Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPUs

Chandan Reddy, Michael Kruse, Albert Cohen

Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPUs

Chandan Reddy Michael Kruse Albert Cohen
INRIA and École Normale Supérieure, Paris, France
first.last@inria.fr

ABSTRACT

Reductions are common in scientific and data-crunching codes, and a typical source of bottlenecks on massively parallel architectures such as GPUs. Reductions are memory-bound, and achieving peak performance involves sophisticated optimizations. There exist libraries such as CUB and Thrust providing highly tuned implementations of reductions on GPUs. However, library APIs are not flexible enough to express user-defined reductions on arbitrary data types and array indexing schemes. Languages such as OpenACC provide declarative syntax to express reductions. Such approaches support a limited range of reduction operators and do not facilitate the application of complex program transformations in presence of reductions. We present language constructs that let a programmer express arbitrary reductions on user-defined data types matching the performance of tuned library implementations. We also extend a polyhedral compilation flow to process these user-defined reductions, enabling optimizations such as the fusion of multiple reductions, combining reductions with other loop transformations, and optimizing data transfers and storage in the presence of reductions. We implemented these language constructs and compilation methods in the PPCG framework and conducted experiments on multiple GPU targets. For single reductions the generated code performs on par with highly tuned libraries, and for multiple reductions it significantly outperforms both libraries and OpenACC on all platforms.

1. INTRODUCTION

A reduction is an associative and commutative operator on a collection of data elements that reduces its dimensionality. Reductions can be found in many computational applications such as image processing, linear-algebra, partial differential equations, computational geometry, statistical computing, machine learning, etc. They are often found in convergence tests of iterative algorithms and are executed repeatedly. For example, in Monte Carlo simulations averages and variances of a vast number of random simulations are repeatedly computed. A poorly optimized reduction will become a bottleneck for the whole program's performance. Imperative implementations of reductions apply a binary operator successively to all elements of the input set; this induces dependences between loop iterations and forces them to execute sequentially. Since reduction operators are associative and commutative, it is safe to ignore the sequential dependence, exposing parallelism and enabling more aggressive loop nest optimizations. Since reduction operators are associative, multiple reductions can also be performed in parallel and eventually combined to produce the final output. Furthermore, the reduction operator's commutativity can enable target-specific optimizations on massively parallel architectures such as GPUs.

To optimize reductions, we first need to identify them. Automatic detection of simple reductions with commonly used reduction operators such as addition, multiplication, minimum and maximum is well understood. However, when the reduction is performed on a user-defined data type or with a more complex operator, no existing compiler can do this automatically, even if it remains easy for the programmer to identify the associative and commutative operations and the variables involved in the reduction. Hence our proposal to introduce new language constructs to convey information about reductions to the compiler. For a compiler to optimize reductions, it needs to know the data type and identity element of the binary operation, the set of loop iterations carrying the reduction, and the actual associative and commutative operator. The language constructs we propose allow all this information to be expressed by the programmer with very little modifications to input code. Unlike the declarative syntax of parallel languages like OpenMP or OpenACC, these constructs are embedded into the semantics of the underlying imperative language.

Even after the reduction is identified, it is not trivial to optimize them on massively parallel architectures such as GPUs. Reduction operations typically have very low arithmetic intensity, performing one operation per load. Hence, they are bandwidth-bound and require many optimizations to achieve peak performance. Libraries such as CUB, Thrust provide a highly efficient implementation of reductions. They often achieve more than 90% percent of theoretical peak performance by means of a diverse range of optimizations, including architecture-specific instructions to accelerate reductions and tuning parameters for a given architecture. These libraries let the programmer customize reductions by providing an associative and commutative operator and its identity element; it is indeed easy to embed a single reduction into a library call. But when encountering multiple reductions on the same input data, or if performing the reduction on only a subset of the input elements, or combining the reduction with pre- and post-processing steps, these libraries are inefficient as they require multiple calls involving multiple array traversals. For example, consider a loop computing the minimal and maximal elements of an array simultaneously. A typical library

PACT '16, September 11–15, 2016, Haifa, Israel.
© 2016 ACM. ISBN 978-1-4503-4121-9/16/09. . . $15.00
DOI: http://dx.doi.org/10.1145/2967938.2967950

approach would need two function calls calls: one to find the minimum and an other one to find the maximum. This is inefficient as one could use a single traversal of the array to find both. In the same spirit, the SLAMBench computer vision application we consider in our experiments has a reduction kernel that performs 32 different reductions [17]. Using library calls in these cases will be highly inefficient. Although library APIs are highly efficient in their optimized use-cases, they are not flexible enough to adapt to all program requirements.

Reductions are usually bound by the maximum available bandwidth of the device. We can improve the arithmetic intensity of reductions by performing certain loop transformations such as loop fusion. Performing two reductions at once can be twice as fast as two separate reductions. In the latter case, input data is scanned twice, whereas in former input data is scanned just once and two reductions are performed per single load. It is also profitable to fuse the parallel map and reduce type of kernels. In order to enable such transformations, we extend the polyhedral framework of compilation to support reductions. In recent times, the polyhedral model has evolved to a powerful, practical and automatic framework to perform loop nest optimizations. We propose a dependence-based abstraction for reductions in the polyhedral model. This abstraction fits well into an existing polyhedral compilation toolchain and also enables loop transformations in the presence of reductions.

We propose a template-based code generator for reductions. It retains the ability to generate code for arbitrary polyhedral program representation, while embedding a reduction template implementing all the optimizations required to achieve peak performance. The code generation algorithm adapts this template to match the reductions defined by the programmer, and to coordinate the scheduling, storage mapping decisions and data transfers across reduction and non-reduction regions of the program. Building on the rich information about the source program's reductions carried by the new language constructs, it is possible to aggressively modify and specialize the template even after loop nest optimizations have been applied. It is also possible to auto-tune the reduction template for a given GPU architecture, to identify the optimal values for parameters such as block size, grid size, and the number reduction elements per thread. Overall, our approach allows generation of highly efficient code for sequences of user-defined reductions embedded into more general array- and loop-intensive computations, and to make that process portable across different architectures.

Building on a start-of-the-art polyhedral compilation flow capable of generating CUDA and OpenCL code from an ISO C program, we extended this framework to handle generalized reduction constructs and to generate efficient code for these. We evaluate our prototype with a selected set of reduction kernels extracted from the SLAMBench, SHOC and Rodinia benchmark suites. We present a performance comparison against highly tuned libraries such as CUB and Thrust, as well as with the PGI OpenACC compiler, targeting three different GPUs to assess the performance portability.

In summary, our main contributions are:

- language constructs embedded into ISO C as builtin functions allowing programmers to express custom reductions on arbitrary data types;
- the support for generalized reductions carried by imperfectly nested loops, and for indirect indexing in reduction arrays (subscripts of subscripts);
- a dependence-based abstraction of user-defined reductions in the polyhedral model to apply loop nest transformations on reductions;
- template-based code generation for custom reductions that are efficient and portable across multiple GPU architectures;
- the evaluation of several single reduction kernels and one complete application with many reductions on multiple GPU architectures.

