Generating Performance Portable Code using Rewrite Rules

From High-level Functional Expressions to High-Performance OpenCL Code

Michel Steuwer
The University of Edinburgh, UK
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

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 performance and code portability. Typically, code is either tuned in an 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 formalise the 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 by describing a subset of the OpenCL programming model in a functional style and by implementing a compiler which generates high performance imperative OpenCL code. Our experiments show that we can automatically derive high-performance hardware-specific implementations from simple functional high-level algorithmic expressions. The performance of the generated OpenCL code is on a par with highly tuned implementations for multicore CPUs and GPUs written by experts.

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

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* usually 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 reduction 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 a tree-based reduction 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.

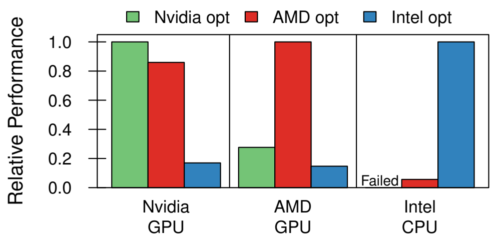


Figure 1: Performance is not portable across devices. Each bar represents the device-specific optimised implementation of a parallel reduction implemented in OpenCL and tuned for an Nvidia GPU, AMD GPU, and Intel CPU respectively. Performance is normalized with respect to the best implementation on each device.

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 sub-optimal 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× slower) when running on the CPU. The CPU optimized version, Intel_opt, achieves less than 20% (*i.e.*, 5× 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 [21] and LiquidMetal [15] allow the programmer to easily express different implementations at the algorithm level. Nvidia's NOVA [9] 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 [7] 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 reduction discussed earlier would be written in Accelerate as:

```
sum xs = let xs' = use xs
         in fold (+) 0 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 used to systematically derive semantically equivalent low-level expressions from high-level algorithms written by the programmer. Once derived, we can automatically generate high performance code based on these expressions.

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. They enable a clear separation of concerns between high-level algorithmic concepts and low-level hardware paradigms while using a unified framework. The rewrite rules define an implementation space that can be automatically searched to produce high performance code.

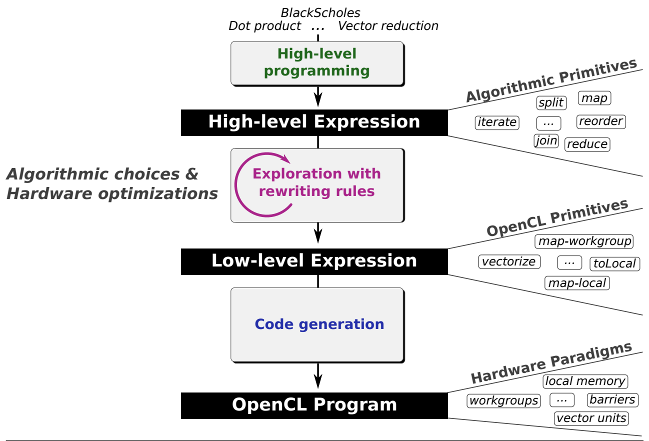


Figure 2: The programmer expresses the problem with high-level algorithmic primitives. These are systematically transformed into low-level primitives using a rule rewriting system. OpenCL code is generated by mapping the low-level primitives directly to the OpenCL programming model representing hardware paradigms.

This paper demonstrates that our approach yields high-performance code with OpenCL as our target hardware platform. We compare the performance of our approach with highly-tuned linear algebra functions extracted from the state-of-the-art libraries and with benchmarks such as BlackScholes. We express them as compositions of high-level algorithmic primitives which are systematically mapped to low-level OpenCL primitives from which OpenCL code is generated.

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;
- we develop a set of **rewrite rules** that systematically express algorithmic and optimization choices, bridging the gap between high-level functional programs and OpenCL;
- we prove the soundness of the rewrite rules with respect to the denotational semantics;
- we achieve **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

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.

$$\lambda\, xs\ .\ \mathit{map}\ (\lambda\, x\ .\ x * 3)\ xs$$

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

rewrite rules

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

(b) Low-level expression 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 3: Pseudo-code representing vector scaling. The user maps a function multiplying an element by 3 over the input array (a). This high-level expression is transformed into a low-level expression (b) using rewrite rules. Finally, our code generator turns the low-level expression into an OpenCL program (c).

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 our *map* primitive as shown in Figure 3a. Our expressions are glued together with lambda abstractions and functions composition; we formally define the syntax in Section 5.

Our technique first rewrites the high-level expression into a low-level expression closer to the OpenCL programming model. This is achieved by applying the rewrite rules presented later in Section 4. Figure 3b shows one possible derivation of the original high-level expression. Starting from the last line, the input (xs) is split into chunks of 1024 elements. Each chunk is mapped onto a group of threads, called *workgroup*, with the *mapWorkgroup* low-level primitive. Within a workgroup, we group 4 elements into a SIMD vector, each mapped to a local thread inside a workgroup via the *mapLocal* primitive. Finally, the *mapVec* primitive applies the vectorized form of the user defined function. The exact meaning of our primitives 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 primitive encountered (Figure 3c). The two map primitives 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 function (`mul3` 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 program representation. This is achieved by systematically transforming the high-level expression into a low-level form suitable for code generation. The next two sections present our high-level and low-level primitives, the code generation mechanism and the rewrite rules.

$$\begin{array}{rcl}
\mathit{map}_{A,B,I} & : & (A \to B) \to [A]_I \to [B]_I \\
\mathit{zip}_{A,B,I} & : & [A]_I \to [B]_I \to [A \times B]_I \\
\mathit{reduce}_{A,I} & : & ((A \times A) \to A) \to A \to [A]_I \to [A]_1 \\
\mathit{split}_{A,I} & : & n \to [A]_{n \times I} \to [[A]_n]_I \\
\mathit{join}_{A,I,J} & : & [[A]_I]_J \to [A]_{I \times J} \\
\mathit{iterate}_{A,I,J} & : & n \to (m \to [A]_{I \times m} \to [A]_m) \to \\
 & & \quad [A]_{I^n \times J} \to [A]_J \\
\mathit{reorder}_{A,I} & : & [A]_I \to [A]_I
\end{array}$$

Figure 4: High-level algorithmic primitives.

