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

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

Adam Betts, Alastair F. Donaldson, and Jeroen Ketema
Imperial College London
Email: {a.betts,alastair.donaldson,j.ketema}@imperial.ac.uk

Javed Absar, Sven van Haastregt, Alexey Kravets, and Anton Lokhmotov†
ARM
Email: first.last@arm.com

Róbert Dávid and Elnar Hajiyev
Realeyes
Email: {robert.david,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 complexity and regain performance portability. We present PENCIL, a rigorously-defined subset of GNU C99—enriched with additional language constructs—that enables compilers to exploit parallelism and produce highly optimized code when targeting accelerators. PENCIL aims to serve both as a portable implementation language for libraries, and as a target language for DSL compilers.**

We implemented a PENCIL-to-OpenCL backend using a state-of-the-art polyhedral compiler. The polyhedral compiler, extended to handle data-dependent control flow and non-affine array accesses, generates optimized OpenCL code. To demonstrate the potential and *performance portability* of PENCIL and the PENCIL-to-OpenCL compiler, we consider a number of image processing kernels, a set of benchmarks from the Rodinia and SHOC suites, and DSL embedding scenarios for linear algebra (BLAS) and signal processing radar applications (SpearDE), and present experimental results for four GPU platforms: AMD Radeon HD 5670 and R9 285, NVIDIA GTX 470, and ARM Mali-T604.

***Keywords*-automatic optimization; intermediate language; polyhedral model; domain specific languages; OpenCL**

I. INTRODUCTION

Software for hardware accelerators is currently written using low-level APIs and languages such as OpenCL [1] and CUDA [2], which have a steep learning curve, are laborious and error-prone to program with, and lack *performance portability*: the performance of an accelerated application may vary dramatically across platforms. Hence, developing software at this level is unattractive and costly.

A compelling alternative for developers is to program in higher-level languages and to rely on compilers 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, although aliasing might not actually occur at runtime. Domain-specific languages (DSLs) can help to side-step this problem: it is often clear how parallelism can be exploited given high-level knowledge about standard operations in a particular domain, such as linear algebra [3], image processing [4] or partial differential equations [5]. A drawback of the DSL approach is the significant effort required to implement a compiler generating highly optimized OpenCL or CUDA code for multiple platforms.

To address the above problems, we present PENCIL, a platform-neutral compute intermediate language. PENCIL aims to serve both as a portable implementation language for libraries, and as a target language for DSL compilers.

PENCIL is a rigorously-defined subset of GNU C99 that enforces a set of coding rules predominantly 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 better optimizations and parallelization when translating a PENCIL program to a lower-level program. PENCIL is also equipped with language constructs such as *assume predicates* and *side effect summaries* for functions, which assist with propagating to a PENCIL compiler optimization-enabling information.

PENCIL is easy to learn, as it is C-based. It also interfaces with non-PENCIL C code, which allows legacy applications to be ported incrementally to PENCIL. From the point of view of DSL compilation, PENCIL offers a tractable target: all a DSL-to-PENCIL compiler has to do is to faithfully encode the semantics of the input DSL program into PENCIL—a PENCIL compiler takes care of auto-parallelization and optimization for multiple accelerator targets. Because

†anton@dividiti.com

DSL-to-PENCIL compilers have tight control over the code they generate, they can aid the effectiveness of the PENCIL compiler by communicating domain-specific information via the language constructs that PENCIL provides.

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. To this end, we consider a number of applications with irregular, data-dependent control and dataflow, making this the first time a fully-automatic polyhedral compilation flow is capable of parallelizing a variety of real-world, non-static-control applications. The applications, which originate from hand-written benchmark suites or were generated by DSL-to-PENCIL compilers, are:

- seven image processing kernels written in PENCIL and covering computationally intensive parts of a computer vision stack used by *Realeyes*, a leader in recognizing facial emotions (http://www.realeyesit.com);
- five benchmarks extracted from the SHOC [6] and Rodinia [7] suites and re-written in PENCIL;
- six kernels generated using the VOBLA linear algebra DSL compiler [3];
- two signal processing radar applications generated from code written in the SpearDE streaming DSL [8].

To assess performance portability, we present an experimental evaluation of generated OpenCL code on four GPU platforms: AMD Radeon HD 5670 and R9 285, Nvidia GTX 470, and ARM Mali-T604. The performance results are promising, considering the implementation efforts for these applications and benchmarks. For example, for the VOBLA linear algebra DSL, we were able to generate code that has performance close to the cuBlas [9] and clMath [10] BLAS libraries [11]. For the Realeyes image processing benchmarks, we could match, and sometimes outperform, the OpenCV image processing library [12].

In summary, our main contributions are:

- PENCIL, a platform-neutral compute intermediate language for direct accelerator programming and DSL compilation;
- a polyhedral compilation flow that leverages the features of PENCIL to handle applications that go beyond the classical restrictions of the polyhedral model, including forms of dynamic, data-dependent control flow and array accesses;
- an evaluation of PENCIL on multiple GPUs and several real-world, non-static-control applications that were previously out of scope for polyhedral compilation.

II. OVERVIEW OF PENCIL

PENCIL is a subset of the C99 language carefully designed to capture static properties essential for implementing advanced loop nest transformations. The language provides constructs that help parallelizing compilers to perform more accurate static analyses and generate efficient target-specific code. The constructs allow communicating information that is difficult for a compiler to extract, but that can be easily captured from DSLs or expressed by expert programmers.

Our aim was for PENCIL to be a strict subset of C99. However, where necessary and when no alternatives existed, we exploited the flexibility of GNU C extensions such as type attributes and pragmas. The pragmas were inspired by familiar annotations for exploiting vector- and thread-level parallelism, but retain a strictly sequential semantics.

PENCIL is not coupled to any particular compiler or target language. However, as we have validated PENCIL using a polyhedral compiler targeting OpenCL, we will refer to this compiler when discussing the implementation of PENCIL.

