On Optimizing Complex Stencils on GPUs

Prashant Singh Rawat†, Miheer Vaidya†, Aravind Sukumaran-Rajam†, Atanas Rountev†,
Louis-Noël Pouchet‡, P. Sadayappan†
†The Ohio State University, USA ‡Colorado State University, USA
{rawat.15,vaidya.56,sukumaranrajam.1,rountev.1,sadayappan.1}@osu.edu, pouchet@colostate.edu

***Abstract*—Stencil computations are often the compute-intensive kernel in many scientific applications. With the increasing demand for computational accuracy, and the emergence of massively data-parallel high-bandwidth architectures like GPUs, stencils have steadily become more complex in terms of the stencil order, data accesses, and reuse patterns. Many prior efforts have focused on optimizing simpler stencil computations on various platforms. However, existing stencil code generators face challenges in optimizing such complex multi-statement stencil DAGs.**

This paper addresses the challenges in optimizing high-order stencil DAGs on GPUs by focusing on two key considerations: (1) enabling the domain expert to guide the code optimization, which may otherwise be extremely challenging for complex stencils; and (2) using bottleneck analysis via runtime profiling to guide the application of optimizations, and the tuning of various code generation parameters.

We implement these abstractions in a prototype code generation framework termed ARTEMIS, and evaluate its efficacy over multiple stencil kernels with varying complexity and operational intensity on an NVIDIA P100 GPU.

I. INTRODUCTION

Stencils are a common computational motif in many scientific and engineering applications. A stencil computation sweeps a computational grid, and updates the value at each point by reading a fixed neighborhood around it. The extent of the neighborhood read from the center along each dimension is called the *stencil order*. The updates can be either time-iterated, involving repeated application of the same stencil operator [1], or spatial and applied to multiple domains in a sequence of steps, as seen in multi-statement stencil DAGs and image processing pipelines [2].

In the recent years, stencil computations have evolved in complexity. From low-order, time iterated stencils, the optimization focus has shifted on high-order, multi-statement, and compute intensive stencil DAGs [3]–[5]. For example, the stencils in many scientific simulations are order-2 and above, performing upwards of 300 double-precision floating point operations (FLOPs), while reading from upwards of 8 3D input domains [3], [4]. We term such stencils as *complex* stencils.

Most stencil computations are inherently bandwidth-bound. Architectures like GPUs provide massive parallelism and higher bandwidth compared to multi-core CPUs, making them an ideal acceleration platform for such stencils. For this reason, there has been a considerable interest in optimizing stencils on GPUs [1], [6]–[15]. Manual optimization of stencil computations on GPUs is daunting, tedious, and error prone. An attractive alternative is to automate the code generation from some input specification of the stencil computation. Halide [2], Forma [11], PPCG [6], [16], Overtile [1], STENCILGEN [9], [17], Physis [18], Mint [19], Patus [20], etc. are some of the code generation frameworks developed over the past few years that generate optimized stencil code from an input specification in a domain-specific language (DSL). Many of these frameworks have demonstrated the effectiveness of tiling, fusion, and autotuning in optimizing both iterative and multi-statement stencils on GPUs. However, there still remains a significant gap between theory and practice when applying these optimization techniques to complex stencil computations. The difficulty in bridging this gap arises from multiple factors, two of which are discussed below.

1. Stencil complexity: The demonstration of high performance via automated code generation in many prior works was limited to low-order, single-precision iterative stencil computations [1], [6]–[8], [21]. Current GPU devices have under 100 KBytes of shared memory per streaming multiprocessor (SM), and allow up to 255 registers per thread during compilation. These hardware constraints are much more severe on complex stencils as compared to simple low-order stencils. For example, to allocate shared memory for 8 or more input domains, a significant reduction in block size will be required, which consequently would increase the volume of data movement as well as recomputations with overlapped tiling, and degrade performance [1], [11]. This raises a question from the code generation perspective: *Can the same optimizations be applied to both simpler, low-order stencils and the more complex, high-order stencils?*

2. Limited effectiveness of performance models in code generators: The current code generators implement a set of optimization strategies, and either (a) apply all the optimizations simultaneously on the input stencil [9]; (b) rely on the user to specify the optimizations [2], [11]; or (c) use heuristics driven by simple cost models to apply selected optimizations [16]. Unfortunately, the internal cost models that have been developed to date are far from satisfactory in their discriminating power to identify the best or close-to-best version of code to generate. For example, real world stencils are often bandwidth-bound at different cache hierarchies, or latency-limited and register-constrained with low occupancy [22]. Ill-applied optimizations on them may be counterproductive, like performing thread coarsening for register-constrained stencils.

The only viable alternative is autotuning, and indeed frameworks like Facebook's Tensor Comprehensions rely heavily on autotuning using a genetic algorithm in order to produce high-

performance code [23]. However, the space of possible GPU code configurations that can be generated for stencils can be extremely large, with many possible choices on buffering different arrays in shared-memory or registers, thread-block sizes along multiple dimensions, thread coarsening factors along multiple dimensions, loop unrolling factors, tiling strategy, prefetching strategy, etc. Thus, a question that cannot currently be effectively answered from a code generation perspective is: *What optimizations should be applied to the given stencil computation at each step to further improve its performance?*

To the best of our knowledge, none of the open-source, automated stencil code generation frameworks address the above-mentioned issues for complex stencils on GPUs. PPCG suffers from inefficient resource assignment heuristics, Forma and Overtile lack optimizations that would accelerate 3D stencils, and Halide relies on the application developer to specify the entire schedule, which requires significant expertise in code optimization. Even the GPU autoscheduler of Halide suffers due to the implemented heuristics, leading to a 2× slowdown in performance for complex stencils [17].