3. Algorithmic and OpenCL Primitives

A key idea of this paper is to expose algorithmic choices and hardware-specific program optimizations in a functional style. This allows for systematic transformations using a collection of rewrite rules (Section 4). The high-level algorithmic primitives can either be used by the programmer directly, as a stand-alone language (or embedded DSL), or be used as an intermediate representation targeted by another language. Once a program is represented by our high-level primitives, we can automatically transform it into low-level hardware primitives. These represent hardware-specific features in a programming model such as OpenCL, the target chosen for this paper. Following the same approach, a different set of low-level primitives might be designed to target other low-level programming models such as MPI.

In this section we give 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 [39] to our setting.

We let I range over sizes. A size can be a size variable m, n, a natural number i, or a product $I \times J$ or power I^J of sizes I and J. We let A, B range over types. We write $A \to B$ for a function from type A to type B and $n \to B$ 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 $\mathbf{1}$ 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 inferred.

3.1 Algorithmic Primitives

Similar to Accelerate we deliberately restrict ourselves to a set of primitives for which we know that high performance CPU and GPU implementations exist. 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 and performs a reduction returning the reduced element in an array of size 1. We assume the given reduction function is associative and commutative in order to admit efficient parallel implementations.

The *split* and *join* primitives transform the shape of array data. The expression *split* $n\ xs$ transforms array xs of size $n \times I$, with elements of type A, into an array of size I with elements that are A arrays of size n; *join* is the inverse of *split*. (In practice A itself may be an array type, in which case we can view *split* as adding a dimension to and *join* as subtracting a dimension from a matrix.)

$$\begin{aligned}
\textit{mapWorkgroup}_{A,B,I} &: (A \to B) \to [A]_I \to [B]_I \\
\textit{mapLocal}_{A,B,I} &: (A \to B) \to [A]_I \to [B]_I \\
\textit{mapGlobal}_{A,B,I} &: (A \to B) \to [A]_I \to [B]_I \\
\textit{mapSeq}_{A,B,I} &: (A \to B) \to [A]_I \to [B]_I \\
\textit{toLocal}_{A,I} &: (A \to B) \to (A \to B) \\
\textit{toGlobal}_{A,I} &: (A \to B) \to (A \to B) \\
\textit{reduceSeq}_{A,B,I} &: ((A \times B) \to A) \to A \to [B]_I \to [A]_1 \\
\textit{reducePart}_{A,I} &: ((A \times A) \to A) \to A \to n \to \\
&\qquad [A]_{I \times n} \to [A]_n \\
\textit{reorderStride}_{A,I} &: n \to [A]_{n \times I} \to [A]_{n \times I} \\
\textit{mapVec}_{A,B} &: (A \to B) \to \langle A \rangle_n \to \langle B \rangle_n \\
\textit{splitVec}_{A,I} &: n \to [A]_{n \times I} \to [\langle A \rangle_n]_I \\
\textit{joinVec}_{A,I,J} &: [\langle A \rangle_I]_J \to [A]_{I \times J}
\end{aligned}$$

Figure 5: Low-level OpenCL primitives used for code generation.

The *iterate* primitive repeatedly applies a given function. The expression *iterate* $n\, f$ applies the function f repeatedly n times. The type of *iterate* is instructive. The function f may change the length of the processed array at each iteration step. We currently restrict the length to stay the same or shrink in each iteration by a fixed factor (given by the implicit subscript I), which is sufficient to express, *e.g.*, iterative reduction (see Section 4). We intend to lift this restriction in the future, which will probably require a richer type system. Given n the type of *iterate* expresses that the input array will shrink by a factor of I^n.

Finally, the *reorder* primitive allows the programmer to express that the order of elements in an array is unimportant, allowing a number of useful optimisations—as we will see in Section 4. This primitive bares obvious similarities to the *unordered* operation of the Ferry query language [17], which asserts that the order of elements in a list is unimportant.

3.2 OpenCL-specific Primitives

In order to achieve high performance on manycore CPUs and GPUs, programmers often use a set of rules of thumb to drive the optimization of their application. Each hardware vendor provides optimization guides [1, 27] that extensively cover hardware idiosynchrasies and optimizations. The main idea behind our work is to identify common optimization patterns and express them with the help of low-level primitives coupled with a rewrite system. Figure 5 lists the OpenCL-specific primitives we have identified.

Maps Each *mapX* primitive has the same high-level semantics as plain *map*, but represents a specific way of mapping computations to the hardware and exploiting parallelism in OpenCL. The *mapWorkgroup* primitive assigns work to a group of threads, called *workgroup* in OpenCL, with every workgroup applying the given function on an element of the input array. Similarly, the *mapLocal* primitive assigns work to a local thread inside a workgroup. As workgroups are optional in OpenCL *mapGlobal* assigns work to a thread not organized in a workgroup. This allows us to map computations in different ways to the thread hierarchy. The *mapSeq* primitive performs a sequential map within a single thread.

Generating OpenCL code for all of these primitives is similar; we describe this using *mapWorkgroup* 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.

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 the *toLocal* function. In the code generator, this will simply change the output pointer of function f to an area in local memory.

Sequential Reduction The *reduceSeq* primitive performs a sequential reduction 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. Therefore, *reduceSeq* has a more general type.

Partial Reduction The *reducePart* primitive performs a partial reduction, *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 reduction as we will see in the next section.

Reorder Stride The high-level semantics of $\textit{reorderStride}_{A,I}\ n$ is just like $\textit{reorder}_{A,I}$. 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 $i/I + i\%I$. 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.

At a high-level, vectors are just a special case of arrays. We write $\langle A \rangle_I$ 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 implementation can only vectorize functions containing simple arithmetic operations such as $+$ or $-$. For more complex functions, we rely on external tools [23] for vectorizing the operations, without performing further analysis.

