Snowflake: A Lightweight Portable Stencil DSL

Nathan Zhang, Michael Driscoll, Armando Fox, Charles Markley
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, California, United States of America
{*nzhang32, chick*}*@berkeley.edu,* {*mbdriscoll, fox*}*@cs.berkeley.edu*

Samuel Williams, Protonu Basu
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California, United States of America
{*swwilliams, pbasu*}*@lbl.gov*

***Abstract*—Stencil computations are not well optimized by general-purpose production compilers and the increased use of multicore, manycore, and accelerator-based systems makes the optimization problem even more challenging. In this paper we present Snowflake, a Domain Specific Language (DSL) for stencils that uses a "micro-compiler" approach, i.e., small, focused, domain-specific code generators. The approach is similar to that used in image processing stencils, but Snowflake handles the much more complex stencils that arise in scientific computing, including complex boundary conditions, higher-order operators (larger stencils), higher dimensions, variable coefficients, non-unit-stride iteration spaces, and multiple input or output meshes. Snowflake is embedded in the Python language, allowing it to interoperate with popular scientific tools like SciPy and iPython; it also takes advantage of built-in Python libraries for powerful dependence analysis as part of a just-in-time compiler. We demonstrate the power of the Snowflake language and the micro-compiler approach with a complex scientific benchmark, HPGMG, that exercises the generality of stencil support in Snowflake. By generating OpenMP comparable to, and OpenCL within a factor of 2× of hand-optimized HPGMG, Snowflake demonstrates that a micro-compiler can support diverse processor architectures and is performance-competitive whilst preserving a high-level Python implementation.**

***Keywords*-Scientific Computing; Domain-Specific Language; Python; GPU; Multicore;**

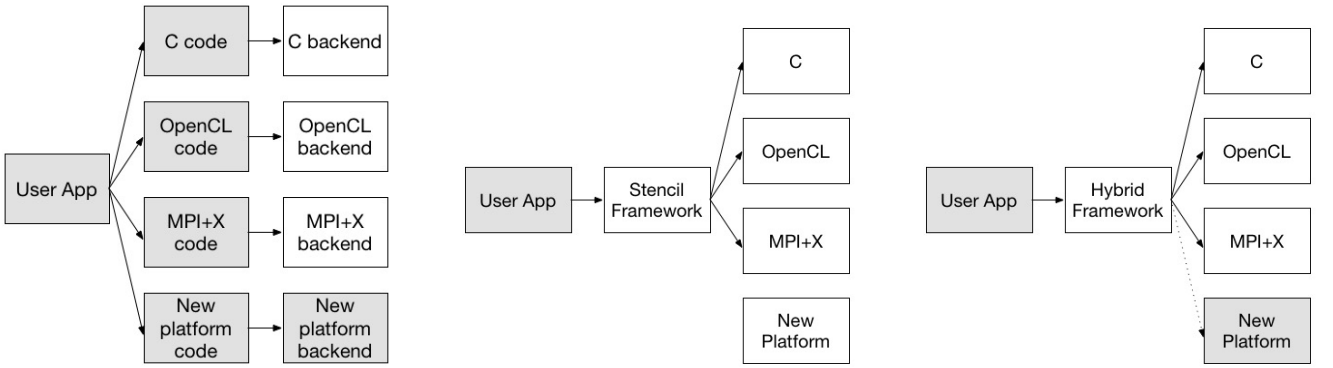
I. Introduction

Stencil computations on structured rectangular grids are common in scientific simulations, used in upwards of 40% of the applications at some supercomputing centers. Production compilers typically perform poorly on these kernels relative to hand-optimized code, and stencils are not easily encapsulated into hand-tuned libraries, because of the large number of stencil operators used in practice [1]. With the advent of multicore and many-core architectures, including GPUs and other accelerators, and a desire to run applications across laptops, clusters and high performance supercomputers, it becomes necessary to write and maintain multiple versions of an application in order to efficiently use each platform [2], [3], [4], [5]. This "multiple codebase, multiple backend" approach requires a variety of programming models and compilers including OpenMP, OpenCL, and CUDA.

The goal is to allow scientific programmers to write a single instance of their source code, and provide performance portability across multiple architectures with multiple backends. For example, Halide [6] is an embedded DSL for image processing kernels that generates optimized code for CPU and GPU based architectures. Similarly, a number of general-purpose compilers and domain specific language projects perform optimizations on stencils from scientific code, but without the full generality needed by real applications.

In this paper we introduce Snowflake, a domain-specific language (DSL) for stencils capable of expressing the complex stencils that appear in scientific code, including strided domains (e.g., red-black decompositions and multigrid operators), variable-coefficient stencils, in-place/out-of-place operations, unions of rectangular domains (used in adaptive mesh refinement), multiple input and output meshes, complex boundary conditions, and asymmetric operators. We also describe the Snowflake compiler architecture, a just-in-time compiler that cleanly separates the front-end parsing and analysis from the "micro-compilers" that target code for different architectures. This makes it easier to target new architectures, including those with fairly different computational models. Snowflake's is embedded in Python, making it an excellent tool for applications that use popular tools such as SciPy [7] and iPython [8]. The Snowflake analysis engine leverages existing symbolic algebra tools in Python and is based on linear Diophantine equation analysis, but is extended to support the full generality of our stencil computations. The interface between the code generator and the parser/optimizer is narrow, cleanly separating backend from front-end architecture. To date, we have created backends for C/OpenMP and C/OpenCL. The performance of the Snowflake generated code on simple stencils is competitive with, and sometimes superior to, handcrafted code produced by human experts, starting from a single source code.