Solution Approach: In this paper, we discuss an approach that we are currently pursuing to address the above mentioned problems. We present a GPU stencil code generation framework called ARTEMIS (github.com/pssrawat/artemis), and experimental results using it on a number of stencils. The key ideas behind ARTEMIS are as follows:

- The code generation framework must incorporate a variety of optimization strategies thereby relieving users of the burden of code generation, but it must allow optional guidance from expert users on optimization options.
- Accurate performance modeling is extremely hard, but effective iterative optimization can be performed by focusing on resource bottlenecks, even without the ability to accurately predict the performance of alternate configurations.
- Autotuning can be a powerful aid to optimization, but the use of generic search strategies like genetic algorithms makes it extremely time consuming. Instead, the use of bottleneck analysis from hardware counter data can enable rapid and effective autotuning.

These key observations are general enough to be incorporated into any code generation framework. Even though developed as a prototype, the novelty of ARTEMIS itself stems from the following three factors:

- It implements a wide variety of optimizations that help accelerate stencils with varying complexity. None of the other stencil code generations frameworks we know of implement all the optimizations covered by ARTEMIS.
- It provides an end-to-end solution to optimize complex stencils by incorporating optimizations, profiling, bottleneck analysis, and autotuning to realize high performance. The performance bottleneck analysis helps prune the autotuning search space, which can be quite vast when explored with tools like OpenTuner [24].
- It automates all tedious aspects of efficient stencil code generation, but also allows application experts to *optionally* provide supplemental information that can guide code generation. This approach fits nicely between the two extreme paradigms in code generation: Halide's approach of separating schedule from computation, and the single-shot optimization strategy of Forma, PPCG, and Overtile.

The rest of the paper is organized as follows. Section II presents a DSL that captures all the information necessary to optimize complex stencils on GPUs. Section III describes a variety of stencil optimizations. However, optimizations are only effective when they target the computation's bottleneck. To this end, Section IV describes a profiling strategy that determines the bottlenecks in the computation, and can be used to decide the utility/futility of certain optimizations. Sections V and VI describe the autotuning strategies for both iterative and spatial stencils, centered around bottleneck analysis and alleviation. Section VII summarizes the integration of these different optimization techniques, and the overall flow of ARTEMIS. Section VIII presents experimental evaluation, comparing the performance of ARTEMIS with other special-purpose stencil code generators. Section IX discusses the related work, and Section X presents our conclusions.

II. DOMAIN-SPECIFIC LANGUAGE

DSLs are often used to expose the three P's (performance, portability, and productivity) in specific domains. Halide [2], PolyMage [25], SDSL [26], Forma [11], and Pochoir [27] are a few popular examples of stencil DSL frameworks. Since all these frameworks were developed independently, each uses its own distinct language to capture the semantics of the computation. Despite this, the common aspects of their languages becomes apparent on closer inspection.

All stencil DSLs require the user to declare the arrays and scalars used in the stencil computation, along with the read-only iterators that are mapped to the unique dimensions of the computational loop nest. For GPUs, the DSLs may also expect the user to specify arrays and scalars that must be copied from host (device) to device (host). The core stencil computation is often expressed in a restricted subset of C: all the memory accesses in the stencil function are scalars or array elements, and the array index expressions are an affine function of the loop iterators and constants [1], [2], [25]–[27]. A loop construct may be used to specify the time loop for iterative stencils [26]. Different DSLs use different techniques to transfer other auxiliary information, like the block/grid size to be used during code generation.

A. A Minimal Stencil Language

Listing 1 shows a minimal DSL, similar in construct to SDSL [26] and Forma [11], that concisely captures the semantics of a 3D 7-point Jacobi stencil from HPGMG [28]. This minimal DSL serves as the starting point of ARTEMIS language. The read-only parameters `L`, `M` and `N` in line 1 are used to describe the dimensions of the input and output arrays. Line 2 declares the read-only iterators from outermost to innermost; these are assumed to be incremented in unit steps by the increment condition of the loop. All declared

Listing 1: Representative DSL for Jacobi stencil [28]

```
1  parameter L=512, M=512, N=512;
2  iterator k, j, i;
3  double in[L,M,N], out[L,M,N], a, b, h2inv;
4  copyin out, in, h2inv, a, b;
5  #pragma stream k block (32,16) unroll j=2
6  stencil jacobi (B, A, h2inv, a, b) {
7    double c = b * h2inv;
8    B[k][j][i] = a*A[k][j][i] - c*(A[k][j][i+1]
9       + A[k][j][i-1] + A[k][j+1][i] + A[k][j-1][i] +
10      A[k+1][j][i] + A[k-1][j][i] - A[k][j][i]*6.0);
11 }
12 jacobi (out, in, h2inv, a, b);
13 copyout out;
```

arrays and scalars in line 3 will be passed as arguments to the generated host function. Line 4 (line 13) specifies the arrays that must be copied from host to device (device to host). Lines 6–11 define the `jacobi` stencil. Many DSLs represent the stencil computation as shift vectors, storing just the offsets from the center point along each dimension [9], [11]. With such DSLs, one cannot express computations involving domains of different dimensionalities. It also means that the user has to spend significant time in rewriting the core stencil computation from C/C++/Fortran to shift vectors. For this reason, we choose to retain the C/C++ flavor for the stencil function in the minimal DSL (lines 8–10). The `#pragma` at line 5 accepts auxiliary information for the optimizations to be applied (described later in Section III): streaming dimension, block size, and the unroll factors. This information will be used in generating code for the stencil definition immediately following the pragma.