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

(a) Iterate decomposition

$$\begin{array}{lcl} \mathit{map}\ f \circ \mathit{reorder} & \rightarrow & \mathit{reorder} \circ \mathit{map}\ f \\ \mathit{reorder} \circ \mathit{map}\ f & \rightarrow & \mathit{map}\ f \circ \mathit{reorder} \end{array}$$

(b) Reorder commutativity

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

(c) Split-join

$$\begin{array}{lcl} \mathit{reduce}\ f\ z & \rightarrow & \mathit{reduce}\ f\ z \circ \mathit{reducePart}\ f\ z\ m \\ \mathit{reducePart}\ f\ z\ m & \rightarrow & \mathit{reduce}\ f\ z \\ & \mid & \mathit{reducePart}\ f\ z\ m \circ \mathit{reorder} \\ & \mid & \textbf{\textit{join}} \circ \mathit{map}\ (\mathit{reducePart}\ f\ z\ m) \circ \textbf{\textit{split}}\ n \\ & \mid & \textbf{\textit{iterate}}\ n\ (\mathit{reducePart}\ f\ z) \end{array}$$

(d) Reduction

$$\begin{array}{lcl} \textbf{\textit{join}} \circ \textbf{\textit{split}}\ n & \rightarrow & \mathit{id} \\ \textbf{\textit{joinVec}} \circ \textbf{\textit{splitVec}}\ n & \rightarrow & \mathit{id} \end{array}$$

(e) Simplification rules

$$\begin{array}{lcl} \mathit{map}\ f \circ \mathit{map}\ g & \rightarrow & \mathit{map}\ (f \circ g) \\ \textbf{\textit{reduceSeq}}\ f\ z \circ \textbf{\textit{mapSeq}}\ g & \rightarrow & \\ \multicolumn{3}{r}{\textbf{\textit{reduceSeq}}\ (\lambda\ acc,x\ .\ f\ acc\ (g\ x))\ z} \end{array}$$

(f) Fusion rules

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

$$\begin{array}{lcl} \mathit{map}\ f & \rightarrow & \textbf{\textit{mapWorkgroup}}\ f \mid \textbf{\textit{mapLocal}}\ f \\ & \mid & \textbf{\textit{mapGlobal}}\ f \mid \textbf{\textit{mapSeq}}\ f \end{array}$$

(a) Map

$$\mathit{reduce}\ f\ z \ \rightarrow\ \textbf{\textit{reduceSeq}}\ f\ z$$

(b) Reduction

$$\mathit{reorder} \ \rightarrow\ \textbf{\textit{reorderStride}}\ n \mid \mathit{id}$$

(c) Stride accesses or normal accesses

$$\begin{array}{lcl} \textbf{\textit{mapLocal}}\ f & \rightarrow & \textbf{\textit{toGlobal}}\ (\textbf{\textit{mapLocal}}\ f) \\ \textbf{\textit{mapLocal}}\ f & \rightarrow & \textbf{\textit{toLocal}}\ (\textbf{\textit{mapLocal}}\ f) \end{array}$$

(d) Local/Global memory

$$\mathit{map}\ f \ \rightarrow\ \textbf{\textit{joinVec}} \circ \mathit{map}\ (\textbf{\textit{mapVec}}\ f) \circ \textbf{\textit{splitVec}}\ n$$

(e) Vectorization

Figure 7: OpenCL-specific rules. Bold functions are known to the code generator.

4. Rewrite Rules

This section presents our rewrite rules, which transform high-level expressions written using the algorithmic primitives 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 vectorization. This contrasts with many prior approaches that provide special vectorized versions of different algorithmic primitives such as map and reduce. By the power of composition 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.

As with the primitives, we distinguish between algorithmic and low-level rules. Algorithmic rules produce derivations that represent the different algorithmic choices and are shown in Figure 6. Figure 7 shows our OpenCL-specific rules which map expressions to OpenCL patterns. Once an expression is in its lowest-level 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 6a expresses the fact that an iteration can be decomposed into several iterations.

Reorder commutativity 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 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.

Reduction The reduction (and associated partial reduction) in Figure 6d 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. The reduction can be derived in a partial reduction combined with a full reduction which ensures we end up with one unique element. The first possible derivation for partial reduction leads to the full reduction. The next possible derivation expresses the fact that it is possible to reorder the elements to be reduced, exploiting the commutativity property which we require from the given reduction function. 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. This exploits the associativity property we require from the reduction function. 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 commonly implemented on a GPU (iteratively reducing within a workgroup using the local memory).

Simplification Rules Figure 6e shows our simplification 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 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 different functions and avoid having to store temporary results. The functional programming community has studied more sophisticated and generic rules for fusion [10, 22, 26]. 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.

Maps The rule in Figure 7a is used to produce OpenCL-specific map implementations that match the OpenCL thread hierarchy. Our implementation maintains context information (not shown for space reason) to ensure the OpenCL thread hierarchy is respected. For instance, it is only legal to nest a *mapLocal* inside a *mapWorkgroup*.

Reduction There is only one low-level rule for reduction (Figure 7b), which expresses the fact that the only implementation known to the code generator is a sequential reduction. Parallel implementations are defined at a higher level by composition of other algorithmic primitives. Most existing approaches treat the reduction directly as an irreducible primitive operation. With our approach it is possible to explore different implementations for the reduction by simply applying different rules.

Reorder Figure 7c 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* function, and *reorderStride*, which reorders elements with a certain stride n. As described earlier, the major use case for the stride reorder is to enable coalesced memory accesses.

Local/Global Figure 7d shows two rules that enable GPU local memory usage. They express the fact that the result of a *mapLocal* can always be stored in local memory or back in global memory. This holds since a *mapLocal* always exists within a *mapWorkgroup* for which the local memory is defined. These rules allow us to determine how the data is mapped to the GPU memory hierarchy and encode the common optimization to load frequently used data from the slow global into the fast local memory. The search strategy, discussed in Section 6, applies this rule to explore opportunities for this optimization.

