text{mapGlobal}
\end{align*}
\]

\[
\begin{align*}
\text{reduceSeq} \quad &\quad \text{reduce}
\end{align*}
\]
\[ \text{asum}_J : [\text{float}]_J \rightarrow [\text{float}]_J \]
\[ \text{asum}_{1 \times J} = \text{reduce}_{\text{float}, 1 \times J} (+) 0 \circ \text{map} \text{abs} \]
\[ \rightarrow \text{reduce}_{\text{float}, 1 \times J} (+) 0 \circ \text{reducePart}_{\text{float}, 1} (+) 0 J \circ \text{map} \text{abs} \]
\[ \rightarrow \text{reduce} (+) 0 \circ \text{join} \circ \text{map} (\text{reducePart} (+) 0 1) \circ \text{split}_{\text{float}, J} I \circ \text{map} \text{abs} \]
\[ \rightarrow \text{reduce} (+) 0 \circ \text{join} \circ \text{map} (\text{reducePart} (+) 0 1) \circ \text{split} I \circ \text{join} \circ \text{map} (\text{map} \text{abs}) \circ \text{split} I \]
\[ \rightarrow \text{reduce} (+) 0 \circ \text{join} \circ \text{map} (\text{reducePart} (+) 0 1) \circ \text{map} (\text{map} \text{abs}) \circ \text{split} I \]
\[ \rightarrow \text{reduce} (+) 0 \circ \text{join} \circ \text{map} (\text{reducePart} (+) 0 1) \circ \text{mapSeq} I \circ \text{map} \text{abs} \circ \text{split} I \]
\[ \rightarrow \text{reduce} (+) 0 \circ \text{join} \circ \text{map} (\text{reduceSeq} (+) 0 1 \circ \text{mapSeq} \text{abs}) \circ \text{split} I \]
\[ \rightarrow \text{reduce} (+) 0 \circ \text{join} \circ \text{map} (\text{reduceSeq} (\lambda (\text{acc}, a). \text{acc} + (\text{abs} a)) 0) \circ \text{split} I \]

**Figure 10:** Derivation of a fused parallel implementation of absolute sum.

### 5.3 Correctness of Rewrite Rules
Using the denotational semantics along with a small amount of equational reasoning, it is straightforward to prove the correctness of the rewrite rules of Section 4. We illustrate the nature of these proofs by giving a proof for the split-join rule (Figure 6c) as an example. The proofs for all other rules can be found in [40].

\[ [[\text{join} \circ \text{map} (\text{map} f) \circ \text{split} n]_{\lambda, \rho}] \]
\[ = \lambda x. (x (i / [I]_n)) (i i [I]_n) \]
\[ = \lambda x. (\lambda i f x i. f (x i) (\lambda x i. (\lambda x i. (\rho f) (x i)) (\lambda x i. x (i \times (i n) + j))) \]
\[ (\beta\text{-reduction}) \]
\[ = \lambda x i. (\rho f) (x (i (i n) + j)) (i / [I]_n) (i i [I]_n) \]
\[ (\beta\text{-reduction}) \]
\[ = \lambda x i. (\rho f) (x (i (i n) + j)) (i i [I]_n) \]
\[ (i < [I]_n) \]
\[ = \lambda x i. (\rho f) (x i) \]
\[ (\text{definition of } [[-]]) \]
\[ = [[\text{map} f]_{\lambda, \rho}] \]

### 5.4 Example Use of Rewrite Rules
We now illustrate how the rewrite rules can be applied to derive optimized implementations. To achieve good performance it is in general beneficial to avoid storing intermediate results. Our rewrite rule 6f allows us to apply this principle and fuse two primitives into one, thus, avoiding intermediate results. Figure 10 shows the derivation of a fused version of the code for calculating the absolute sum of an array of numbers, asum, from a high-level expression written by the programmer. The derivation consists of a sequence of rewrites. The annotations on rewrites refer to the rules from Figure 6 and Figure 7.