Because Snowflake's DSL is embedded in Python, the application writer can use all of the other features of the high-level language (graphing, file processing, and so on) and even use other embedded DSLs. As a proof of concept, we have created a Python reference implementation of the High-Performance Geometric Multigrid (HPGMG) [9] solver, a complex supercomputing benchmark that takes advantage of strided domains, complex boundaries, and other



(a) The traditional multiple codebase multiple backend model. Users write the sections in gray, generating large amounts of repetitive work

(b) The more modern single codebase multiple backend model, which emphasizes reusability. However, support for new platforms is sometimes lacking, especially if the end-user develops a non-standard system

(c) The proposed hybrid model which attempts to solve both problems by enabling the user to write backends on top of the single codebase multiple backend model

Figure 1: The two major paradigms when dealing with platform portability are multiple codebase multiple backend and single codebase multiple backend, both of which suffer from portability issues.

Snowflake features; From a single, common Python source, Snowflake generates C code with OpenMP directives or OpenCL operations and is competitive with hand-optimized code.

The contributions of this paper are:

- Development of a domain-specific language, Snowflake, that is powerful enough to express complex scientific stencils, including boundary conditions, strided regions, higher order operators, and multiple input and output grids.
- Construction of a just-in-time program analysis framework that leverages tools in Python for dependence analysis within and across stencil sweeps.
- Integration of a Diophantine-based analysis algorithm that generalizes prior work to handle the full generality of Snowflake.
- Performance analysis of a scientific benchmark that exercises Snowflake features and shows performance close to that of hand-optimized code for both CPU and GPU architectures.

The paper is organized as follows. Section II describes the design of the language, whose organizing principle is the ability to specify application of a stencil over domains consisting of arbitrary unions of hyperrectangles in arbitrary dimension. This simplification means, for example, that boundaries are not treated differently from stencil interiors—they are simply stencils over different domains. It also describes the computations Snowflake can represent, many of which are unsupported by existing stencil frameworks.

Sections III and IV describe the dependency analysis and compiler, respectively. In particular, they illustrate how the DSL's representation of stencils and domains detects dependencies that could cause write conflicts during parallelization. Analysis is based on solving systems of Diophantine equations, and it works both within one stencil operation and across a group of several serial stencil operations.

Section V presents the results of our implementation of Snowflake with OpenMP and OpenCL backends. We ported the High Performance Geometric Multigrid (HPGMG) benchmark to Python/Snowflake and compare Snowflake-generated OpenMP and OpenCL code to hand tuned code on comparable problem sizes. We show that the performance of Snowflake-generated code is competitive with hand-tuned code on CPU platforms, and within a factor of $2\times$ of hand-tuned CUDA on GPU-accelerated platforms. Moreover, as no changes to the Python source code are required, Snowflake provides a performance-portable implementation of HPGMG across CPU and GPU architectures. In Sections VI and VII, we review related and future work.

II. Language Design

Table I and Figure 2 summarize the main data structures in Snowflake, which represent *weight arrays, stencils,* and *domains*. This section gives a rapid overview of the language by explaining the roles of these data structures and giving some examples from typical scientific codes.

A. Language Elements and Expressiveness

A 1D `WeightArray` specifies stencil weights, with the middle element corresponding to the stencil center point. In 2D, the array elements are themselves arrays, specifying the weights along each dimension, so that (for example) a 2D 3×3 stencil whose center point is $0, 0$ would have the weight array $[[w_{-1,-1}, w_{0,-1}, w_{-1,1}],\ [w_{0,-1}, w_{0,0}, w_{0,1}],\ [w_{1,-1}, w_{1,0}, w_{1,1}]]$ The notation generalizes so that an N-dimensional weight array consists of arrays nested N-deep.

Element	Description
`WeightArray`	In 1D, an array specifying stencil weights with middle element corresponding to the stencil center point; in N dimensions, arrays nested N deep, specifying weight array in each dimension.
`SparseArray`	Alternative representation of a `WeightArray` specified as a hashmap whose keys are vector offsets (relative to stencil center) and values are weights at that vector offset.
`Component`	Associates a `WeightArray` or `SparseArray` with a grid.
`RectDomain`	Specifies a start, end, and stride in arbitrary dimensions
`DomainUnion`	A union of `RectDomains`
`Stencil`	Associates a `Component`, `RectDomain` or `DomainUnion`, and output grid (which can be one of the input grids, for in-place stencils), and exposes a `compile` method that generates native code and returns a Python callable
`StencilGroup`	A series of `Stencils` to be performed consecutively; exposes a `compile` method that generates native code and returns a Python callable

Table I: Snowflake's main data structures, as used in the example in Figure 4.

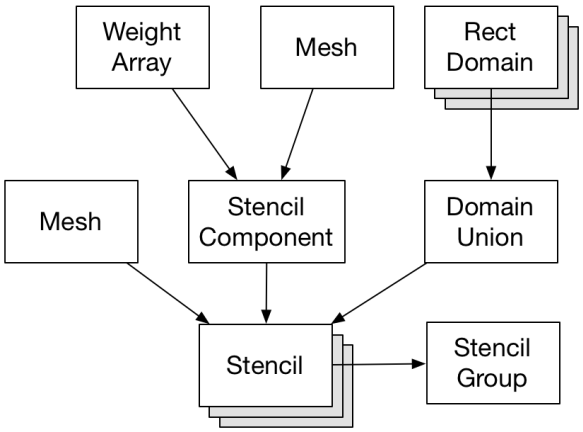


Figure 2: Relationships among the main stencil computation elements described in Table I.

