A Transformation-Based Approach to Developing High-Performance GPU Programs

Bastian Hagedorn[1], Michel Steuwer[2], and Sergei Gorlatch[1]

[1] University of Münster, Germany,
b.hagedorn@wwu.de, gorlatch@wwu.de,
[2] University of Glasgow, UK
michel.steuwer@glasgow.ac.uk

Abstract. We advocate the use of formal patterns and transformations for programming modern many-core processors like Graphics Processing Units (GPU), as an alternative to the currently used low-level, *ad hoc* programming approaches like CUDA or OpenCL. Our new contribution is introducing an intermediate level of *low-level patterns* in order to bridge the abstraction gap between popular high-level patterns (*map*, *fold/reduce*, *zip*, etc.) and imperative, executable code for many-cores. We define our low-level patterns based on the OpenCL programming model which is portable across parallel architectures of different vendors, and we introduce semantics-preserving rewrite rules that transform programs with high-level patterns into programs with low-level patterns, from which executable OpenCL programs are automatically generated. We show that program design decisions and optimizations, which are usually applied *ad-hoc* by experts, are systematically expressed in our approach as provably-correct transformations for high- and low-level patterns. We evaluate our approach by systematically deriving several differently optimized OpenCL implementations of parallel reduction that achieve performance competitive with OpenCL programs which are manually written and highly tuned by performance experts.

Keywords: Parallel programming, rewrite rules, algorithmic patterns, GPU, OpenCL, code generation, skeletons, transformations

1 Motivation and Related Work

Although systems with many-core accelerators, like Graphics Processing Units (GPUs) and Intel Xeon Phi are an inherent part of modern high-performance computing, achieving high application performance on these systems remains a challenging task even for experienced programmers. Usually, an initial, intuitively correct version of an application is iteratively improved by applying *ad-hoc* optimizations using experience-motivated "rules of thumb". Writing high-performance code for many-core architectures requires explicit management of available resources which nowadays comprise of: 1) a hierarchy of several hundreds or even thousands of processing units (cores) able to execute multiple concurrent threads which are additionally organized in groups, warps, etc., and 2)

a memory hierarchy consisting of multiple cache levels, software-managed local memory for groups of threads, and global memory. Therefore, high-performance code is usually written by experts using low-level programming approaches like OpenCL [10] or CUDA [8] which require the programmer to explicitly manage both thread and memory hierarchies.

The challenges of the state-of-the-art GPU programming are demonstrated in the recent GPU programming guide [8], where performance experts of Nvidia Corp. consider an allegedly very simple application – the reduction of an array (e. g., the summation of array elements). In order to achieve high performance on GPUs, they develop seven differently optimized implementations for this simple example . The eventually achieved speedup is up to 30 times compared to the initial version, which on the one hand emphasizes the importance of program optimizations for GPUs, and on the other hand demonstrates how difficult and hardware-specific the optimization process is, even for simple applications.

In this paper, we propose a systematic approach to program development for systems with GPUs. A program is expressed using high-level algorithmic patterns and then systematically transformed into a program with novel low-level patterns, from which high-performance GPU code is automatically generated.

Our approach is inspired by early work on transformational programming [3–5]. We improve the state of the art by formally introducing low-level, OpenCL-specific patterns and formalizing the device- and application-specific transformations for parallel GPU programs, which before were only informally described in optimization guides of hardware vendors like [8]. The Lift framework [13, 16] provides a prototype implementation of our low-level patterns and rewrite rules, and it demonstrates that they work well for a broad class of real-world applications. There exist libraries based on the concept of algorithmic patterns (skeletons [7]): SkelCL [15], MUESLI [11] or FastFlow [1]. While they rely on hard-coded and hardware-specific implementations with a fixed set of optimizations and are therefore inherently not performance-portable, our approach allows to systematically derive optimizations by rewrite rules which enable applying different optimizations for different devices. Functional approaches like Accelerate [6], Harlan [9] and Obsidian [17] rely either on predefined implementations or on time-consuming code analysis. In contrast, our approach expresses hardware-relevant paradigms of OpenCL in functional code which allows us to express low-level optimizations using rewrite rules.

In this paper, we make the following main contributions: in Sections 2 and 3, we introduce a novel transformation process to systematically develop programs for parallel systems using high-level patterns, and in Section 4, we formalize low-level optimizations as rewrite rules allowing the systematic derivation of optimized programs from a high-level program. In Section 5, we experimentally evaluate programs generated by our approach: and we demonstrate their performance in comparison to high-performance libraries like Nvidia cuBLAS [12].

2 Our Approach: Patterns and Transformations

We use well-known high-level patterns like *map* and *reduce*, also known as algorithmic skeletons [7], to express applications as computations on (multi-dimensional) arrays. Although these patterns themselves are simple, they can specify many real-world applications like N-body-simulations [16] or medical imaging [14]. In our approach, the transformation process systematically rewrites a high-level pattern-based program describing *what* to compute into a low-level pattern-based program describing *how* the computation is organized within the OpenCL programming model. The resulting low-level program is used to generate executable OpenCL code. We use OpenCL as our low-level programming model since it is portable on a broad variety of architectures including GPUs, multicore CPUs, Intel Xeon Phi, and FPGAs.

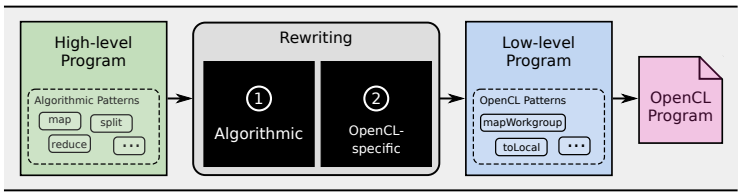

Fig. 1: Transformation approach: a high-level, pattern-based program is systematically transformed into a low-level program in two phases: 1. Algorithmic Rewriting: the high-level program is decomposed in sub-parts that can be executed in parallel. 2. OpenCL-specific Rewriting: the decomposed program is transformed and optimized using OpenCL-specific patterns, from which OpenCL code is generated.

