PENCIL: a Platform-Neutral Compute Intermediate Language for Accelerator Programming

Riyadh Baghdadi Ulysse Beaugnon
Albert Cohen Tobias Grosser
Michael Kruse Chandan Reddy
Sven Verdoolaege
INRIA
first.last@inria.fr

Adam Betts Alastair F. Donaldson
Jeroen Ketema
Imperial College London
a.betts@imperial.ac.uk
alastair.donaldson@imperial.ac.uk
j.ketema@imperial.ac.uk

Javed Absar Sven Van Haastregt
Alexey Kravets Anton Lokhmotov
ARM
first.last@arm.com

Robert David Elnar Hajiyev
RealEyes
robert.david@realeyesit.com elnar@realeyesit.com

Abstract

Programming accelerators such as GPUs with low-level APIs and languages such as OpenCL and CUDA is difficult, error prone, and not performance-portable. Automatic parallelization and domain specific languages (DSLs) have been proposed to hide this complexity and to regain some performance portability. We present PENCIL, a rigorously-defined subset of GNU C99 with specific programming rules and few extensions. Adherence to this subset and the use of these extensions enable compilers to exploit parallelism and to better optimize code when targeting accelerators. We intend PENCIL both as a portable implementation language to facilitate the acceleration of applications, and as a tractable target language for DSL compilers.

We validate the potential of PENCIL as a front-end to a state-of-the-art polyhedral compiler, extending the applicability of the compiler to dynamic, data dependent control flow and non-affine array accesses. To this end, we have the polyhedral compiler generate highly optimized OpenCL code for a set of standard benchmark suites (Rodinia and SHOC), image processing kernels, and DSL embedding scenarios for linear algebra (BLAS) and signal processing radar applications (SpearDE). To assess *performance portability*, we present experimental results on four GPU platforms: AMD Radeon HD 5670 and Radeon R9 285, Nvidia GTX470, and ARM Mali-T604 GPU.

1. Introduction

The use of special-purpose accelerators such as GPUs can be more appealing than the use of general-purpose processors due to their performance and energy efficiency. Software for such accelerators is currently written using low-level APIs, such as OpenCL [22] and CUDA [17]. These low-level programming models require a high level of expertise to work with, are laborious and error-prone, and do not offer *performance portability*: the performance of an accelerated application may vary dramatically across platforms. These factors mean that there is a high cost associated with developing software at this level.

A compelling alternative for developers is to work with higher-level programming languages, and to leverage compilation technology to automatically generate efficient low level code. For general-purpose languages in the C family, this approach is hindered by the difficulty of static analysis in the presence of pointer aliasing. The possibility of aliasing often forces a parallelizing compiler to assume that it is *not* safe to parallelize a region of source code, even though aliasing might not actually occur at runtime. Domain-specific languages (DSLs) can circumvent this problem: it is often clear how parallelism can be exploited given high-level knowledge about standard operations in a given domain, such as linear algebra [4], image processing [21] or partial differential equations. The drawback of the DSL approach is the significant effort required to lower code all the way from the DSL level to highly optimized OpenCL or CUDA. The effort involved is even more significant if optimization is required for multiple platforms.

We present the design and implementation of PENCIL, a platform-neutral compute intermediate language. PENCIL aims to serve both as a portable implementation language to facilitate the acceleration of new and legacy applications on modern accelerators, and as a tractable target language for DSL compilers.

PENCIL is a rigorously-defined subset of GNU C99, and enforces a set of coding rules principally related to restricting the manner in which pointers can be manipulated. These restrictions make PENCIL code "static analysis-friendly": the rules are designed to enable a compiler to perform better optimization and parallelization when translating PENCIL to a lower-level formalism such as OpenCL. PENCIL is also equipped with specific language constructs, including *assume predicates* and *side effect summaries* for functions, that enable communication of domain-specific information to the PENCIL compiler, to be used for optimization.

Because it is based on C, the learning curve for PENCIL is gentle. By design, PENCIL interfaces with non-PENCIL C code, so that legacy C applications can be incrementally ported into PENCIL. From the point of view of DSL compilation, PENCIL offers a tractable target because all a DSL-to-PENCIL compiler must do is faithfully encode the semantics of the input DSL program into PENCIL; auto-parallelization and optimization for multiple accelerator targets is then taken care of by the downstream compiler. Because DSL-to-PENCIL compilers have tight control over the code they generate, such compilers can aid the effectiveness of the downstream PENCIL compiler by careful generation of code, and by

communicating domain-specific information via the language constructs PENCIL provides for this purpose.

We demonstrate the capabilities of PENCIL and its novel static analysis-friendly features in a state-of-the-art polyhedral compilation flow, extended with a PENCIL front-end and implementing advanced combinations of loop and data transfer optimizations. We illustrate this flow on irregular, data-dependent control and dataflow, generating efficient CUDA and OpenCL code on a variety of targets. This is the first time a fully-automatic polyhedral compilation flow is capable of parallelizing a variety of real-world, non-static-control applications.

We evaluate PENCIL by considering hand-written benchmark suites and code generated by DSL-to-PENCIL compilers:

- an image processing benchmark suite containing seven kernels written in PENCIL and covering computationally intensive parts of a computer vision stack used by *RealEyes*, a leader in the automatic recognition of facial emotions and eye tracking[1];
- six benchmarks extracted from the SHOC [11] and the Rodinia [8] benchmark suites re-written in PENCIL;
- six kernels generated using the VOBLA linear algebra DSL compiler [4];
- two signal processing radar applications generated from the SpearDE streaming DSL and modeling environment [14].

To assess performance portability, we present an experimental evaluation in which we target four GPU platforms: AMD Radeon HD 5670 and Radeon R9 285, Nvidia GTX470, and ARM Mali-T604. The performance gains compared to the implementation efforts for these applications and benchmarks are very encouraging. For example, for the VOBLA linear algebra DSL, we were able to generate code that has performance close to the cuBlas [18] and clMath [10] BLAS linear algebra libraries [13]. For the RealEyes image processing benchmark, we were able to match and sometimes outperform the OpenCV image processing library [19].

In summary, our main contributions are:

- PENCIL, a platform-neutral compute intermediate language for direct accelerator programming and DSL compilation;
- a polyhedral compilation framework that leverages the features of PENCIL to handle applications that do not fit in the classical restrictions of the polyhedral model, including forms of dynamic, data-dependent control flow and array accesses;
- the evaluation of PENCIL on multiple GPUs and on several real-world, non-static-control applications that were previously out of scope for polyhedral compilation.

2. Overview of PENCIL

PENCIL is a subset of the C99 language carefully designed to capture static properties essential to the implementation of advanced loop nest transformations. It provides a set of language constructs that helps parallelizing compilers to perform more accurate static analyses and to generate efficient target-specific code. These specific constructs provide information that is difficult for a compiler to extract but that can be easily captured from a DSL, or expressed by an expert programmer. Our aim was for PENCIL to be a strict subset of C99. Where necessary, we have exploited the flexibility of GNU extensions to C99 such as type attributes, and pragmas when no alternative was available. The latter have been inspired by standard pragmas used as annotations for SIMDization and thread-level parallelism, but retain strictly sequential semantics in PENCIL.