A `Component` associates a weight array with a particular grid.

A stencil is applied over a `RectangularDomain`, which describes a start, end, and stride in arbitrary dimensions, or over a `DomainUnion`, which is a union of `RectangularDomains`. `RectangularDomains` may contain negative start and end parameters, in which case they are relative to the grid size. This allows interior, boundary, and other general stencil definitions to be reused without redefining the iteration space on each grid size.

Finally, the stencil operator itself associates a `Component`, `RectangularDomain` or `DomainUnion`, and another `Mesh` specifying where the output should go. The output grid can be one of the input grids, for in-place stencils.

These language features allow expressing the following complex stencil behaviors, typical in scientific applications:

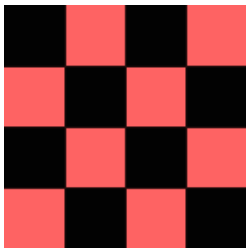
1) *Striding* is the application of the stencil to a subset of elements in a multidimensional array separated by constant sized jumps in each dimension (common operations such as restriction or interpolation). Striding is supported by defining a `RectangularDomain`. *Striding and coloring* (Figures 3a-3b) are supported by constructing the `DomainUnion` of multiple such domains.
2) *In-place operations* write the result of a stencil into one of the input arrays, rather than into a new write-only output array. While rare in image processing, in-place operations are common in techniques such as Gauss-Seidel Red-Black stencils and Chebyshev smoothing.
3) *Boundary Conditions* are restrictions on boundary values or values just outside of the boundary to enforce expected behavior across multiple iterations. These are also expressed as stencils with (sometimes) large offsets, or as *asymmetric stencils* (Figure 3c-3d).
4) *Variable-coefficient stencils* are those whose coefficients at each point differ; they are common in applications such as heat flow where the medium may be heterogeneous, requiring the stencil to read values such as flow coefficients from a separate array.

An example serves to illustrate how the language supports these features.

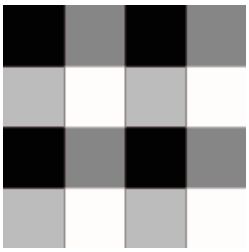
B. Example: Complex Smoothing

The code in Figure 4 shows how Snowflake expresses a fairly complex stencil, typical of a scientific application. The red-black pattern, or checkerboard pattern, is an iteration ordering used in scientific computing. In a red-black pattern, or multi-color stencils in general, each element of the array has a color determined by a union of striding patterns. Typically in scientific computing, values at the grid boundary are also constrained. For example, in a Dirichlet boundary, the value at the boundary of the grid is constrained to be zero. One common method of emulating this boundary value when using linear operators is by setting values outside of the grid in a ghost zone or halo; in this example, the ghost value immediately outside a boundary cell is set to the negative of the value immediately inside, to cancel its effect.

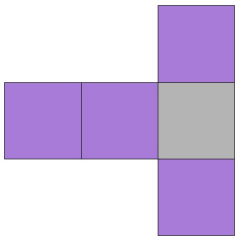
The Snowflake language makes no distinction between interior passes and exterior boundaries; both are described as the application of a stencil over a domain, which is any union of hyperrectangular regions within a multidimensional



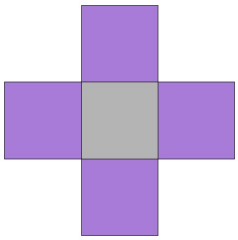
(a) Red-Black tiling



(b) 4-color tiling



(c) Asymmetric stencil used near grid boundary



(d) 5-point Jacobi stencil

Figure 3: (a) Red-black tiling allows cross-point updates simultaneously at points of the same color, so an update operation takes only 2 passes. (b) 4-color tilings are common when each update requires the surrounding 3-by-3 neighborhood. Like red-black tiling, all points of the same color in a 4-color tiling can be updated simultaneously. (c) An asymmetric stencil, sometimes used near the grid boundary of a standard 5-point stencil (d), results in odd dependency patterns. Purple points are read from, gray points are written to.

```
1  top =  Component("beta_x", WeightArray([[1]])
2  bot =  Component("beta_x", WeightArray([[0], [1], [0]])
3  left = Component("beta_y", WeightArray([[1]]))
4  right = Component("beta_y", WeightArray([[0, 0, 1]])
5  Ax = Component("mesh", WeightArray([[0,top,0], [left, left+top+bot+right, bot], [0, bot,
       0]]))
6  b = Component("rhs", WeightArray([[1]]))
7  difference = b - Ax
8  original = Component("mesh", WeightArray([[1]])
9  lambda_term = Component("lambda", WeightArray([[1]]))
10 final = original + lambda_term * difference
11 red = RectDomain((1,1), (-1,-1), (2,2)) + RectDomain(((2, 2), (-1, -1), (2, 2))
12 black = RectDomain((1,1), (-1,-1), (2,2)) + RectDomain(((2,2), (-1,-1), (2,2))
13 red_stencil = Stencil(final, "mesh", red)
14 black_stencil = Stencil(final, "mesh", black)
15 # Dirichlet zero boundary: 1 of 4 stencils shown...
16 top_boundary = Stencil("mesh", Component("mesh", WeightArray([[ 0],[ 0],[-1]])),
17                         RectangularDomain((1, -1), (-1, -1), (1, 0)))
18 # ...others are rotationally equivalent
```

Figure 4: This complex-smoothing operation a strided colored (red-black) stencil with Dirichlet boundaries and variable-coefficients. Nominally, we are solving $-\nabla \cdot \beta \nabla x = b$, and are doing so by applying the Jacobi operator without dampening over the red and black points on a checkerboard on alternating iterations. These operators and iteration domains can be constructed at run-time with no additional cost.

grid. Stencils such as the Gauss-Seidel Red-Black (GSRB) stencil operate in-place, meaning that the output of the red points reads from the black points and writes back the red points, and vice versa.