B. ARTEMIS-*Specific Extensions to the Minimal Language*

1) User-Guided Resource Assignment: An efficiently optimized GPU code must allocate appropriate GPU memory resources, like shared/constant memory and registers, to the input/output/intermediate arrays. Most code generators automatically determine this resource assignment (or mapping), but without considering the resource limits of the underlying GPU device. The challenges in resource assignment exacerbate for high-order stencils, since the resource consumption increases with the stencil order. To illustrate this, consider the *rhs4center* kernel from SW4 routine [4]. The kernel is an order-2 double-precision stencil, reading from five 3D input arrays, namely *u0, u1, u2, mu*, and *la*. Even if we assume that all the code generators are capable of generating a code with streaming, most code generators will still use 5 shared memory buffers per input array. This implies that on a GPU device with 48KB shared memory, we must use a block size strictly smaller than 16×16, since $\frac{48*1024}{16*16*8*5*5} < 1$. Apart from reduced occupancy, a consequence of such small block size is an increase in recomputation volume, which adversely affects performance.

Proposed Extension: Clearly the code generator must automatically determine the resource mapping. However, we propose that the domain expert be allowed to *optionally* specify some resource assignments in the input specification that the code generator *must* adhere to. We extend the representative DSL of listing 1, and allow the user to write "`#assign shmem` (*u0,u1,u2*), `gmem` (*mu,la*)" within the stencil function, indicating that arrays *u0, u1, u2* must be cached in shared memory, and arrays *mu, la* must be read directly from global memory. Now we can use a 16×16 block, alleviating some occupancy drop and reducing the recomputation volume.

2) User-Guided Resource Rationing: Current code generators do not take register pressure into account while performing resource assignment or optimizations. It has been shown that some kernels perform better at lower occupancy due to lesser contention at L2 cache [29]. Also, the performance of compute-intensive kernels with many-to-many reuse can be extremely sensitive to register pressure [22]. For example, even if the compiler generates spill-free code at both 128 or 255 registers, instruction-level parallelism (ILP) can be significantly lower at 128 registers, restraining performance. Often, a domain expert has a better understanding of such performance quirks, and may want to target a lowered occupancy, thereby allowing more shared memory and registers per thread block.

Proposed Extension: Since statically estimating an optimal occupancy is extremely error-prone [30], we propose that the domain expert be allowed to *optionally* specify the target occupancy for kernels. We can extend the `pragma` of Listing 1 with clause "`occupancy` t", where $0 < t \leq 1$, indicating that tX out of X threads must be active per SM. If the shared memory per block prevents the code generator from generating code for the targeted occupancy, then the resource mapping algorithm must choose a shared memory buffer with minimum number of accesses, and demote its storage to global memory. This process is repeated till the shared memory usage is no longer a bottleneck in achieving the targeted occupancy.

III. OPTIMIZATIONS OF INTEREST FOR STENCILS

A. *Current State-of-the-Art*

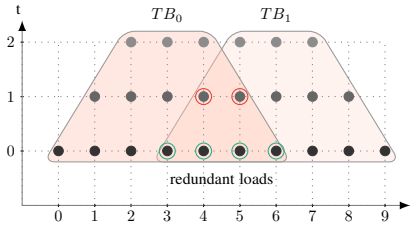
It will not be an exaggeration to say that optimizations form the crux of stencil code generation frameworks. We briefly discuss four optimizations that are common to multiple stencil code generators, including ARTEMIS.

1) Overlapped Tiling: Overlapped tiling [1] is used to achieve concurrent execution of thread blocks assigned to different tiles of the output domain [1], [2], [8], [9], [13]. Figure 1b shows the overlapped tiling scheme to time-tile the *j1d3pt* stencil of Figure 1a. The domain points highlighted in red (green) are the redundant computations (loads), which is a consequence of the overlap.

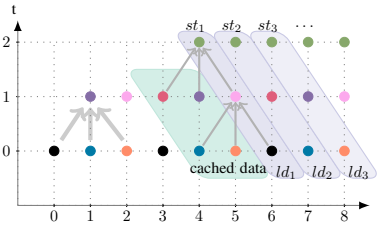
2) Serial Streaming: Streaming [8], [14], [15] is an optimization to reduce the redundancy in overlapped tiling. The idea of streaming is based on the observation that in an order-k 3D stencil, at most $2k + 1$ planes of an input domain are needed in memory to compute an output plane. Therefore, one can overlap-tile just the two dimensions of the 3D domain, and *stream* along the third dimension. This strategy exposes spatial reuse along the streamed dimension, and reduces the shared memory utilization, thereby allowing an increase in the thread block size to reduce redundancy. Figure 1c depicts the streaming strategy for *j1d3pt* stencil. Different colors at different time steps correspond to distinct shared memory buffers. In the steady state, two out of the three points at each time step are cached in shared memory. One input point needs

```
1 double A[2][N], c;
2 for (int t=0; t<2; t++) {
3   int w = (t+1)%2, r = t%2;
4   for (int i=0; i<N; i++) {
5     A[w][i] = c*(A[r][i+1] +
6           A[r][i] +
7           A[r][i-1]) / 3.0;
8   }
9 }
```

(a) Two time steps of *j1d3pt*



(b) Overlapped tiling



(c) Streaming

Fig. 1: Different tiling schemes for 3-point 1D Jacobi computation

to be loaded along the streaming dimension at time step 0 to compute a new output value at other time steps.