We begin by applying reduce rules 6d twice: first to decompose reduce into reduce 0 reducePart (1) and second to expand reducePart (2). Next we expand map abs (3), deforest the adjacent split and join (4), and fuse the adjacent maps (5). We now realize the inner map as a sequential mapSeq (6), and the inner reducePart as a sequential reduceSeq (7). Finally, we fuse the inner reduceSeq and mapSeq (8). The resulting expression yields a more efficient implementation than the original code as the intermediate result does not need to be materialized.

### 6. Searching for Good Derivations
We now present an automatic search strategy to find good expressions by applying the rules presented in Section 4.

#### 6.1 Automatic Search
The rules presented earlier define a search space of possible implementations. In order to find the best possible low-level expressions for a given target device, we have developed a simple automatic search strategy based loosely on Bandit-based optimization [17]. Our current search strategy is rather basic and just designed to prove that it is possible to find good implementations automatically. We envision replacing this exploration strategy in the future by using machine-learning techniques to avoid having to search the space at all. However, this is orthogonal to the work presented in this paper.

Our search strategy starts with the high-level expression and determines all the valid rules that can be applied. We use a Monte-Carlo method for evaluating the potential impact of each rule by randomly walking down the search tree. We execute the code generated from the randomly chosen expressions and measure its performance. The rule that promises the best performance following the Monte-Carlo descent is chosen and the resulting derivation fixed and used as a starting point for the next random walk. This process is repeated until we reach a terminal expression. In addition to selecting the rules, we also search at the same time for the parameters controlling our primitives such as the parameter for the split n. We limit the choices for these numerical parameters to a reasonable set appropriate for our test hardware.

In order to speed up the search process, we incorporate macro rules to guide the optimization process more efficiently. Macro rules are rules which perform multiple small steps at once by applying a set of rules in a predefined order. One example of such a macro rule is the fusion of map and reduce as discussed in Figure 10. While not strictly necessary, these macro rules provide shortcuts for the most commonly used sequences of derivations.

#### 6.2 Found Expressions
Figure 11 shows several low-level expressions found by applying the automatic search technique described in Section 6.1. We started from the high-level expression for the sum of absolute use-case (asum) and tested it on two GPUs and one CPU (described later in Section 8). We can make several important observations. First, in all of the expressions the fusion macro rule merging map and reduce was applied. The second observation is that none of the
asum for the search to achieve the best performance on two GPUs and one CPU.

6.3 Search Efficiency

We now present some evidence that our search strategy is effective. Figure 12 shows how many expressions were evaluated during the search to achieve the best performance on two GPUs and one CPU for the asum application. The performance of the best expression found is discussed in Section 9, here we focus on the search efficiency. Each evaluated expression is represented as a point grouped from left to right by the number of fixed derivations in the search tree. The red line connects the fastest expressions found so far.

The performance improves steadily for all three platforms before reaching a plateau. For both GPUs the best performance is reached after testing \( \approx 40 \) expressions. At this point we have fixed five derivations and found a subtree offering good performance for some expressions. Nevertheless, even in the later stages of the search many expressions offer bad performance, which is partly due to the sensitivity of the GPU to the particular numerical parameters. On the CPU performance converges quicker and more expressions offer good performance. This shows that the CPU is easier to optimize for an not as sensitive when selecting numerical parameters.

Overall the search took less than an hour to complete on all platforms, with an average execution time per expression of around 1/2 of a second, including OpenCL code generation, compilation, data transfers, and execution. We believe an implementation optimized for fast code generation could significantly reduce the search time.

7. Benchmarks

We now discuss how applications can be represented as expressions composed of our high-level algorithmic primitives using a set of easy to understand benchmarks from the fields of linear algebra, mathematical finance, and physics.
7.1 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 13 shows how we express vector scaling, sum of absolute values, dot product of two vectors and matrix vector multiplication using our high-level primitives. While three benchmarks perform computations on vectors, matrix vector multiplication illustrates a computation using a 2D data structures, where we exploit nested parallelism.