Vectorization Figure 7e shows the vectorization rule. SIMD vectorization is a key aspect of modern hardware architectures. In our approach vectorization is achieved by using the *splitVec* and corresponding *joinVec* primitives, 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.

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

$$\boxed{\Delta \vdash I}$$

$$\textsc{IVar}\ \frac{n \in \Delta}{\Delta \vdash n} \qquad \textsc{INat}\ \frac{}{\Delta \vdash i} \qquad \textsc{ITimes}\ \frac{\Delta \vdash I \qquad \Delta \vdash J}{\Delta \vdash I \times J}$$

$$\textsc{IPower}\ \frac{\Delta \vdash I \qquad \Delta \vdash J}{\Delta \vdash I^J}$$

$$\boxed{\Delta \vdash A}$$

$$\textsc{TInt}\ \frac{}{\Delta \vdash \mathbf{int}} \qquad \textsc{TFloat}\ \frac{}{\Delta \vdash \mathbf{float}} \qquad \textsc{TUnit}\ \frac{}{\Delta \vdash \mathbf{1}}$$

$$\textsc{TProduct}\ \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \times B} \qquad \textsc{TFun}\ \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}$$

$$\textsc{TFunI}\ \frac{\Delta, n \vdash B}{\Delta \vdash n \to B} \qquad \textsc{TArray}\ \frac{\Delta \vdash A \qquad \Delta \vdash I}{\Delta \vdash [A]_I}$$

$$\boxed{\Delta; \Gamma \vdash M : A}$$

$$\textsc{Const}\ \frac{c : A \in \Gamma}{\Delta; \Gamma \vdash c : A} \qquad \textsc{Var}\ \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}$$

$$\textsc{Unit}\ \frac{}{\Delta; \Gamma \vdash () : \mathbf{1}} \qquad \textsc{Pair}\ \frac{\Delta; \Gamma \vdash M : A \qquad \Delta; \Gamma \vdash N : B}{\Delta; \Gamma \vdash (M, N) : A \times B}$$

$$\textsc{Lam}\ \frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda x^A.M : A \to B}$$

$$\textsc{App}\ \frac{\Delta; \Gamma \vdash M : A \to B \qquad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M\, N : B}$$

$$\textsc{LamI}\ \frac{\Delta, n; \Gamma \vdash M : B}{\Delta; \Gamma \vdash \lambda n.M : n \to B}$$

$$\textsc{AppI}\ \frac{\Delta; \Gamma \vdash M : n \to B \qquad \Delta \vdash I}{\Delta; \Gamma \vdash M\, I : B}$$

Figure 8: Typing Rules for the Core Language

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 $\mathbb{F}$ 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, $X \to Y$ denotes the set of functions from the set X to the set Y. Size-dependent functions are interpreted as size-dependent func-

Sizes

$$\begin{aligned}
\llbracket n\rrbracket_\iota &= \iota(n)\\
\llbracket i\rrbracket_\iota &= i\\
\llbracket I\times J\rrbracket_\iota &= \llbracket I\rrbracket_\iota \times \llbracket J\rrbracket_\iota\\
\llbracket I^J\rrbracket_\iota &= \llbracket I\rrbracket_\iota^{\llbracket I\rrbracket_\iota}
\end{aligned}$$

Types

$$\begin{aligned}
\llbracket \mathbf{int}\rrbracket_\iota &= \mathbb{Z}\\
\llbracket \mathbf{float}\rrbracket_\iota &= \mathbb{F}\\
\llbracket \mathbf{1}\rrbracket_\iota &= \mathbf{1}\\
\llbracket A\times B\rrbracket_\iota &= \llbracket A\rrbracket_\iota \times \llbracket B\rrbracket_\iota\\
\llbracket A\to B\rrbracket_\iota &= \llbracket A\rrbracket_\iota \to \llbracket B\rrbracket_\iota\\
\llbracket n\to B\rrbracket_\iota &= (i:\mathbb{N}) \to \llbracket B\rrbracket_{\iota[n\mapsto i]}\\
\llbracket [A]_I\rrbracket_\iota &= [0..\llbracket I\rrbracket_\iota) \to \llbracket A\rrbracket_\iota
\end{aligned}$$

Size environments

$$\begin{aligned}
\llbracket \cdot\rrbracket &= \emptyset\\
\llbracket \Delta, n\rrbracket &= \llbracket \Delta\rrbracket[n\mapsto \mathbb{N}]
\end{aligned}$$

Type environments

$$\begin{aligned}
\llbracket \cdot\rrbracket_\iota &= \emptyset\\
\llbracket \Gamma, x:A\rrbracket_\iota &= \llbracket \Gamma\rrbracket[x\mapsto \llbracket A\rrbracket_\iota]
\end{aligned}$$

Terms

$$\begin{aligned}
\llbracket x\rrbracket_{\iota,\rho} &= \rho(x)\\
\llbracket ()\rrbracket_{\iota,\rho} &= ()\\
\llbracket (M,N)\rrbracket_{\iota,\rho} &= (\llbracket M\rrbracket_{\iota,\rho}, \llbracket N\rrbracket_{\iota,\rho})\\
\llbracket \lambda x^A.M\rrbracket_{\iota,\rho} &= \lambda v.\llbracket M\rrbracket_{\iota,\rho[x\mapsto v]}\\
\llbracket M\,N\rrbracket_{\iota,\rho} &= \llbracket M\rrbracket_{\iota,\rho}(\llbracket N\rrbracket_{\iota,\rho})\\
\llbracket \lambda n.M\rrbracket_{\iota,\rho} &= \lambda i.\llbracket M\rrbracket_{\iota[n\mapsto i],\rho}\\
\llbracket M\,I\rrbracket_{\iota,\rho} &= \llbracket M\rrbracket_{\iota,\rho}(\llbracket I\rrbracket_\iota)
\end{aligned}$$

Primitives

$$\begin{aligned}
\llbracket \mathit{map}_{A,B,I}\rrbracket_{\iota,\rho} &= \lambda f\,x\,i.f\,x\,i\\
\llbracket \mathit{reduce}_{A,I}\rrbracket_{\iota,\rho} &= \lambda(\oplus)\,e\,x\,i.\\
&\quad x(0)\oplus(x(1)\oplus(\ldots\oplus(x(\llbracket I\rrbracket_\iota-1)\oplus e)\ldots))\\
\llbracket \mathit{zip}_{A,B,I}\rrbracket_{\iota,\rho} &= \lambda x\,y\,i.(x(i),y(i))\\
\llbracket \mathit{split}_{A,I,J}\rrbracket_{\iota,\rho} &= \lambda n\,x\,i\,j.x((i\times n)+j)\\
\llbracket \mathit{join}_{A,I,J}\rrbracket_{\iota,\rho} &= \lambda x\,i.x(i/\llbracket I\rrbracket_\iota)(i\%\llbracket I\rrbracket_\iota)\\
\llbracket \mathit{iterate}_{A,I,J}\rrbracket_{\iota,\rho} &= \lambda n\,f\,x\,i.f(i_1)\,(f(i_2)(\ldots\,(f(i_n)\,x\,i)\ldots))\\
&\text{where } i_j = \llbracket I\rrbracket_\iota^{n-j}\times\llbracket J\rrbracket_\iota
\end{aligned}$$

Figure 9: Denotational Semantics for the Core Language

tions in the meta language. Arrays are interpreted in the obvious way as functions from sizes to elements.