A. 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 restricted cases.

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

Portability. A standard, non-parallelizing C99 compiler supporting GNU C extensions should be able to compile the language. This ensures portability to platforms without specialized PENCIL support and allows existing tools to be used for debugging (unparallelized) PENCIL code.

Sequential semantics. The language should have a sequential semantics to simplify DSL compiler development and direct programming in PENCIL, and, importantly, to avoid committing to any particular parallel patterns.

In designing the PENCIL extensions to C99, we analyzed numerous benchmarks and DSLs [13] and identified language constructs that would be helpful in exposing parallelism and enabling compiler optimizations. In deciding which language features to include, we were 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 means that only those properties that are useful for improved static analysis and target mapping need to be expressible in PENCIL. Domain-specific properties that are not useful for optimization do not have to be conveyed and should thus not be a part of PENCIL. This keeps PENCIL general-purpose, sequential and lightweight.

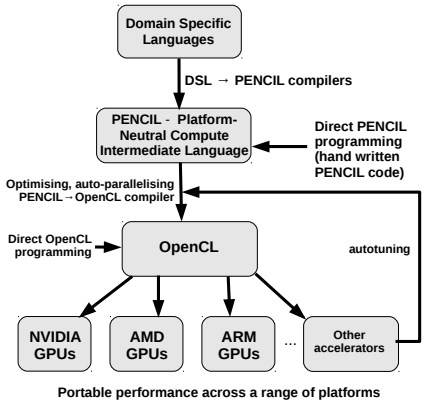

Figure 1. A high level overview of the PENCIL compilation flow

Figure 1 gives a high level overview of a typical PENCIL usage scenario. First, a program written in a DSL is translated into PENCIL. Some domain-specific optimizations may be applied prior to or during this translation, while delaying target-specific optimizations to later compilation stages. Second, the generated PENCIL code is combined with hand-written PENCIL that implements library functions; PENCIL is used here as a standalone language. The combined code is then optimized and parallelized. Finally, highly optimized OpenCL code is generated. The generated code is autotuned through profiling-based iterative compilation.

B. PENCIL *Coding Rules*

We detail the most important restrictions imposed by PENCIL from the point of view of enabling GPU-oriented compiler optimizations. For more details, see [14], [15].

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 of functions. Pointer dereferencing is also not allowed except for accessing C99 arrays. These restrictions essentially eliminate aliasing problems, which is important for parallelization and data movement between the different address spaces of accelerators such as GPUs.

No recursion. Recursive function calls are not allowed, as they are forbidden in languages such as OpenCL.

Sized, non-overlapping arrays. Arrays must be declared using the C99 variable-length array syntax [16], and the declaration of each function argument that is of array type must use `pencil_attributes`, a macro expanding to the C99 `restrict` and `const` type qualifiers followed by the `static` keyword (see Figure 4). During optimization, the PENCIL compiler thus knows the length of each array (in parametric form), and knows that arrays do not overlap.

Structured `for` loops. A PENCIL `for` loop must have a single iterator, invariant start and stop values, and a constant increment (step), where invariant means that the value does not change in the loop body. Precisely specifying the loop format avoids the need for sophisticated induction variable analyses which may fail under unpredictable conditions.

A further guideline—which is not mandatory as it cannot be statically checked in general—is that array accesses should not be linearized. Linearization obfuscates affine subscript expressions, hindering effective compilation. Multidimensional arrays should be used instead.

PENCIL also supports OpenCL scalar builtin functions such as `abs`, `min`, `max`, `sin`, `cos`, using a target-independent and explicitly typed naming scheme (using suffixes to distinguish between `float` and `double` builtins).

C. *Assume Predicates*

We now describe *assume predicates*, the first main construct introduced by PENCIL. The other new constructs—the `independent` directive, summary functions, and the `__pencil_kill` function—follow in Sections II-D–II-F.

An *assume predicate*, written `__pencil_assume`(e), with e a Boolean expression, indicates that e is guaranteed to hold whenever the control flow reaches the predicate. This knowledge is taken on trust by the PENCIL compiler, and may enable generation of more efficient code. If e is violated during execution, the semantics of the PENCIL program is undefined. This is *not* checked at runtime, but optional runtime checking, for debugging, could be provided. In the context of DSL compilation, an assume predicate allows a DSL-to-PENCIL compiler to communicate high level facts.

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 input pixel using a kernel matrix `kern_mat` for the weights. The convolution code is part of an image processing benchmark from Realeyes (see also Section IV-A).

In Realeyes's production environment, the size of the `kern_mat` never exceeds 15×15, as indicated by the assume predicates. While the image processing experts know this, without the predicates the compiler must assume that the kernel matrix can be arbitrarily large. When compiling for a GPU target the compiler must thus either allocate the kernel matrix in the GPU's global memory rather than in fast local memory, or must generate multiple variants—one to handle large kernel matrix sizes and another for smaller kernel matrix sizes—selecting between variants at runtime. Instead, the `__pencil_assume` statements in the code communicate limits on the size of the array, allowing the compiler to store the whole array in local memory.

D. *The Independent Directive*

The `independent` directive is used as a loop annotation, and is semantically similar to the equally named High Performance Fortran directive [17]. The directive indicates that the result of executing the loop does not depend on the execution order of the data accesses from different loop

```
#define clampi(val, min, max) \
  (val < min) ? (min) : (val > max ) ? (max):(val)

__pencil_assume(ker_mat_rows <= 15);
__pencil_assume(ker_mat_cols <= 15);

for (int i = 0; i < rows; i++)
  for (int j = 0; j < cols; j++) {
    float prod = 0.0f;
    for (int e = 0; e < ker_mat_rows; e++)
      for (int r = 0; r < ker_mat_cols; r++) {
        row = clampi(i+e-ker_mat_rows/2, 0, rows-1);
        col = clampi(j+r-ker_mat_cols/2, 0, cols-1);
        prod += src[row][col] * kern_mat[e][r];
      }
    conv[i][j] = prod;
  }
```