PENCIL is not coupled to any particular parallelizing compiler or target language. However, we validated PENCIL using a polyhedral compilation toolchain targeting OpenCL, and will thus refer to (a generalized form of) polyhedral compilation generating OpenCL code when discussing the implementation of PENCIL.

2.1 Design Goals

We designed PENCIL with four main goals in mind:

Ease of analysis. The language should simplify static code analysis, to enable a high degree of optimization. The main impact of this is that the use of pointers is disallowed, except in specific cases.

Support for domain-specific information. PENCIL should provide facilities that allow a domain expert or a DSL-to-PENCIL compiler to convey, in PENCIL, domain-specific information that can be exploited by the PENCIL compiler during optimization. For example, PENCIL should allow the user to indicate bounds on array sizes, enabling placement of arrays in the shared memory of a GPU.

Portability. A standard non-parallelizing C99 compiler that supports GNU C attributes should be able to compile PENCIL. This ensures portability to platforms without OpenCL support and allows existing tools to be used for debugging (unparallelized) PENCIL code.

Sequential Semantics. We chose a sequential semantics for PENCIL in order to simplify DSL compiler development and the work of a domain expert directly developing in PENCIL, and more importantly, to avoid committing to any particular pattern(s) of parallelism.

The design of the extensions to C99 that are a part of PENCIL took place in two phases. First, numerous DSLs (and benchmarks) were analyzed, and based on this analysis, a list of the properties that are expressed in these DSLs was created. This list was then filtered and only a few properties were kept and became language constructs in PENCIL. The decision of which properties to include in PENCIL was guided by the principle that all domain-specific optimizations should be performed at the DSL compiler level, while the PENCIL compiler should be responsible only for parallelization, data locality optimization, loop nest transformations, and mapping to OpenCL. This separation means that, in PENCIL, only the properties that are necessary to improve static analysis and GPU mapping are needed. Any domain-specific property that is not necessary for the PENCIL compiler does not have to be conveyed through PENCIL and thus should not be a part of the PENCIL language. This choice has the advantage of keeping PENCIL general-purpose, sequential and lightweight.

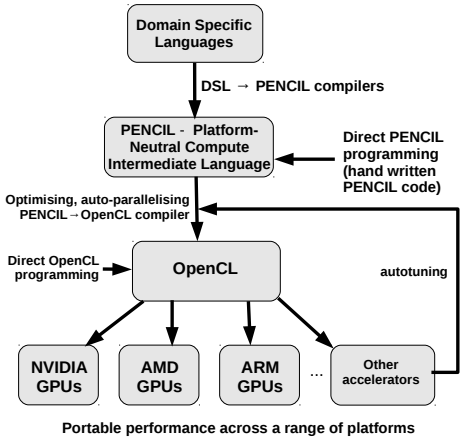

Figure 1: A high level overview of the PENCIL compilation flow

Figure 1 shows a high level overview of a typical PENCIL usage scenario. First, a program written in a DSL is translated into PENCIL. Domain specific optimizations are applied during

[1] http://www.realeyesit.com

this translation. Second, the generated PENCIL code is combined with hand-written PENCIL code that implements library functions. PENCIL is used here as a standalone language. The combination of the two pieces of code is then optimized and parallelized (in this paper, we use a polyhedral framework for this purpose). Finally, highly specialized OpenCL code is generated. The generated code is tuned through profiling-based iterative compilation and auto-tuning.

2.2 PENCIL Coding Rules

We detail the most important restrictions imposed by PENCIL from the point of view of enabling GPU-oriented compiler optimizations. The PENCIL specification [3] contains the rules in full.

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 function calls are not allowed, because accelerator programming languages such as OpenCL forbid this.

Sized, non-overlapping arrays. Arrays must be declared using the C99 variable-length array syntax [12]; array function arguments must be declared using `pencil_attributes`, a macro expanding to the `restrict` and the `const` C99 type qualifiers and to the `static` C99 keyword. During optimization, the PENCIL compiler thus knows the length of arrays, and that arrays do not overlap.

Structured `for` loops. A PENCIL `for` loop must have a single iterator, an invariant start value, an invariant stop value and a constant increment (step). Invariant in this context means that the value does not change in the loop body. By precisely specifying the loop format we avoid the need for a sophisticated induction variable analysis. Such an analysis is not only complex to implement, but more importantly results in compiler analyses succeeding or failing under conditions unpredictable to the user.

An additional programming guideline (which is not mandatory as it cannot be statically checked in general) is that array accesses should not be linearized. Linearization tends to obfuscate affine subscript expressions, hindering the effectiveness of the PENCIL compiler. Multidimensional C99 arrays should be used instead.

PENCIL also supports the OpenCL scalar builtin functions such as `abs`, `min`, `max`, `sin`, `cos` and `log`, using a target-independent and explicitly typed naming scheme (e.g., prefixes to differentiate float and double).

The main constructs introduced by PENCIL include the assume builtin function, the `independent` directive, summary functions and the kill builtin function. They are described below.

2.3 PENCIL Assume

An intrinsic function, `__pencil_assume`(e), where e is a logical expression, indicates that e is guaranteed to hold at a given program point. This knowledge is taken on trust by the PENCIL compiler, and may enable generation of more efficient code. In the context of DSL compilation, an assume statement allows a DSL-to-PENCIL compiler to communicate high level facts in the generated code. The truth of an expression e appearing in an assume statement is *not* checked at runtime (support for runtime checking could be optionally provided for debugging purposes).

The *general 2D convolution* example of Figure 2 illustrates 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. This kernel is part of an image processing benchmark written by RealEyes, and is presented in detail in Section 4.1.

```
1  #define clampi(val, min, max) \
2    (val < min) ? (min) : (val > max ) ? (max):(val)
3
4  __pencil_assume(ker_mat_rows <= 15);
5  __pencil_assume(ker_mat_cols <= 15);
6
7  for (int i = 0; i < rows; i++)
8    for (int j = 0; j < cols; j++) {
9      float prod = 0.;
10     for (int e = 0; e < ker_mat_rows; e++)
11       for (int r = 0; r < ker_mat_cols; r++) {
12         row = clampi(i+e-ker_mat_rows/2, 0, rows-1);
13         col = clampi(j+r-ker_mat_cols/2, 0, cols-1);
14         prod += src[row][col] * kern_mat[e][r];
15       }
16     conv[i][j] = prod;
17   }
```

Figure 2: General 2D convolution

It is sufficient, for RealEyes' requirements in production, to consider that the size of the array `kern_mat` does not exceed 15×15, as conveyed by the assume statements.

While well known to image processing experts, the compiler does not have this knowledge and must assume that the kernel matrix can be arbitrarily large. When compiling for a GPU target the compiler must thus allocate the kernel matrix in GPU global memory, rather than in fast shared memory, or must generate multiple variants of the kernel – one to handle large kernel matrix sizes and another optimized for smaller kernel matrix sizes – selecting between variants at runtime. The use of `__pencil_assume` tells the compiler about the limits on the size of the array, allowing it to store the whole array in shared memory.