The core of the GSRB stencil is a 5-point stencil with variable coefficient arrays β_x, β_y representing the physical properties of the space (lines 1–4). In the smooth, we take the difference between the right hand side of the problem and the Ax component to find the difference $b - Ax$ (lines 5–7). We finally adjust the grid by this difference multiplied by a factor λ (lines 8–10).

Having defined the operation, we define the red and black domains; each is defined as the union (+) of two domains offset from each other and strided by 2 in each dimension (lines 11–12). We can now define the main red-black stencil by associating the operation, its output, and its domain (lines 13–14).

The last step is generating the boundary for a uniform linear Dirichlet condition in 2 dimensions. This requires four stencils (top, bottom, left, and right boundaries); for each one, the cell immediately outside the boundary should be set to the negative of the value inside the boundary, to make the boundary cell be zero. Lines 16–17 show how to set up the stencil for the top boundary; the others are rotationally equivalent.

Finally, the red and black stencils (lines 13–14) and

the boundary stencils (lines 16–17, plus three rotationally equivalent boundary stencils omitted for brevity) can be combined into a `StencilGroup`, which allows analysis to identify parallelism across all these stencils as well as within each one. The next section describes how the analysis is done.

III. Analysis

One major goal of the Snowflake DSL was to make analysis of stencils easier in order to ensure correctness and ease the burden on the optimization process. Given the highly regular access patterns of stencils and stencil groups, the inherent parallelism is statically determinable in many nontrivial cases [10]. These dependencies reduce to a system of Diophantine equations that determine whether or not a stencil interferes with itself and other stencils. Diophantine equations are equations where integer solutions are sought. For example, the equation $x^2+y^2 = 1$ has an infinite number of general solutions, but only 4 integer solutions: $(\pm 1, \pm 1)$.

Snowflake can analyze dependencies to identify parallelism across multiple stencils or domains. For example, it can find parallelism in computing two stencils on different parts of a grid, or in computing a series of stencils over a rectangular domain composed of a boundary plus the interior, as in the example in the previous section.

A. Diophantine Analysis

The primary purpose of analysis is to determine whether a stencil can be applied in parallel over a union of domains. We use the open-source Python library SymPy[1] to find solutions to the system of Diophantine equations that indicate the presence of a loop-carried dependency. Furthermore, the presence of a loop-carried dependency within even polynomial indexing systems is decidable as the corresponding Diophantine equations can be solved using the extended greatest common divisor algorithm. The general Diophantine equation is intractable according to the Davis-Putnam-Robinson-Matiyasevich Theorem [11], but by appropriately restricting the types of stencil we permit under our language, we avoid this problem. We allow the usage of polynomial indexing, but place primary focus on affine indexing due to its usage in the restriction and interpolation operators in stencils. Specifically, affine and polynomial Diophantine equations can be solved or shown to be unsatisfiable using the extended Euclidean algorithm, thus allowing straightforward analysis.

This technique is used for both verification and auto-parallelizing within stencils as well as detecting data hazards across stencil iterations, and can also be used for eliminating dead stencils and reordering computations. This analysis is platform independent, and can handle the full generality of scientific stencils, which do not exercise exponential Diophantine problems, and thus do not incur the MRDP problem.

[1]SymPy: http://www.sympy.org/

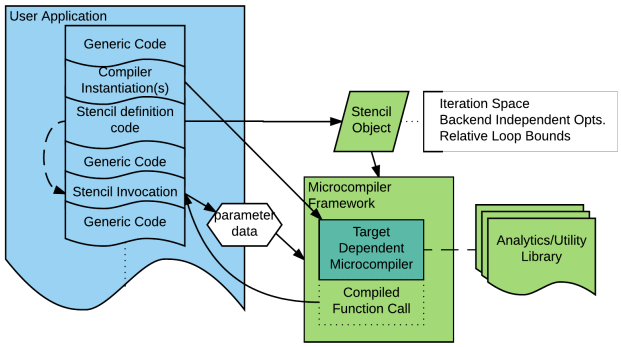


Figure 5: In the Snowflake workflow, there are two classes of end users: the scientist *(light blue)* and the compilers/-platform expert *(teal)*. Snowflake framework code, which is provided but extensible, is in *lime green*.

In Snowflake, our analyses are performed assuming finite domains, which allows for executing multiple stencils on the same set of grids simultaneously assuming that they do not interfere with each other. This is done by first using well-understood Diophantine techniques for obtaining symbolic expressions describing points of potential conflict, followed by satisfiability computations handled by Sympy.

For example, boundary conditions are able to be expressed as simple stencils in Snowflake and do not create false dependencies which infinite-domain analyses such as Halide's interval analysis would flag.

IV. JIT Micro-Compiler

The Snowflake framework invokes a user configurable JIT stencil compiler with the python AST of the stencil codes. On the fly, this AST is modified by by multiple analysis, optimization and translation passes. The result is rendered into the configured performance language, which is then handed to an appropriate compiler, producing a binary which is wrapped in a python callable function which is then called. These call-ables are cached, for subsequent use. The compiler architecture cleanly separates the front-end (platform-agnostic) parsing and analysis from the backend (platform-specific) code generation, allowing new backends to be added by users. The encapsulation allows the programmer to freely mix Snowflake with other code generation frameworks.