Figure 2. PENCIL code for general 2D convolution

```
/* 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

iterations. As such, the accesses from different iterations may be executed in parallel.

In practice, **independent** is used to indicate that a loop has no loop carried dependences. The directive can also be used when some dependences exist but the user 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. It may be necessary to invoke external non-PENCIL functions to enable parallelization of 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, the purpose of which is to enable parallelization of loops whose only dependences are on variables into reductions are computed. For brevity we do not discuss this clause further.

Figure 3 shows a code fragment of our PENCIL implementation of the breadth-first search benchmark from the Rodinia [7] benchmark suite. The benchmark computes the minimal distance from a given source node to each node in the input graph. The algorithm maintains a frontier and computes the next frontier by examining all unvisited nodes adjacent to the nodes in the current frontier. All nodes in a frontier have the same distance from the source node.

The `for` loop of Figure 3 can be parallelized because each node in the current frontier can be processed independently. This creates a possible race condition on the `cost` and `next_frontier` arrays, but this race condition can be ignored, because all conflicting threads will write the same value. By specifying the **independent** pragma, the programmer guarantees that the race condition is benign, enabling parallelization.

E. Summary Functions

The effect of a function call on its array arguments is usually derived from analyzing the called function. In some cases, the results of such an analysis may be too inaccurate, and in the extreme case, when no code is available, the compiler must conservatively consider the possibility that all elements of each array argument are accessed. To mitigate this problem, PENCIL allows the user to associate a *summary function* with each function. A summary function has a signature identical to the function it is associated with, and the association informs the PENCIL compiler that it may derive the memory accesses from the summary function.

In practice, summary functions are used to describe the memory access patterns of *library functions* called from PENCIL code (and whose source code is usually not available for analysis), and of *non*-PENCIL *functions* called from PENCIL code, as they may be difficult to analyze otherwise. To associate a summary function with a function `foo()`, a programmer uses the attribute **pencil_access**`(name)`, where `name` is the name of summary function describing the accesses of `foo()`.

Summary functions are not executed, but only used for analyzing memory footprints: A summary function must access the same memory elements as the function it is associated with, or an over-approximation thereof. Providing a summary function can enable more precise static analysis than the default conservative assumption that all elements of all array arguments can be accessed. In general, a summary can be simpler than the function it summarizes: it only needs to capture sets of accesses, not their order and number of occurrences. As an example, if a function were to be executed on a processor having no direct access to main memory, the compiler could use its summary to determine the memory elements that would need to be marshaled into and out of the function (cf. [18]).

The functions **__pencil_use** and **__pencil_def** are designed to be used in summary functions to mark memory accesses. A call to **__pencil_use**`(A[`e`])` indicates that a read from array `A` at index e *may* occur, while a call to **__pencil_def**`(A[`e`])` indicates that a write to array `A` at index e *must* occur.

For writes, *may* information can also be conveyed by

```
__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. Code from Adaptive Beamformer, illustrating summary functions

using a **__pencil_maybe** predicate, which evaluates to a Boolean value unknown at compile-time. 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 such as Boogie [19].

Figure 4 shows a loop nest extracted from the *Adaptive Beamformer* (ABF) benchmark presented in Section IV-D. The code calls a function `fft32` (a Fast Fourier Transform). The function only reads and modifies (in place) 32 elements of its input array `in`, it does not modify any other parts of the array. The function is not analyzed by the PENCIL compiler because it is not a PENCIL-function. Without a summary function the compiler would conservatively assume that the whole array passed to `fft32` is accessed for reading and writing, preventing parallelization. The summary function indicates that each iteration of the loop nest only reads and writes 32 elements of the input array, allowing the compiler to parallelize the loop nest.

Writing summary functions for library routines is the most common use case for summaries, and is the library developer's responsibility. The summary functions should be provided in the library's header files and are used directly by the PENCIL compiler. In less common cases, summary functions are either written by the PENCIL programmer or automatically generated by a DSL compiler.

F. Kill Statements

The **__pencil_kill** builtin function allows the user to refine dataflow information within and across any control flow region. The **__pencil_kill** function is polymorphic and signals that its argument (a variable or array element) is dead at the program point where the call to the function occurs, meaning that no data flows through this argument from any statement instance executed before the kill to any statement instance executed after.

The information is used in several ways, as explained in detail in [20]. The effect of **__pencil_kill** is illustrated by the following example:

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

If the above loop is mapped to a GPU kernel, then the `A` array needs to be copied out from the GPU to the host after computation, because some elements of `A` may be written to by the loop. This copy-out overwrites the original contents of `A` on the host. Since not all elements of `A` may be written to, the array must in principle also be copied in to ensure that the elements not written to retain their original values after the copy-out. The **__pencil_kill**`(A)` statement indicates that the data in `A` is not expected to be preserved by the region and that the copy-in may be omitted.

III. POLYHEDRAL COMPILATION OF PENCIL CODE

We next explain how specific PENCIL features can be compiled with a polyhedral compiler. (But, to reiterate, PENCIL is not tied to any particular compilation technique.)

A. Polyhedral Compilation

Polyhedral compilation uses an abstract mathematical representation to model programs. Each statement in a program is represented using three pieces of information: an *iteration domain*, *access relations* and a *schedule*. The representation is first extracted from the program's 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 execution instances of the statement (a statement in a loop has an execution instance for each loop iteration upon which it executes). Each execution instance of a statement in a loop nest is uniquely represented by an identifier and a tuple of integers (typically, the values of the outer loop iterators). These integer tuples are compactly described by quasi-affine constraints. For example, the statement on Line 9 of Figure 2, call it S_0, has the following iteration domain:

$$\{\ S_0(i,j)\ :\ 0 \le i < rows \wedge 0 \le j < cols\ \}$$

A quasi-affine constraint is a constraint over integer values and integer variables involving only the operators `+`, `-`, $\times$, `/`, `%`, `&&`, `||`, `<`, `<=`, `>`, `>=`, `==`, `!=`, and the ternary `?:` operator, where the second argument of `/` and `%` must be a (positive) integer literal, and where at least one of the arguments of $\times$ must be a piece-wise constant expression. An example of a quasi-affine constraint for a statement in a loop nest is $10 \times i + j + n > 0$, where i and j are loop iterators and n is a *symbolic constant* (i.e., a variable that has an unknown but fixed value for the duration of an execution). Examples of non-quasi-affine constraints are $i \times i > 0$ and $n \times i > 0$.