Figure 1 shows how a high-level program is transformed to high-performance OpenCL code in a two-phase transformation process:

1. Algorithmic Rewriting In the first phase, we rewrite the algorithmic structure of a high-level program: we decompose it into parts which can be executed in parallel. For example, instead of reducing an array in one step, we derive a program that expresses the reduction as a tree-based reduction (processing parts in parallel) followed by a final reduction as suggested by the Nvidia experts in [8].

2. OpenCL-Specific Rewriting In the second phase, we transform the decomposed high-level program to a program with low-level, OpenCL-specific patterns by means of rewrite rules that map high-level patterns to the OpenCL's thread hierarchy and data to the OpenCL's memory hierarchy. Efficient OpenCL code can be automatically generated from the resulting low-level program.

3 Algorithmic Patterns and Rewriting

Our high-level patterns are similar to those used in functional programming approaches [3,4], which allows us to reuse already proved rewrite rules. Compared

to existing languages and formalisms, we use only few selected patterns for which we can generate high-performance parallel code. This allows us to limit the variety of required rewrite rules while providing an expressive enough language to develop a broad class of high-performance applications.

3.1 High-Level Algorithmic Patterns

All our patterns are defined as (higher-order) functions on arrays. An array xs of length n containing elements x_i is denoted as $[x_0, \ldots, x_{n-1}]$. Higher-dimensional data structures like matrices or cubes are represented as nested multidimensional arrays. Instead of using recursive cons-lists, as used for example in the BMF, we define our patterns on arrays because we target the generation of high-performance OpenCL codes which work on plain C-arrays. We use a notation similar to the BMF and denote function application by a whitespace: $f\ x$. We use the $\circ$ operator to denote function composition which associates to the right, e. g., $(f \circ g)\ x = f\ (g\ x)$, and has a lower precedence than function application which associates to the left, e. g., $f\ x \circ g\ y$ is read as $(f\ x) \circ (g\ y)$.

Definition 1 (High-Level Algorithmic Patterns).

$$\mathit{map}\ f\ [x_0, \ldots, x_{n-1}] = [f\ x_0, \ldots, f\ x_{n-1}] \tag{1}$$

$$\mathit{reduce}\ (\oplus)\ [x_0, \ldots, x_{n-1}] = [x_0 \oplus \cdots \oplus x_{n-1}] \tag{2}$$

$$\mathit{zip}\ [x_0, \ldots, x_{n-1}]\ [y_0, \ldots, y_{n-1}] = [\langle x_0, y_0 \rangle, \ldots, \langle x_{n-1}, y_{n-1} \rangle] \tag{3}$$

$$\mathit{split}\ m\ [x_0, \ldots, x_{n-1}] = [[x_0, \ldots, x_{m-1}], \ldots, [x_{n-m-1}, \ldots, x_{n-1}]] \tag{4}$$

$$\mathit{join}\ [[x_0, \ldots, x_{m-1}], \ldots, [x_{n-m-1}, \ldots, x_{n-1}]] = [x_0, \ldots, x_{m-1}, \ldots, x_{n-m-1}, \ldots, x_{n-1}] \tag{5}$$

The *map* pattern applies a unary function to all elements of an array. *reduce* combines all elements of an array using an associative binary operator and returns a singleton array. Returning a singleton array instead of a scalar value simplifies the formulation of some rewrite rules. The *zip* pattern combines two arrays of the same length element-wise to an array of pairs. The *split* pattern splits its input into chunks of the specified size m, i. e., it adds another array dimension. The *join* pattern, also known as *concat*, reduces the dimension of a given array by flattening its two outermost dimensions into one.

3.2 Algorithmic Rewrite Rules

In order to systematically transform high-level programs in a semantics-preserving way, we define a set of rewrite rules, also known as algebraic identities, which we denote as $A = B$. In an arbitrary program, the left-hand side expression (A) of a rule can be replaced with the right-hand side expression (B) and vice versa.

For example, the map-distribution rule [4] states that *map* distributes over function composition:

$$\mathit{map}\ f \circ \mathit{map}\ g = \mathit{map}\ (f \circ g) \tag{6}$$

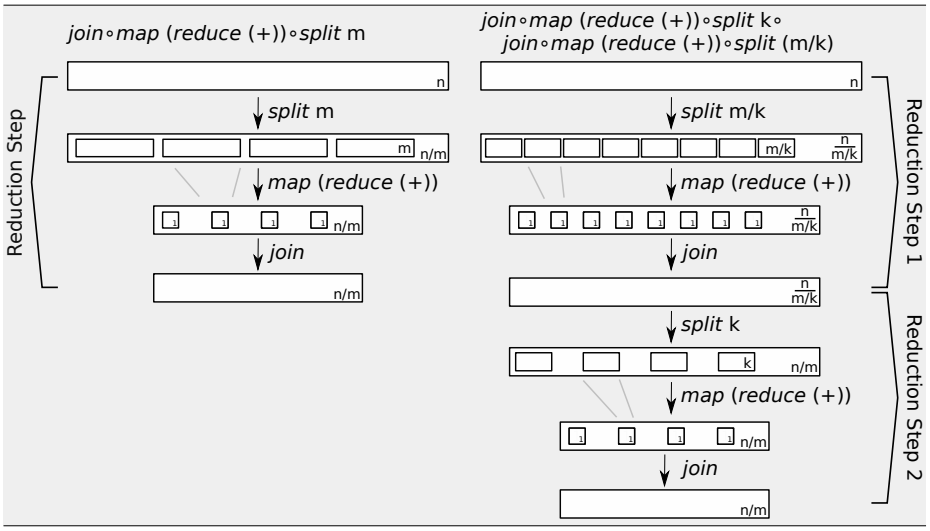

Fig. 2: Tree reduction rule: a reduction of m elements (left) is the same as first reducing m/k elements and then reducing the temporary results (right).