Since Snowflake comes prepackaged with Python, sequential C, C with OpenMP, and C with OpenCL micro-compilers, the compiler expert is only needed when additional optimizations are requested or unsupported backends are needed. These existing micro-compilers are implemented

purely in Python, and share their analysis and frontend modules. The scientist or application programmer is the typical end-user of other DSLs, and is responsible for constructing the stencils as well as invoking the kernels. Stencils and their components may be passed around arbitrarily and be used to compose StencilGroups or more complex stencils.

To compile the DSL code for a stencil, we call the `compile` method on either the `Stencil` or on a `StencilGroup`, which specifies a series of stencils to be run consecutively and allows the compiler to do cross-stencil optimizations. The `compile` method returns a callable that will do the actual computation. For backends that generate compiled shared object files, we currently use Python's built-in Foreign Function Interface in order to pass parameters down to lower-level code.

A. Design of the OpenMP Backend

The OpenMP backend makes heavy use of the dependency analysis in prior sections in order to establish barrier points in the generated OpenMP code. Since this paper describes a language and a DSL framework architecture, the OpenMP backend was implemented using a naive scheduling system. Each stencil is generated as an OpenMP Task, with larger stencils being split into subtasks.

The backend inserts barriers if the output of a group of stencils is needed by the next group of stencils. These groups were formed greedily; the compiler maintains a list of stencils in the current group and places a barrier only when the next stencil depends on the stencils in the existing group. This dependency analysis technique is used to generate a DAG of stencil dependencies. The task farming system promotes greater system usage when encountering NUMA issues when compared to naive parallel code.

In the OpenMP backend of Snowflake, we implemented tiling and multicolor reordering, both implemented as transformations on the Snowflake AST tree. Tiling is an arbitrary-dimension blocking algorithm, which allows the user to specify a tiling size when compiling the stencil, and provides a method of tuning tiling sizes. Multicolor reordering is a loop-interchange optimization that allows Red-Black or other non-unit stride stencil operations in order to decrease slow-memory reads. When examining the variable-coefficient GSRB performance, it is evident that Snowflake needs to implement more efficient tiling and iteration methods in order to achieve speeds closer to those of hand-optimized code.

B. Design of the OpenCL Backend

The OpenCL backend uses a tall-skinny blocking method, using two-dimensional tiles which are then rolled upwards through the remaining dimensions. Work on non-unit stride stencils such as those found in GSRB is currently in progress.

V. Performance

We evaluate Snowflake's ability to provide single-source performance-portability on stencil computations running on CPU and GPU-accelerated platforms. Given its complexity and use of many different stencils on varying array sizes, we use HPGMG, the High Performance Geometric Multigrid supercomputer benchmark developed by Lawrence Berkeley National Lab, as a driver [9]. Geometric multigrid is a structured grid-based linear algebra algorithm that uses a series of restriction, smoothing, and interpolation operations to numerically approximate solutions to partial differential equations in near-linear time. Since the publicly-maintained reference implementation of HPGMG uses MPI+OpenMP for parallelism and blends functionality with hand optimizations, porting it to new platforms is difficult and time consuming. A happy side effect of porting it to Python/Snowflake is that we now have a reference implementation that is source- and performance-portable, requiring only a new Snowflake backend to port the benchmark to other platforms.

A. Experimental Setup

For our evaluations, we used two platforms. All CPU experiments were conducted on an Intel Core i7-4765T running at 2.0GHz without exploiting TurboMode or Hyper-Threading (superfluous for memory-intensive computations on out-of-order processors). GPU experiments were conducted on a NVIDIA K20c GPU. The CPU has a STREAM Triad bandwidth of about 22.2GB/s while the GPU has an Empirical Roofline Toolkit [12] bandwidth of about 127GB/s. We had exclusive access to all systems.

For comparison purposes, we compare Snowflake performance (number of unknowns / solve time) to the 2nd order, hand-optimized OpenMP and OpenMP+CUDA versions of HPGMG [9], [13]. Note, as our CPU-based system has only a single NUMA node, we have disabled MPI parallelism in HPGMG and have only used OpenMP. We configured our benchmarks (Snowflake or native) to run 10 V-Cycles instead of the default 1 F-Cycle, and configured it to use two GSRB smooths (4 stencil sweeps) for pre- and postsmoothing.

In order to provide context, we evaluate the performance of three standalone stencil computations — the canonical 7-point constant-coefficient Laplacian, a Jacobi smoother ($x^{n+1} = x^n + \frac{2}{3}D^{-1}(f - Lx^n)$ where L is the 7-point constant-coefficient Laplacian above), and a Gauss-Seidel, Red-Black smoother using a variable-coefficient 7-point Laplacian. Note, each of these operators requires an interspersed (boundary / red / boundary / black) Dirichlet boundary condition stencil to be applied over the surface of the problem domain. As multigrid requires applying a smoother across multiple grid spacings, we also evaluate the performance of the variable-coefficient GSRB (in isolation) across a range of problem sizes. Finally, we build a complete geometric multigrid solver using Snowflake representations

for the smoother, residual, restriction, interpolation, and boundary condition stencils.

Owing to the limitations of our testbeds and an incompatibility with the Intel compiler, Snowflake was compiled with GCC version 4.9 with `-std=c99 -O3 -fgcse` and `-fPIC` flag for linking. The OpenCL backend additionally used `-lOpenCL` with OpenCL version 1.2. HPGMG was compiled with ICC 14.0 with `-O3 -openmp`. In all experiments, we conducted an untimed warmup phase (e.g. 10 v-cycles) followed by the benchmarking phase.