Size environments are interpreted as *size maps*, partial maps from size variables to natural numbers. Type environments are interpreted as *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 ι from size variables to natural numbers (that is, the interpretation of a size environment). Similarly, terms are interpreted with respect to a partial map ρ from term variables to values. We overload λ-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 *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:

$$M\circ N \equiv \lambda x.M\,(N\,x) \qquad f\circ g\equiv \lambda v.f(g(v))$$

Theorem 1 (Type soundness)**.**

$$\Delta;\Gamma\vdash M:A \Rightarrow \llbracket M\rrbracket_{\llbracket\Delta\rrbracket,(\llbracket\Gamma\rrbracket_{\llbracket\Delta\rrbracket})}\in\llbracket A\rrbracket_{\llbracket\Delta\rrbracket}$$

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 *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:

$$\begin{aligned}
&\llbracket \mathit{mapWorkgroup}\rrbracket = \llbracket \mathit{mapLocal}\rrbracket = \llbracket \mathit{mapGlobal}\rrbracket =\\
&\quad\llbracket \mathit{mapSeq}\rrbracket = \llbracket \mathit{mapWorkgroup}\rrbracket = \llbracket \mathit{mapVec}\rrbracket = \llbracket \mathit{map}\rrbracket
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathit{reduceSeq}\rrbracket &= \llbracket \mathit{reduce}\rrbracket\\
\llbracket \mathit{toLocal}\rrbracket &= \llbracket \mathit{toGlobal}\rrbracket = \lambda x.x\\
\llbracket \mathit{splitVec}\rrbracket &= \llbracket \mathit{split}\rrbracket\\
\llbracket \mathit{joinVec}\rrbracket &= \llbracket \mathit{join}\rrbracket
\end{aligned}$$

The semantics of the remaining two primitives is as follows.

$$\begin{aligned}
\llbracket \mathit{reducePart}_{A,I}\rrbracket_{\iota,\rho} &= \lambda(\oplus)\,e\,n\,x\,i.\\
&x(j)\oplus(x(j+1)\oplus(\ldots\oplus(x(j+\llbracket I\rrbracket_\iota-1)\oplus e)\ldots))\\
&\text{where } j = i\times\llbracket I\rrbracket_\iota\\
\llbracket \mathit{reorderStride}_{A,I}\rrbracket_{\iota,\rho} &= \lambda n\,x\,i.x(i/\llbracket I\rrbracket_\iota + n\times(i\%\llbracket I\rrbracket_\iota))
\end{aligned}$$

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:

$$\begin{aligned}
&\llbracket \mathit{join}\circ \mathit{map}\,(\mathit{map}\,f)\circ \mathit{split}\,n\rrbracket_{\iota,\rho}\\
=\ &(\text{definition of } \llbracket - \rrbracket \text{ and } \circ)\\
&\lambda x\,i.x(i/\llbracket I\rrbracket_\iota)(i\%\llbracket I\rrbracket_\iota)\,\big(\\
&\quad\lambda f\,x\,i.f(x(i))\,(\lambda x\,i.(\rho(f))(x(i)))\,(\lambda x\,i\,j.x((i\times\iota(n))+j))\big)\\
=\ &(\beta\text{-reduction})\\
&\lambda x.(\lambda i\,j.(\rho(f))(x((i\times\iota(n))+j))\,(i/\llbracket I\rrbracket_\iota)\,(i\%\llbracket I\rrbracket_\iota))\\
=\ &(\beta\text{-reduction})\\
&\lambda x\,i.(\rho(f))(x(((i/\llbracket I\rrbracket_\iota)\times\iota(n))+(i\%\llbracket I\rrbracket_\iota)))\\
=\ &(i<\llbracket I\rrbracket_\iota)\\
&\lambda x\,i.(\rho(f))(x(i))\\
=\ &(\text{definition of } \llbracket - \rrbracket)\\
&\llbracket \mathit{map}\,f\rrbracket_{\iota,\rho}
\end{aligned}$$

5.4 Applying 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 patterns into one, thus, avoiding intermediate results. Figure 10 shows how we can derive a fused version for calculating the sum of absolute value, *asum*, from the high-level expression written by the programmer. We write the derivation as a sequence of equations. The numbers above the equality sign refer to the rules applied.