2. ABOUT PENCIL

Our work builds upon PENCIL [1], a Platform-Neutral Compute Intermediate Language. This section reviews its essential design and syntactic elements.

PENCIL is intended as a target language for DSL compilers and as a high level portable implementation language for programming accelerators. A program domain expert with high-level knowledge about the relevant operations in a given domain knows a lot of information that could be useful to an optimizing compiler. Information regarding aliasing, parallelization, high level data flow and other domain specific information can be exploited by the compiler to perform more accurate static analysis, additional optimizations and to generate more efficient target-specific code. Such information can be difficult to extract by a compiler but might be easily captured from a DSL, or expressed by an expert programmer. PENCIL is a rigorously-defined subset of GNU C99 and provides language constructs that enable communication of this domain-specific information to the PENCIL compiler.

PENCIL was designed with following main objectives:

Sequential Semantics. Its sequential semantics simplifies DSL-to-PENCIL compiler development and flattens the learning curve of an expert directly developing in PENCIL. Note that the sequential semantics does not preclude the expression of information essential to parallelization, such as interprocedural side-effects and (in)dependence properties; but the user is not bound to any particular pattern(s) of parallelism.

Portability. Any standard C99 compiler that supports GNU C attributes is able to compile PENCIL. This ensures portability to platforms without OpenCL/CUDA support and allows existing tools to be used for debugging sequential PENCIL code.

Ease of analysis. The language simplifies static code analysis for a high degree of optimization. The main restriction of this is that the use of pointers is disallowed, except in specific read-only cases.

Support for domain-specific information. PENCIL helps domain experts and high-level DSL compilers to convey, in PENCIL, domain-specific information that can be exploited by the PENCIL compiler during optimization.

Figure 1 shows a high level overview of a PENCIL compiler framework. At the top level a DSL program is translated into PENCIL by a DSL-to-PENCIL compiler, applying domain-specific optimizations in the process.

PPCG is a polyhedral optimizer that performs loop nest transformations, parallelization, data locality optimization and generates efficient OpenCL or CUDA code [34]. PPCG was modified by the authors of PENCIL to handle the language's extensions [1]. It extracts the additional information provided through PENCIL extensions and uses it to apply the aforementioned transformations. This separation of domain-specific optimizations and general loop-level optimizations make PENCIL a lightweight general-purpose language applicable to a wide range of domains, and made it a

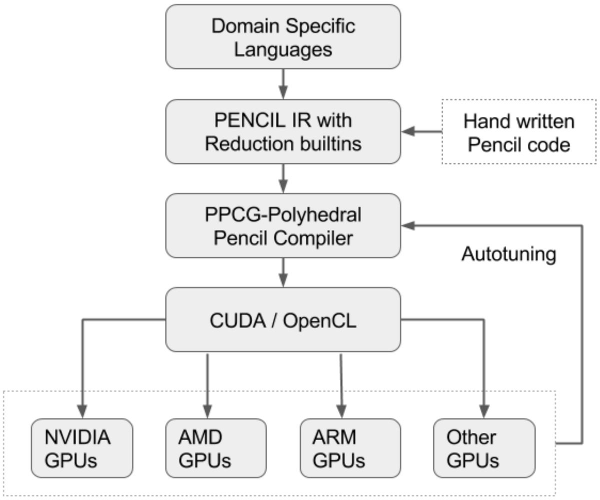
Domain Specific Languages
PENCIL IR with Reduction builtins
Hand written Pencil code
PPCG-Polyhedral Pencil Compiler
Autotuning
CUDA / OpenCL
NVIDIA GPUs
AMD GPUs
ARM GPUs
Other GPUs

Figure 1: A high level overview of the PENCIL compilation flow

prime candidate for the prototyping of our reduction constructs and compilation methods. Our framework also supports auto-tuning of the generated code to find optimal parameter values for tile sizes, block sizes, grid sizes, etc. for a given target architecture.

We detail the most important restrictions imposed by PENCIL from the point of view of enabling GPU-oriented compiler optimizations. The PENCIL specification [2] contains the rules in full.

Sized, non-overlapping arrays. Arrays must be declared through the C99 variable-length array syntax [11], with the `static const restrict` C99 type qualifiers/keywords. As a shortcut, the `pencil_attributes` macro expands to the latter. Optimizations in the PENCIL compiler know about the length of arrays, and that arrays do not overlap.

Pointer restrictions. Pointer declarations and definitions are allowed in PENCIL, but pointer manipulation (including arithmetic) is not, except that C99 array references are allowed as arguments in function calls. Pointer dereferencing is also not allowed except for accessing C99 arrays. The restricted use of pointers is important for moving data between different address spaces of hardware accelerators, as it essentially eliminates aliasing problems.

No recursion. Recursive calls are not allowed, because they are not supported by accelerator programming languages such as CUDA or OpenCL.

Structured for loops. A PENCIL `for` loop must have a single iterator, a loop-invariant stop value and a constant increment (step).

The main constructs introduced by PENCIL include the `assume` and `kill` builtin functions, the `independent` directive and summary functions. They are described here very briefly, in a form extracted from the more complete description of [2, 1].

2.1 Summary Functions

Summary functions are used to describe the memory access patterns of (1) library functions called from PENCIL code, for which source code is not available for analysis, and (2) non-PENCIL functions called from PENCIL code to enable more precise static analysis. The concept is inspired from stub functions in PIPS [10] and shares the motivations of access specifications in Jade [29] and Access/Execute metadata [9]. As a distinctive feature, the concrete semantics of summary functions captures both may- and must-access information from a single imperative implementation.

```
__attribute__((pencil_access(summary_fft32)))
void fft32(int i, int j, int n,
           float in[pencil_attributes n][n][n]);

int ABF(int n, float in[pencil_attributes n][n][n])
{
  // ...
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      fft32(i, j, n, in);
  // ...
}

void summary_fft32(int i, int j, int n,
           float in[pencil_attributes n][n][n]);
{
  for (int k = 0; k < 32; k++)
    __pencil_use(in[i][j][k]);
  for (int k = 0; k < 32; k++)
    __pencil_def(in[i][j][k]);
}
```

Figure 2: Example code illustrating the use of summary functions

Figure 2 shows an example use of summary functions. The code calls the function `fft32` (Fast Fourier Transform). This function only reads and modifies (in-place) 32 elements of its input array `in`, but does not modify any other parts of the input array. Without a summary function the compiler must conservatively assume that the whole array passed to `fft32` is accessed for reading and writing. Such a conservative assumption prevents parallelization. The effect of a calling `fft32` is given by its summary function `summary_fft32`. The PENCIL compiler derives accurate memory access information (reads and writes of 32 elements) `summary_fft32` enabling parallelization of the loop nest.

Summary functions are a powerful construct enabling polyhedral analysis and transformations for non-affine code. Traditional polyhedral compilers handle control flow with only affine conditionals. When dealing with non-affine parts and data dependent conditionals, one may create a function encapsulating such control flow regions and provide conservative memory accesses information through a summary function. A polyhedral compiler can now perform analyses and transformations based on the memory access information in the summary function. For example, we were able to pencilize the large SLAMBench application, rich in non-affine and data dependent control flow, using summary functions.