The *map-promotion* rule [4]:

$$map\ f \circ join = join \circ map\ (map\ f) \tag{7}$$

describes handling of two-dimensional arrays: instead of applying f to the flattened array (produced by *join*) it is also possible to apply $(map\ f)$ on each outer array and join the resulting arrays afterwards. A variation of this rule allows to explicitly introduce an additional array dimension using *split*:

$$map\ f = join \circ map\ (map\ f) \circ split\ m \tag{8}$$

Adding additional dimensions using this rule will become useful when we map computations to the hierarchically structured OpenCL programming model.

Rules can also define relations between more complex compositions of patterns as the following *tree-reduction* rule, provided that k divides m:

$$\begin{aligned} join \circ map\ (reduce\ (\oplus)) \circ split\ m = join \circ map\ (reduce\ (\oplus)) \circ split\ k \circ \\ join \circ map\ (reduce\ (\oplus)) \circ split\ \tfrac{m}{k} \end{aligned} \tag{9}$$

Figure 2 visualizes the tree-reduction rule. The left-hand side shows a single reduction step that consists of: 1) dividing the input into disjoint chunks using *split*; 2) reducing all chunks independently using $map\ (reduce\ (\oplus))$; 3) combining the results using *join*.

The right-hand side shows two reduction steps, where the first step reduces the array from n elements to an array of $\frac{n}{m/k}$, before the second step reduces the array to n/m elements, which corresponds to the right-hand side (9). Applying

the tree-reduction rule recursively leads to multiple steps that reduce the input array in a tree-like fashion. Computing reductions as tree-based reductions is one of the optimizations suggested by the Nvidia experts in [8].

A list containing more rewrite rules is given in appendix A.

Correctness of Rewrite Rules All of the rules in this paper are provably correct with respect to a standard functional denotational semantics of our patterns as defined in [13]. Applying semantics-preserving rewrite rules to pattern-based programs allows us to guarantee the correctness of the derivation process, thus, a derived program always computes the same result as the original.

An example of proving (25) using equational reasoning is given in appendix B.

3.3 Transformation Using Algorithmic Rewrite Rules

Let us consider an example of how the summation of the elements of an array is systematically transformed using rewrite rules starting from the high-level program (HLP): *reduce* (+). We deliberately choose this concise example (which is nevertheless non-trivial for implementing on GPUs as shown in [8]) to discuss our formal approach in depth.

In the following, we use a superscript to denote the composition of the same function, e.g. $(f \circ f \circ f) = f^3$. Starting from the high-level program, we systematically perform the following transformations:

$$
\begin{aligned}
& \mathit{reduce}\ (+) && \text{(HLP)}\\
&\{\text{(25) in Appendix A}\}\\
&= \mathit{reduce}\ (+) \circ \mathit{join} \circ \mathit{map}\ (\mathit{reduce}\ (+)) \circ \mathit{split}\ m\\
&\{\text{(25) in Appendix A}\}\\
&= \mathit{reduce}\ (+) \circ \mathit{join} \circ \mathit{map}\ (\\
&\quad \mathit{reduce}\ (+) \circ\\
&\quad \mathit{join} \circ \mathit{map}\ (\mathit{reduce}\ (+)) \circ \mathit{split}\ m\\
&\quad) \circ \mathit{split}\ m\\
&\{\text{(9)}\}\\
&= \mathit{reduce}\ (+) \circ \mathit{join} \circ \mathit{map}\ (\\
&\quad \mathit{reduce}\ (+) \circ\\
&\quad \mathit{join} \circ \mathit{map}\ (\mathit{reduce}\ (+)) \circ \mathit{split}\ 2 \circ\\
&\quad \mathit{join} \circ \mathit{map}\ (\mathit{reduce}\ (+)) \circ \mathit{split}\ {}^{m}\!/_{2}\\
&\quad) \circ \mathit{split}\ m\\
&\{\text{(9)}\}\\
&= \mathit{reduce}\ (+) \circ \mathit{join} \circ \mathit{map}\ (\\
&\quad \mathit{reduce}\ (+) \circ\\
&\quad (\mathit{join} \circ \mathit{map}\ (\mathit{reduce}\ (+)) \circ \mathit{split}\ 2)^2 \circ\\
&\quad \mathit{join} \circ \mathit{map}\ (\mathit{reduce}\ (+)) \circ \mathit{split}\ {}^{m}\!/_{4}\\
&\quad) \circ \mathit{split}\ m\\
&\{\text{(9) applied } (\log m) - 2 \text{ times}\}\\
&= \mathit{reduce}\ (+) \circ \mathit{join} \circ \mathit{map}\ (\\
&\quad \mathit{reduce}\ (+) \circ (\mathit{join} \circ \mathit{map}\ (\mathit{reduce}\ (+)) \circ \mathit{split}\ 2)^{\log m} && \text{(TP)}\\
&\quad) \circ \mathit{split}\ m
\end{aligned}
$$

Interestingly enough, our transformed program (TP) follows the algorithmic structure of the first version of the parallel reduction described by the Nvidia experts in [8]. The parameter m is an arbitrary value (as long as it divides the size of the input) used to split the input into chunks of size m, e. g., $m = 128$ as suggested in the Nvidia example. The input is divided by *split* into distinct chunks; then each chunk is iteratively reduced into a single temporary result by pairwise combining and reducing neighboring elements; finally, all temporary results, i.e., the sums of all chunks, are summed up by the leftmost *reduce*.