2.4 Independent Directive

The `independent` directive is used to annotate loops, and is semantically similar to the High Performance Fortran directive of the same name [15]. 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 can be used to indicate that the marked loop does not have any loop carried dependence (i.e., it could be run in parallel).

The `independent` directive can also be used when some dependences exist but the user nonetheless wants to ignore them. In such cases the execution order of the data accesses may have to be constrained using specific synchronization constructs. Examples include reductions implemented via atomic regions, and the use of low-level atomics to give semantics to so-called "benign races", where the same value is written to a location by multiple threads in parallel. In general, it can sometimes be necessary to invoke external non-PENCIL functions when parallelizing an algorithm that can tolerate arbitrarily-ordered execution of intermediate steps.

The `independent` directive has an effect only on the marked loop, not on any nested or outside loops. It accepts a `reduction` clause which, for brevity, we do not discuss here.

Figure 3 shows a code fragment of our PENCIL implementation of the breadth-first search benchmark from the Rodinia [8] benchmark suite. This benchmark computes the minimal distance from a given source node to each node of the input graph. The algorithm maintains a frontier and computes the next frontier by examining all unvisited nodes adjacent to the nodes of the current frontier. All nodes in a frontier have the same distance from the source node.

The `for` loop shown in Figure 3 can be parallelized since each node of the current frontier can be processed independently. This creates a possible race condition on the `cost` and `next_frontier` arrays. The race condition can be ignored, however, because each conflicting thread will write the same values. By specifying the

```
/* Examine nodes adjacent to current frontier */
#pragma pencil independent
for (int i = 0; i < n_nodes; i++) {
  if (frontier[i] == 1) {
    frontier[i] = 0;
    /* For each adjacent edge j */
    for (int j = edge_idx[i];
             j < edge_idx[i] + edge_cnt[i]; j++) {
      int dst_node = dst_node_index[j];
      if (visited[dst_node] == 0) {
        /* benign race: threads write same values */
        cost[dst_node] = cost[i] + 1;
        next_frontier[dst_node] = 1;
      }
    }
  }
}
```

Figure 3: PENCIL code fragment for breadth-first search

`independent` pragma, the programmer guarantees that the race condition is benign, enabling parallelization the loop.

2.5 Summary Functions

The effect of a function call on its array arguments is derived from an analysis of the called function. In some cases, the results of this analysis may be too inaccurate. In the extreme case, no code may be available for the function and the compiler can then only assume that every element of the passed arrays is accessed. In order to obtain more accurate information on memory accesses, the user may tell the compiler to derive the memory accesses not from the actual function body, but from some other function with the same signature. Such a function is called a *summary function*.

In practice, summary functions are used to describe the memory access patterns of:

- **library functions** called from PENCIL code, for which source code is not available for analysis;
- **non-PENCIL functions** called from PENCIL code, as they are otherwise difficult to analyze.

The use of summary functions enables more precise static analysis. The accesses to each array passed as argument to the function must be described by the summary function. To indicate the summary function of a function `foo()`, one uses the attribute `pencil_access(summary)`, where `summary` is the name of the summary function that describes the memory accesses in `foo()`. The summary function is not meant to be executed, and is instead only used for the analysis of memory footprints. Each and every array element accessed in a function should be accessed in its summary. Yet a summary is generally simpler than the function it summarizes: it only captures sets of accesses, not their ordering and number of occurrences.

The builtin functions `__pencil_use` and `__pencil_def` are designed to be used in summary functions to mark memory accesses. A `__pencil_use(A[`e`])` annotation indicates that a read *may* occur from array `A` at index e, while a `__pencil_def(A[`e`])` annotation indicates that a write *must* occur to array `A` at index e. In the case of writes, it can also be useful to communicate *may* information. This can be achieved using the `__pencil_maybe` construct, which has the semantics of evaluating to a nondeterministic Boolean value. It allows a single summary to carry both *must* and *may* information. More specifically, the conditional

`if (__pencil_maybe) __pencil_def(A[`e`]);`

indicates that a write *may* occur to array `A` at index e. This nicely fits any static analysis capable of extracting *may* and/or *must* information from conditional expressions and is also consistent with the usage of wildcards in intermediate verification languages.[2]

[2] http://research.microsoft.com/en-us/projects/boogie

```
__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 4: Example code extracted from Adaptive Beamformer, illustrating the use of summary functions

Figure 4 shows a loop nest extracted from the *ABF* benchmark presented in Section 4.4. The code calls the function `fft32` (Fast Fourier Transform). This function only reads and modifies (in place) 32 elements of its input array `in`, it does not modify any other parts of the input array. Such a function is not analyzed by the PENCIL compiler as it is not a PENCIL function. Without a summary function the compiler conservatively assumes that the whole array passed to `fft32` is accessed for reading and writing. Such a conservative assumption prevents parallelization. The use of a summary function in this case indicates to the compiler that each iteration of the loop nest reads and writes 32 elements of the input array, allowing for parallelization of the loop nest.

Writing the summaries of library functions is the library developer's responsibility. It is also the most common use case. Such summary functions should be provided in the library's header files and are used directly by the DSL compilers or PENCIL programmers. In other less common cases, summary functions are either written by the PENCIL programmer or generated automatically by the DSL compiler (only the sets of read and written elements for each function argument need to be provided in this case).

2.6 PENCIL Kill

The `__pencil_kill` builtin function allows the user to refine dataflow information within and across any control flow region in the program. It is a polymorphic function that signifies 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 from any statement instance executed before the kill to any statement instance executed after.

This information is used in several ways, as explained in detail in [24]. The effect of the `kill` builtin is illustrated on the following example code:

```
__pencil_kill(A);
for (int i = 0; i < n; i++) {
  if (B[i] > 0)
    A[i] = B[i];
}
```

If this loop is mapped to a device kernel, then the `A` array needs to be copied out from the device to the host after the computation because some elements of `A` may be written inside the loop. This copy-out overwrites the original contents of `A` on the host. Since not all elements may be written by the loop, the array would in principle also need to be copied in first to ensure that after the copy-out, the elements not written by the loop retain their original values. The `__pencil_kill(A)` statement is used to indicate that the data

in A is not expected to be preserved by the region and that therefore this copy-in can be omitted.

3. Polyhedral Compilation of PENCIL Code

We now explain how specific PENCIL features can be compiled using a polyhedral compiler, although PENCIL itself is not tied to any particular compilation technique.

In polyhedral compilation, an abstract mathematical representation is used to model the program. Each statement in the program is represented using three pieces of information: an *iteration domain*, *access relations* and a *schedule*. This representation is first extracted from the program AST, it 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. For example, the statement in Line 9 in Figure 2 has the following iteration domain (let us call the statement S_0): { S_0(i,j) : 0 ≤ i < rows and 0 ≤ j < cols}. A quasi-affine constraint is a constraint over integer values and integer variables involving only the operators +, −, ∗, /, %, &&, ||, <, <=, >, >=, ==, != or the ternary ?: operator. The second argument of the / and the % operators is required to be a (positive) integer literal, while at least one of the arguments of the ∗ operator is required to be a piece-wise constant expression. An example of a quasi-affine constraint used in a loop nest is $10 * i + j + n > 0$ 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 $i * i > 0$ and $n * i > 0$.

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

In this paper, we used an existing polyhedral compiler for GPUs, PPCG [26], and extended it to handle PENCIL code. PPCG relies on the pet library [25] to extract the iteration domain and the access relations while the dependence analysis is performed by the isl library [23]. A new schedule is computed by isl using a variant of the Pluto algorithm [5] (the computation of a new schedule is the step where most loop nest transformations are applied).

We made several changes to this flow to support PENCIL applications. For a detailed description of these changes, including support for arrays of structures, we refer to [24].

Assume builtin. The pet library already keeps track of constraints on the symbolic constants of the program (variables that have an unknown but fixed value throughout the execution). These constraints are automatically derived from array declarations and index expressions. In particular, the values of the symbolic constants that necessarily result in negative array sizes or negative array indices are excluded (negative indices are not allowed because they could result in aliasing within an array). The constraints are used during the generation of an AST from a schedule to simplify the generated AST expressions.

__pencil_assume allows the user to provide additional constraints on the symbolic constants of the program that cannot be derived automatically from the code, For example, Line 4, and Line 5 in Figure 2 provide additional constraints on the symbolic constants ker_mat_rows and ker_mat_cols that are used throughout code generation whenever needed.

Kill builtin. A kill statement in pet represents the fact that no dataflow on the killed data elements can pass through any instance of the kill statement. This information can be used during dataflow analysis to stop the search for potential sources on data elements killed by the statement.

Whenever pet comes across a variable declaration, two kill statements that kill the entire array are introduced, one at the location of the variable declaration and one at the end of the block that contains the variable declaration. A user can introduce explicit kills by adding a __pencil_kill.

Non static-affine array accesses. In order to handle accesses that may not be static-affine, pet has been modified to make a distinction between *may*-writes and *must*-writes. Any index expression that cannot be statically analyzed or that is not affine, is treated as *possibly* accessing any index. This typically results in more dependences as more pairs of statement instances may possibly access the same array element.

Non static-affine conditionals and loop bounds, while loops, break and continue. A non static-affine conditional or a loop with non static-affine loop bounds is treated by PPCG as follows: the statement and its body are all treated as one macro-statement (i.e., as one statement that encapsulates the control statement and its body). Any write inside this macro-statement is treated as a may-write. For example, the statement in Line 2 of Figure 5 is governed by the condition in Line 1 which cannot be analyzed. The if-statement and its body therefore are considered as one macro-statement and the access to sup is treated as a may-write.

```
1  if (se[e][r] != 0)
2    sup = max(sup, img[cand_row][cand_col]);
```

Figure 5: Code extracted from *Dilate* (image processing)

While-loops and loops containing a break or a continue statement are treated similarly: the loop and its body are treated as one macro-statement (one single statement). For example, due to the break statement in Line 7 in Figure 6, the whole loop in Line 3 is treated by PPCG as a single statement. This means that PPCG can schedule (i.e., change the order of execution of) the loop in Line 3 and its body as a whole, but cannot schedule the statements in the body individually.