2.2 Assume Builtin

`__pencil_assume(e)` is an intrinsic function where e is a logical expression, indicating that e is guaranteed to hold whenever the control flow reaches the intrinsic. This knowledge is taken on trust by the PENCIL compiler, and may enable generation of more efficient code. An assume statement allows a programmer to communicate high level facts to the optimizer.

A *general 2D convolution* in image processing is a a good example that demonstrates the use of `__pencil_assume`. This image processing kernel calculates the weighted sum of the area around each pixel using a kernel matrix for weights. It can be helpful to know that the size of the convolution matrix (the matrix that holds the convolution kernel) is at most 15 × 15. This can be expressed using the assume builtin as follows:

```
__pencil_assume(kernel_matrix_rows <= 15);
__pencil_assume(kernel_matrix_cols <= 15);
```

This information enables a Pencil compiler to perform optimizations such as unrolling the inner convolution loops or placing kernel matrix into shared memory.

2.3 Independent Directive

The `independent` directive is used to annotate loops. It indicates that the desired result of the loop execution does not depend in any way upon the execution order of the data accesses from different iterations. In particular, data accesses from different iterations may be executed simultaneously. In practice, the `independent` directive is used to indicate that the marked loop does not have any loop-carried dependences (i.e., it could be run in parallel).

Note that reductions are *not* allowed in a loop annotated with the `independent` directive, as they carry dependences along the successive iterations of the loop. This is a major limitation of Pencil that we address in this paper.

2.4 Pencil Kill

The **`__pencil_kill`** builtin function allows the user to refine data flow information within and across any control flow region in the program. It is a polymorphic function that signals that its argument (a variable or an array element) is dead at the program point where **`__pencil_kill`** is inserted, meaning that no data flows through the argument from any statement instance executed before the kill to any statement instance executed after. The accurate data flow information avoids unnecessary data transfers between the host and accelerator.

3. REDUCTIONS IN PENCIL

Let us now present the Pencil extensions to represent user-defined reductions. Figure 3 shows a sample reduction code of complex number multiplications. Automatic techniques for the detection of reductions have been proposed; see [6] for a survey of the polyhedral ones. However, most of these techniques can only detect a simple reduction with standard operators such as sum, max, min, etc. Real-world applications often involve user-defined reductions with custom reduction operators and user-defined data types. Moreover, reductions might be applied only to a range of array indexes. Automatic detection techniques are expensive and fail to detect such complex user-defined reductions. Often the programmer or the code generator for a high-level, domain-specific language, readily knows all information related to reductions. Hence, we provide Pencil extensions that will allow a programmer to easily express arbitrary reductions.

We analyzed many benchmarks from different suites (SHOC, Rodinia, PolyBench, SLAMBench, etc.) and many DSLs (linear algebra, image processing, signal processing, etc.). Based on these, and considering the semantic constraints of embedding custom reduction information into a statically compiled imperative language with first order functions, we designed the following extensions that are natural and flexible enough to express all the reductions we have encountered. The programmer or DSL compiler needs to convey the following pieces of information regarding reductions to the underlying compiler: *reduction operator*, *identity element* and *reduction domain*. The reduction operator is a commutative and associative function of two arguments that returns one value of the same type. The identity element of the reduction operator is used to initialize the temporary variables that hold the intermediate result. The reduction domain represents the set of iterations for which the reduction operator is called, possibly spanning multiple nested loops.

Figure 4 shows a sample reduction kernel taken from the Srand benchmark of the Rodinia suite. Figure 5 shows the Pencil version of Figure 4. All the information regarding reductions is expressed using two extensions of Pencil:
`__pencil_reduction_var_init` and **`__pencil_reduction`**.

```
typedef struct COMPLEX {
  int a;
  int b;
} Complex;

Complex multiply(Complex x,Complex y) {
  Complex z;
  z.a = x.a*y.a - x.b*y.b;
  z.b = x.a*y.b + x.b*y.a;
  return z;
}

Complex Reduce(const int N, Complex input[N]){
  int i;
  Complex product = {1.0, 0.0};

  for(i=0;i<N;i++)
    product = multiply(product, input[i]);

  return product;
}
```

Figure 3: Complex number multiplication

3.1 Reduction Initialization Builtin

`__pencil_reduction_var_init` is an intrinsic function that is used to express the identity element of the reduction operator. The first argument of this function is the address of the reduction variable. A reduction variable can be a scalar or an array element of a built-in type or a user-defined data type as in the example in Figure 6. The second argument is a function that will take reduction variable as an input and initialize it with the identity element. Having a simple function to initialize reduction variable provides the flexibility to handle initialization of any user-defined data types such as complex numbers. The function provided in **`__pencil_reduction_var_init`** is called by the compiler whenever it wants to initialize the temporary variable required to compute the reductions in parallel. This intrinsic function also marks the beginning of reduction domain.

3.2 Reduction Builtin

`__pencil_reduction` is another intrinsic function that is used to express a single reduction operation. The first argument for this function must be the address of the reduction variable. The second argument must be the current reduction element. The third argument is a function that implements the reduction operator. This function must accept two variables which are of the same type as the reduction variable and return the result of reduction operator. This function is assumed to be both commutative and associative. Every reduction variable must be introduced through an associated **`__pencil_reduction_var_init`** and operated upon through **`__pencil_reduction`**.

The domain of the reduction is the set of all dynaminc instances of **`__pencil_reduction`**.

We believe this intrinsic function approach is flexible enough to express most user-defined reductions.

3.3 Related work

Many programming languages provide high-level abstractions to express user-defined reductions. Google's Map Reduce [4] framework provides parallelization API through which a user can specify custom reduction functions. Intel TBB [28] provides parallel reduction templates that can be specialized through custom reduction methods. Similar to TBB, a user can express reductions in PPL [15]

```
int srand_reduction(int niter, int Nr, int Nc,
                    float image[Nr][Nc]){
  for (int iter=0; iter<niter; iter++) {
    float sum = 0.0; //S1
    float sum2 = 0.0; //S2

    for (int i = 0; i < Nr; i++) {
      for (int j = 0; j < Nc; j++) {
        sum += image[i][j]; //S3
        sum2 += image[i][j] * image[i][j]; //S4
      }
    }

    float meanROI = sum / NeROI; //S5
    float varROI = (sum2 / NeROI) - meanROI*meanROI;
    float q0sqr = varROI / (meanROI*meanROI);

    diffusion( Nr, Nc, q0sqr, image, c);
  }
}
```

Figure 4: Reduction from Rodinia's Srand benchmark

```
void initialize(float *val) {
  *val = 0.0;
}
float reduction_sum(float v1, float v2){
  return v1 + v2;
}

void srand_reduction(int niter, int Nr, int Nc,
                     float image[Nr][Nc]){
  for (int iter=0; iter<niter; iter++) {
    float sum;
    float sum2;
    __pencil_reduction_var_init(&sum2, initialize);
    __pencil_reduction_var_init(&sum, initialize);

    for (int i = 0; i < Nr; i++) {
      for (int j = 0; j < Nc; j++) {
        __pencil_reduction(&sum, image[i][j],
                           reduction_sum);
        __pencil_reduction(&sum2, image[i][j]*image[i][j],
                           reduction_sum);
      }
    }

    float meanROI = sum / NeROI;
    float varROI = (sum2 / NeROI) - meanROI*meanROI;
    float q0sqr = varROI / (meanROI*meanROI);

    diffusion( Nr, Nc, q0sqr, image, c);
  }
}
```

Figure 5: Example from Rodinia's Srand reduction in PENCIL

using parallel reduction templates. In both cases the user needs to pass explicitly the range of values to be reduced. Cilk++ [7] supports a limited number of reduction operators which are used to eliminate the contention of shared reduction variables by performing reductions in a lock-free manner.