3) Loop Unrolling: Loop unrolling is traditionally used to expose ILP in the computation. Consider a warp in a block where each thread is responsible for computing two output points, i.e., the unrolling factor is 2. Then, the unrolled version can have the following two work distribution strategies: (1) *cyclic*, where thread in lane i computes output points indexed at $m+i$ and $m+32+i$; and (2) *blocked*, where thread in lane i computes output points indexed at $m+2*i$ and $m+2*i+1$. There will be a better register-level reuse with blocked distribution due to the near-neighbor dependences of stencil computations; this reuse can be further optimized for CSE or register pressure using other DSL code generators [22].

4) Prefetching: The main computational loop with streaming (Figure 1c) comprises three phases: (1) computation of an output point per time step; (2) shifting of the data in the shared memory buffers; and (3) loading a new point from the input array into the least recently written shared memory buffer. To guarantee correctness, there must be a synchronization barrier between the first phase and the second phase. This barrier prevents an overlap of compute and memory access operations between the phases. With prefetching, one can load the elements for the third phase into *prefetch* registers simultaneously with the computation in the first phase. Since there is no synchronization barrier between the load and compute phase now, the compiler interleaves the memory and compute operations to hide the data transfer latency.

Different frameworks implement different optimizations, depending on the targeted stencil application. Halide implements inlining for nodes with producer-consumer relationship in image processing pipelines. Overtile and Forma implement time tiling for iterative stencils. Zhang et al. [10] implement thread coarsening for spatial stencils. Streaming is implemented in [2], [9], [12], [13], and loop unrolling is implemented in [1], [2], [6], [7], [10]. Prefetching is implemented in frameworks optimizing GEMM kernels [31].

B. ARTEMIS-*Specific Optimizations*

Below, we discuss four other optimization strategies that, to the best of our knowledge, are novel with respect to automation in ARTEMIS.

1) Concurrent streaming: The streaming discussed in Section III-A2 is serial, where the $N-1$ dimensions of an N-dimensional domain are tiled, and a thread block traverses the untiled dimension in serial. In concurrent streaming, all N dimensions are overlap-tiled, and a thread block traverses one of the tiled dimension in serial. Concurrent streaming improves the performance of computations that suffer from low execution concurrency with serial streaming [8].

2) Statement decomposition and retiming: The manual retiming approach of Rawat et al. [8] involves leveraging operator associativity and distributivity to decompose a stencil statement into a set of accumulation sub-statements. A direct consequence of retiming is better balancing of GPU resource usage between memory and registers. STENCILGEN [9], [17] retimes the computation if the stencil statements are manually massaged to a form that is amenable to retiming. We eliminate the need for any manual transformation by automating both statement decomposition and retiming in ARTEMIS.

For retiming, ARTEMIS checks if the RHS of each sub-statement is *homogenizable*. An expression can be homogenized if the offset along the streaming dimension (or the slowest varying dimension if streaming is disabled) can be reduced to 0 for all the accesses in it. For example, when streaming along k dimension, the RHS of statement $B[k][j][i] = A[k-1][j][i]$ can be homogenized by adding 1 to the array access expression involving k on both sides. In contrast, the RHS in the statement $B[k][j][i] = C[k+1][j][i] * A[k-1][j][i]$ cannot be homogenized. If the RHS in all the sub-statements can be homogenized, then ARTEMIS retimes the computation.

3) Thread block load/compute adjustment: Assume that for an order-k stencil, the output tile size is $b_y \times b_x$, and the input tile size is $(b_y + 2k) \times (b_x + 2k)$. Most code generation frameworks use one of the following strategies to choose thread block size: (1) *output perspective*: create $b_y \times b_x$ thread blocks, so that each thread computes one output point. The k threads at each block boundary may need to load additional k elements from the input domain, while the internal threads remain idle [10]; and (2) *input perspective*: create $(b_y+2k) \times (b_x+2k)$ thread blocks, so that each thread loads exactly one input element. The $2k*(b_x+2k)+2k*b_y$ threads at the boundary may not be involved in the computation of the output domain [11].

A thread block with output perspective may issue non-coalesced load transactions at boundary, which can be particularly wasteful for bandwidth-bound stencils. However, when the occupancy is really low, then input perspective can be detrimental to performance, since at least $2k$ warps along the y dimension will be idle after the initial loads. Clearly, which perspective to choose should depend on the target occupancy of the kernel. Apart from these two perspectives, ARTEMIS provides an additional *mixed* perspective with block

Listing (2) Optimized code for the spatial 3D 7-point stencil

```
in_shm_c0[j-j0][i-i0] = in[1][j][i];
in_reg_m1 = in[0][j][i];
in_reg_p1 = in[2][j][i];
for (k=1; k<N-1; k++) {
  __syncthreads ();
  if (j>=j0+1 & j<=min(j0+blockDim.y-2, M-2) & ...)
    out[k][j][i] = a*in_shm_c0[j-j0][i-i0] - ...;
  __syncthreads ();
  in_reg_m1 = in_shm_c0[j-j0][i-i0];
  in_shm_c0[j-j0][i-i0] = in_reg_p1;
  in_reg_p1 = in[max(L-1,k+2)][j][i];
}
```

Listing (3) Reducing memory footprint for Listing 2

```
in_shm_c0[j-j0][i-i0] = in[0][j-j0][i-i0];
in_reg_m1 = in[0][j-j0][i-i0];
in_reg_p1 = in[0][j-j0][i-i0];
for (k=1; k<N-1; k++) {
  __syncthreads ();
  if (j>=j0+1 & j<=min(j0+blockDim.y-2, M-2) & ...)
    out[0][j-j0][i-i0] = a*in_shm_c0[j-j0][i-i0] - ...;
  __syncthreads ();
  in_reg_m1 = in_shm_c0[j-j0][i-i0];
  in_shm_c0[j-j0][i-i0] = in_reg_p1;
  in_reg_p1 = in[0][j-j0][i-i0];
}
```

size $b_y \times (b_x + 2k)$ which eliminates any idle warps along the y dimension, and any non-coalesced loads along x dimension if $b_x + 2k$ is a multiple of the warp size.