We continue with the obtained transformed program (TP) for the parallel reduction in the next section and further transform it into a program with OpenCL-specific patterns. The fully optimized low-level program and the OpenCL code generated from it are presented and evaluated at the end of the paper.

4 OpenCL-specific Patterns and Rewriting

In this section, we introduce OpenCL-specific patterns to specify how programs are mapped onto OpenCL, i. e., how computations are assigned to OpenCL's thread hierarchy, and how data are stored in OpenCL's memory hierarchy. Furthermore, we introduce rewrite rules that transform high-level programs into programs using low-level patterns, which are ultimately transformed to OpenCL.

4.1 Exploiting the Thread Hierarchy using Low-Level Map Patterns

OpenCL [10] is currently a de-facto standard for portable programming of systems with multi- and many-core processors. The OpenCL model differentiates between a *host*, in our case a CPU, and a *device*, e. g., a GPU. The parallel execution of a program, called *kernel*, is performed on the device by multiple threads in the OpenCL's thread hierarchy: threads, called *work-items*, are organized in *work-groups*. Computations within a work-item are performed sequentially. Each work-item has two IDs: a unique *global-id* and a *local-id* which is unique within the work-item's work-group; work-groups have unique IDs themselves. The IDs allow to exploit the thread hierarchy using either only global IDs and therefore work-items directly, or using work-groups and local IDs within them.

Nvidia GPUs add a third level to the thread hierarchy: work-groups are further divided into so-called *warps*, i. e., groups of 32 work-items that are called *lanes* and are executed together in a lock-step manner, i. e., all lanes execute the same instruction at the same time. Although warps and lanes are not captured by the OpenCL 1.2 standard, it is performance-critical to optimize the warp execution for Nvidia GPUs: in particular, since work-items of the same warp are implicitly synchronized, the costly barrier synchronization can be avoided. The current practice of GPU programming requires that low-level, device-specific optimizations are carefully applied by experts with knowledge of the target architecture. We propose low-level, OpenCL-specific patterns and rewrite rules that introduce such optimizations systematically.

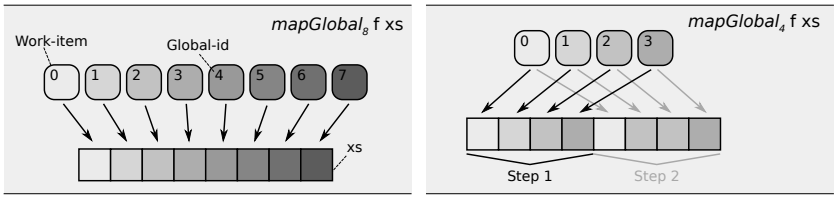

Fig. 3: Visualization of the *mapGlobal* pattern: The input is divided among available work-items which apply the given function to their assigned elements

We start by introducing several low-level variants of the high-level *map* pattern. Each of these low-level patterns represents a possible realization of the high-level *map* semantics (1) using the different levels of the OpenCL's thread hierarchy. Our first pattern – *mapGlobal* – specifies how m work-items, identified by their global IDs, apply function f to all n elements of an array in parallel:

$$mapGlobal_m\ f\ [x_0, \ldots, x_{n-1}] = [y_0, \ldots, y_{n-1}],\ \text{where}\ y_i = f_{(i \bmod m)}\ x_i \quad (10)$$

Here, we annotate function f with a subscript indicating which work-item computes which element: e. g., $f_0\ x_0$ denotes that the work-item with the global ID = 0 applies function f to element x_0. Definition (10) specializes the definition (1) of the high-level map by prescribing which work-items perform computations. We omit in *mapGlobal* the subscript m that specifies the number of work-items, when all available work-items take part in the computation.

Figure 3 shows two possible situations of using the *mapGlobal* pattern. In the simplest case on the left-hand side, the number of work-items equals the size of the input ($m = n$), such that each work-item applies f to the input element whose index in the array equals the work-item's global ID. If there are fewer work-items than input elements ($m < n$), as on the right-hand side of Figure 3, then all work-items start applying f to the m leftmost elements in the array and then proceed to the next m elements, until f is applied to all input elements.

Figure 4 shows the OpenCL pseudo-code which implements *mapGlobal*: a `for`-loop iterates over the global IDs and applies function f to all input elements.

mapGlobal f xs

⇓

```
for (int g_id = get_global_id(0); g_id < n; g_id += m) {
  output[g_id] = f( xs[g_id] ); }
```

Fig. 4: OpenCL pseudo-code implementing definition (10) of *mapGlobal*

Our next patterns – *mapWorkgroup* and *mapLocal* – are used to exploit the two-level thread hierarchy of work-groups and work-items in OpenCL. These patterns are defined similarly to *mapGlobal*, with the difference that the subscript of f corresponds to the work-group ID and local work-item ID within the work-group, correspondingly. The subscript m defines the number of work-items in

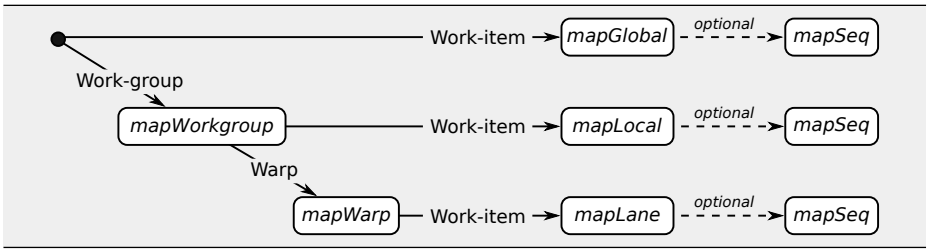

Fig. 5: Valid nestings of low-level maps expressing the OpenCL thread hierarchy.

a work-group for *mapLocal* and the number of work-groups for *mapWorkgroup*. To respect the OpenCL thread hierarchy, the *mapLocal* pattern can only occur nested in the *mapWorkgroup* pattern. The OpenCL implementation pseudo-code for *mapLocal* and *mapWorkgroup* is almost identical with the code of *mapGlobal* in Figure 4. The only difference is that the call to `get_global_id` is replaced with a corresponding call to obtain the local or work-group ID.