MPI [16] includes support for several built-in reduction operators as well as the ability to define custom reduction operators in the context of distributed computing. In MPI, a user needs to create a custom function with fixed syntax which will be called by runtime while reduction is performed across multiple nodes. ZPL [5] relies on overloading for the specification of user defined reductions. The user needs to create two functions with specific signatures, one to return the identity element and another function that implements reduction operator.

OpenMP 3.0 and OpenACC [35] support built-in reduction operators through declarative pragma syntax. OpenMP 4.0 introduced user-defined reductions, specifying the custom associative and commutative function and identity element through a new directive: `#pragma omp declare reduction`. More recently, OpenMP 4.5 introduced pointwise reductions over arrays in C and C++.[1] One may apply user-defined and pointwise array reductions to a specific for loop which forms the domain of the reduction. If multiple, imperfectly nested loops form the reduction domain, it is not directly possible to express it through OpenMP pragmas; the programmer needs to modify the loop structure to handle such cases. In our proposed approach the nesting depth of the reduction and its domain are inferred from reduction builtin locations, and such that the programmer does not need to modify the loop structure. Array reductions that are not pointwise—such as histograms or reductions over unstructured meshes—cannot be expressed in OpenMP or OpenACC.

```
void initialize(float *val){
  *val = 0.0;
}
float reduction_sum(float v1, float v2){
  return v1 + v2;
}

void kernel_correlation(int M, int N,
                        float data[M][N], float mean[M]){
  for (int j = 0; j < M; j++) {
    __pencil_reduction_var_init(&mean[j], initialize);
    for (int i = 0; i < N; i++)
      __pencil_reduction(&mean[j], data[i][j],
                         reduction_sum, NULL);
      mean[j] /= N;
  }
}
```

Figure 6: Example from PolyBench's correlation benchmark

4. MODELING REDUCTIONS WITHIN A POLYHEDRAL FRAMEWORK

In polyhedral compilation, an abstract mathematical representation is used to model the program and its transformations.

4.1 Polyhedral Framework

Each statement in the program is represented using three pieces of information: an *iteration domain*, an *access relations* and a *schedule*. This representation is first extracted from the program AST, which is then analyzed and transformed (loop optimizations are applied during this step), and finally it is converted back into an AST.

The *iteration domain* of a statement is a set that contains all the execution instances of the statement (a statement in a loop has an execution instance for each iteration where it is executed). Each execution instance of the statement in the loop nest is represented individually by an identifier for the statement and a sequence of integers (typically, the values of the outer loop iterators) that uniquely identifies the execution instance. Instead of listing all the integer tuples in the *iteration domain*, the integer tuples are described using quasi-affine constraints.

A quasi-affine constraint is a constraint over integer values and integer variables involving only the operators `+`, `-`, `*`, `/`, `%`, `&&`, `||`, `<`, `<=`, `>`, `>=`, `==`, `!=` or the ternary `?:` operator. An example of a quasi-affine constraint used in a loop nest is $10 \times i + j + n > 0$ where i and j are the loop iterators and n is a *symbolic constant* (i.e., a variable that has an unknown but fixed value throughout the execution). Examples of non quasi-affine constraints are $i \times i > 0$ and $n \times i > 0$.

In order to be able to extract the polyhedral representation, all loop bounds and conditions need to be quasi-affine with respect to

[1] See http://www.openmp.org for the complete specification.

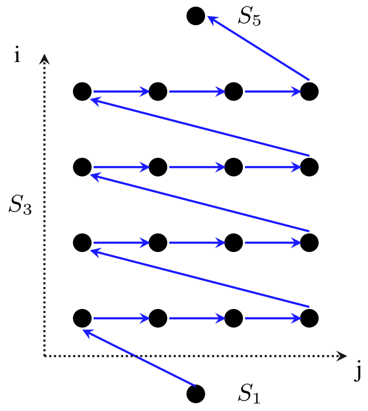

Figure 7: Original reduction domain and dependences

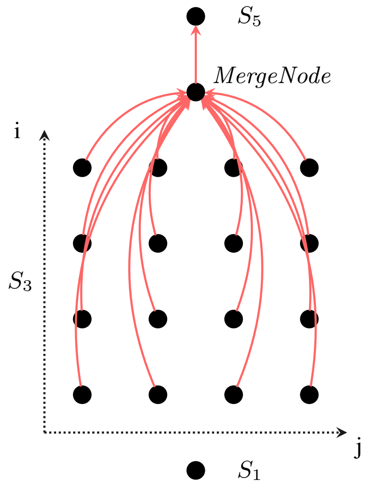

Figure 8: Modified reduction dependences

the loop iterators and a fixed set of symbolic constants. We will abbreviate this condition as *static-affine*.

The *access relations* map statement instances to the array elements that are read or written by those instances, where scalars are treated as zero-dimensional arrays. An accurate representation requires that the index expressions in the input program are static-affine. The *dependence relations* map statement instances to statement instances that depend on them for their execution. These dependence relations are derived from the access relations and the original execution order. In particular, two instances depend on each other if they (may) access the same array element, if at least one of those accesses is a write and if the first is executed before the second. Finally, the *schedule* determines the relative execution order of the statement instances. Program transformations are performed by modifying the schedule.

4.2 Reduction Domain

The reduction domain is defined as the set of all iterations on which the reduction operator is applied. In pragma-based languages such as OpenMP or OpenACC, the reduction domain is simply the loop that is annotated with a reduction pragma. In some benchmarks such as the Rodinia's Srand benchmark shown in Figure 4, the reduction domain is not a single for-loop. The programmer needs to coalesce (flatten) multiple loops into a single one before annotating it with pragmas. Whereas in PENCIL, since the programmer uses builtin functions that are not associated with a specific loop, no modifications of the control flow is required. Note that the reduction domain may not be same as the iteration domain of the `__pencil_reduction` statement. For example, the iteration domain of statement S_3 in Figure 4 is $\{S_3(iter, i, j) : 0 \leq iter \leq niter; 0 \leq i < nr; 0 \leq j < nc\}$ whereas, the reduction domain for the reduction in S_3 is $\{S_3(i, j) : 0 \leq i < nr; 0 \leq j < nc\}$. The reduction operation always starts with the initialization of the reduction variable, using `__pencil_reduction_var_init`. The reduction domain is derived from the iteration domains of the `__pencil_reduction_var_init` and `__pencil_reduction` statements. For each `__pencil_reduction` statement, the reduction variable—the first argument of the function—is looked up to find the dominating `__pencil_reduction_var_init` defining the same reduction variable. The reduction domain is obtained from the iteration domain of `__pencil_reduction` by considering the loop iterators of `__pencil_reduction_var_init` as parameters.