For scaling (scal), the map primitive applies a function to each element which multiplies it with a constant $a$. The sum of absolute values (asum) and the dot product (dot) applications both produce scalar results by performing a summation, which we express using the reduce primitive combined with addition. For dot product, a pair-wise multiplication of the two input vectors is performed before reducing the result using addition.

The gemv benchmark performs matrix vector multiplication as defined in BLAS: $\vec{y} = \alpha A\vec{x} + \beta \vec{y}$. To multiply matrix $A$ with $\vec{x}$, we map the computation of the dot-product with the input vector $\vec{x}$ over each row of the matrix $A$. Notice how we are reusing the high-level expressions for dot-product and scaling as building blocks for the more complex matrix-vector multiplication. Expressions describing algorithmic concepts can be reused, without committing to a particular low-level implementation. After optimisation, the dot-product from gemv might be implemented in a completely different way from a stand-alone dot-product.

7.2 Mathematical Finance Application

The BlackScholes application uses a Monte-Carlo method for option pricing and computes for each stock price a pair of call and put options. Figure 13 shows the BlackScholes implementation, where the function compCallPut computes the call and put option for a single stock price. It is applied to all stock prices using the map primitive. A detailed discussion of a similar financial benchmark can be found in [2], which is also parallelized using map.

7.3 Physics Application

Another application we consider is the molecular dynamics (md) application from the SHOC benchmark suite [15]. It calculates the sum of all forces acting on a particle from its neighbors. Figure 13 shows the implementation using our high-level primitives.

The function updateF updates the force $f$ of particle $p$ by computing and adding the force between a single particle and one of its neighbors, based on the neighbor’s index $nId$ and the vector storing all particles $p$. It only updates the force if the computed distance between the two particles is below a given threshold $t$.

For computing the force for all particles $ps$, we use the zip primitive to build a vector of pairs, where each pair combines a single particle with the indices of all of its neighboring particles. Computing the resulting force exerted by all the neighbors on one particle is done by applying reduce on vector $ns$ storing the neighboring indices and using updateF as the reduce operation.

7.4 Limitations

In our experimental evaluation, we have chosen to mainly focus on linear algebra kernels; these kernels have been studied in depth and have specialized high-performance libraries implementations on many devices. While our approach is currently limited by the small number of high-level primitives we support, it can be easily extended to support more complex applications found in benchmark suites such as Rodinia [10] or SHOC [15]. However, the two larger applications already demonstrate the applicability of our approach beyond linear algebra kernels. In the future, we intend to extend our set of primitives to support additional patterns found in more complex benchmarks such as stencil applications.

$$
scal = \lambda a. \text{map} \ (\ast a)
$$

$$
asum = \text{reduce} \ (+) \ 0 \ \circ \ \text{map} \ \text{abs}
$$

$$
dot = \lambda xs\ ys. \ (\text{reduce} \ (+) \ 0 \ \circ \ \text{map} \ (+)) \ (\text{zip} \ xs \ ys)
$$

$$
gemv = \lambda \text{mat} \ ys\ \alpha\ \beta. \ \text{map} \ (+) \ \{ \text{zip} \ (\text{map} \ (\alpha \ \circ \ \text{dot} \ xs) \ \text{mat}) \ (\alpha \ \beta \ ys) \}
$$

blackScholes = \text{map} \ \text{compCallPut}

$$
md = \lambda p\ nsbs. \ \text{map} \ (\lambda (p, ns). \ \text{reduce} \ (\lambda f\ nId. \ \text{updateF} \ f\ nId\ p\ ps\ t) \ 0\ ns) \ (\text{zip} \ ps\ nsbs)
$$

Figure 13: Our benchmarks expressed using our high-level algorithmic primitives. The operators (+) and (*) operate on a single pair instead of two scalar values.

8. Experimental Setup

8.1 Implementation Details

Our system is implemented in C++11 using the LLVM/Clang compiler infrastructure and making heavy use of C++ templates. Our primitives are expressed as C++ functions and expressions as compositions of those. When generating code two basic steps are performed: First, the Clang compiler library parses the input expression and produces 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 primitives.