For Nvidia GPUs, we utilize the additional, third level of the thread hierarchy by introducing two patterns, *mapWarp* and *mapLane*, defined similar to the *mapGlobal* pattern. For *mapWarp*, the subscript of f corresponds to the warp ID which is calculated as $\lfloor {}^{local\_id}/_{32} \rfloor$. In case of *mapLane*, the subscript of f corresponds to the ID of a lane which is calculated as ($local\_id \mod 32$); furthermore, *mapLane* always has to be nested in a *mapWarp*.

Finally, the *mapSeq* pattern expresses *map* computed sequentially.

Figure 5 provides an overview of the introduced low-level *map* variants and visualizes the nestings of them which respect the OpenCL thread hierarchy. To enforce this nesting structure, we introduce a set of rewrite rules to transform nestings of the high-level *map* pattern into equivalent low-level expressions:

$$map \rightarrow mapGlobal \tag{11}$$

$$map\ (map\ f) \rightarrow mapWorkgroup\ (mapLocal\ f) \mid mapGlobal\ (mapSeq\ f) \tag{12}$$

$$map\ (map\ (map\ f)) \rightarrow mapWorkgroup\ (mapWarp\ (mapLane\ f)) \mid mapWorkgroup\ (mapLocal\ (mapSeq\ f)) \tag{13}$$

$$map\ (map\ (map\ (map\ f))) \rightarrow mapWorkgroup\ (mapWarp\ (mapLane\ (mapSeq\ f))) \tag{14}$$

4.2 Exploiting the Memory Hierarchy using Low-Level Patterns

In the following, we introduce a collection of low-level patterns to utilize OpenCL's memory hierarchy consisting of four disjoint memory spaces: *global*, *local*, *private*, and *constant memory*. The global and constant memory are available to all work-items executing a kernel and correspond to the GPU's main memory. The local memory is shared by all work-items of a work-group and corresponds

to the fast on-chip memory of a GPU. The private memory is owned by a single work-item and corresponds to the registers of a GPU. Accessing the small private and local memory is several hundred times faster than accessing the larger global memory. Therefore, efficient utilization of the GPU's memory hierarchy is mandatory in order to achieve high performance.

To make the different memory spaces explicit in our low-level programs, we extend the array type with a memory space notation: e. g., $[A]_n^{\texttt{global}}$ denotes the type of an array with n elements of type A residing in global memory. The global memory is the default memory space for arrays in OpenCL and input arrays are always allocated in global memory.

We introduce low-level patterns to allow the programmer to change the memory space. In particular, they allow work-items of a work-group to copy data from the global memory to the fast local memory, which is a well-known optimization in OpenCL that can significantly speed up the execution of a kernel. The *toLocal* pattern is defined as follows for an arbitrary memory space M: $\mathit{toLocal} : [A]_n^{\texttt{M}} \to [A]_n^{\texttt{local}}$. Intuitively, this pattern takes an array located in an arbitrary memory space and returns the same array stored in the local memory. The *toLocal* pattern is, therefore, a hint to our code generator to copy the array into the GPU's local memory space. The *toGlobal* and *toPrivate* patterns are defined analogously, allowing the utilization of the corresponding memory spaces. There is no *toConstant* pattern, because the constant memory is read-only.

For example, consider an array xs of type $[[A]_4]_2^{\texttt{global}}$. The following program applies f to all elements using two work-groups with four work-items per group, utilizing local memory and writing the result into the global memory:

$$
\begin{array}{l}
ys = \mathit{mapWorkgroup}_2\ (\\
\quad \mathit{toGlobal} \circ \qquad\qquad\qquad\qquad \text{// copy to global memory} \\
\quad\quad \mathit{mapLocal}_4\ f \circ \qquad\quad \text{// apply f in local memory} \\
\quad\quad\quad \mathit{toLocal} \qquad\qquad\qquad\ \text{// copy to local memory} \\
)\ xs
\end{array}
$$

The following rewrite rules specify how the low-level memory patterns can be used together with the previously introduced low-level *map* patterns:

$$
\begin{aligned}
\mathit{mapWorkgroup}\ f &\to \mathit{mapWorkgroup}\ (\mathit{toGlobal} \circ f) \\
&\mid \mathit{mapWorkgroup}\ (f \circ \mathit{toLocal}) \qquad (15)\\
\mathit{mapLocal}\ f &\to \mathit{toGlobal}\ \circ \mathit{mapLocal}\ f \\
&\mid \mathit{mapLocal}\ f \circ \mathit{toPrivate} \qquad (16)\\
\mathit{mapGlobal}\ f &\to \mathit{toGlobal}\ \circ \mathit{mapGlobal}\ f \\
&\mid \mathit{mapGlobal}\ f \circ \mathit{toPrivate} \qquad (17)\\
\mathit{mapLane}\ f &\to \mathit{toGlobal}\ \circ \mathit{mapLane}\ f \\
&\mid \mathit{mapLane}\ f \circ \mathit{toPrivate} \qquad (18)
\end{aligned}
$$

These rules allow individual work-items to use their private memory, and (15) describes the possibility to use the local memory for computations in a work-

group (the use of local memory outside a workgroup is forbidden in OpenCL). To generate a valid OpenCL kernel, the final result of a kernel has to reside in the global memory to be accessible from the host, therefore, a *toLocal* or *toPrivate* has to be followed by a *toGlobal* in a correct low-level program.