4.3 Reduction Dependences

The Figure 7 depicts the Read-after-Write (RAW) dependences of a single reduction for the program in Figure 4. The arrows between iterations indicate the flow of data, i.e., a value produced by the source iterator is read at the destination iteration. Because the reduction variable is read and written in all iterations of the reduction domain, there is a serial dependence between them. This dependence between iterations of the reduction domain is called a reduction dependence. This dependence usually forces the serial execution of the reduction. Since the reduction operator is associative, we can perform the reductions in parallel and then combine the partial results to obtain a final reduction value. The associativity of the reduction operation is abstracted by relaxing the reduction dependences. We can precisely compute these reduction dependences from the reduction domain. The reduction of partial results and synchronization needed to produce the end result is modeled by introducing an additional node in the dependence graph, called a *merge node*, and adding a new dependence from all iterations in reduction domain to the merge node as shown in Figure 8. These are called *reduction isolation dependences* are necessary to prevent the use of the reduction variable before the reduction operation is complete when the reductions are performed in parallel. The merge node is inserted right after the reduction loops in the input AST. The relaxation of the serial reduction dependences into parallelism-friendly reduction isolation dependences will accurately model the flow of data for the parallel execution of reductions.

4.4 Reduction-Enabled Scheduling

A schedule represents the execution order of statement instances in a program. Various transformations are performed by changing the schedule. Dependences are used to find the valid set of schedules. A schedule is said to be valid if does not violate dependences in the input program, i.e., all the source iterations of dependences are executed before the destination iterations. State-of-the-art polyhedral compilers are seriously limited in the application of affine transformations in the presence of reductions, because of the serial reduction dependence. They do not exploit the associativity of the reduction operator. The explicit dependence manipulation as explained in the previous section enables transformations such as tiling and parallelization of reduction loops. Because the serial reduction dependences are relaxed, the polyhedral scheduler can now safely reorder the reduction iterations.

A practical and automatic scheduling algorithm like Pluto takes a dependence graph as input and recursively constructs schedule. At each level of the recursion, the algorithm first checks for the components that do not depend on each other and hence can be scheduled independently. Within each component, the algorithm uses an ILP solver to construct a sequence of one-dimensional affine functions (hyperplanes), such that each of these functions independently respect all dependences and are optimal based on heuristics such as induced communication. After the construction of a band is completed, the dependence graph is updated to only contain dependences that are mapped to the same values by the current hyperplane, and the process is repeated until the number of hyperplanes found is equal to the dimensionality of the loop. Our dependence based abstraction of reductions fits well with the automatic scheduling algorithms enabling affine transformations for user defined reductions. Because the relaxation of serial reduction dependences, the reduction iterations can now be reordered and parallelized. The new merge dependences represent the required data-flow of combining the partial reduction results to produce the final reduction value. It is essential for the scheduler to know about the cost of performing reductions in parallel and merge dependences accurately represent this cost. For example, consider a loop nest with a parallel loop and a reduction loop. The scheduler should choose the parallel loop with no loop-carried dependences as the outermost loop rather than the reduction loop because of the cost of merge dependences. Hence, with this approach reduction parallelism is exploited and yields communication-free parallelism. Many of the previous approaches that just relax the reduction dependence have the problem of treating both types of parallelism equally which could be lead to poor schedules and additional communication.

4.5 Related work

Modeling reductions was commonly done implicitly, e.g., by ignoring the reduction dependences during a post parallelization step [12, 13, 21, 25, 22, 24, 36, 33]. The first one to introduce reduction dependences, where Pugh and Wonnacott [23]. Similar to most other approaches [26, 31, 27, 8] the detection and modeling of reductions was performed on imperative statements and utilizing a precise but costly access-wise dependence analysis. In the works of Redon and Feautrier [27, 26] the reductions are modeled as Systems of Affine Recurrence Equations (SAREs). Array expansion allow to eliminate the memory-based dependences induced by reductions and facilitate the recognition of induction patterns. They also propose a polyhedral scheduling algorithm that optimally schedules reductions together with other statements, assuming reductions are computable in single time step. Such atomic reduction computation simplifies scheduling choices while preventing schedules that reorder or interleave reduction statement instances with other statement instances. Gupta et al. [8] extended the work of Redon and Feautrier [26] by removing the restriction of an atomic reduction computation. Their scheduling algorithm tries to minimize the latency while scheduling reductions. Compared to these works our dependence based abstraction works on existing practical polyhedral scheduling algorithms such as Pluto [3]. Our approach does not require any preprocessing such as array expansion while dependences accurately model the specific atomicity constraints of reductions.

In contrast to polyhedral optimizations for reductions, Gautam and Rajopadhye [8] propose techniques to exploit the associativity and commutativity of reductions to decrease the complexity of a computation in the context of dynamic programming. Their method reuses intermediate results of reduction computations.

Stock et al. [30] describe how reduction properties can be used to reorder stencil computations, to better exploit register reuse and to eliminate loads and stores. However neither do they describe the detection method nor does their method enable scheduling for generic reductions.

Doerfert et al. [6] propose compiler techniques to automatically detect reductions on LLVM IR. They also propose a model to relax memory-based dependences to enable polyhedral scheduling in the presence of reductions. However, their techniques cannot detect reductions on arbitrary data types such as complex number multiplication and their reduction modeling do not account for the cost of reductions while scheduling them.

5. CODE GENERATION

Parallelizing reductions on GPUs is a challenging task. Reductions typically have low arithmetic intensity performing just one operation per memory load and hence are bound by the device's maximum memory bandwidth. The performance of reductions is often measured by the effective bandwidth achieved by a given implementation. Vendor and highly tuned libraries such as CUB and Thrust provide an optimized implementation of reductions achieving performance above 90% percent of the peak bandwidth.

5.1 Optimizations for Reductions on GPUs

The problem of Optimizing reductions in CUDA is well studied and the list of all the optimizations and their impact on performance for a single reduction is explained in [14]. The important optimizations are listed below.

Kernel Decomposition. The reduction is parallelized at two levels, matching the thread hierarchy of the GPUs. At the first level, each thread block is allocated a portion of the reduction domain. Within each thread block, a tree-based decomposition is used to perform local reductions and to produce partial results. These partial results are stored in global memory. At the second level, another kernel is launched to reduce the partial results and produce the final value.

Shared memory utilization. Within each thread block, multiple threads use shared memory to keep the intermediate reduction values. The tree-based reduction is performed on these values while avoiding divergent branches. Shared memory bank conflicts are avoided by reordering memory loads making use of the reduction operator's commutativity.

Complete unrolling. The kernel is specialized by completely unrolling the reduction tree. This is done using templates to generate a specialized kernel for different block sizes. This eliminates unnecessary control flow and synchronization between the threads in a thread block as the reduction proceeds.

Using shuffle instructions. Shuffle instructions [20] can be used to accelerate fine-grained reductions within a warp. These dedicated instructions control the exchange of data between threads, eliminating data accesses to shared memory and the associated synchronizations.