```
1  for (int i = 0; i < N; i++)
2    for (int j = 0; j < M; j++)
3      for (int k = 0; k < M; k++) {
4        B[i][j][k] = 0;
5
6        if (A[i][j][k] == 0)
7          break;
8      }
```

Figure 6: Example of code containing a break statement

Handling while loops is currently very basic in PPCG. The current work does not have any contribution in that direction.

Independent directive. When the `independent` directive is used to annotate a loop, the iterations of that loop may be freely reordered with respect to each other, including reorderings that result in (partial) overlaps of distinct iterations. In particular, the user asserts through this directive that no dependences need to be introduced to prevent such reorderings. The directive assumes that a variable that is declared inside the loop is considered private to any given iteration. `pet` currently handles the `independent` directive by building a relation between the statement instances that are excluded from depending on each other, as well as the set of variables that are local to the marked loop. This set of local variables is used by `PPCG` to ensure that their live ranges do not overlap in any affine transformation in a way similar to [2], and to privatize them if needed when generating parallel code.

Summary functions. `pet` has been modified to extract access information from called functions. Whenever a summary function is provided, this information is extracted from the summary function instead of the actually called function.

4. PENCIL Evaluation

We evaluate the performance of OpenCL code generated from PENCIL using a development version of `PPCG` [26]. To show that PENCIL can be used as a standalone language as well as an intermediate language for DSL compilers, we present experimental results covering benchmark suites (written in PENCIL) and code generated by DSL compilers. The benchmark suites written in PENCIL are the RealEyes image processing benchmark suite (Section 4.1), and a selected set of benchmarks from Rodinia and SHOC (Section 4.2). The DSL compilers are the VOBLA DSL compiler (Section 4.3) and the SpearDE DSL compiler (Section 4.4).

The experiments evaluate whether PENCIL enables the parallelization (mapping to OpenCL) of kernels that cannot be parallelized with the current state-of-the-art polyhedral compilers (Pluto [5]). They also evaluate whether PENCIL enables the generation of more efficient code and how close the performance of the automatically generated code is to hand-crafted code. The experiments evaluate the whole PENCIL framework on a relatively large set of real world applications and test platforms.

We developed an autotuning compiler framework to facilitate the retargeting of our framework to very different GPU architectures. We only apply autotuning to the `PPCG`-generated code. For one, autotuning the reference code (which in mostly implemented as libraries) does not make sense because library code is not designed to be autotuned (for example, workgroup sizes are hardcoded in the libraries, the use of shared and private memory requires manual code modification of the kernels, etc.). Moreover, BLAS libraries (clMath [10] and cuBlas [18]) do not require autotuning because these libraries are already configured with a set of optimal parameters for their target architectures. Our autotuning framework searches for the most appropriate optimizations (compiler flags) by generating many different code variants and executing each of them on the target hardware. It searches through combinations of `PPCG`'s compiler flags that include the work group sizes, tile sizes, whether to use shared memory, whether to use private memory and which loop distribution heuristic to use (out of two possible heuristics). The autotuning of each benchmark suite takes several hours (except for the six kernels generated from VOBLA, which altogether take up to two days as the search space is larger).

The code generated by `PPCG` for a given kernel is optionally instrumented to measure the wall clock execution time of that kernel. This time includes kernel execution, data copy (between host memory and GPU device memory), and any kernel code executed on the host CPU. It does not include device initialization and release, nor kernel compilation time. This measured wall clock execution time is the time we report below. In order to exclude compilation time, we either invoke a dry-run computation that is not timed beforehand (caching the compiled kernels), or subtract the compilation time from the total duration, depending on how the reference benchmark compiles and invokes its kernels. We use OpenCL profiling tools to further analyze the performance of the reference code and the `PPCG` generated code (to get the number of cache misses, the number of device global memory accesses, the GPU occupancy, etc.). Each test is run 30 times and the median of the speedups over the reference benchmark is reported.

We use four GPU platforms for the experimental evaluation: an Nvidia GTX470 GPU (with AMD Opteron Magny-Cours, 2×12 cores and 16GB of RAM), an ARM Mali T604 GPU (with dual-core ARM Cortex-A15 CPU and 2 GB of RAM), an AMD Radeon HD 5670 GPU (with Intel Core2 Quad CPU Q6700 and 8 GB RAM) and an AMD Radeon R9 285 GPU (with Intel Xeon CPU E5-2640, 8 cores and 32 GB of RAM).

4.1 Image Processing Benchmark Suite

We studied a set of image processing kernels covering computationally intensive parts of a computer vision stack of RealEyes. The benchmark suite includes simple image filters as well as composite image processing algorithms. For each kernel in the benchmark suite, we compare a straightforward PENCIL implementation of the kernel (without any optimization), with a call to the equivalent kernel in the OpenCL implementation of the OpenCV image processing library [19].

The benchmark suite contains 7 image processing kernels: *affine warping*, *image resize*, *general 2D convolution*, *gaussian smoothing*, *color conversion*, *dilate* and *basic image histogram* (calculates the tonal distribution in an image).

One important characteristic of image processing kernels is that they contain non static-affine code: non static-affine array accesses, non static-affine if conditionals and non static-affine loop bounds that a classical polyhedral compiler does not handle efficiently since they do not fit the traditional restrictions of the polyhedral model. The conditional `if (se[e][r] != 0)` in Figure 5 is an example of such non static-affine code.

The benchmark exhibits many patterns of non static-affine code: 5 out of 7 kernels have non static-affine conditionals, 5 out of 7 kernels have non static-affine read accesses, 1 kernel has non static-affine write accesses. To be able to efficiently handle these kernels, a polyhedral compiler needs to be able to handle not only the non static-affine conditionals, and the non static-affine read accesses, but also the non static-affine write accesses. Write array accesses are more difficult to handle because they prevent the compiler, in general, from determining whether the loop is parallel or not.

Benchmark	support for non static-affine code	independent	assume	kill
resize	required	-	-	33% ↑
dilate	required	-	-	10% ↑
color conversion	-	-	-	34% ↑
affine warping	required	-	-	23% ↑
2D convolution	required	-	20% ↑	21% ↑
gaussian smoothing	required	-	-	47% ↑
basic histogram	-	required	-	-

Table 1: Effect of enabling support for individual PENCIL features on the ability to generate code and on gains in speedups

The kernels in this benchmark require the following PENCIL features: support for non static-affine code, `independent` directive, and the `__pencil_assume` and `__pencil_kill` builtins. Table 1 shows the list of PENCIL features that were useful in the image processing benchmark suite. It shows whether a given PENCIL feature is required for OpenCL code generation and shows the gain in speedup obtained when support for that feature is enabled (compared to the case where support for that feature is disabled). We only show the effect on performance on one test platform (Nvidia GTX), the effect on the other platforms is similar. The

symbol "-" indicates that the absence of the feature does not have any effect on the generated code.

The table shows that support for non static-affine code is required to be able to generate OpenCL code for 5 out of 7 kernels in the benchmark. In *basic histogram*, the use of the `independent` directive enables the parallelization and OpenCL code generation for the kernel which is difficult otherwise. In *dilate*, assuming that the size of the structuring element (the array that represents the neighborhood used to compute each pixel) is less than 16×16 enables PPCG to map that array to shared memory. Using this assumption allows PPCG to generate code that is 20% faster compared to the case where the assumption is not used. Using `__pencil_kill` allows PPCG to generate code that is 28% faster compared to the case where the `kill` builtin is not used. `__pencil_kill` in this case mainly eliminates extra data copies that PPCG generates to move data between host and device memories.