B. Roofline Performance Bound

We use the Roofline Model [14], [15] to qualify the performance we attain relative to the capabilities of the target CPU and GPU platforms. To that end, for each stencil, we calculate the asymptotic compulsory memory traffic per stencil (24, 40, and 64 bytes per stencil for constant-coefficient 7-point Laplacian, constant-coefficient Jacobi, and variable-coefficient GSRB respectively), and for each platform, we use the modified STREAM [16] benchmark shown in Figure 6 to measure the memory bandwidth for the read-dominated memory access patterns endemic to stencil computations. The ratio of these two terms provides a fundamental, speed-of-light performance bound (stencils/s) for each operator for each platform. Note, we assume no capacity or conflict misses and always assume write-allocate cache behavior coupled with the inability of the compiler to automatically generate cache bypass instructions. Similarly, we use DRAM bandwidth regardless of problem size.

```
1 void tuned_STREAM_Dot(double scalar)
2 {
3       int j;
4   #pragma omp parallel for
        reduction(+:beta)
5       for (j=0; j<N; j++)
6           beta += a[j]*b[j];
7 }
```

Figure 6: Modified STREAM benchmark using the dot product in order to approximate the read-dominated memory access patterns of stencil computations.

C. CPU and GPU Performance

Multigrid solvers, and applications in general must compose a number of stencils together. It is imperative any system deliver performance for a variety of stencil computations. Figure 7 presents Snowflake performance with either the OpenMP or OpenCL backends for three different stencils/smoothers on a fixed 256^3 problem. We include performance comparisons to the equivalent operations in HPGMG and HPGMG-CUDA as well as to a Roofline-inspired DRAM performance bound. Unfortunately, HPGMG-CUDA

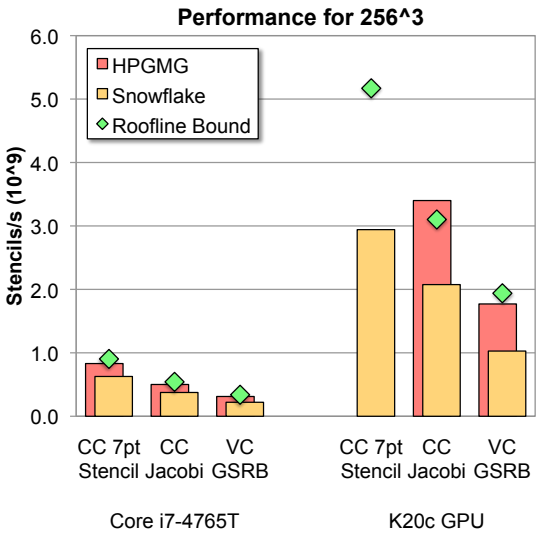


Figure 7: Snowflake productively delivers performance across architecture and operators — generated OpenMP and OpenCL compared to HPGMG, HPGMG-CUDA, and the Roofline (DRAM) performance bound for a fixed 256^3 problem. Note, HPGMG-CUDA does not include a constant-coefficient 7-point Stencil.

does not include a bare 7-point constant-coefficient Laplacian stencil, but only includes it in the context of a smoother. As we can see Snowflake/OpenMP performance does very well, delivering performance close to HPGMG and the Roofline. Conversely, it is clear the additional low-level optimizations found in NVIDIA's HPGMG-CUDA are necessary as Snowflake's OpenCL backend underperforms. As we assume a write-allocate cache architecture, GPU Roofline estimates for the Laplacian and Jacobi may underestimate performance potential (HPGMG-CUDA exceeds a Roofline underestimate). Nevertheless, it is clear Snowflake was able to deliver performance portability within a factor of $2\times$ across CPUs and GPUs from a single-source description.

In order to realize a high-performance multigrid solver — O(N) solve time in the number of variables N — one must deliver constant performance across a range of exponentially-varying problem sizes. Figure 8 shows performance for the variable-coefficient GSRB smoother across the range of problem sizes found in a multigrid solver. Observe that runtime decreases with problem size as bound by Roofline. Moreover, Snowflake OpenMP and OpenCL performance track the hand-optimized HPGMG and HPGMG-CUDA performances. Note, the smallest 32^3 problem likely fits in the CPU L3 cache and can thus greatly surpass the (DRAM-based) Roofline bound.

Figure 9 shows the overall (smooth, residual, interpolation, restriction, boundary condition stencils for all grid sizes) multigrid solver performance for both the hand-

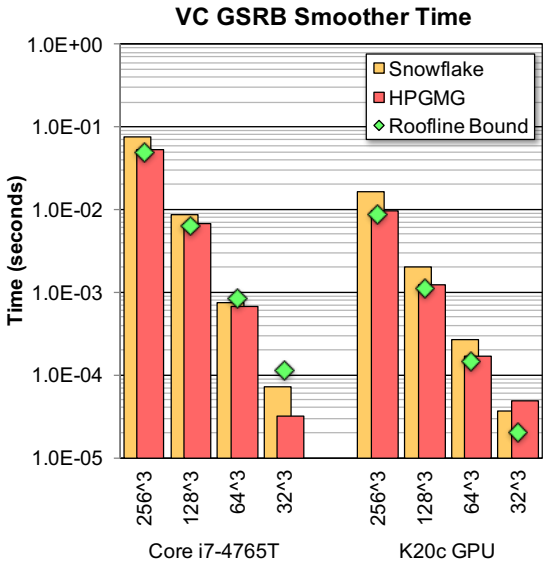


Figure 8: Snowflake execution time for the variable-coefficient GSRB smoother with generated OpenMP and OpenCL compared to HPGMG, HPGMG-CUDA, and Roofline as a function of problem size. Note, small problems exceed the DRAM-based Roofline bound because they fit in cache. Performance across scales is critical to realizing a high-performance multigrid solver.