As part of the first step, we have developed our own type system which plays a dual role. First, it prevents the user producing incorrect expressions. Secondly, the type system encodes information for code generation, such as the array size information 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 code analysis which makes our design very different compared to traditional optimizing compilers.

8.2 Hardware Platforms and Evaluation Methodology

The experiments were performed on three different hardware platforms: 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. The OpenCL runtimes from Nvidia (CUDA-SDK 5.5), AMD (AMD-APP 2.8.1), and Intel (XE 2013 R3) were used. The GPU drivers installed were 310.44 for Nvidia and 13.1 for AMD.

The profiling APIs from OpenCL and CUDA were used to measure kernel execution time and the gettimeofday function for the CPU implementation. Following the Nvidia benchmarking methodology [23], the data transfer time to and from the GPU is excluded from the results. Each experiment was repeated 1000 times and we report median runtimes.

The experiments were performed with multiple 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, an input matrix of 4096×4096 elements (64MB) and a vector size of 4096 elements (16KB) were used for the small input size. For the large input size, the matrix size was 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.
9. Results

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

9.1 Comparison vs. Portable Implementation

First, we show how our approach performs across three platforms. We use the cBLAS OpenCL implementations written by AMD as our baseline for this evaluation since it is inherently portable across all different platforms. Figure 14 shows the performance of our approach relative to cBLAS. We achieve better performance than cBLAS on most platforms and benchmarks. The speedups are highest for the CPU, with up to 20× for the `asum` benchmark with a small input size. The reason is that cBLAS was written and tuned specifically for an AMD GPU which usually exhibits a larger number of parallel processing units. As we saw in Section 6, our systematically derived expression for this benchmark is specifically tuned for the CPU by avoiding creating too much parallelism, whereas our implementation systematically and automatically generates these parallel kernels.

Figure 14 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 a 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 on par with the OpenCL implementation on all platforms.

9.2 Comparison vs. Highly-tuned Implementations

We compare our approach with a state of the art implementation for each platform. For Nvidia, we pick the highly tuned CUBLAS implementation of BLAS written by Nvidia. For the AMD GPU, we use the same cBlAS 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.

Similar to the high performance libraries our approach results in device-specific OpenCL code with implementation parameters tuned for specific data sizes. In contrast, existing library approaches are based on device-specific manually optimized implementations whereas our approach systematically and automatically generates these specialized versions.

Figure 15a 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 show that our technique is able to achieve high performance.

On the AMD GPU, we are surprisingly up to 4.5× faster than the cBLAS implementation on `gemv` small input size as shown in Figure 15b. The reason for this is found in the way cBLAS is implemented; cBLAS performs automatic code generation using fixed templates. In contrast to our approach, it only generates one implementation since it does not explore different template compositions.

For the Intel CPU (Figure 15c), our approach beats MKL for one benchmark and matches 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 a par with MKL for both benchmarks. The `asum` implementation in the MKL does not use thread level parallelism, whereas our implementation does; hence we achieve a speedup of up to 1.78 on the larger input size.

9.3 Summary

We have demonstrated that our approach generates performance portable code which is competitive with highly-tuned platform specific implementations. Our systematic approach is generic and generates optimized kernels for different devices and data sizes. The results show that high performance is achievable for different input sizes and for a range of benchmarks.
10. Related Work

Algorithmic Patterns   Algorithmic patterns (or algorithmic skeletons) [11] have been around for more than two decades. Early work already covers algorithmic skeletons in the context of performance portability [16]. Patterns are parts of popular frameworks such as Map-Reduce [18] from Google. Current pattern-based libraries for platforms ranging from cluster systems [37] to GPUs [41] have been proposed with recent extensions to irregular algorithms [20]. Lee et al. [28] discuss how nested parallel patterns can be mapped efficiently to GPUs. Compared to our approach, most prior work relies on hardware-specific implementations to achieve high performance. Conversely, we systematically generate implementations using fine-grain OpenCL patterns combined with rewrite rules.