4) Storage and Computation Folding: A common motif in many spatial stencils is an element-wise operation between two or more arrays. If all the accesses to arrays $A_r, 0 \leq r \leq n$ are of the form $\bigodot_{r=0}^{n} A_r[i]$ at each domain point i, where $\odot$ is any point-wise mathematical operation, then instead of storing $A_0, \ldots, A_n$ independently in shared memory or register, we can simply store the result of $\bigodot_{r=0}^{n} A_r[i]$ in shared memory or register. This not only reduces the resource usage, but also optimizes away the recomputations at source level.

Automating such a wide set of optimizations within a single framework is a significant engineering effort. Armed with these optimizations, ARTEMIS can accelerate both time-iterated 2D/3D stencils, and complex spatial stencils with many-to-many reuse alike.

IV. PROFILING FOR BOTTLENECKS

A detail that is well known to the application developers, but usually not addressed by the current generation of automated code generators is the fact that *an optimization is only effective if it alleviates one or more performance bottleneck of the computation.* Merely implementing a set of optimization strategies in itself is not enough. There should be an analysis to estimate the bottlenecks of the computation, and the optimizations applied must be driven by those bottlenecks. This section describes a simple profiling technique implemented in ARTEMIS, that is used to gauge the profitability of some of the optimizations described in Section III. Additionally, as elaborated later in Section VI-A, the profiling component is also used in tandem with autotuning to determine the degree of fusion in iterative stencils.

While executing on GPU, the stencil data is cached at different levels in the GPU memory hierarchy, e.g., DRAM, texture/L2 cache, shared memory, registers, etc. Depending on the complexity of the accesses and computation, the kernel may be bottlenecked at different GPU resources. A crucial step in optimizing the performance is to analyze the performance bottlenecks for different GPU resources.

Performance characterization of computations on GPUs is a vast research topic in itself. Several analytical models have been proposed with the objective of estimating the execution time of the computation on GPUs [32]–[35], but they can be quite inaccurate across applications. On the other hand, cycle-accurate simulators are much slower than executing the application itself. Both these approaches are therefore unsuitable for performance bottleneck analysis.

The profiling component implemented in ARTEMIS overcomes the unreliability of analytical models by using the roofline model [36] in conjunction with NVIDIA's profiling tool, nvprof [37]. It first uses nvprof to execute and profile the kernel to collect the counters for metrics of interest, and then uses those metrics to compute the operational intensity (OI) for different memory levels in the memory subsystem. For memory level M, the operational intensity, OI_M, is defined as the ratio of the FLOPs computed relative to the data accesses in bytes from M. ARTEMIS currently computes the OI for DRAM (OI_{dram}), texture cache (OI_{tex}), and shared memory (OI_{shm}) only. For memory level M, the maximum FLOPs that can be computed per byte accessed from M is $\frac{\alpha}{\beta_M}$, where α is the peak performance of the device, and β_M is the peak bandwidth for memory level M. The user is expected to provide these theoretical peak values for the GPU device to ARTEMIS. Then, using the roofline model [36], ARTEMIS classifies a kernel as:

- bandwidth-bound at memory M, if $\text{OI}_M \ll \frac{\alpha}{\beta_M}$. This happens when the data transfers dominate the execution time.
- compute-bound at memory M, if $\text{OI}_M \geq \frac{\alpha}{\beta_M}$. This happens when the computations dominate the execution time.

Additionally, a kernel can be latency-bound when the GPU does not have enough concurrency to hide the latency of arithmetic instructions. Such kernels are neither bandwidth-bound at any memory level, nor compute-bound.

When OI_M is closer to $\frac{\alpha}{\beta_M}$, categorizing the kernel as bandwidth- or compute-bound is difficult. For such kernels, ARTEMIS uses *code differencing*: for an input V, it automatically generates a modified version V' with drastically reduced accesses to M. For example, Listing 3 illustrates the modified version V' generated for the code V in Listing 2. For a 32×32 block, the values of `i-i0` and `j-j0` (highlighted in blue in Listing 3) correspond to the thread index along i and j dimension, respectively. Therefore, the DRAM accesses in Listing 2 are confined to a 32×32 block for each global array in Listing 3. Consequently, DRAM transactions can no longer be a bottleneck in V'. If the performance of V' improves over V, then V is classified as DRAM bandwidth-bound.

A. Key Insights from Profiling

With the profiling information, ARTEMIS internally decides which optimizations to apply. In some cases, it also generates alternate versions for the user to optimize, and outputs some textual optimization hints based on the following guidelines:

- If the stencil is compute-bound, then shared memory optimizations, or optimizations like unrolling that improve ILP, are not useful, and turned off. On the other hand, optimizations that reduce the FLOPs are applied.
- If the stencil exhibits high register pressure or register spills, then loop unrolling is turned off. Instead, ARTEMIS generates versions of the kernel with varying degree of fission (Section VI-B).
- If an iterative stencil is bandwidth-bound at texture cache and/or DRAM, then further fusion will help by reducing texture cache accesses; ARTEMIS generates a kernel with higher fusion degree for such stencils (Section VI-A).
- If a spatial stencil is texture cache bandwidth-bound, then shared memory optimization is turned on by default.
- If a spatial stencil is highly DRAM bandwidth-bound with shared memory, then ARTEMIS generates its global memory version for the user to optimize. This is because the shared memory version may experience a performance degradation due to the additional shared load/store transactions. ARTEMIS also hints the user to manually perform any algorithmic optimization that would possibly reduce DRAM accesses or stencil order.
- If a kernel is bandwidth-bound at shared memory, then the register-level optimizations from Section III are turned on.