Multiple reductions per thread. Each thread loads and reduces multiple elements into shared memory before the tree-based reduction in shared memory. More work per thread will help to hide the memory latency. The optimal number of elements per thread depends on the architecture and is determined by tuning this parameter along with the number of blocks and threads per block.

5.2 Template Specialization for User-Defined Reductions

A generic polyhedral optimizer such as PPCG can perform various optimizations such as shared memory allocation and register promotion, memory coalescing, etc. for generic programs [34]. It uses heuristics to determine the profitability of such optimizations. However, it cannot perform all the specialized optimizations required for highly efficient reductions. Some of these optimizations are specific to reductions, and applicable only to specific hardware or device characteristics. Hence it is difficult to come up generic heuristics for these optimizations in order to make them applicable to generic programs. These optimizations are crucial to achieve close-to-peak performance on GPUs. Hence, we follow a template-based approach for generating efficient code for user-defined reductions. We precisely identified the reductions through programmer-provided constructs, hence can generate efficient reduction code capturing all the optimizations above, even on user-defined reductions.

User-defined reduction operator. A reduction is characterized by the reduction operator, its identity element, and the domain of the reduction. The new reduction operator is expressed in a separate PENCIL function. It is easy to adapt the parallel template to user-defined reductions by replacing the reduction operation with the user-defined function, and similarly for the partial value initialization. All these function calls are inlined to eliminate overhead. The reduction domain is equally distributed among multiple thread blocks.

Shared memory allocation. The parallel reduction template allocates temporary storage for each thread, to store partial results in the CUDA shared memory. In the case of a scalar, or pointwise array reduction, each thread only needs one scalar reduction accumulator to be stored in shared memory; generalized reductions may involve larger array segments. Yet the total shared memory available varies from device to device and depends on the optimal number of threads for each device. Therefore, the available shared memory limits certain transformations and thread block sizes. The maximum amount of shared memory on the device is an input to our framework. We then compute the shared memory required to implement the reduction and use it to calculate the maximum thread block size.

Fusing multiple reductions. The template can also be adapted to generate code for multiple fused reductions. Fusion can help increase the arithmetic intensity of the reduction. We use the following heuristic to determine when to fuse. Two reductions are fused if there is an overlap with global data accessed between them. For example, if two separate reductions operate on same input array, then these two reductions are fused and a single reduction template is generated. This optimization is always profitable since by fusion we are enabling data reuse in shared memory. Existing polyhedral techniques can be used to assess the correctness of such a fusion transformation, taking into account all side-effects involved in associated computations involved in the reductions (inductively or not). Code generation for fused reductions is a straightforward extension of the above-mentioned technique: the shared memory required is the sum of shared memory requirement for individual reductions and the two reduction functions are called one after another in the template.

Auto-tuning. The performance of the template depends on the target GPU. The following two parameters are tuned to obtain optimal performance on a given architecture: number of threads, and number of thread blocks. The reduction template supports powers of two only for the number of threads, so there are only few values considered starting from 2 and up to the maximum allowed by the device. The maximum is also limited by the required shared memory for the reduction. The thread block size varies over the 2 to 256 interval.

6. EXPERIMENTAL EVALUATION

We implemented PENCIL reduction support in a developmental branch of PPCG. Our framework takes a C program with PENCIL functions as input, with reduction kernels according to the specification. It generates CUDA code automatically, which is then auto-tuned to a particular GPU architecture. We selected a reduction template for scalar and pointwise array reductions, hence the absence of histogram (subscript of subscript) reductions in the experiments below. Since the auto-tuning search space is relatively small for our reduction template, we could conduct an exhaustive search of the optimization space within minutes on all examples.

We compare the performance of the generated code with highly tuned libraries such as CUB v1.5.1 [18] and Thrust v1.8.1 [19]. Both of these libraries provide APIs for performing reductions. The user can customize them by specifying a reduction operator and an identity value. Note that these libraries are optimized for a particular GPU architecture. CUB, for example, has an internal database of the optimal values for thread block size, number of threads and number of items per thread for all known architectures. We also compare the performance with the OpenACC PGI 2015 compiler [32], which provides high-level directives, including reduction pragmas, to program accelerators. Note that the PGI OpenACC compiler currently only supports a limited set of pre-defined reduction operators.

We evaluate performance on the following three GPU architectures: a desktop NVIDIA GTX 470 with peak memory bandwidth of 133.9 GB/s, the more advanced NVIDIA Quadro K4000 with peak bandwidth of 134.9 GB/s, and the low power embedded GPU NVIDIA Jetson TK1 with peak bandwidth of 14.78 GB/s. We do not present OpenACC numbers on TK1 because the PGI compiler does not support this architecture. All the benchmarks are compiled with the NVIDIA CUDA 7.5 toolkit, with the `-O3` optimization flag. Reduction performance is measured as the effective bandwidth utilized by the benchmark, and is computed using

$$\text{Bandwidth} = \text{InputArraySize}/\text{ReductionTime}.$$

Each benchmark is run 100 times and the average of these times is taken as the reduction time. We also perform a dry-run before benchmarking to hide device configuration and kernel compilation time. In the graphs below, performance numbers are presented as a percentage of peak bandwidth of the device.

6.1 Single Reduction Kernel

The first benchmark is a single sum reduction over an array of 2^{22} elements. This corresponds to a single call in CUB and Thrust, and to a single for-loop marked with a reduction pragma in OpenACC. The performance of the sum benchmark for three different data types is shown in Figures 9, 10 and 11, abbreviated as SumInt, SumFloat and SumDouble. CUB achieves an impressive 92.4% of the peak device bandwidth for int and 81.5% for double on the GTX 470. The performance of Thrust is slightly lower at 73.8% for int and float. OpenACC is only able to achieve 23.7% for int and 45.2 for double, whereas the PENCIL reduction code generated by our framework achieves 90.2% of the peak. This shows the impact of the various optimizations described in the Section 5. There