To be able to extract a 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. This condition is called *static-affine*.

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 the index expressions in the input program to be static-affine.

Finally, the *schedule* determines the relative execution order of the statement instances. Program transformations are performed via modifications of the schedule and depend on *dependence relations*. These relations map statement instances to statement instances that depend on them for their execution, and are derived from the access relations and the original execution order. In particular, two statement 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.

B. Compilation of PENCIL

We adapted `PPCG` [21], an existing polyhedral compiler for GPUs, to handle PENCIL. `PPCG` relies on the `pet` library [22] to extract the iteration domain and access relations; the dependence analysis is performed by the `isl` library [23]. A new schedule is computed by `isl` using a variant of the Pluto algorithm [24] (this latter step applies most loop nest transformations).

We next discuss the changes we made to `PPCG` to support PENCIL. For more details, including details on support for arrays of structures, we refer the reader to [20].

Assume predicates. `pet` keeps track of constraints on the symbolic constants of a program (i.e., of variables that have an unknown but fixed value throughout an execution). The constraints are automatically derived from array declarations and index expressions. In particular, constraints are derived that exclude negative array sizes and negative array indices (negative indices are not allowed because they could result in aliasing within an array). The constraints are used by `PPCG` when generating an AST from a schedule to simplify the generated AST expressions.

An assume predicate provides `pet` with additional constraints on the symbolic constants that may not be automatically derivable. For example, Lines 4, and 5 in Figure 2 provide additional constraints on the symbolic constants `ker_mat_rows` and `ker_mat_cols`. Although the argument of a **`__pencil_assume`** statement can be any expression, `PPCG` currently only exploits quasi-affine ones.

The kill builtin. A kill statement in `pet` represents the fact that no dataflow on the killed data elements can pass through an instance of the statement. This information can be used during dataflow analysis to stop the search for potential sources of data elements. When `pet` comes across a variable declaration, two kill statements that kill the variable are introduced, one at the location of the variable declaration and one at the end of the block that contains the variable declaration. The use of the **`__pencil_kill`** builtin introduces additional kill statements to `pet`.