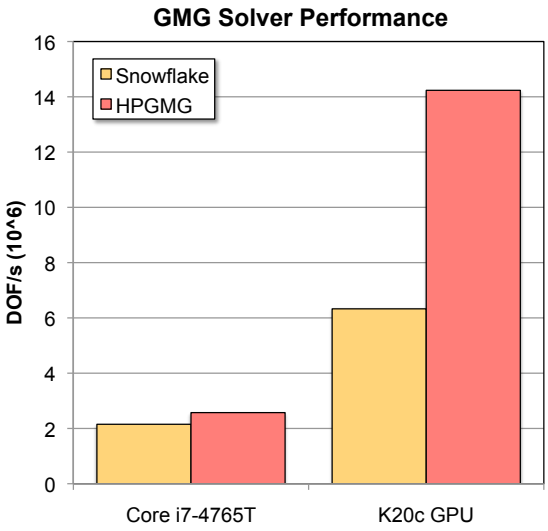


Figure 9: Single source Snowflake GMG solve performance is comparable to hand-optimized HPGMG on CPUs and productively attains nearly half the performance of optimized HPGMG-CUDA on GPUs for a 256^3 problem.

optimized HPGMG and HPGMG-CUDA as well as the Snowflake OpenMP and OpenCL generated code running on CPUs and GPUs. Observe that Snowflake delivers very similar performance on a CPU to the hand-optimized HPGMG (memory bandwidth bound). Conversely, Snowflake performance on the GPU is roughly half the performance of hand-optimized CUDA — no surprise given multigrid is usually dominated by operations on the finest grids (largest arrays) for which Figure 8 highlighted performance differences. Future optimization effort will attempt to improve the quality of the generated OpenCL code.

VI. Related Work

Much work has been focused on optimizing stencils (a special case of loop nests) by improving general static analysis and transformation techniques. PLuTo [17] and PENCIL [18] apply the polyhedral model to recognize, estimate the profitability of, and effect loop transformations. The CHiLL framework [19], [20] also uses polyhedral analysis, but it eschews automatic techniques in favor of user-provided scripts that direct loop transformations. General-purpose compilers usually fall short of hand-optimized implementations, and their generality inhibits portability across architectural backends.

Domain-specific languages are another means of achieving fast, portable stencils, but DSLs for stencils have yet to handle the full complexity of scientific codes and present a challenge for inter-operability with legacy code. PATUS [21] separates the stencil definition from the implementation strategy to enable auto-tuning. Halide [6] enforces a similar separation, and optimizes pipelines of image processing stencils via interval analysis. Halide does not support boundary conditions because they conflict with the assumption that stencils are implicitly applied over infinite grids. The Pochoir Compiler [22] implements stencils via a cache-oblivious algorithm. It supports complex boundary conditions, but it does not support variable-coefficient stencils because each stencil can only be associated with one array. SDSL [23] attempts to combine domain-specific knowledge with the polyhedral model. It supports boundary conditions and variable-coefficient stencils, but SDSL cannot express stencils that do interpolation or restriction because it defines stencils in terms of additive offsets, rather than the requisite multiplicative offsets. Lastly, neither Halide, Pochoir, or SDSL support in-place stencils, and thus cannot express in-place GSRB.

The platform portability problem has been a perennial challenge and there have been many attempts to leverage a higher-level language to abstract away platform differences. Perhaps the most popular example is the Java programming language, designed to separate the user from the system.

While JIT code generation is not new, our Python-based code generation is based on the Python-to-C and Python-to-GPU work done by Catanzaro et al. [4] and Kamil et al. [24]. Like their work, we expose code generation at the pattern level, as opposed to the assembly level or

intermediate-representation level. Like Kamil's work, we selectively specialize only those parts of a Python program that use our embedded DSL, so the same application can combine Snowflake with other code-generation frameworks.

Like the above work and also Olukotun et al. [3], our goal is that scientists can use a productivity-level language to describe algorithms most of whose code will ultimately be run in an efficiency-level language. DSL design and implementation has been a popular method for resolving the platform portability problem by specializing a subset of computations. Pochoir [22] and Halide [6] have been successful within the stencil domain.

Our backend optimization framework is similar to Rose [25], a compiler framework and toolkit for creating code-to-code translators, optimizers, and compilers.

Diophantine equations have been used to analyze individual stencils but we believe our extension to detect dependencies across multiple stencils (e.g. between the face stencils and the interior stencils) and to do this analysis on unions of domains for multi-color stencils (i.e. GSRB) is novel. Traditional Diophantine analysis assumes an infinite domain, while our analysis works with finite domains, which allows it to recognize (for example) that our linear Dirichlet edges don't interact with our faces. Finite-domain dependency analysis also lets us run multiple different stencils on the interior at the same time if they are non-overlapping. The Halide framework assumes an infinite domain, and doesn't support in-place stencils.

VII. Future Work

We plan to evaluate Snowflake on a more diverse set of platforms including the Intel Knights Landing manycore processor. In addition, the performance on GPUs is less than ideal, and in order to fully and efficiently accommodate scientific stencils the performance will need to be closer to hand-written code. We plan to augment the OpenCL micro-compiler and explore the creation of CUDA, OpenACC, or OpenMP 4 micro-compilers.