We start by applying the reduction rule 6d twice: first to replace *reduce* with *reduce* ∘ *part-red* and then a second time to expand *par-red*. To get (2) we expand *map*, which can be simplified by removing the two corresponding *join* and *split* patterns. In the step from (3) to (4) two *map* patterns are fused and in the next step the nested *map* is transformed into the *map-seq* pattern to obtain (5). By first transforming *part-red* back into *reduce* (using rule 6d) and then applying the rule 7b we get (6). Finally, we apply rule 6f

$$\begin{aligned}
\mathit{asum} = \ & \mathit{reduce}\ (+)\ 0 \circ \mathit{map}\ \mathit{abs} \overset{6d}{=} \mathit{reduce}\ (+)\ 0 \circ \mathit{join} \circ \mathit{map}\ (\mathit{reducePart}\ (+)\ 0) \circ \mathit{split}\ n \circ \mathit{map}\ \mathit{abs} && (1)\\
\overset{6c}{=} \ & \mathit{reduce}\ (+)\ 0 \circ \mathit{join} \circ \mathit{map}\ (\mathit{reducePart}\ (+)\ 0) \circ \mathit{split}\ n \circ \mathit{join} \circ \mathit{map}\ (\mathit{map}\ \mathit{abs}) \circ \mathit{split}\ n && (2)\\
\overset{6e}{=} \ & \mathit{reduce}\ (+)\ 0 \circ \mathit{join} \circ \mathit{map}\ (\mathit{reducePart}\ (+)\ 0) \circ \mathit{map}\ (\mathit{map}\ \mathit{abs}) \circ \mathit{split}\ n && (3)\\
\overset{6f}{=} \ & \mathit{reduce}\ (+)\ 0 \circ \mathit{join} \circ \mathit{map}\ (\mathit{reducePart}\ (+)\ 0 \circ \mathit{map}\ \mathit{abs}) \circ \mathit{split}\ n && (4)\\
\overset{7a}{=} \ & \mathit{reduce}\ (+)\ 0 \circ \mathit{join} \circ \mathit{map}\ (\mathit{reducePart}\ (+)\ 0 \circ \mathit{mapSeq}\ \mathit{abs}) \circ \mathit{split}\ n && (5)\\
\overset{6d\&7b}{=} \ & \mathit{reduce}\ (+)\ 0 \circ \mathit{join} \circ \mathit{map}\ (\mathit{reduceSeq}\ (+)\ 0 \circ \mathit{mapSeq}\ \mathit{abs}) \circ \mathit{split}\ n && (6)\\
\overset{6f}{=} \ & \mathit{reduce}\ (+)\ 0 \circ \mathit{join} \circ \mathit{map}\ (\mathit{reduceSeq}\ (\lambda\ acc, a\ .\ acc + (\mathit{abs}\ a))\ 0) \circ \mathit{split}\ n && (7)
\end{aligned}$$

Figure 10: Derivation for *asum* to a fused parallel version. The numbers refer to the rules from Figure 6 and Figure 7.

to fuse the *map-seq* and *reduce-seq* into a single *reduce-seq*. This sequence of transformations results in expression (7), which allows for a more optimal implementation since no temporary storage is required for the intermediate result.

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 expression for a given target device, we have developed a simple automatic search strategy based loosely on Bandit-based optimization [13]. Our current search strategy is very 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 speedup the search process, we added *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 expression 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 on two GPUs and one CPU (described later in Section 8). We can make several important observations. First, in all the expressions the fusion macro rule merging *map* and *reduce* was applied. The second observation is, that none of the versions make use of the local memory (although our systems fully support it). It is common wisdom that using local memory on the GPU enables high performance and in fact the highly tuned hand-written implementation of *asum* use local memory on the GPU. However, as we will see later in the results section, our automatically derived version is able to perform as well without using local memory. The third key observation is, that each thread performs a large sequential reduction independent of all other threads, which does not require thread synchronization, avoiding overheads.

While these observations are the same for all platforms, there are also crucial differences between the different low-level expressions. Both GPU versions make use of the *reorderStride* primitive, allowing for coalesced memory accesses. The AMD and Intel versions are vectorized with a vector length of two and four respectively. The Nvidia version does not use vectorization since this platform does not benefit from vectorized code. On the CPU, the automatic search picked numbers for partitioning into work groups and then into work items in such a way that inside each work group only a single work item is active. This corresponds to the fact that there is less parallelism available on a CPU compared to GPUs.

Interestingly, the results of our search have some artifacts in the expressions. For example, we perform unnecessary copies on AMD and Intel by performing a *mapSeq* with the identity nested inside. While this does not seem to affect performance much, a better search strategy could probably get rid of these artifacts and achieve a slightly better performance.

6.3 Search Efficiency

We now present some evidence that our search strategy is effective at finding good derivations. 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 later in Section 9, here we want to focus on the efficiency of the search. The evaluated expressions are grouped from left to right by the number of fixed derivations in the search tree. The red line connects the fastest expression found so far.

As can be seen the performance improves steadily for all three platforms before reaching a plateau. For both GPUs the best performance is reached after testing ≈ 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 GPU for selecting appropriate numerical parameters. On the CPU performance converges quicker and more expressions offer good performance. This shows that the CPU is easier to optimize for and not as sensitive when selecting numerical parameters.