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

Figure 5. Code extracted from dilate

```
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. Code containing a `break` statement

Non-static-affine array accesses. To handle non-static-affine accesses, `pet` has been modified to distinguish *may*-writes vs. *must*-writes. Any index expression that cannot be statically analyzed or that is not affine, is treated as *possibly* accessing any index. This over-approximation typically results in the compiler statically identifying more dependences than will actually be exhibited at runtime.

Non-static-affine conditionals and loop guards. `PPCG` treats any non-static-affine conditional or loop with a non-static-affine loop guard as a single macro-statement together with its body (i.e., as a statement encapsulating both control and body). Any write inside such a macro-statement is treated as a may-write. For example, the conditional of Figure 5, extracted from the *dilate* benchmark, cannot be analyzed. The if-statement and its body are therefore considered as one macro-statement and the assignment to `sup` is treated as a may-write.

While loops, break and continue. While loops and loops containing break and continue statements are treated like non-static-affine conditionals: the loop and its body are considered to be a single macro-statement. For example, due to the `break` in Figure 6, `PPCG` treats the entire loop headed at Line 3 as a single statement. This means that `PPCG` can schedule (i.e., change the order of execution of) the loop headed at Line 3 and its body as a whole, but it cannot schedule the individual statements in the body.

The 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 distinct iterations accessing overlapping data. Through the directive the user asserts that no dependences need to be introduced to prevent such reorderings and that any variable declared inside the loop is private to each iteration. `pet` handles the **`independent`** directive by building a relation between the statement instances that excludes them from depending on each other. Moreover,

pet builds a set of variables that are local to the loop. This set of variables is used by PPCG to ensure that their live ranges do not overlap in affine transformations, and to privatize them if needed when generating parallel code.

Summary functions. pet has been modified to extract access information from called functions. If a summary function is provided, the information is extracted from the summary instead.

IV. Experimental Evaluation

We evaluated the performance of OpenCL code generated from PENCIL using pencilcc, a version of PPCG incorporating a runtime library and the changes discussed in the previous section.[1] To verify that PENCIL can be used both as a standalone language and intermediate language for DSL compilers, we used both benchmarks written directly in PENCIL and code generated by DSL compilers. The set of benchmarks written directly in PENCIL consists of a image processing benchmark suite by Realeyes (Section IV-A) and a selected set of benchmarks from the Rodinia and SHOC suites (Section IV-B). The code generated by DSL-to-PENCIL compilers originates from the VOBLA and SpearDE DSLs (Sections IV-C and IV-D).

We used four GPU platforms for our experiments: an Nvidia GTX 470 (with an AMD Opteron Magny-Cours 2×12 core CPU and 16GB RAM), an ARM Mali-T604 (with a dual-core ARM Cortex-A15 CPU and 2GB RAM), an AMD Radeon HD 5670 (with an Intel Core2 Quad Q6700 CPU and 8GB RAM) and an AMD Radeon R9 285 (with an Intel Xeon E5-2640 8 core CPU and 32GB RAM). Hence, we covered both a relatively large set of real-word applications and a relatively diverse range of platforms.

Our experiments were designed to evaluate (a) whether PENCIL enables the parallelization (mapping to OpenCL) of kernels that cannot be parallelized with current state-of-the-art polyhedral compilers (Pluto [24]), and (b) whether PENCIL enables the generation of efficient code (by comparing the performance of the automatically generated code to hand-crafted code).

Autotuning. We developed an autotuning compiler framework to facilitate the retargeting of our compiler to different GPU architectures. We applied autotuning to the pencilcc-generated code only. Autotuning the hand-crafted reference code (mostly implemented as libraries) would be difficult, because the code is not designed to be autotuned (work group sizes are hard-coded, changing the use of local and private memory requires manual modifications, etc.). Moreover, the BLAS libraries (clMath [10] and cuBlas [9]) do not require autotuning: they 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 them on the target GPUs. The search covers combinations of pencilcc's compiler flags, including different work group and tile sizes, whether to use local and/or private memory, and which loop distribution heuristic to use (out of two possible heuristics). Autotuning each benchmark takes several hours (except for the six VOBLA kernels, which take up to two days due to the large search space).

Measurements. For our experiments, we let pencilcc instrument the generated code to measure the wall clock execution time, which includes the GPU kernel execution time, duration of any data copies (between the host and the GPU), and the time taken to execute on the host any program code that was not offloaded to the GPU. The measured times do not include device initialization and release, and kernel compilation times. In order to exclude compilation time, we either invoked a dry-run computation beforehand that was not timed (caching compiled kernels), or subtracted the compilation time from the total execution time, depending on the way in which the reference implementation compiled and invoked its kernels. We used OpenCL profiling tools to further analyze the performance of the reference implementations and the pencilcc-generated code (obtaining the number of cache misses, device global memory accesses, device occupancy, etc.). Each test was run 30 times. Below, we report the median of the speedups over the reference implementations.

A. Image Processing Benchmark Suite

The image processing benchmark suite consists of a set of kernels covering computationally intensive parts of the computer vision stack by Realeyes ranging from. simple image filters to composite image processing algorithms. For each kernel in the benchmark suite we compared a straightforward (non-hand-optimized) PENCIL implementation with the equivalent OpenCL kernel from the OpenCV version 2.4.10 image processing library [12].

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

An important characteristic of the image processing kernels is that they contain non-static-affine code, which a classic polyhedral compiler does not handle efficiently due to the restrictions of the polyhedral model. The conditional `if (se[e][r] != 0)` in Figure 5 is an example of such non-static-affine code.

Five kernels from the benchmark suite have non-static-affine conditionals and read accesses. One kernel has non-static-affine write accesses. Hence, the compiler needs to handle all of these. Non-static-affine write accesses are difficult to

[1] Version 0.4 of pencilcc is available at https://github.com/Meinersbur/pencilcc. The experiments in this section were performed using an older, development version: https://github.com/Meinersbur/pencil-driver/tree/7a0dd59708253cb121cadf0b6529bd792b35c3fd.

Table I
EFFECT OF ENABLING SUPPORT FOR INDIVIDUAL PENCIL FEATURES ON THE IMAGE PROCESSING BENCHMARKS

Benchmark	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 II
SPEEDUPS OF THE CODE GENERATED BY PENCILCC OVER OPENCV FOR THE IMAGE PROCESSING BENCHMARKS

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

handle because they prevent the compiler, in general, from determining whether a loop is parallelizable.