We will incorporate dead-stencil elimination and reordering optimizations into our dependency analysis framework. We will extend the analysis to mark stencils for fusion and reordering by analyzing dependencies and memory access patterns. Extended dependency analysis will likely also include extracting dependency graphs of stencils and we hope to eventually make decisions on splitting across multiple backends (i.e. CPU and GPU) in order to increase parallelism.

Finally, we're exploring the development of new backends to target distributed-memory systems via MPI or UPC++. Happily, this will also provide performance on NUMA node architectures (e.g. multisocket Xeons) by running one process per NUMA node.

VIII. Conclusions

In this paper, we presented a novel solution to the performance portability problem for stencils on structured grids, probably the most important computational kernel in scientific computing due to both it prevalence across scientific domains and the difficulty of packaging all possible stencil operators into a hand-optimized library. Our Snowflake language balances the simplicity of a domain-specific approach, while providing the full generality of boundary conditions, strided iteration spaces, update-in-place operations, and other features required for real scientific applications. Our simple yet powerful analysis engine is able to handle the full generality of Snowflake stencils, including variable-coefficient stencils, through the JIT approach and micro-compiler backend architecture, which makes it easy to retarget Snowflake to new architectures. While we do not yet support all known stencil optimizations, our code is competitive with hand-optimized code for CPU multicore and GPU-accelerated architectures. This is even more significant because Snowflake is embedded in Python. Moreover, we included a Roofline analysis to show that the Snowflake code is close to the best possible performance of a given system.

Both functional and performance portability of scientific applications has become the primary software engineering concern in scientific computing today, with the diverse architectural models requiring special language annotations or completely new languages. Trends suggest that this problem will only get worse as architectures are emerging with new levels of memory and new kinds of accelerators optimized for particular types of computation. The Snowflake approach shows that by tailoring the language to a narrow domain, and leveraging powerful algebraic tools in analysis, one can produce highly optimized code across diverse architectures in a framework that is extensible to new architectures and manageable in terms of code size and complexity.

Acknowledgments

This research used resources in Lawrence Berkeley National Laboratory and the National Energy Research Scientific Computing Center, which are supported by the U.S. Department of Energy Office of Science's Advanced Scientific Computing Research program under contract number DE-AC02-05CH11231. Research at UCB partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Hewlett-Packard, Huawei, LGE, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

REFERENCES

[1] K. Yelick, "Programming Models for SOCs in HPC (A play in 3 Acts)," http://www.socforhpc.org/wp-content/uploads/2015/10/PACT2015_Yelick-SOCHPC-2015.pdf, SOCHPC Keynote, 2015.

[2] J. Holewinski, T. Henretty, K. Stock, L. Pouchet, A. Rountev, and P. Sadayappan, "High-performance computing for stencil computations using a high-level domain-specific language," hpc.pnl.gov/conf/wolfhpc/2011/talks/JustinHolewinski.pdf.

[3] T. Rompf, A. K. Sujeeth, H. Lee, K. Brown, H. Chafi, M. Odersky, and K. Olukotun, "Building-blocks for performance oriented dsls," *IFIP Working Conference on Domain-Specific Languages*, 2011.

[4] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "Sejits: Getting productivity and performance with selective embedded jit specialization," *Workshop on Programming Models for Emerging Architectures*, 2009.

[5] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 35–46. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941561

[6] J. Ragan-Kelly, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Programming Language Design and Implementation*, 2013.

[7] E. Jones, T. Oliphant, P. Peterson *et al.*, "Open source scientific tools for Python," 2001.

[8] F. Pérez and B. E. Granger, "IPython: a system for interactive scientific computing," *Computing in Science and Engineering*, vol. 9, no. 3, pp. 21–29, May 2007. [Online]. Available: http://ipython.org

[9] "HPGMG: High-performance geometric multigrid repository," https://bitbucket.org/hpgmg/hpgmg/.

[10] J. Holewinski, L. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," *International Conference on Supercomputing*, 2012.

[11] M. Davis, Y. Matiyasevich, and J. Robinson, "Hilbert's tenth problem. diophantine equations: positive aspects of a negative solution," *Proc. Symp. Pure Math*, vol. 28, pp. 323–378, 1976.

[12] T. Ligocki, "Roofline toolkit." [Online]. Available: https://bitbucket.org/berkeleylab/cs-roofline-toolkit

[13] "HPGMG-CUDA: High-performance geometric multigrid (CUDA) repository," https://bitbucket.org/nsakharnykh/hpgmg-cuda.

[14] S. Williams, A. Watterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Communications of the ACM*, April 2009.

[15] S. Williams, "Auto-tuning performance on multicore computers," Ph.D. dissertation, EECS Department, University of California, Berkeley, December 2008.

[16] J. D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," https://www.cs.virginia.edu/stream/.

[17] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2008.

[18] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, "Pencil: A platform-neutral compute intermediate language for accelerator programming," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct 2015, pp. 138–149.

[19] C. Chen, J. Chame, and M. Hall, "CHiLL: A framework for composing high-level loop transformations," University of Utah, Tech. Rep., 2008.

[20] P. Basu, A. Venkat, M. Hall, S. Williams, B. V. Straalen, and L. Oliker, "Compiler generation and autotuning of communication-avoiding operators for geometric multigrid," in *20th Annual International Conference on High Performance Computing*, Dec 2013, pp. 452–461.

[21] M. Christen, O. Schenk, and H. Burkhart, "PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 676–687. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.70

[22] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2011, pp. 117–128.

[23] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 311–320. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304619

[24] S. Kamil, D. Coetzee, and A. Fox, "Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization," in *10th Python in Science Conference (SciPy 2011)*, Austin, TX, July 2011.

[25] "Rose compiler infrastructure," http://rosecompiler.org.