V. HIERARCHICAL AUTOTUNING

Stencil code generation frameworks often incorporate an autotuning component to find the near-optimal thread block configuration and/or unrolling factors [26], [38], [39]. For example, Overtile use a customized autotuner, whereas Halide uses OpenTuner [24], that can be customized to different frameworks. A general concern with autotuning is the amount of time taken to explore the vast optimization space.

We incorporate a customized autotuner in ARTEMIS to tune the parameter configurations like unrolling factors, thread block size, etc. Instead of relying on an analytical model, our autotuner, like OpenTuner, generates multiple versions of the code, and executes them to measure performance. The autotuner initially prunes the search space by making the following three logical choices based on the computation: (1) the block sizes and unrolling factors are limited to powers of two along each dimension; (2) the block sizes are lower-bounded by 4 and upper-bounded by 256 along each dimension; and (3) the unrolling factor are upper-bounded by 8 and 4 along each dimension for theoretically bandwidth-bound and compute-bound stencils, respectively. These choices conform to the tuned parameters discovered by other autotuners [10], [26].

The search space is further pruned with *hierarchical autotuning*, i.e., autotuning in steps. To illustrate hierarchical autotuning, assume that the profiling information indicates that all the optimizations described in Section III can benefit the computation. As a first step, the autotuner tunes the computation for just the *high-impact* optimizations like loop unrolling factors and thread block size variations, with serial streaming enabled by default if shared memory is used. It then selects a few high-performing candidates, and applies optimizations like prefetching, concurrent streaming, and thread block load-/compute adjustment to just those candidates. Depending on the desired granularity of tuning, ARTEMIS allows the user to define their own hierarchy of optimizations for autotuning. The unrolled versions are explored in a sequence so that the number of stencil statements post unrolling monotonically increases. For example, if the unrolling factors are (u_z, u_y, u_x) along dimensions (z, y, x), then the autotuner explores the versions in increasing order of $u_z \times u_y \times u_x$. The higher the unrolling factor, the more registers each thread is expected to use. This allows the autotuner to dynamically increment the registers per thread (between 32, 64, 128, and 255) during autotuning, so that only non-spill configurations are explored. For a spatial 7-point Jacobi stencil, OpenTuner took more than 24 hours for exhaustive autotuning, whereas hierarchical autotuning arrived at a version with similar performance in less than 5 hours.

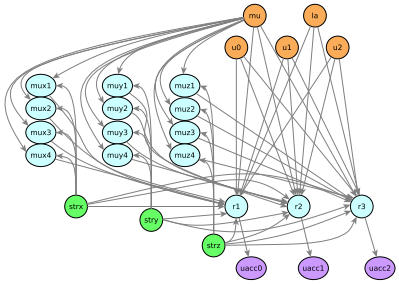
VI. MISCELLANEOUS OPTIMIZATIONS AND TUNING

The hierarchical autotuning described in Section V only tunes for the kernel/launch parameters, without changing the kernel semantics. However, in some cases, optimizations like time tiling (kernel fusion), and kernel fission, which change the number of kernels launched on the device, are key to improving the performance in the generated code. Kernel fusion in stencil DAGs is a well researched problem [2], [9], [12], [13], [25]. However, we show in Section VI-B that for register-constrained stencils, the opposite optimization, i.e., kernel fission is more promising. We have therefore implemented a novel, non-trivial kernel fission strategy in ARTEMIS for spatial stencils. This section describes the kernel fusion and fission strategies implemented in ARTEMIS.

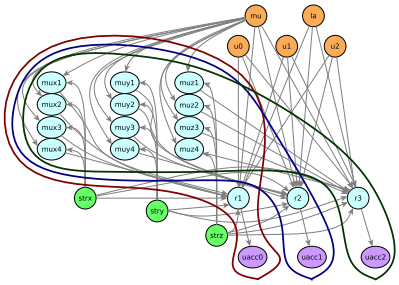
A. Fusion for Arbitrary Time Iterations

Iterative stencils are usually central to PDE solvers [28], imaging pipelines [40], etc. The time iteration count in many such applications is variable, since the accuracy of the solution depends on it. For example, the desired degree of smoothing determines the time iterations of the smoothers in HPGMG [28]. In such cases, an important tuning problem is to determine a profitable fusion degree for arbitrary time iterations. If a bandwidth-bound, order-1 stencil S has only 2 time iterations, then the choice is straightforward – fuse the two time steps. However, if in the next invocation, the time iterations of S increase, then perhaps a different degree of fusion may achieve better performance. ARTEMIS supports *deep tuning* to determine the profitable fusion for iterative stencils with varying time iterations.