The kernels require support for non-static-affine code, the **independent** directive, and the **__pencil_assume** and **__pencil_kill** builtins. Table I lists the features per benchmark. In the case of non-static-affine code and the **independent** directive, the table lists whether the feature was required for OpenCL code generation. For the builtins, the table shows the speedup obtained when support for the feature was enabled (vs. disabled). The speedup shown is for the Nvidia GTX 470, the effect on the other platforms was similar. A '-' indicates that a feature was not used in a benchmark or its use did not affect code generation.

Support for non-static-affine code was required to generate OpenCL code for five kernels. For *basic histogram*, the use of the **independent** directive enabled parallelization and OpenCL code generation, which is difficult otherwise. For *dilate*, assuming that the size of the structuring element (the array representing the neighborhood used to compute each pixel) is less than 16×16 enabled `pencilcc` to map the element to local memory, and allowed it to generate code that was 20% faster compared to when it did not assume this. The speedups associated with using **__pencil_kill** are mainly due to the builtin enabling `pencilcc` to eliminate redundant data copies.

Table II presents the speedups of the `pencilcc`-generated OpenCL code over the baseline OpenCV OpenCL implementation. We used the same image to evaluate all kernels (a 2880×1607, 1.5MB image).

On the AMD Radeon R9 285 platform, the speedup of the `pencilcc`-generated kernels over the OpenCV reference implementations was due to slow data copies used by OpenCV. On this platform, OpenCV used OpenCL's `clEnqueueWriteBufferRect`, which copies data from host to device while at the same time padding the data for aligned memory accesses. `pencilcc`, on the other hand, used OpenCL's `clEnqueueWriteBuffer`, which copies data but does not perform any padding. OpenCV's approach was $7\times$ slower on the AMD Radeon R9 285 platform, explaining the significant speedups we obtain. Note that, although the use of `clEnqueueWriteBufferRect` may be less efficient for these benchmarks, 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.

Other than the difference in data copies, there was no significant difference in the speedups obtained on the AMD Radeon R9 285 HD 5670 platforms, and we focus in the latter AMD platform in the remainder of this section.

`pencilcc` does not apply any optimizations to data copies other than eliminating spurious copies when the user provides appropriate **__pencil_kill** statements. For each of the image processing benchmarks, the amount of data copied by the `pencilcc`-generated code (when using **__pencil_kill**) was equal to the amount of data copied by the reference implementation. Consequently, the listed speedups (or slowdowns) were solely due to faster (or slower) kernel execution times (except for the R9 285, as discussed above).

The speedups of *resize* and *color conversion* on Nvidia, ARM and AMD Radeon HD 5670 were due to the tiling of the 2D loop nest in these two kernels, which considerably enhanced data locality (up to 56% fewer L1 cache misses on Nvidia for *color conversion*). In the case of *affine warping*, the speedup was due to two optimizations: thread coarsening, which merges multiple work items, leading to less redundant computation, and tiling, which enhanced data locality (up to 65% fewer L1 cache misses on Nvidia).

For *basic histogram*, the code generated by `pencilcc` was generally slower than the OpenCV reference implementation. The OpenCV version was faster, because each work group computes a histogram in local memory, and the local histograms are only combined into one global histogram during a final reduction. Automatic generation of such reductions is not yet supported by `pencilcc`.

In the case of *dilate*, the OpenCV reference implementation was vectorized, while `pencilcc` currently does not support the generation of vectorized code. The lack of vectorization affected the performance most on AMD and ARM. In addition, the OpenCV reference implementation mapped the input image array to local memory while `pencilcc`'s local memory heuristic decided not to apply this mapping. As a consequence, the `pencilcc`-generated code accessed global GPU memory 175 times more than the OpenCV implementation, which led to a decrease in performance. The same problem with the local memory heuristic applied to *gaussian smoothing*.

The performance of *2D convolution* matched that of the OpenCV reference implementation on Nvidia and AMD.

Table III
SELECTED BENCHMARKS FROM THE RODINIA AND SHOC SUITES

Benchmark (Suite)	Data set size	Description/notes
2D stencil (SHOC)	100 iter., 4096×4096 grid	On structured grid
Gaus. elim. (Rodinia)	1024×1024 matrix	Dense matrix
SRAD (Rodinia)	100 iter., 502×458 image	Image enhancement
SpMV (SHOC)	16384 rows	Sparse matrix-vector multipl.
BFS (Rodinia)	4 million nodes	Breadth-first search on graph

Table IV
EFFECT OF ENABLING SUPPORT FOR INDIVIDUAL PENCIL FEATURES ON THE RODINIA AND SHOC BENCHMARKS

Benchmark	Non-static-affine code	Independent
2D stencil	-	-
Gaussian elimination	-	-
SRAD	required	-
SpMV	required	-
BFS	required	required

The reference implementation could not be run on the ARM Mali GPU, as it used hardcoded local memory and work group sizes that exceeded hardware limits.

B. Rodinia and SHOC Benchmark Suites

Our second set of benchmarks consists of reverse-engineered benchmarks from the Rodinia [7] and SHOC [6] suites. We selected the benchmarks, listed in Table III, based on diversity (i.e., covering different Berkeley 'motifs' [25] such as dense and sparse linear algebra, structured grids, and graph traversal), and for their ability to pose a challenge to traditional polyhedral compilers due to their use of non-static-affine code. We compared the performance of `pencilcc`-generated code for these benchmarks with the Rodinia and SHOC reference implementations.

Table IV lists the PENCIL features required for each of the benchmarks and shows the effect of the features on `pencilcc`'s ability to generate OpenCL code. Support for non-static-affine code is required by three benchmarks, which use non-static-affine read accesses, conditionals, and write accesses. The non-static-affine write accesses, in *BFS*, prevent the compiler from parallelizing the code, and require use of the **`independent`** directive. We did not make use of **`__pencil_kill`** annotations for the benchmarks in this suite. Assume predicates were useful in providing optimization hints to the compiler for the *2D Stencil*, *SpMV* and *BFS* benchmarks. This was especially important for enabling generation of OpenCL code that could be automatically vectorized by the ARM Mali compiler, but for this benchmark suite we did not conduct a controlled measurement of performance with vs. without assume predicates.

Table V shows the speedups over the OpenCL reference implementations. The speedups for *2D Stencil* and *Gaussian Elimination* are mainly due to tiling which enhanced data locality and reduced cache misses (we observed $4\times$ fewer L1 cache misses for *2D Stencil* on Nvidia GTX 470). For *SRAD*, the PENCIL-generated OpenCL code was significantly slower than the reference implementation, mainly

Table V
SPEEDUPS FOR THE OPENCL CODE GENERATED BY `PENCILCC` OVER THE RODINIA AND SHOC REFERENCE IMPLEMENTATIONS

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
Gaussian elimination	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
BFS	0.65	0.78	0.43	0.72

because `pencilcc` did not map a reduction to OpenCL (`pencilcc` currently does not support the generation of parallel reductions). This leads to additional data transfers between the host and the device. For *BFS*, the generated OpenCL code was also slower than the reference code, again due to additional data transfers. These data transfers were due to `pencilcc` only mapping the *bodies* of while loops to the device and generating data transfers at the beginning and end of each loop iteration.