As our final low-level pattern we introduce *reorderStride* which enforces a special reordering of arrays in global memory, for $m = s \times n$, as follows:

$$\begin{aligned} \mathit{reorderStride}\ s\ [x_0, \ldots, x_{n-1}] = [y_0, \ldots, y_{n-1}],\ \text{where} \\ y_i = x_{((i-1)\ \text{div}\ n + s \times ((i-1)\ \text{mod}\ n))} \end{aligned} \tag{19}$$

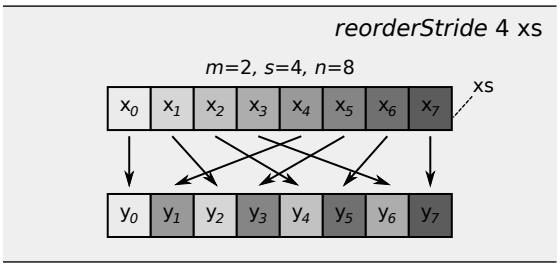

Fig. 6: The *reorderStride* pattern: input elements are reordered using a given stride

Figure 6 visualizes the reordering of an array xs with 8 elements using a stride of 4. Reordering the elements of an array following (19) ensures that when all work-items access their elements, consecutive work-items access consecutive memory elements at the same time. This access pattern corresponds to so-called *coalesced memory accesses*, which are beneficial on GPUs as multiple memory accesses of work-items can be fused to a single memory access. Here, x_2 is reordered to position y_4, because $4 = (2-1)\ \text{div}\ 2 + 4 \times ((2-1)\ \text{mod}\ 2)$. In the generated OpenCL code, reordering will not be performed by copying the array, but rather by reading the array elements in a different order.

The following rewrite rule introduces *reorderStride* in reductions:

$$\mathit{reduce}\ (\oplus) \rightarrow \mathit{reduce}\ (\oplus) \circ \mathit{reorderStride}\ n \tag{20}$$

We only apply *reorderStride* if the reordered array is reduced afterwards. Therefore, we only change the order in which the elements are combined using $\oplus$; this requires that $\oplus$ is associative and commutative.

4.3 Using Low-Level Patterns to Implement High-Level Reduction

Using the rewrite rules for thread and memory hierarchies, we further derive the parallel program for reduction systematically, by introducing OpenCL-specific low-level patterns. We start with the transformed program (TP) obtained in Section 3.3:

$$
\begin{aligned}
& \mathit{reduce}\ (+) \circ \mathit{join} \circ \mathit{map}\ (\\
& \quad \mathit{reduce}\ (+) \circ (\mathit{join} \circ \mathit{map}\ (\mathit{reduce}\ (+)) \circ \mathit{split}\ 2)^{\log m}\ ) \circ \mathit{split}\ m \\
& \overset{\{(12)\}}{=} \mathit{reduce}\ (+) \circ \mathit{join} \circ \mathit{mapWorkgroup}\ (\\
& \quad \mathit{reduce}\ (+) \circ \\
& \quad\quad (\mathit{join} \circ \mathit{mapLocal}\ (\mathit{reduce}\ (+)) \circ \mathit{split}\ 2)^{\log m}\ ) \circ \mathit{split}\ m \\
& \overset{\{(15)\ \text{applied twice}\}}{=} \mathit{reduce}\ (+) \circ \mathit{join} \circ \mathit{mapWorkgroup}\ (\\
& \quad \mathit{toGlobal} \circ \mathit{reduce}\ (+) \circ \\
& \quad\quad (\mathit{join} \circ \mathit{mapLocal}\ (\mathit{reduce}\ (+)) \circ \mathit{split}\ 2)^{\log m} \circ \mathit{toLocal} \qquad \text{(LLP1)} \\
&) \circ \mathit{split}\ m
\end{aligned}
$$

The obtained low-level program LLP1 closely resembles the first optimized OpenCL kernel for parallel reduction as informally described in [8]. In appendix C we show two more low-level programs, LLP2 and LLP3, also similar to versions from [8]. Like the program LLP1, these are derived from the same high-level program HLP by varying the choice of rewrite rules applied.

4.4 Code Generation

Starting with a high-level program consisting of the high-level pattern *reduce*, we systematically derived optimized low-level versions LLP1–LLP3 of the parallel reduction. In this section, we briefly explain how to transform such low-level programs into imperative OpenCL code using our code generator.

Listing 1 shows the OpenCL program OCL1 generated by our code generator [16] from the low-level program LLP1. The generator does not apply any optimizations, but rather transforms low-level patterns into imperative OpenCL code as indicated in Figure 4 for the low-level *map* patterns. Multidimensional arrays have a flat representation in our imperative OpenCL code: therefore, no code is emitted when visiting patterns that change the data layout, like *split*, *join*, or *reorderStride*; these patterns rather influence the generation of how data is accessed by subsequent patterns.