(a) Nvidia GPU

$$\begin{aligned}&\lambda x.(\mathit{reduceSeq} \circ \mathit{join} \circ \mathit{join} \circ \mathit{mapWorkgroup}\ (\\ &\quad \mathit{toGlobal}\ \big(\mathit{mapLocal}\ (\mathit{reduceSeq}\ (\lambda a\, b.\ a + (\mathit{abs}\ b))\ 0)\big) \circ \mathit{reorderStride}\ 2048\\ &)\circ \mathit{split}\ 128 \circ \mathit{split}\ 2048)\ x\end{aligned}$$

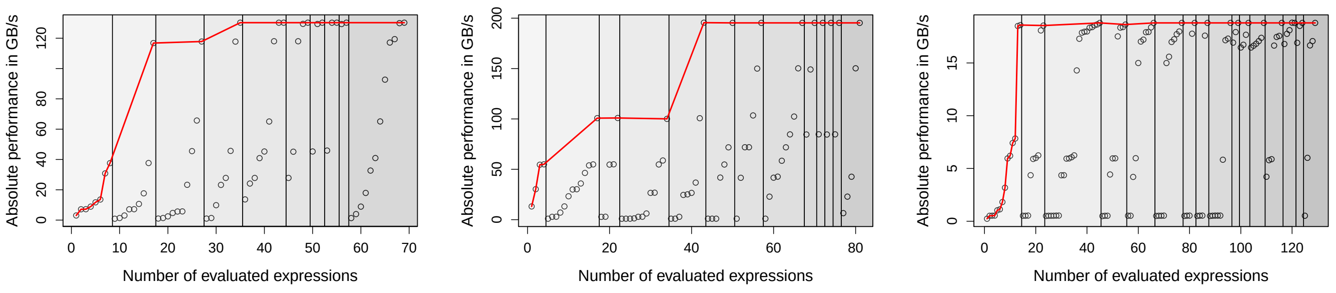
(b) AMD GPU

$$\begin{aligned}&\lambda x.(\mathit{reduceSeq} \circ \mathit{join} \circ \mathit{joinVec} \circ \mathit{join} \circ \mathit{mapWorkgroup}\ (\\ &\quad \mathit{mapLocal}\ (\mathit{mapSeq}\ (\mathit{mapVec}\ 2\ \mathit{id}) \circ \mathit{reduceSeq}\ (\mathit{mapVec}\ 2\ (\lambda a\, b.\ a + (\mathit{abs}\ b)))\ 0 \circ \mathit{reorderStride}\ 2048\\ &)\circ \mathit{split}\ 128 \circ \mathit{splitVec}\ 2 \circ \mathit{split}\ 4096)\ x\end{aligned}$$

(c) Intel CPU

$$\begin{aligned}&\lambda x.(\mathit{reduceSeq} \circ \mathit{join} \circ \mathit{mapWorkgroup}\ (\mathit{join} \circ \mathit{joinVec} \circ \mathit{mapLocal}\ (\\ &\quad \mathit{mapSeq}\ (\mathit{mapVec}\ 4\ \mathit{id}) \circ \mathit{reduceSeq}\ (\mathit{mapVec}\ 4\ (\lambda a\, b.\ a + (\mathit{abs}\ b)))\ 0\\ &)\circ \mathit{splitVec}\ 4 \circ \mathit{split}\ 32768) \circ \mathit{split}\ 32768)\ x\end{aligned}$$

Figure 11: Low-level expressions performing the sum of absolute values. These expressions are automatically derived by our system from the high-level expression $\mathit{asum} = \mathit{reduce}\ (+)\ 0\ \circ \mathit{map}\ \mathit{abs}$.



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

Figure 12: Search efficiency. The vertical partitioning represents the number of fixed derivations in the search tree. The red line connects the fastest expressions found so far.

7. Benchmarks

We now discuss how applications can be represented as expressions composed of our high-level algorithmic primitives.

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 the addition. For dot product, a pair-wise multiplication of its two input vectors is performed before applying the reduction expressed using the *zip* and *map* primitives.

gemv shows matrix vector multiplication as defined in BLAS: $\vec{y} = \alpha A\vec{x} + \beta\vec{y}$. To multiply matrix A with $\vec{x}$, the *map* primitive maps the computation of the dot-product with the input vector $\vec{x}$ to each row of the matrix A. Notice how we are reusing the high-level expressions for dot-product. Expressions describing algorithmic concepts can be reused, without committing to a particular low-level implementation. The dot-product from *gemv* might be implemented in a completely different way from the 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.

7.3 Physics Application

Another application we consider is the molecular dynamics (*md*) application from the SHOC benchmark suite. 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 $\mathit{updateF}$ 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 the *reduce* primitive on vector ns storing the neighboring indices and using $\mathit{updateF}$ as reduction operation.

$$
\begin{aligned}
scal &= \lambda a\; xs.\; map\; (\lambda x. a * x)\; xs \\
asum &= \lambda xs.\; (reduce\; (+)\; 0\; \circ map\; abs)\; xs \\
dot &= \lambda xs\; ys.\; (reduce\; (+)\; 0\; \circ map\; (*))\; zip\; xs\; ys \\
gemv &= \lambda mat\; xs\; ys\; \alpha\; \beta. \\
&\qquad map\; (+)\; \big(zip\; (map\; (scal\; \alpha \circ dot\; xs)\; mat) \\
&\qquad\qquad (scal\; \beta\; ys)\big)
\end{aligned}
$$

$$
blackScholes = map\; compCallPut
$$

$$
\begin{aligned}
md &= \lambda ps\; nbhs\; t.\; map\; \big(\lambda(p, ns). \\
&\qquad reduce\; (\lambda f\; nId.\; updateF\; f\; nId\; p\; ps\; t)\; 0\; ns\big) \\
&\qquad (zip\; ps\; nbhs)
\end{aligned}
$$

Figure 13: Our benchmarks expressed using our high-level algorithmic primitives.

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

We used three 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. We used the OpenCL runtimes from Nvidia (CUDA-SDK 5.5), AMD (AMD-APP 2.8.1), and Intel (XE 2013 R3). The GPU drivers installed 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 [19], the data transfer time to and from the GPU is excluded. We repeat each experiment 1000 times and report median runtimes.

We have performed experiments 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*, 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.

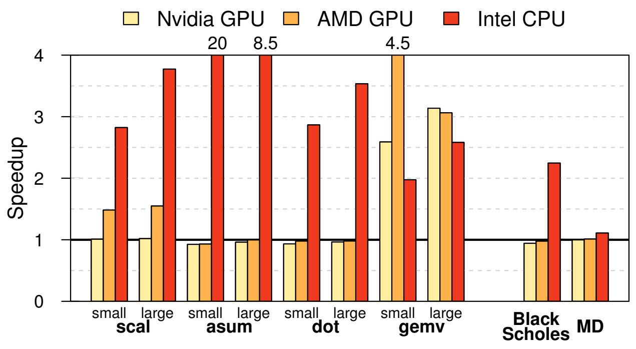


Figure 14: Performance of our approach relative to a portable OpenCL reference implementation (clBLAS). We outperform it on most benchmarks and platforms.

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 clBLAS 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 clBLAS. As can be seen, we achieve better performance than clBLAS on most platforms and benchmarks. The speedups are the highest for 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, 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.