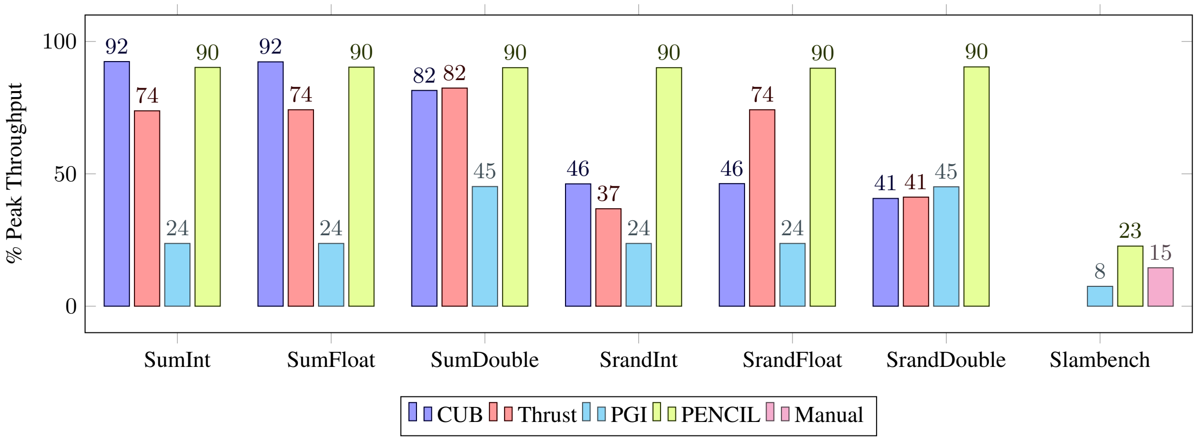

Figure 9: Performance on NVIDIA GeForce GTX 470

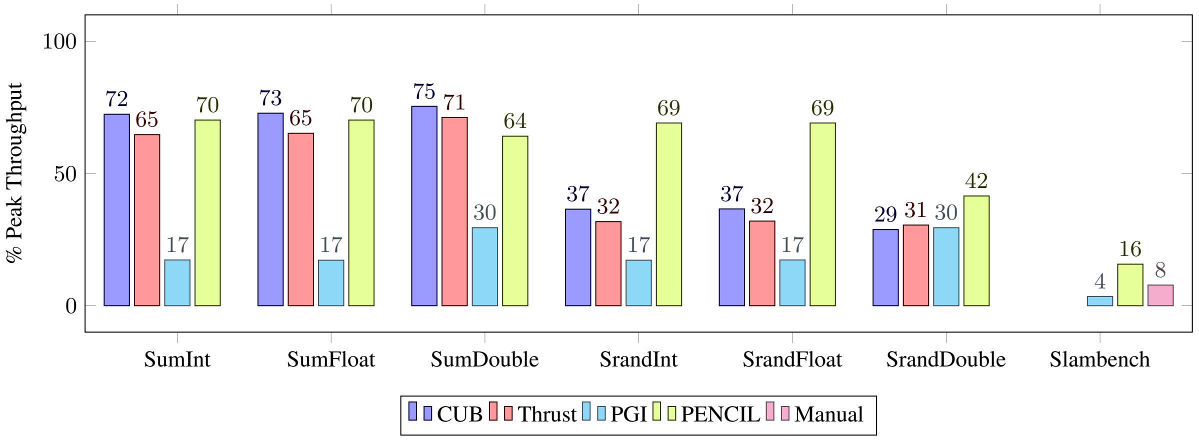

Figure 10: Performance on NVIDIA Quadro K4000

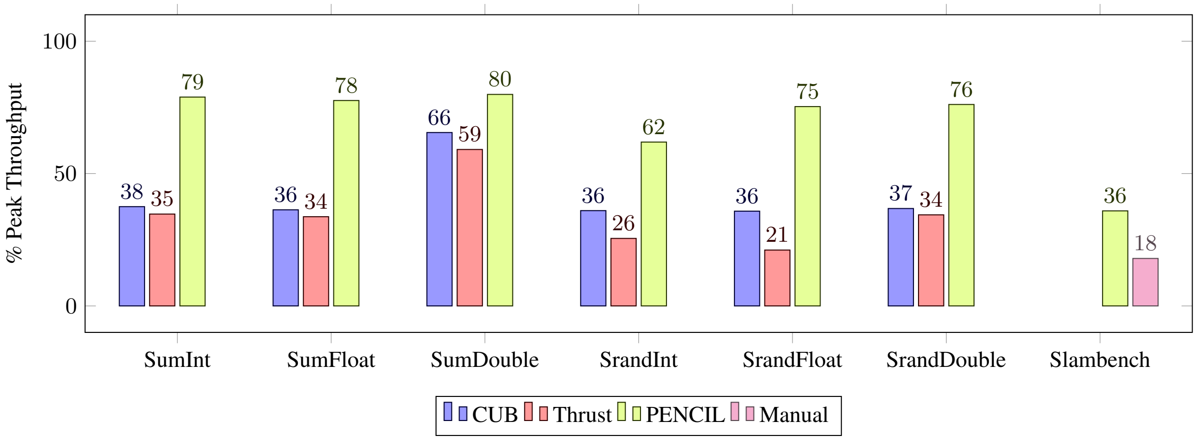

Figure 11: Performance on NVIDIA TK1

was a $0.53\times$ slowdown with and without shared memory access reordering. This optimization is essential to eliminate shared memory bank conflicts and is applicable only if the reduction operator is commutative. Hence we expect the reduction operator to be both associative and commutative.

The performance of CUB on the embedded TK1 is only 37.5% of peak for int and 65.5% for double, whereas the PENCIL version reaches 78.9% and 79.91%, respectively. We suspect that both CUB and Thrust libraries are not tuned for the TK1 platform, whereas our generated code benefited from auto-tuning. This shows the importance and effectiveness of auto-tuning even after all optimizations have been applied and a suitable template is being used. Table 1 collects the optimal values for the number of thread blocks and the number of threads per block. The number of reductions per thread depends on the problem size, grid size and block size. This parameter is tuned implicitly while tuning for the optimal grid and block size parameters. Note that the exhaustive search does not take much time because of the limited search space for reduction templates.

Benchmark	GTX 470	Quadro K4000	TK1
SumInt	84, 128	128, 128	222, 128
SumFloat	84, 128	128, 128	103, 512
SumDouble	120, 256	124, 64	253, 256

Table 1: Optimal parameter values for the number of thread blocks and for the the thread block size

Benchmark	OpenACC	PENCIL
GTX 470	0.51×	1.53×
Quadro K4000	0.45×	2.00×
TK1	–	1.97×

Table 2: Speedup of the SLAMBench reduction kernel relative to the manual implementation

6.2 Srand Reduction kernel

The Srand reduction kernel shown in Figure 4 consists of two reductions on the same array. It is quite straightforward to express such reductions in OpenACC and PENCIL, whereas CUB or Thrust involve two library calls along with some additional processing. One reduction call to compute sum, an intermediate step to compute the square of the input, and then another reduction call to compute the sum of squares. This is clearly inefficient as input array is traversed twice to perform reductions separately. This kernel illustrates the limitations of library-based approaches in real world applications: Figures 9, 10 and 11 show the performance of the Srand kernel abbreviated as SrandInt, SrandFloat and SrandDouble on different GPU architectures. The performance of PENCIL is almost identical to the case of a single reduction kernel. Because the array is scanned only once, reduction time does not vary much. On the other hand, for both CUB and Thrust, the time taken is twice that of a single reduction because of the two array traversals. This kernel illustrates the benefits of fusing multiple reductions, as evaluating multiple reduction operators per memory access made much more effective use of the bandwidth. The OpenACC compiler was also able to fuse the two reductions because both reductions appeared in the same loop. This restriction does not apply to our framework, which supports the fusion of different reduction kernels even if they are two separate loops in the input code.