Table 2 shows the speedups of the PPCG generated OpenCL code over the baseline OpenCV 2.4.10 OpenCL implementation. We use the same image to evaluate all the kernels (2880×1607, 1.5 MB image).

Benchmark	Nvidia GTX 470	ARM Mali T604	AMD Radeon HD 5670	AMD Radeon R9 285
resize	1.00	1.25	2.47	8.09
dilate	0.59	0.32	0.25	2.91
color conversion	1.32	2.37	1.56	1.11
affine warping	1.06	1.93	2.44	2.85
2D convolution	0.91	-	0.95	2.53
gaussian smoothing	0.92	0.97	0.51	1.61
basic histogram	0.45	0.42	0.16	4.34

Table 2: Speedups of the OpenCL code generated by PPCG over OpenCV

`__pencil_kill` helps PPCG to eliminate spurious data copies but no other optimization on the data copies is applied by PPCG. In all the kernels, the amount of data copied by the PPCG generated code (when `__pencil_kill` is used) is exactly equal to the amount of data copied by the reference benchmark code. As a consequence, the speedups (or slowdowns) listed in the table are due to faster (or slower) kernel executions and not to a difference in data copy time. This is true for all the test platforms except for the AMD Radeon R9 285 platform. The particularity of this platform is discussed later in this section.

The speedup in *color conversion* and in *resize* for the Nvidia, ARM and AMD Radeon HD 5670 test platforms is due to the tiling of the 2D loop nest in each of these two kernels which enhances data locality considerably (up to 56% less L1 cache misses on the Nvidia platform for *color conversion*). In *affine warping*, the speedup is due to two optimizations: thread coarsening where multiple work-items are merged together leading to less redundant computations and tiling which enhances data locality (up to 65% less L1 cache misses on the Nvidia platform).

In the *basic histogram* kernel, the code automatically generated by PPCG still does not meet the performance of the hand optimized OpenCV implementation of the histogram for all the test platforms. The OpenCV code is faster because each workgroup computes its own local histogram placed in shared memory, and then the different local histograms are combined into one final histogram (a reduction). Automatic generation of OpenCL code that exploits this kind of reductions is not yet supported by PPCG.

For *dilate*, the OpenCV code is vectorized while the current PPCG OpenCL backend still does not support the generation of vectorized code. The lack of vectorization in the PPCG generated code affects the performance more on the AMD and the ARM test platforms. Moreover, in the OpenCV code for *dilate*, the input image array is mapped into shared memory while PPCG's shared memory heuristic decides not to map this array into shared memory. As a consequence, the PPCG generated code accesses global GPU memory $175\times$ more often compared to the OpenCV code, which leads to a decrease in performance. The same problem in the shared memory heuristic applies to *gaussian smoothing*.

While PPCG can generate code for *2D convolution*, the OpenCV reference implementation for *2D convolution* could not be run on the ARM Mali GPU, as it uses hardcoded shared memory and workgroup sizes that both exceed its limits.

On the AMD Radeon R9 285 platform, the speedups of the PPCG generated kernels over the OpenCV kernels are due to the slow data copies that OpenCV performs. The data copies are slower because OpenCV, on this platform, decides to add padding to the input image for aligned memory accesses. In order to do that, OpenCV uses the OpenCL `clEnqueueWriteBufferRect` function which copies data from host to device memory and adds padding at the same time. PPCG in contrast uses the `clEnqueueWriteBuffer` OpenCL function which only copies data from host to device memory. Using `clEnqueueWriteBufferRect` is $7\times$ slower than the use of `clEnqueueWriteBuffer`. This difference explains the high speedups that were obtained for the PPCG generated code on this platform. Other than this difference in data copies, there is no other significant difference between the speedups obtained on the AMD Radeon R9 285 platform and the AMD Radeon HD 5670 platform. Note that, although the use of `clEnqueueWriteBufferRect` may be less efficient in these tests, it may be more efficient in other cases where only one data copy is performed and many filters are applied on the same input image.

4.2 Rodinia and SHOC Benchmark Suites

When choosing benchmarks from the Rodinia [8] and SHOC [11] benchmark suites for writing in PENCIL (reverse-engineering from OpenCL to PENCIL), we decided to focus our resources on a selection of benchmarks that offer diversity (cover different Berkeley "motifs" [1] such as dense and sparse linear algebra, structured grids, and graph traversal), and pose a challenge to traditional polyhedral compilers arising from non static-affine code. We chose six benchmarks (presented in Table 3). Four of these benchmarks are particularly challenging for a polyhedral compiler as they exhibit patterns of non static-affine code. We show the benefit of using PENCIL to implement these benchmarks and compare the performance of the PPCG generated code for each benchmark with the reference Rodinia/SHOC implementations.

Benchmark	Suite	dataset sizes	Description / notes
2D Stencil	SHOC	100 iterations, 4096×4096 grid	On structured grid
Gauss. Elim.	Rodinia	1024×1024 matrix	Dense matrix
SRAD	Rodinia	100 iterations, 502×458 image	Image enhancement
SpMV	SHOC	16384 rows	Sparse matrix-vector multiplication
Radix Sort	SHOC	16777216 elements	Integer sorting
BFS	Rodinia	4 million nodes	Breadth-first search on a graph

Table 3: Selected benchmarks from Rodinia and SHOC

Benchmark	support for non static-affine code	independent
2D Stencil	-	-
Gauss. Elim.	-	-
SRAD	required	-
SpMV	required	-
Radix Sort	required	required
BFS	required	required

Table 4: PENCIL Features that are useful for SHOC and Rodinia benchmarks

The benchmarks require support for non static-affine code and the use of the `independent` directive. Table 4 shows the effect of these features on the ability of PPCG to generate OpenCL code. Other PENCIL features do not have any effect on these benchmarks.

The table shows that supporting non static-affine code is required for OpenCL code generation in 4 out of 6 benchmarks. The

non static-affine code patterns in these benchmarks include non-affine read accesses, non-affine conditionals and non-affine write accesses. The non-affine write accesses, in *BFS* and in *Radix Sort*, are particularly difficult to handle since they prevent the compiler from parallelizing the code, requiring the use of the `independent` directive.

Table 5 shows speedups. The speedups in *Stencil* and *Gaussian* are mainly due to tiling which enhances data locality and reduces cache misses (4× less L1 cache misses for *Stencil* on Nvidia GTX 470). For *SRAD*, the PENCIL-generated OpenCL code is significantly slower than the reference benchmark, mainly because `PPCG` did not map a reduction in *SRAD* to OpenCL (as `PPCG` does not support the generation of parallel reductions yet). This leads to unnecessary data transfers between the part of *SRAD* mapped to host and the part of *SRAD* mapped to device hence the slowdown. In *BFS*, the generated OpenCL code is slightly slower than the reference code also due to unnecessary data transfers that `PPCG` generates. This happens because `PPCG` does not handle while loops yet (currently only the while loop body is mapped to GPU, which makes `PPCG` generate a data copy between host and device at the beginning and end of each while loop iteration).

Benchmark	Nvidia GTX 470	ARM Mali T604	AMD Radeon HD 5670	AMD Radeon R9 285
2D Stencil	3.44	3.04	2.68	5.76
Gauss. Elim.	0.67	1.54	4.39	2.58
SRAD	0.22	0.34	0.43	0.56
SpMV	1.17	1.67	1.04	1.08
Radix	x	0.06	x	x
BFS	0.65	0.78	0.43	0.72

Table 5: Speedups for the OpenCL code generated by `PPCG` for selected Rodinia and SHOC benchmarks

Rodinia and SHOC as well as the previously analyzed image processing benchmarks are examples of the use of PENCIL as a standalone language. In the next two sections, we show two other examples where PENCIL is used as an intermediate language for DSL compilers.

4.3 VOBLA DSL for Linear Algebra

VOBLA is a domain specific language designed by ARM for implementing linear algebra algorithms [4]. It provides a compact and generic representation for linear algebra algorithms using an imperative programming style.

The main control flow operators defined in VOBLA include `if`, `for`, `forall` and `while`. The `if` and the `while` operators have semantics similar to their counterparts in C and PENCIL. The `for` and `forall` operators are used to iterate over a scalar range (using a multi-dimensional iteration space) or over an array. `forall` loops are used to indicate that the iterations of a loop can be executed in any order.

We only provide a brief description of VOBLA and its compilation into PENCIL. A detailed description is available in [4].