Let us denote a version that has y invocations of a fused kernel, each with time tile size of x, as $(x \times y)$. Thus, if the stencil S has 13 time iterations, the possible output versions



(a) dependency graph for *rhs4center* kernel



(b) trivial fission into three sub-kernels

```
stencil sw4_0 (uacc0, ...) {
mux1 = ...; ... muz4 = ...;
r0 = ...; uacc0 = ...; }
stencil sw4_1 (uacc1, ...) {
mux1 = ...; ... muz4 = ...;
r1 = ...; uacc1 = ...; }
stencil sw4_2 (uacc2, ...) {
mux1 = ...; ... muz4 = ...;
r2 = ...; uacc2 = ...; }
sw4_0 (uacc0, ...);
sw4_1 (uacc1, ...);
sw4_2 (uacc2, ...);
```

(c) Generated DSL specification

Fig. 3: Trivial fission for the *rhs4center* kernel from SW4 [4]

are (1×13), $(2\times6\oplus1\times1)$, $(3\times4\oplus1\times1)$, $(4\times3\oplus1\times1)$, $(3\times3\oplus4\times1)$, etc., where $\oplus$ denotes kernel composition. If S is bandwidth-bound, then it is possible that version $(2\times6\oplus1\times1)$ performs better than (1×13), version $(3\times4\oplus1\times1)$ performs better than $(2\times6\oplus1\times1)$, and so on, up to a certain point where either the kernel is not bandwidth-bound, or any further benefit from fusion is outweighed by the increase in recomputation volume with overlapped tiling [9]. Due to this trade-off, some configurations will never be beneficial from the perspective of fusion. For example, with overlapped tiling, version (8×1) of a 3D 7-point stencil is not likely to be better than (4×2) on any of the current generation of GPU devices.

ARTEMIS leverages this insight to deep-tune the iterative stencils with variable time iterations. It automatically generates version $(x\times1)$ of the stencil, where x is both the time tile size and the time iteration count, starting with $x=1$. The version $(x\times1)$ is first autotuned, and its execution time and tuned parameter configuration is recorded. It is then profiled for bottlenecks. Since fusion helps bandwidth-bound stencils by reducing the data transfer volume, we need to tune version $((x+1)\times1)$ only if version $(x\times1)$ is determined to be bandwidth-bound at DRAM, texture cache, or shared memory by the profiler. Let k be the number of versions thus generated and tuned by ARTEMIS. Once the execution time and parameter configurations for the $(x\times1)$, $1\leq x\leq k$ versions are recorded, a good fusion schedule can be found for any time iteration count T using a dynamic program with the following optimal substructure:

$$opt(T)=\begin{cases}0 & T=0\\ min(\underset{x=1}{\overset{min(k,T)}{\forall}} f(x)+opt(T-x)) & \text{otherwise}\end{cases}$$

where $f(x)$ is the execution time for version $(x\times1)$. Usually, $k\leq4$ for most order-1 stencils, and much smaller for high-order stencils. Therefore, due to the synergistic coupling of autotuning and profiling, at most 4 fusion candidates must be tuned by ARTEMIS to find profitable fusion degree for any arbitrary time iteration. Furthermore, the deep tuning is done only once. For most applications, its cost will be amortized over the stencil invocations.

B. Kernel Fission for Register-Constrained Stencil DAGs

One can visualize fusion as constructing a forest of disjoint trees, where the leaf nodes correspond to individual stencil statements, and an internal node corresponds to a state where all its descendants are fused. A forest with a single tree would correspond to a maximal fusion, and a forest with trees having just one leaf node corresponds to a minimal fusion. Given just the leaf nodes, there are various ways to construct distinct forest of trees. One among such forests corresponds to an optimal fusion state. Clearly, finding it is non-trivial. However, we make the following two observations: (1) increase in recomputations post fusion can offset any fusion benefit. For example, in our experimentation, we never encountered an iterative order-1 or order-2 3D stencil that benefited from a time tile size upwards of 4 or 2, respectively; and (2) for stencils where fusion does not increase the recomputation volume, a higher degree of fusion is often achievable without incurring register spills, by using more registers per thread. Through experimentation, we observe that in many such cases, the occupancy loss is compensated by the increase in ILP, resulting in performance improvement.

For optimizing stencil DAGs on GPUs, we can find a near-optimal fusion state quickly if we start from a forest where the root of each tree has at most 4 descendants involved in a RAW dependence that results in an increase in recomputation volume, and then perform fission on the root of such trees such that there are no register spills and/or excessive recomputations. These observations motivate the incorporation of a fission component in ARTEMIS.

For automated fission, ARTEMIS generates split versions from the input stencil specification, and writes them out as DSL specification files. Should the users desire, they can then optimize the generated DSL files of fission candidates. ARTEMIS currently generates the following three DSL specification versions:

1) *maxfuse*: all the stencil functions with the same domain are fused to create the *maxfuse* version.
2) *trivial-fission*: each distinct array output in the computation is placed in a separate kernel, along with the temporary values required to compute it.
3) *recompute-fission*: The array outputs in a stencil function are split so that the recomputation halo for each kernel is $\leq max(4,r)$, where r is the maximum among stencil order of individual statements in the kernel.

Fission (fusion) may require replication (elimination) of statements computing temporary scalars across kernels, as illustrated in Figure 3. Figure 3a shows the data dependence

graph for the *rhs4center* kernel from SW4 [4]. In the figure, all the 3D input arrays are in orange, 1D arrays are in green, temporary scalars computed are in blue, and the 3D output arrays are in violet. Figure 3b shows the sub-kernels created with trivial fission. The three outputs *uacc0, uacc1, uacc2* are placed in three sub-kernels, along with *r1, r2, r3* respectively. The temporary input values *mux1*, ..., *muz4* contribute to *r1, r2, r3*, and hence get replicated in all three kernels. The generated DSL specification is shown in Figure 3c.