6.3 SLAMBench reduction kernel

SLAMBench [17] is a computer vision benchmark for 3D modeling and tracking. The benchmark contains a reduction kernel with 32 different reductions. It is straightforward to port these reductions in PENCIL and OpenACC whereas it is tedious and inefficient to do so in CUB and Thrust, as they involve multiple calls to the reduction API. The comparison of our framework with OpenACC and with SLAMBench's manual CUDA implementation is shown in Figures 9, 10 and 11. Because there are 32 reductions performed for a single element, the memory bandwidth is no longer the bottleneck. PENCIL outperforms both OpenACC and the manual implementation. The latter is almost twice as fast as OpenACC. The SLAMBench implementation uses shared memory to store intermediate reduction values while reductions are performed in parallel. In PENCIL, the reduction kernel uses all the optimizations listed in Section 5 for all 32 reductions and auto-tuning results in a 2× improvement on NVIDIA TK1.

7. CONCLUSION

We presented language constructs and compilation methods to express arbitrary reductions on user-defined data types. We also presented dependence-based abstractions of reductions that enable polyhedral loop nest optimizations in the presence of loops carrying reductions. Combining polyhedral and template-based code generation, we are able to perform complex optimizations for user-defined reductions on GPUs, reaching close to peak performance. This approach enables a generic polyhedral compiler to produce highly efficient code for reductions while offering maximum expressiveness to the programmer. We demonstrated the approach on generalized reduction patterns and we believe it may be extended to prefix scans and radix sort algorithms.

Acknowledgments. This work was partly supported by the European Commission and French Ministry of Industry through the ECSEL project COPCAMS id. 332913, and by the French ANR through the European CHIST-ERA project DIVIDEND. We also acknowledge the support of ARM through the Polly Labs initiative. Our experiments with SLAMBench k-fusion benefited from numerous exchanges with Bruno Bodin, Luigi Nardi and Ali Zaidi, with whom we are also working on a PENCIL release of the application. And finally, none of this work would have been possible without the tremendous work of Sven Verdoolaege and Tobias Grosser on isl and PPCG.

8. REFERENCES

[1] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, J. Absar, S. van Haastregt, A. Kravets *et al.*, "PENCIL: A platform-neutral compute intermediate language for accelerator programming," *Proc. Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[2] R. Baghdadi, A. Cohen, T. Grosser, S. Verdoolaege, A. Lokhmotov, J. Absar, S. Van Haastregt, A. Kravets, and A. Donaldson, "PENCIL Language Specification," INRIA, Research Report RR-8706, May 2015. [Online]. Available: https://hal.inria.fr/hal-01154812

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *ACM SIGPLAN conference on Programming Language Design and Implementation*, vol. 43, no. 6. ACM, 2008, pp. 101–113.

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[5] S. J. Deitz, B. L. Chamberlain, and L. Snyder, "High-level language support for user-defined reductions," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 23–37, 2002.

[6] J. Doerfert, K. Streit, S. Hack, and Z. Benaissa, "Polly's polyhedral scheduling in the presence of reductions," *arXiv preprint arXiv:1505.07716*, 2015.

[7] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other Cilk++ hyperobjects," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA. New York, NY, USA: ACM, 2009, pp. 79–90. [Online]. Available: http://doi.acm.org/10.1145/1583991.1584017

[8] G. Gupta and S. V. Rajopadhye, "Simplifying reductions." in *ACM Symposium on Principles of Programming Languages (POPL)*, vol. 6, 2006, pp. 30–41.

[9] L. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. J. Kelly, "Towards metaprogramming for parallel systems on a chip," in *Proceedings of the 2009 International Conference on Parallel Processing*, ser. Euro-Par. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 36–45. [Online]. Available: http://dl.acm.org/citation.cfm?id=1884795.1884803

[10] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: An overview of the pips project," in *ACM International Conf. on Supercomputing (ICS)*, Cologne, Germany, Jun. 1991.

[11] ISO, "The ANSI C standard (C99)," ISO/IEC, Tech. Rep. WG14 N1124, 1999.

[12] P. Jouvelot, "Parallelization by semantic detection of reductions," in *ESOP 86*. Springer, 1986, pp. 223–236.

[13] P. Jouvelot and B. Dehbonei, "A unified semantic approach for the vectorization and parallelization of generalized reductions," in *Proceedings of the 3rd international conference on Supercomputing*. ACM, 1989, pp. 186–194.

[14] Mark Harris, "Optimizing parallel reduction in CUDA." [Online]. Available: https://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf

[15] Microsoft, "Parallel patterns library." [Online]. Available: https://msdn.microsoft.com/en-us/library/dd470426.aspx#parallel_reduces

[16] MPIF, "MPI-2: Extensions to the message-passing interface," *University of Tennessee, Knoxville*, 1996.

[17] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. J. Kelly, A. J. Davison, M. Luján, M. F. P. O'Boyle, G. D. Riley, N. Topham, and S. Furber, "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM," *CoRR*, vol. abs/1410.2167, 2014. [Online]. Available: http://arxiv.org/abs/1410.2167

[18] Nvidia, "CUB's collective primitives." [Online]. Available: https://nvlabs.github.io/cub/

[19] Nvidia, "Thrust C++ library." [Online]. Available: https://developer.nvidia.com/thrust/

[20] Nvidia forum, "Faster parallel reductions on Kepler." [Online]. Available: https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/

[21] S. S. Pinter and R. Y. Pinter, "Program optimization and parallelization using idioms," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 305–327, 1994.

[22] B. Pottenger and R. Eigenmann, "Idiom recognition in the polaris parallelizing compiler," in *Proceedings of the 9th international conference on Supercomputing*. ACM, 1995, pp. 444–448.

[23] W. Pugh and D. Wonnacott, "Static analysis of upper and lower bounds on dependences and parallelism," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 4, pp. 1248–1278, 1994.

[24] L. Rauchwerger and D. A. Padua, "The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 2, pp. 160–180, 1999.

[25] X. Redon and P. Feautrier, "Detection of recurrences in sequential programs with loops," in *Parallel Architectures and Languages Europe (PARLE)*. Springer, 1993, pp. 132–145.

[26] X. Redon and P. Feautrier, "Scheduling reductions," in *Proceedings of the 8th international conference on Supercomputing*. ACM, 1994, pp. 117–125.

[27] X. Redon and P. Feautrier, "Detection of scans," *J. of Parallel Algorithms and Applications*, vol. 15, no. 3-4, pp. 229–263, 2000.

[28] J. Reinders, *Intel Threading Building Blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[29] M. C. Rinard and M. S. Lam, "The design, implementation, and evaluation of Jade," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 3, pp. 483–545, May 1998. [Online]. Available: http://doi.acm.org/10.1145/291889.291893

[30] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 65–76.

[31] T. Suganuma, H. Komatsu, and T. Nakatani, "Detection and global optimization of reduction operations for distributed parallel machines," in *Proceedings of the 10th international conference on Supercomputing*. ACM, 1996, pp. 18–25.

[32] The Portland Group, "PGI accelerator compilers with OpenACC directives." [Online]. Available: http://www.pgroup.com/resources/accel.htm/

[33] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, "Non-affine extensions to polyhedral code generation," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 185.

[34] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization (TACO)*, Jan. 2013, selected for presentation at the HiPEAC 2013 Conference.

[35] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC: first experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 859–870.

[36] D. N. Xu, S.-C. Khoo, and Z. Hu, "Ptype system: A featherweight parallelizability detector," in *Programming Languages and Systems*. Springer, 2004, pp. 197–212.