5 Evaluation

In this section, we experimentally evaluate the OpenCL kernels generated from the three low-level programs LLP1-LLP3 (the latter two shown in Appendix C) which have been systematically derived from our initial high-level program *reduce* (+). Interestingly, the low-level program LLP1 describes the computation as implemented in the first version by Nvidia [8], the code for LLP2 is very similar to the fourth implementation, and the code for LLP3 corresponds to the fully optimized, seventh Nvidia's version in the same paper.

```
float sumUp(float x, float y) { return x+y; }
kernel void OCL1(global float* g_idata, global float* g_odata,
                 unsigned int N, local float* sdata) {
    local float* sdata1 = sdata;
    local float* sdata2 = &sdata1[128];
    local float* sdata3 = &sdata2[64];
  for (int wg_id = get_group_id(0); wg_id < (N / (128));
       wg_id += get_num_groups(0)) {
    int l_id = get_local_id(0);
    sdata1[l_id] = g_idata[(wg_id * 128) + l_id];
    barrier(CLK_LOCAL_MEM_FENCE);
    int size = 128;
    local float* sin = sdata1;
    local float* sout = ((7 & 1) != 0) ? sdata2 : sdata3;
    for (int j = 0; j < 7; j += 1) {
      int l_id = get_local_id(0);
      if (l_id < size / 2) {
        float acc = 0.0f;
        for(int i = 0; i < 2; ++i) {
          acc = sumUp(acc, sin[(l_id * 2) + i]); }
        sout[l_id] = acc;
      }
      barrier(CLK_LOCAL_MEM_FENCE);
      size = ( size / 2 );
      sin  = ( sout==sdata3 ) ? sdata3 : sdata2;
      sout = ( sout==sdata3 ) ? sdata2 : sdata3;
    }
    int l_id = get_local_id(0);
    if (l_id < 1)
      g_odata[wg_id + l_id] = sdata2[l_id];
  }
}
```

mapWorkgroup · toGlobal · tree-reduction · mapLocal · toLocal

Listing 1: OCL1: Generated OpenCL code for the low-level program LLP1

We use an Nvidia GeForce GTX 480 GPU to conduct our experiments using the OpenCL runtime from Nvidia's CUDA-SDK 5.5 and driver version 310.44. To measure kernel run times, we use the OpenCL profiling API and we exclude data transfer times to focus on the quality of the generated OpenCL kernels. Each experiment is repeated 100 times, we report the median run time.

Figure 7 shows the performance of our OpenCL kernels (OCL1, OCL2, OCL3) obtained by means of formal transformations and automatic code generation as compared to the hand-written and manually tuned kernels provided by Nvidia in [8](reduce1, reduce4, reduce7). We also compare our programs to the kernels from two libraries which implement manually optimized versions of the parallel reduction for GPUs: cuBLAS [12] and Thrust [2].

In order to compare our performance to the peak performance of the GPU, we report our results as achieved bandwidth in GB/s by dividing the input data size in gigabytes by the elapsed run time in seconds. We observe in Figure 7 that in all cases the performance of our code is on par with the performance of the corresponding manually-tuned codes from [8]. Our most optimized version OCL3 generated from LLP3 slightly outperforms the `reduce7` code from [8] and the Thrust code, and we almost exactly match the performance of the cuBLAS library, which is the currently best known parallel implementation of reduction.

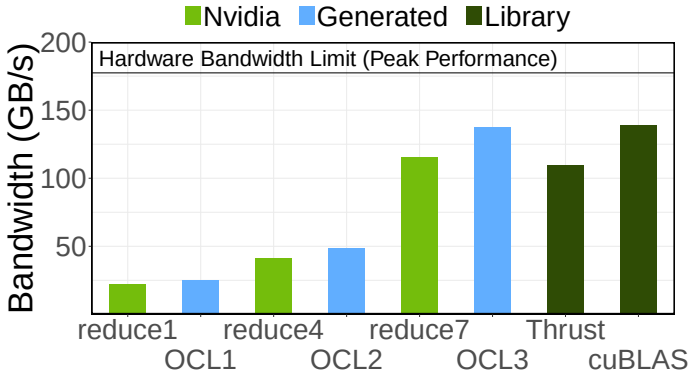

Fig. 7: Performance comparison for generated code against hand-tuned OpenCL code

6 Conclusion

In this paper we present a transformation-based approach to developing high-performance GPU programs using patterns and rewrite rules. We introduce novel, OpenCL-specific low-level patterns to map our computations to the thread and memory hierarchy of the GPU hardware, explicitly describing implementation choices. We formalized well-known optimizations in order to systematically transform high-level programs to provably-correct, optimized low-level programs, rather than apply *ad-hoc* optimizations following the informal optimization guides from GPU vendors. We choose the OpenCL model because it allows to target a wide range of parallel accelerators. However, our transformational programming approach is not limited to OpenCL: we may also use other models like CUDA or OpenMP.

While this paper focuses on formalizing low-level OpenCL-related patterns and rewrite rules, the order in which to apply these rules remains an open research question. Since multiple rewrite rules might be applicable at the same time and some rules can be applied infinitely often, the space of possible low-level expressions needs to be efficiently searched. An automatic randomized search strategy in [13] already leads to well performing programs. One possibility to prune the search space is to package often occurring combinations of rules in so-called macro-rules to encode specific optimizations like tiling. Analytical cost models or heuristics based on machine learning can guide the optimization process.

Experiments show that our transformation-based approach achieves performance which is competitive or even better than hand-tuned code written by performance experts and used in the modern vendor libraries for accelerators.

Acknowledgments

This work was supported by the German Research Council (DFG) within the Cluster of Excellence CiM (University of Muenster), by the German Ministry of Education and Research (BMBF) within the project HPC^{2}SE, and by a EuroLab-4-HPC collaboration. We thank Nvidia for their generous hardware donation used in our experiments.