The VOBLA-to-PENCIL compiler is quite simple. It does not perform any sophisticated analyses or optimizations. Advanced loop nest optimizations and the mapping to OpenCL code are all handled by the PENCIL compiler. The VOBLA-to-PENCIL compiler generates only the following PENCIL features: the `independent` directive, the `assume` builtin and the `restrict` type qualifier. No other PENCIL feature needs to be generated. The `kill` builtin is only useful to eliminate spurious data copies in non-static control code and is not needed for the purely static control code generated by VOBLA. Summary functions are only needed when library functions are called from PENCIL which is not the case in the VOBLA generated code.

The `independent` directive, the `assume` builtin and the `restrict` type qualifier are generated in the following way:

- `forall` VOBLA operators are translated into `for` loops that are annotated with the `independent` directive.
- `assume` builtins are inferred from the relations between array sizes in the VOBLA program. Such information allows the PENCIL compiler to simplify the generated code and avoid unnecessary checks.

  For example, thanks to the statement `C = A + B`, the compiler can infer that the sizes of `A` and `B` are equal and can generate a `__pencil_assume` to indicate that. This information enables the PENCIL compiler to avoid handling the case where `(size_A != size_B)`. This information is useful for instance if the PENCIL compiler decides to fuse two loops over `A` and `B`.
- The VOBLA compiler annotates all the arrays with the `restrict` type qualifier in the PENCIL generated code. This is correct because of the way VOBLA is designed. In VOBLA, the only way to create aliasing is by calling a function and passing the same array to that function more than once in the same call. But even in this case, no aliasing heppens in the generated pencil function because the compiler creates a new version of that function where the different arrays that actually represent the same array are merged into a single array.

We used VOBLA to implement a set of linear algebra kernels including *gemver* (vector multiplication and matrix addition), *2mm* (2 matrix multiplications), *3mm* (3 matrix multiplications), *gemm* (general matrix multiplication), *atax* (matrix transpose and vector multiplication) and *gesummv* (scalar, vector and matrix multiplication). Many of these kernels are a sequence of BLAS function calls.

We compare the code generated from `PPCG` for these kernels with equivalent code that calls BLAS library functions. The VOBLA implementation is first compiled to PENCIL using the VOBLA-to-PENCIL compiler and then PENCIL is mapped to the GPU using `PPCG`. We compare the generated code with two highly optimized BLAS library implementations.

- We use the clMath 2.2.0 [10] BLAS library provided by AMD for comparison on AMD platforms.
- We use the cuBlas 5.5 [18] BLAS library provided by Nvidia for comparison on the Nvidia platform. In this case we use `PPCG` to generate CUDA code instead of OpenCL code.

We do not provide a comparison on the Mali GPU as no reference BLAS library is available for Mali to this date. We use 4096×4096 as a matrix size for all the benchmarks.

The only PENCIL features that are beneficial to the VOBLA generated code are the `assume` builtin and the `restrict` type qualifier. The `independent` directive is not needed as the kernels do not contain any non-affine write accesses. The `restrict` type qualifier is mandatory to elimiante aliasing problems.

Table 6 shows the gains in speedups obtained when support for the `assume` builtin is enabled (measured on the Nvidia platform). When the `assume` builtin is used, the generated code is significantly faster. For example, the generated code for *gemm*, when `assume` is used, is 71% faster than the generated code without `assume`. This happens because `PPCG` simplifies the control flow in the generated kernels using the information provided through the `assume` builtin.

Benchmark	assume
gemver	06% ↑
2mm	84% ↑
3mm	91% ↑
gemm	71% ↑
atax	13% ↑
gesummv	02% ↑

Table 6: Gains in performance obtained when support for the `assume` builtin is enabled

Benchmark	Nvidia GTX 470	AMD Radeon HD 5670	AMD Radeon R9 285
gemver	1.17	2.14	0.39
2mm	0.91	0.62	0.14
3mm	0.87	0.66	0.12
gemm	1.09	0.69	0.19
atax	0.88	1.79	0.37
gesummv	1.03	1.83	0.33

Table 7: Speedups obtained with PPCG over highly optimized BLAS libraries

Table 7 shows the speedups of the kernels generated by PPCG over BLAS libraries. The PPCG generated kernels for the Nvidia and the AMD HD 5670 platforms were close in performance to the highly optimized BLAS library calls for *2mm*, *3mm*, *atax* and *gemm* (e.g. $0.69\times$ for *gemm* on the AMD platform). The main optimizations applied in these kernels are tiling, loop fusion, and the use of shared and private memories. The BLAS code still outperforms the PPCG generated code as it implements many other optimizations including vectorization (clMath) and the use of register tiling (cuBlas) which are not yet supported by PPCG. The speedups for *gesummv* and *gemver* are due to loop fusion and tiling that are performed across different library calls. The *gemver* kernel, for example, is a sequence of 6 BLAS library calls. Although the individual BLAS library functions are highly optimized, better performance can be obtained by fusing and tiling across function calls. PPCG is able to perform these optimizations, and thus outperforms the sequence of BLAS library calls by a factor of $2.14\times$ on AMD Radeon. clMath is highly vectorized and tuned for the AMD Radeon R9 285, since PPCG still does not support vectorization it fails to reach the performance levels for clMath on this platform.

4.4 SpearDE DSL for Data-Streaming Applications

SpearDE [14] is a domain-specific modeling and programming framework for signal processing applications, designed by *Thales Research and Technology*. We evaluate PENCIL using two representative SpearDE applications: Space-Time Adaptive Processing (*STAP*) and Adaptive Beamformer (*ABF*). Both are common signal processing applications for radar systems. We compare the PPCG parallelized code with the sequential CPU version because no parallel version is available to us.

The code for these two applications is relatively large. The ABF code consists of 38 statements in the polyhedral representation (with a loop depth reaching five) while STAP consists of 88 statements (with a loop depth reaching seven). The STAP code is distributed across 12 separate PENCIL functions that are optimized independently. The code is separated because PPCG's optimization pass currently does not scale to a fully inlined version reaching about 1000 lines of code in each benchmark.

ABF and STAP benefit from the following PENCIL features: support for non static-affine code (data-dependent conditionals and non-affine array subscripts), the `independent` directive, summary functions and the `__pencil_kill` builtin. Table 8 shows how ABF and STAP benefit from these PENCIL features.

ABF calls the `fft32` (Fast Fourier Transform) function presented in Section 2.5. Without a summary function, the compiler assumes that the function modifies its whole input array and thus cannot parallelize (a part of) the code. The use of the `independent` directive in STAP enables the parallelization of a loop with non-affine array accesses.

In both, ABF and STAP, PENCIL is only used for compute intensive parts of the code. Many temporary arrays used in these parts are allocated in non-PENCIL regions of the code. The PENCIL compiler however does not assume that these arrays are temporary as it does not analyze non-PENCIL regions. The use of `__pencil_kill`, in this case, allows PPCG to infer that the arrays do not need to be copied between host and device memory. In the case of STAP, copying these temporary arrays to and from host memory cannot be avoided completely as the PENCIL code is distributed across multiple PENCIL functions and the temporaries are used in more than one of these functions. `__pencil_assume` had very little effect on ABF and STAP (on ABF for example, only some expressions could be simplified). None of these effects caused a noticeable change in performance.

Benchmark	summary functions	support for non static-affine code	independent	kill
ABF	required	required	-	14% ↑
STAP	-	required	06% ↑	04% ↑

Table 8: Effect of enabling support for individual PENCIL features on the ability to generate code and on gains in speedups

Table 9 shows the speedups of PPCG generated code over the sequential code. It ranges from a slowdown of 2 to a nearly 4-fold speedup, compared to single-CPU execution. Given the large amount of parallelism available on the GPU devices, this is somewhat disappointing.

Benchmark	Nvidia GTX 470	ARM Mali T604	AMD Radeon HD 5670	AMD Radeon R9 285
ABF	11.00	1.88	2.05	3.69
STAP	2.94	0.51	0.89	1.72

Table 9: Speedups of the PPCG generated code compared to sequential CPU code for *STAP* and *ABF*

Let us comment on the performance anomalies for both STAP and ABF.

On all platforms, the speedup in ABF comes from parallelization and tiling. The generated code did not make use of shared/local memory, but privatization of scalars was essential for making parallelization possible. This is also the case for STAP, except that the generated kernel code does not perform well on the short-vector SIMD architectures (ARM Mali and on AMD Radeon HD 5670). Especially that the embedded ARM Mali system suffered from the lack of automatic vectorization in our flow. The lack of explicit vectorization is clearly identified as a weakness of the current optimization flow.