Algebra of Programming   Bird and Meertens, amongst others, developed formalisms for algebraic reasoning about functional programs in the 1980s [5]. Our rewrite rules are in the same spirit and many of our rules are similar to equational rules presented by Bird, Meertens, and others. Skillicorn [38] describes the application of the algebraic approach for parallel computing. He argues that it leads to architecture-independent parallel programming — which we call performance portability in this paper. Our work can be seen as an application of the algebraic approach to the generation of efficient code for contemporary parallel processors.

Functional Approaches for GPU Code Generation   Accelerate is a Haskell embedded domain specific language aimed at generating efficient GPU code [9, 30]. Obsidian [42] and Harlan [24] are earlier projects with similar goals. Obsidian exposes more details of the underlying GPU hardware to the programmer. Harlan is a declarative programming language compiled to GPU code. Bergstrom and Reppy [4] compile NESt, which is a first-order dialect of ML supporting nested data-parallelism, to GPU code. Recently, Nvidia introduced NOVA [12], a new functional language targeted at code generation for GPUs, and Copperhead [7], a data parallel language embedded in Python. HiDP [46] is a hierarchical data parallel language which maps computations to OpenCL. All of these projects rely on code analysis 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.

Rewrite-rules for Optimizations   Rewrite rules have long been used as a way to automate the optimization process of functional programs [26]. Recently, rewriting has been applied to HPC applications [32] as well, where the rewrite process is driven by user annotations on imperative code. Spiral [34] uses rewrite rules to optimize signal processing programs and was more recently adapted to linear algebra [39]. One difference is that our rules and OpenCL hardware patterns are expressed at a finer-grained level, allowing for highly specialized and optimized code generation.

Automatic Code Generation for GPUs   A large body of work has explored how to generate high performance code for GPUs. Dataflow programming models such as StreamIt [43] and LiquidMetal [19] have been used to produce GPU code. Directive based approaches such as OpenMP to CUDA [29], OpenACC to OpenCL [36], and hiCUDA [22] compile sequential C code for the GPU. X10, a language for high performance computing, can also be used to program GPUs [14]. However, this 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 [44]. Delite [6, 8], a system that enables the creation of domain-specific languages, can also target multicore CPUs or GPUs. Alas, none of these approaches currently provides full performance portability, as they assume a fixed platform and the optimizations and implementations are targeted at a specific device.

Finally, Petabricks [3] takes an alternative approach by letting the programmer specify different implementations of an algorithm. The compiler and runtime choose the most suitable implementation based on an adaptive mechanism, and produces OpenCL code [33]. Compared to our work, this technique relies on static analysis to optimize code. Our code generator does not perform any analysis since optimization happens at a higher level within our rewrite rules.

11. 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 high-performance device-specific implementations. We presented a formalism, which we use to prove the correctness of the presented rewrite rules. Our approach results in a clear separation of concerns between high-level algorithmic concepts and low-level hardware optimizations which pave the way for fully automated high performance code generation.

To demonstrate our approach in practice, we have developed OpenCL-specific primitives and rules together with an OpenCL code generator. The design of the code generator is straightforward given that all optimization decisions are made with the rules and no complex analysis is needed. We achieve performance on a par with highly tuned platform-specific BLAS libraries on three different processors. For some benchmarks such as matrix vector multiplication we even reach a speedup of up to 4.5. We also show that our technique can be applied to more complex applications such as BlackScholes and molecular dynamics simulation.

Acknowledgments

This work was supported by a HiPEAC collaboration grant, EPRSRC (grant number EP/K034413/1), the Royal Academy of Engineering, Google and Oracle. We are grateful to the anonymous reviewers who helped to substantially improve the quality of the paper. We would like to thank Sergei Gorlatch for his support of the first author in the HiPEAC collaboration and the following people for their involvement in the discussions on formalization: Robert Atkey, James Cheney, Stefan Fehrenbach, Adam Harries, Shayan Najd, and Philip Wadler.

References