C. VOBLA DSL for Linear Algebra

The image processing benchmarks and Rodinia and SHOC benchmarks of Sections IV-A and IV-B demonstrate the use of PENCIL as a standalone language. Here and in Section IV-D, we consider benchmarks in which PENCIL is used as an intermediate language for DSL compilers.

VOBLA is a domain specific language for implementing linear algebra algorithms, providing a compact and generic representation using an imperative programming style [3]. The main control flow operators of VOBLA are `if`, `while`, `for`, and `forall`. The `if` and `while` operators have standard semantics. The `for` and `forall` operators iterate over a scalar range (e.g., `0:3`) or arrays. `forall` indicates that the iterations of a loop can be executed in any order.

The VOBLA-to-PENCIL compiler is fairly simple and does not perform any sophisticated optimizations. Advanced loop nest transformations are handled by `pencilcc`. The VOBLA compiler only uses assume predicates and the **`independent`** directive. The **`__pencil_kill`** builtin is only useful to eliminate spurious data transfers in non-static control code and is not needed for the purely static control code of VOBLA. Summary functions are only needed when library functions are called, but these are not generated by the VOBLA compiler.

The VOBLA compiler infers assume predicates from relations between array sizes. For example, for the statement `C = A + B`, the VOBLA compiler infers that the sizes of `A` and `B` are equal and generates a **`__pencil_assume`** statement that indicates this. As a consequence, `pencilcc` does not need to generate code to handle the case in which the sizes of `A` and `B` differ. This information can, e.g., be exploited when `pencilcc` decides to fuse loops that iterate over `A` and `B`, respectively.

The VOBLA compiler generates the **`independent`** directive when translating `forall` operators: each `forall`

Table VI
PERFORMANCE GAINS FOR BENCHMARKS COMPILED FROM VOBLA WHEN ASSUME PREDICATES ARE ENABLED

Benchmark	Nvidia GTX 470
gemver	6% ↑
2mm	84% ↑
3mm	91% ↑
gemm	71% ↑
atax	13% ↑
gesummv	2% ↑

Table VII
SPEEDUPS OBTAINED WITH `PENCILCC` OVER THE BLAS LIBRARIES

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

operator is translated into a PENCIL `for` loop that is annotated with **`independent`**.

We used VOBLA to implement a set of linear algebra kernels and compared the code generated by `pencilcc` with equivalent code that calls BLAS library functions. The kernels are *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).

The VOBLA implementations were first compiled to PENCIL using the VOBLA compiler and then mapped to OpenCL using `pencilcc`. We compared the code with two highly optimized BLAS library implementations:

- the clMath 2.2.0 [10] BLAS library provided by AMD and used for comparison on the AMD platforms, and
- the cuBlas 5.5 [9] BLAS library provided by Nvidia and used for comparison on the Nvidia platform. In this case we used `pencilcc` to generate CUDA code instead of OpenCL code.

We do not provide a comparison for the ARM platform, as no BLAS library is available on that platform. We used a matrix size of 4096×4096 for all benchmarks.

Table VI shows that the code obtained for the Nvidia GTX 470 was significantly faster with assume predicates enabled. For example, the code generated for *gemm* with assume predicates is 71% faster than without.

Table VII shows the speedups for the kernels generated by `pencilcc` over the BLAS libraries. The `pencilcc`-generated kernels for the Nvidia and the AMD HD 5670 platforms were close in performance to the highly optimized BLAS libraries for *2mm*, *3mm*, *atax* and *gemm* (e.g., $0.69\times$ for *gemm* on the AMD platform). The main optimizations applied to these kernels were tiling, loop fusion, and the use of local and private memory. The BLAS library code still outperforms the `pencilcc`-generated code as it implements many other optimizations such as vectorization (clMath) and the use of register tiling (cuBlas). The speedups for *gesummv* and *gemver* were due to loop fusion and tiling across different BLAS library calls. For example, the *gemver* kernel consists of a sequence of 6 BLAS library calls and although the individual BLAS library functions are highly optimized, better performance can be obtained by fusing and tiling across function calls. clMath is highly vectorized and tuned for the AMD R9 285. Since `pencilcc` does not perform vectorization, it fails to reach the performance levels for clMath on this platform.

D. SpearDE DSL for Data-Streaming Applications

SpearDE [8] is a domain-specific modeling and programming framework for signal processing applications, designed by *Thales Research and Technology*. We evaluated PENCIL using two representative SpearDE applications: *Adaptive Beamformer* (*ABF*) and *Space-Time Adaptive Processing* (*STAP*). Both are common signal processing applications for radar systems. We compared the `pencilcc`-generated code with the sequential CPU version, because no parallel version was available to us.

AFB and *STAP* are relatively large: *ABF* consists of 38 statements in the polyhedral representation (with a loop depth reaching five), and *STAP* consists of 88 statements (with a loop depth reaching seven). The *STAP* code is distributed across 12 separate PENCIL functions. The functions were optimized individually, because `pencilcc`'s optimization pass currently does not scale to a fully inlined version reaching about 1000 lines of code.

As shown in Table VIII, *ABF* and *STAP* benefit from support for non-static-affine code, the **`independent`** directive, summary functions, and the **`__pencil_kill`** builtin. The speedups reported are again for the Nvidia GTX 470.

As mentioned in section II-E, *ABF* calls a fast Fourier transform function. Without a summary, the compiler assumes that the function modifies its whole input array, making parallelization impossible. The use of the **`independent`** directive in *STAP* enables the parallelization of a loop with non-static-affine array accesses.

Both *ABF* and *STAP* use PENCIL only for the computationally intensive parts of the code. Many temporary arrays used in these parts are allocated outside the PENCIL regions. However, as `pencilcc` does not analyze non-PENCIL code, it cannot assume that the arrays are temporary. Using **`__pencil_kill`** allows the compiler to infer that the arrays do not need to be copied between host and device. In the case of *STAP*, copying the temporary arrays cannot be completely avoided, as the code is distributed across multiple functions and the temporaries are used in several of them.

Table IX shows the speedups of the `pencilcc`-generated code over the sequential code. On all platforms, the speedup for *ABF* was due to parallelization and tiling. The generated code did not make use of local memory, but privatization

Table VIII
EFFECT OF ENABLING SUPPORT FOR INDIVIDUAL PENCIL FEATURES ON THE SPEARDE BENCHMARKS

Benchmark	Summary functions	Non-static-affine	Independent	Kill
ABF	required	required	-	14% ↑
STAP	-	required	6% ↑	4% ↑

Table IX
SPEEDUPS WITH `PENCILCC` OVER THE SEQUENTIAL CPU CODE FOR THE SPEARDE BENCHMARKS

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

of scalars was essential for making parallelization possible. This was also the case for *STAP*, except that the generated kernel code did not perform well on short-vector architectures (ARM Mali and AMD Radeon HD 5670), which suffer from no automatic vectorization in `pencilcc`.