References

1. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems*. Wiley-Blackwell, 2011.
2. AMD. *Bolt C++ Template Library*.
3. J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.
4. R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.
5. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.
6. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *DAMP*, pages 3–14. ACM, 2011.
7. S. Gorlatch and M. Cole. Parallel skeletons. In *Encyclopedia of Parallel Computing*, pages 1417–1422. Springer, 2011.
8. M. Harris et al. Optimizing parallel reduction in CUDA. *NVIDIA Developer Technology*, 2(4), 2007.
9. E. Holk, W. E. Byrd, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Declarative parallel programming for GPUs. In *PARCO*, pages 297–304, 2011.
10. Khronos OpenCL Working Group. *The OpenCL Specification*.
11. H. Kuchen. A skeleton library. Lecture Notes in Computer Science, pages 620–629. Springer, 2002.
12. Nvidia. *CUDA Basic Linear Algebra Subroutines (cuBLAS)*. Version 6.5.
13. M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance opencl code. In *ICFP*, pages 205–217. ACM, 2015.
14. M. Steuwer and S. Gorlatch. High-level programming for medical imaging on multi-gpu systems using the skelcl library. In *ICCS*, volume 18 of *Procedia Computer Science*, pages 749–758. Elsevier, 2013.
15. M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL: a portable skeleton library for high-level GPU programming. In *HIPS @ IPDPS*, pages 1176–1182. IEEE, 2011.
16. M. Steuwer, T. Remmelg, and C. Dubach. Lift: a functional data-parallel IR for high-performance GPU code generation. In *CGO*, pages 74–85. ACM, 2017.
17. J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *IFL*, volume 5836 of *Lecture Notes in Computer Science*, pages 156–173. Springer, 2008.

Appendix

A Additional Rewrite Rules

$$f = map\ id \circ f = f \circ map\ id \tag{21}$$

$$f = \lambda\ x.\ f\ x \tag{22}$$

$$id = join \circ split\ n \tag{23}$$

$$reduce\ (\oplus) \circ join = reduce\ (\oplus) \circ join\ \circ map\ (reduce\ (\oplus)) \tag{24}$$

$$reduce\ (\oplus) = reduce\ (\oplus) \circ join\ \circ map\ (reduce\ (\oplus)) \circ split\ m \tag{25}$$

B Proof of a Rewrite Rule

Rewrite rules are proved using equational reasoning. As an example we prove rule (25) which introduces layers in the computation hierarchy of a reduction: first a partial reduction is computed, followed by a reduction combining all temporary results.

Proof (Reduce-Promotion Variant). Let n be a number divisible by m.

$$(reduce\ (\oplus) \circ join \circ map\ (reduce\ (\oplus)) \circ split\ m)\ [x_1, \ldots, x_n]$$
{ Def. *split* (4) }
$$= (reduce\ (\oplus) \circ join \circ map\ (reduce\ (\oplus)))\ [[x_1, \ldots, x_m], \ldots, [x_{n-m}, \ldots, x_n]]$$
{ Def. *map* (1) }
$$= (reduce\ (\oplus) \circ join)\ [reduce\ (\oplus)\ [x_1, \ldots, x_m], \ldots, reduce\ (\oplus)\ [x_{n-m}, \ldots, x_n]]$$
{ Def. *reduce* (2) }
$$= (reduce\ (\oplus) \circ join)\ [[x_1 \oplus \cdots \oplus x_m], \ldots, [x_{n-m} \oplus \cdots \oplus x_n]]$$
{ Def. *join* (5) }
$$= reduce\ (\oplus)\ [x_1 \oplus \cdots \oplus x_m, \ldots, x_{n-m} \oplus \cdots \oplus x_n]$$
{ Def. *reduce* (2), associativity of $\oplus$ }
$$= [x_1 \oplus \cdots \oplus x_m \oplus \cdots \oplus x_{n-m} \oplus \cdots \oplus x_n]$$
{ Def. *reduce* (2) }
$$= reduce\ (\oplus)\ [x_1, \ldots, x_n] \quad \square$$

C Derived Low-Level Reduction Programs

```
reduce (+) ∘ join ∘ mapWorkgroup (
  join ∘ toGlobal (mapLocal (mapSeq id)) ∘ split 1 ∘
    ( λ xs . join ∘ mapLocal (reduce (+)) ∘ split 2 ∘ reorderStride ((size xs)/2) xs )^7 ∘
      join ∘ toLocal (mapLocal (reduce (+))) ∘ split 2 ∘
        reorderStride 128 ) ∘ split (2 × 128)
```
(LLP2)

```
reduce (+) ∘ join ∘ mapWorkgroup ( join ∘ toGlobal (mapLocal (mapSeq id)) ∘ split 1 ∘ join ∘ mapWarp (
    join ∘ mapLane (reduce (+)) ∘ split 2 ∘ reorderStride 1 ∘
    join ∘ mapLane (reduce (+)) ∘ split 2 ∘ reorderStride 2 ∘
    join ∘ mapLane (reduce (+)) ∘ split 2 ∘ reorderStride 4 ∘
    join ∘ mapLane (reduce (+)) ∘ split 2 ∘ reorderStride 8 ∘
    join ∘ mapLane (reduce (+)) ∘ split 2 ∘ reorderStride 16 ∘
    join ∘ mapLane (reduce (+)) ∘ split 2 ∘ reorderStride 32
) ∘ split 64 ∘ join ∘ mapLocal (reduce (+)) ∘ split 2 ∘ reorderStride 64 join∘
  toLocal (mapLocal (reduce (+))) ∘ split (blockSize/128) ∘ reorderStride 128) ∘ split blockSize
```
(LLP3)

Fig. 8: Two more low-level programs implementing parallel reduction. They are equivalent to the fourth and the (seventh) most optimized version described in [8], correspondingly