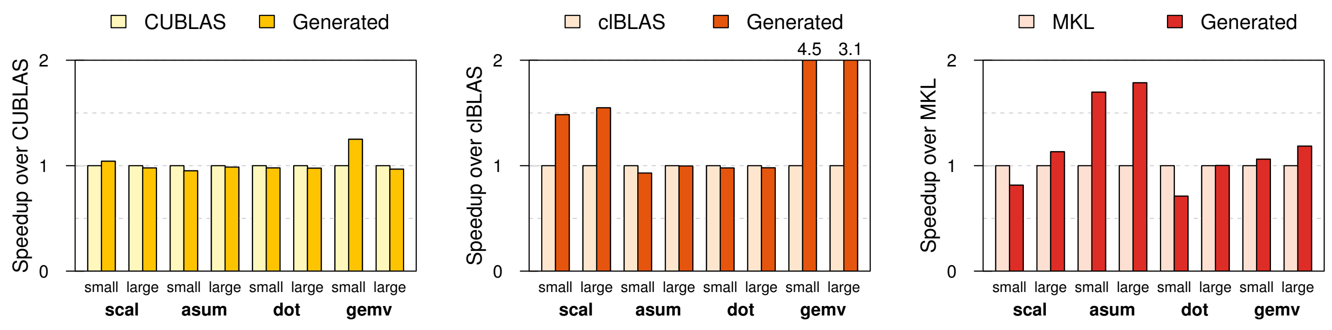
Figure 14 also shows the results we obtain relative to the Nvidia SDK *BlackScholes* and SHOC molecular dynamics *MD* benchmark. For *BlackScholes*, we see that our approach is on par with the performance of the Nvidia implementation on both GPUs. On the CPU, we actually achieve a 2.2× speedup due to 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 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 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 clBLAS implementation on *gemv* small input size as shown in Figure 15b. The reason for this is found in the way clBLAS is



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

Figure 15: Performance comparison with state of the art platform-specific libraries; CUBLAS for Nvidia, clBLAS for AMD, MKL for Intel. Our approach matches the performance on all three platforms and outperforms clBLAS in some cases.

implemented; clBLAS performs automatic code generation using fixed templates. In contrast to our approach, they only generate one implementation since they do 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 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 that our approach generates *performance portable* code which is competitive with highly-tuned platform specific implementations.

10. Related Work

Algorithmic Patterns Algorithmic pattern (or algorithmic skeletons [8]) have been around for more than two decades. Early work already discussed algorithm skeletons in the context of performance portability [12]. Patterns are parts of popular frameworks such as Map-Reduce [14] from Google. Current pattern-based libraries for platforms ranging from cluster systems [33] to GPUs [35] have been proposed with recent extension to irregular algorithms [16]. Lee et al., [24] discuss how nested parallel patterns can be mapped efficiently to GPUs. 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 embedded into Haskell to support GPU acceleration [7, 26]. Obsidian [36] and Harlan [20] 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 [3] compile NESL, which is a first-order dialect of ML supporting nested data-parallelism, to GPU code. Recently, Nvidia introduced NOVA [9], a new functional language targeted at code generation for GPUs, and Copperhead [5], a data parallel language embedded in Python. HiDP [40] is a hierarchical data parallel language which maps computations to OpenCL. All 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.

Halide [31] is a domain specific approach that targets image processing pipelines. It separates the algorithmic description from optimization decisions. Our work is domain agnostic and takes a different approach. 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 as a way to automate the optimization process of functional programs [22]. Recently, rewriting has been applied to HPC applications [28] as well, where the rewrite process uses user annotations on imperative code. Similar to us, Spiral [30] uses rewrite rules to optimize signal processing programs and was more recently adapted to linear algebra [34]. In contrast, our rules and OpenCL hardware patterns are expressed at a much finer 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 [37] or LiquidMetal [15] have been used to produce GPU code. Directive based approaches such as OpenMP to CUDA [25], OpenACC to OpenCL [32], or hiCUDA [18] compile sequential C code for the GPU. X10, a language for high performance computing, can also be used to program GPUs [11]. 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 [38]. 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 choose the most suitable one based on an adaptive mechanism and produces OpenCL code [29]. 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 optimizations decisions are made with the rules and no complex analysis is needed. We achieve performance on 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 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. PLDI. ACM, 2009.

[3] L. Bergstrom and J. H. Reppy. Nested data-parallelism on the GPU. ICFP. ACM, 2012.

[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. PACT. ACM, 2011.

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

[7] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. DAMP. ACM, 2011.

[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. ARRAY. ACM, 2014.

[10] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. ICFP. ACM, 2007.

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

[12] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. PARLE. Springer, 1993.

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

[14] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communication of the ACM*, 51(1), 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). PLDI. ACM, 2012.

[16] C. H. González and B. B. Fraguela. An algorithm template for domain-based parallel irregular algorithms. *International Journal of Parallel Programming*, 42(6):948–967, 2014.

[17] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. SIGMOD. ACM, 2009.

[18] T. D. Han and T. S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1), Jan. 2011.

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

[20] E. Holk, W. E. Byrd, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Declarative parallel programming for GPUs. PARCO. IOS Press, 2011.

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

[22] 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.

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

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

[25] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. PPoPP. ACM, 2009.

[26] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. ICFP. ACM, 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. ICPP. IEEE, 2012.

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

[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. *IEEE special issue on "Program Generation, Optimization, and Adaptation"*, 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. PLDI. ACM, 2013.

[32] R. Reyes, I. López-Rodríguez, J. Fumero, and F. de Sande. accULL: an OpenACC implementation with CUDA and OpenCL support. Euro-Par. Springer, 2012.

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

[34] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. CGO. ACM, 2014.

[35] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - a portable skeleton library for high-level GPU programming. HIPS Workshop. IEEE, 2011.

[36] J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. IFL. Springer, 2008.

[37] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. CC. Springer, 2002.

[38] 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.

[39] H. Xi and F. Pfenning. Dependent types in practical programming. POPL. ACM, 1999. .

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