The performance of *ABF* and *STAP* was also affected by limitations of `pencilcc`'s two loop fusion/distribution heuristics. The first tries to fuse loops as much as possible, which maximizes temporal locality, but does not take into account resource limits (register pressure), resulting in a loss of performance on GPUs. The second heuristic tries to distribute loops as much as possible, which maximizes parallelism but may damage locality (e.g., 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 Pluto's smartfuse heuristic [24] would allow a better trade-off between parallelism and data locality and would enhance performance.

E. Discussion of Results

As our experiments show, the **`independent`** directive and (in the case of SpearDE) summary functions improve `pencilcc`'s ability to generate OpenCL. Assume predicates and the **`__pencil_kill`** builtin enhance the quality of the generated code. Performance-wise, 72% of the generated kernels reach at least 50% of the performance of the hand-optimized reference implementations, and in 47% of the cases the generated kernels outperform the reference implementation. Our experiments also expose some limitations of the current setup. In particular, the inability of `pencilcc` to generate parallel reductions, its limited loop fusion heuristics, handling while loops as black boxes, and the lack of vectorization and register tiling.

We have not discussed the performance of our autotuning framework. In brief: it performed well on small PENCIL benchmarks, but for larger benchmarks (e.g., the SpearDE ones), we ran into problems due a combinatorial explosion in the number of compiler options. This warrants further investigation into search heuristics and predictive modeling.

V. RELATED WORK

Summary functions have first been proposed as abstract domain transformers of numerical libraries in PIPS [26]. As a language construct, they find their origin in the decoupled access/execute (æcute) model [18], which allows expressing memory access patterns and execution constraints of kernels. PENCIL's summary functions are, to the best of our knowledge, the first attempt to abstract interprocedural access patterns in C99.

PENCIL's directives are inspired by directive-based languages such as OpenMP [27] and OpenACC [28]. 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`**, occurs in High Performance Fortran [17]. What sets PENCIL apart is its sequential semantics. As a subset of C, it is designed to allow advanced compilers to perform better static analysis, enabling automatic parallelization.

PENCIL builtins such as **`__pencil_assume`** allow the PENCIL compiler to receive additional information from a DSL compiler or from an expert programmer. The compiler can exploit this information to enable further optimizations. Microsoft Visual C and clang 3.6 support, respectively, semantically identical **`__assume`** and **`__builtin_assume`** builtins. These builtins could be used as a substitute when available.

DSL compilers targeting GPUs typically map DSL code directly to OpenCL and CUDA, relying on DSL constructs that express parallelism. Using such an approach, DSL compilers such as Halide [4] and Diderot [29] (for image processing) and OoLaLa [30] (for linear algebra) show promising results. Our complementary goal is to build a more generic framework and intermediate language to be used by different domain specific optimizers.

Delite [31] is a generic framework for building DSL compilers. Delite relies on information from a DSL to decide whether a loop is parallel but has no facilities for advanced loop nest transformations. We therefore believe that generic DSL frameworks like Delite can benefit from using PENCIL and a polyhedral compiler.

VI. CONCLUSION

We have presented PENCIL, a portable intermediate language designed to enable productive and efficient accelerator programming. PENCIL is unique in its design combining a sequential semantics, strict compliance with C, and a rich set of attributes and pragmas that enable static analysis. PENCIL makes many forms of non-static-affine code and access patterns amenable to advanced loop transformation and parallelization within the polyhedral framework.

We have evaluated the design and implementation of PENCIL on a representative set of benchmarks across several

GPU-accelerated platforms. Some of these benchmarks are written in a domain-specific language and then compiled to PENCIL. Our experiments validate the use of PENCIL together with an optimizing compiler as a valuable building block for enabling performance-portable accelerator programming.

ACKNOWLEDGMENTS

This work was partly supported by the EU FP7 project CARP (287767) and the ARTEMIS COPCAMS project (332913). We would like to thank our partners at Thales Research and Technology for their collaboration on a PENCIL-based interface within the SpearDE framework.

REFERENCES

[1] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, 2010.

[2] Nvidia, "Nvidia CUDA C programming guide 4.0," 2011. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[3] 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*, 2014, pp. 115–124.

[4] 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*, 2013, pp. 519–530.

[5] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, "Unified form language: A domain-specific language for weak formulations of partial differential equations," *ACM Trans. Math. Softw.*, vol. 40, no. 2, 2014.

[6] 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*, 2010, pp. 63–74.

[7] 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*, 2009, pp. 44–54.

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

[9] Nvidia, *cuBLAS Library User Guide*, 2012. [Online]. Available: http://docs.nvidia.com/cuda/cublas

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

[11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Softw.*, 1979.

[12] OpenCV Developers Team, "Open source computer vision library," 2002. [Online]. Available: http://opencv.org

[13] A. Kravets, G. Kouveli, A. Lokhmotov, E. Hajiyev, L. Marak, and T. Virolainen, "CARP deliverable D2.2.A: requirements analysis," 2012. [Online]. Available: http://carp.doc.ic.ac.uk/external/publications/D2.2A.pdf

[14] R. Baghdadi, A. Cohen, T. Grosser, S. Verdoolaege, A. Lokhmotov, J. Absar, S. van Haastregt, A. Kravets, and A. F. Donaldson, "PENCIL language specification," INRIA, Research Rep. RR-8706, 2015. [Online]. Available: https://hal.inria.fr/hal-01154812

[15] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. F. Donaldson, "PENCIL: Towards a platform-neutral compute intermediate language for DSLs," in *WOLFHPC*, 2012.

[16] ISO, "ISO/IEC 9899:1999, Programming languages – C," 1999.

[17] D. B. Loveman, "High performance Fortran," *Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 25–42, 1993.

[18] L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. J. Kelly, "Deriving efficient data movement from decoupled access/execute specifications," in *HiPEAC*, 2009, pp. 168–182.

[19] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO*, 2006, pp. 364–387.

[20] S. Verdoolaege, "PENCIL support in pet and PPCG," INRIA, Tech. Rep. RT-457, 2015. [Online]. Available: https://hal.inria.fr/hal-01133962

[21] 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.*, vol. 9, no. 4, 2013.

[22] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *IMPACT*, 2012.

[23] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *ICMS*, vol. 6327, 2010, pp. 299–302.

[24] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI*, 2008, pp. 101–113.

[25] 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," University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[26] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: An overview of the PIPS project," in *ICS*, 1991.

[27] OpenMP Architecture Review Board, "OpenMP application program interface, v3.0," 2008.

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

[29] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer, "Diderot: A parallel DSL for image analysis and visualization," in *PLDI*, 2012.

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

[31] 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*, 2011, pp. 35–46.