Also, we only explored a small search space for autotuning as PPCG compilation time for these two applications is significant. The generated code consists of many OpenCL kernels (ABF: up to 16, STAP: up to 26; depends on optimization options), each with different characteristics and a different amount of parallelism and therefore a different set of optimal optimization options. Currently, our tuning framework only allows the use of one set of optimization options for all the kernels, which is far from being optimal.

The current performance of ABF and STAP is also affected by limitations in the loop fusion/distribution heuristic that is implemented in PPCG. PPCG currently supports only two loop fusion/distribution heuristics. The first heuristic tries to fuse loops as much as possible, which maximizes temporal locality, possibly breaking out of resource limits (register pressure), resulting in a loss of parallelism on GPUs. The second heuristic tries to distribute loops as much as possible, which maximizes parallelism but may damage locality (for example, the imaginary and real parts of complex-valued arithmetic are computed in separate OpenCL kernels when this heuristic is applied). The implementation of a heuristic similar to the smartfuse heuristic implemented by Pluto [5] would allow a better trade-off between parallelism and data locality and would enhance performance.

The platforms Nvidia GTX 470 and AMD Radeon R9 285 are powerful enough to compensate for the suboptimal block sizes and code partitioning selected by the autotuner. Some loop dimensions in ABF and STAP have less than 10 iterations which limits the amount of parallelism that can be extracted by PPCG and makes the current dimension-per-dimension mapping heuristic in PPCG inefficient.

Overall, while PENCIL may convey sufficient information for a compiler to apply target-specific optimizations competitive with

expert-written code, the tool flows still require major investment efforts to fully take advantage of PENCIL features.

Discussion of the results. The evaluation section shows how PENCIL features improve the ability of PPCG to generate OpenCL code in two ways: first, by enabling the generation of OpenCL (through features such as the `independent` directive, and summary functions), and second, by enhancing the quality of the generated code (through features such as the `assume` and `kill` builtins). Second, the experiments provide an assessment about the efficiency of the generated code compared to a large set of highly optimized reference codes: 72% of the generated kernels have speedups above $0.5\times$ and 47% outperform the reference implementations. Yet, the evaluation also shows some limitations in the current tools including limitations in generating parallel reductions, loop fusion heuristics, in handling while loops and the lack of vectorization and register tiling. Although the auto-tuning framework was very suitable for small kernels, it was less suitable for larger applications such as STAP and ABF as the exploration of possible optimizations for each kernel creates a large search space. This motivates the integration of parametric tiling and better heuristics in the compilation flow.

5. Related Work

PENCIL language constructs such as the `independent` directive are inspired from directive-based languages such as OpenMP [20] and OpenACC [6], but unlike those, PENCIL has sequential semantics. In PENCIL, the `independent` directive describes the absence of loop carried dependences and such information can be used to enable a range of loop nest transformations rather than enabling loop parallelization alone. A semantically similar directive, also called `independent`, has been part of High Performance Fortran [15].

PENCIL constructs such as `__pencil_assume`, not defined in OpenMP or in OpenACC, allow the compiler to receive additional information from the DSL (or directly from an expert programmer), and to exploit this information to enable further optimizations. Microsoft Visual C supports a proprietary `__assume` statement and a `__builtin_assume` statement has been introduced in clang 3.6. Such builtins have semantics identical to `__pencil_assume` and could be used as substitutes if available. As a subset of C, PENCIL is designed to allow advanced compilers to perform better static analysis, enabling automatic parallelization which is not addressed by OpenMP and OpenACC.

DSL compilers in general map the DSL code directly to GPU relying on parallelism information provided by the domain specific language constructs that express parallelism. Using such an approach, DSL compilers like Halide [21] and Diderot [9], designed for image processing, and OoLaLa [16], designed for linear algebra, show promising results. Our goal is complementary, as we aim to build a more generic and reusable framework and intermediate language that can be used for different domain specific optimizers.

Delite [7] is a more generic DSL framework and run-time designed to simplify building DSL compilers. Delite relies on information from the DSL to decide whether a loop is parallel and does not use any framework for advanced loop nest transformations. We believe that the use of PENCIL and the polyhedral framework with generic DSL frameworks like Delite can leverage automatic parallelism detection and more complex loop nest transformations.

6. Conclusion

We presented PENCIL, an intermediate language for DSL compilers, domain experts, and optimization experts, designed to simplify static code analysis. The design of PENCIL is unique in its combination of sequential semantics, strict compliance with the syntax and semantics of C, and a rich set of static analysis helpers through attributes and pragmas. It makes many forms of non static-affine code and access patterns amenable to advanced loop transformation and parallelization techniques based on the polyhedral framework. We ported a representative set of benchmarks to PENCIL, some of which written in a DSL and compiled to PENCIL. We parallelized these applications automatically on GPU platforms, demonstrating unprecedented expressiveness capabilities for a polyhedral framework. Our experiments validate the use of PENCIL together with an optimizing compiler as building blocks for the implementation of languages and compilers aiming for performance portability.

Acknowledgments. This work was partly supported by the European FP7 project CARP id. 287767.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, Dec 2006.

[2] R. Baghdadi, A. Cohen, S. Verdoolaege, and K. Trifunovic. Improved loop tiling based on the removal of spurious false dependences. *TACO*, 9(4):52, 2013.

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

[4] U. Beaugnon, A. Kravets, S. van Haastregt, R. Baghdadi, D. Tweed, J. Absar, and A. Lokhmotov. VOBLA: A vehicle for optimized basic linear algebra. In *LCTES*, pages 115–124, 2014.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, pages 101–113, 2008.

[6] CAPS Enterprise, Cray Inc., Nvidia, and the Portland Group. The OpenACC application programming interface, v1.0, Nov. 2011.

[7] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPoPP*, pages 35–46, 2011.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.

[9] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A parallel DSL for image analysis and visualization. In *PLDI*, pages 111–120, 2012.

[10] clMath Developers Team. OpenCL math library, 2013. URL https://github.com/clMathLibraries.

[11] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU*, pages 63–74, 2010.

[12] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[14] E. Lenormand and G. Edelin. An industrial perspective: A pragmatic high end signal processing design environment at Thales. In *SAMOS*, pages 52–57, 2003.

[15] D. B. Loveman. High performance Fortran. *Parallel & Distributed Technology: Systems & Applications*, 1(1):25–42, 1993.

[16] M. Luján, T. L. Freeman, and J. R. Gurd. Oolala: An object oriented analysis and design of numerical linear algebra. In *OOPSLA*, pages 229–252, 2000.

[17] Nvidia. Nvidia CUDA programming guide 4.0, 2011.

[18] Nvidia. *cuBLAS Library User Guide*, Oct. 2012.

[19] OpenCV Developers Team. Open source computer vision library, 2002. URL http://opencv.org/.

[20] OpenMP Architecture Review Board. OpenMP application program interface, v3.0, May 2008.

[21] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, pages 519–530, 2013.

[22] J. E. Stone, D. Gohara, and G. Shi. OpenCL: a parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12 (3):66–73, 2010.
[23] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *ICMS 2010*, volume 6327, pages 299–302, 2010.
[24] S. Verdoolaege. Pencil support in pet and PPCG. Technical Report RT-457, INRIA Paris-Rocquencourt, Mar. 2015.
[25] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *IMPACT*, 2012.
[26] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4), Jan. 2013.