VII. End-to-End Use Summary

We summarize the steps taken by Artemis to optimize an input stencil S.

- Using the auxiliary information passed by the user with `#pragma` in the DSL (Section II), Artemis generates a baseline version for S.
- If the user wants to tune the baseline version of S, Artemis *automatically* profiles the baseline version to determine the (un)profitable optimizations (Section IV) and prune the autotuning space. Artemis then performs hierarchical autotuning (Section V) to find profitable parameter configurations. Finally, it profiles the best-performing version discovered via autotuning to determine the performance bottlenecks. Based on the bottlenecks, Artemis may generate some optimization hints for the user in the form of textual output, or some fission/fusion candidates (Section VI-B) that the user might want to optimize.
- The user must *manually* decide if some constraints/guidance must be added to the input DSL based on the generated hints, or if the generated fission/fusion candidates must be explored based on the profiling feedback.
- If S is a time-iterated stencil, the user can perform deep tuning with Artemis to find a near-optimal fusion schedule for it (Section VI-A). Artemis can *automatically* generate fusion candidates for S, record their execution times and parameter configurations, and then use the information to generate the near-optimal fusion schedule. Here, profiling is used to determine the number of fusion candidates that must be tuned.

VIII. Experimental Evaluation

A. Evaluation Setup

We explore a set of 11 3D double-precision stencils that are listed in Table I. The first three stencils are smoothers from the geometric multigrid benchmark, HPGMG [28]. *denoise* is a multi-statement stencil from the CDSC image processing pipeline [40]. *miniflux* is a computational fluid dynamics benchmark from [5]. The next two stencils are extracted from ExpCNS Compressible Navier-Stokes Department of Energy (DoE) mini-application [3]. SW4lite is a DoE Exascale Computing Project proxy application suite, that is designed to optimize the performance of key numerical kernels from the Geodynamics Seismic Wave (SW4) application [4]. The last four stencils are the compute-intensive kernels in SW4lite. The benchmarks are a mix of iterative stencil patterns representative of computations evaluated in other efforts, and complex, high-order spatial stencils from mini-applications that are more recent, relatively unexplored, and of significant interest to the Exascale research community.

Benchmark	Domain	T	k	# Flops	# IO Arrays
7pt-smoother	512^3	12	1	10	2
27pt-smoother	512^3	12	1	32	2
helmholtz	512^3	12	2	17	2
denoise	512^3	12	1	61	4
miniflux	320^3	1	2	135	25
hypterm	320^3	1	4	358	13
diffterm	320^3	1	4	415	11
addsgd4	320^3	1	2	373	10
addsgd6	320^3	1	3	626	10
rhs4center	320^3	1	2	666	8
rhs4sgcurv	320^3	1	2	2126	13

T: time tile size, k: stencil order

TABLE I: Characteristics of the 3D benchmarks

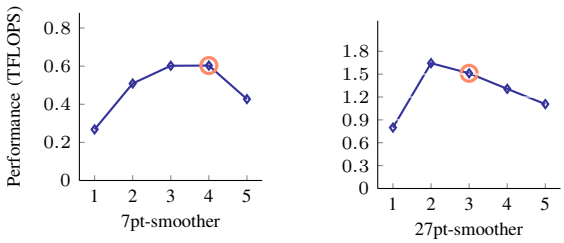


Fig. 4: Deep tuning for arbitrary time iterations

We evaluate the benchmarks on an NVIDIA Pascal P100 GPU device (double-precision α = 4.7 TFLOPS, $\frac{\alpha}{\beta_{dram}}$ = 6.42, $\frac{\alpha}{\beta_{tex}}$ = 2.35, $\frac{\alpha}{\beta_{shm}}$ = 0.49 [41]). The code is compiled with NVCC-9.1, using flags *-O3 -maxrregcount=r –use_fast_math -Xptxas "-dlcm=ca"*. The value of r in *maxrregcount* corresponds to the registers available to a thread, and must be strictly below 256. *dlcm* modifies the caching behavior; with *ca*, the global memory accesses are cached in both L1 and L2 cache. The *use_fast_math* flag allows the compiler to optimize some math library calls, and use fused multiply-add instructions for better performance [42].

B. Optimizing Iterative Stencils

The first four benchmarks from Table I are bandwidth-bound iterative stencils, and therefore would benefit from fusion. We use the deep tuning strategy described in Section VI-A to find an optimal fusion degree for these stencils. Figure 4 plots the results of deep tuning for *7pt-smoother* and *27pt-smoother*. For each time tile size $1 \leq t \leq 5$, version $(t \times 1)$ was extensively autotuned for at most 5 hours to find profitable parameter configurations. As the time tile size increases, the resource constraints limit the feasible configurations, and the autotuning time drastically reduces. For example, while autotuning the (1×1) version took 5 hours in some cases, autotuning the (5×1) version took less than 15 minutes in all cases. The autotuning time can be further reduced if the smaller domain sizes are used for autotuning.

Deep tuning results in Figure 4 reveal a cusp – the performance improves till a certain fusion degree, and then drops. To illustrate this trend, we compute the OI at each memory level

M	global	1×1	2×1	3×1	4×1	5×1
OI_{dram}	0.97	0.97	2.01	2.84	4.26	5.90
OI_{tex}	0.29	0.98	3.06	4.51	5.56	6.42
OI_{shm}	-	0.22	0.25	0.24	0.22	0.21

TABLE II: OI for different fusion degree of *7pt-smoother*

for different versions of *7pt-smoother* stencil. The results are presented in Table II. We observe that with an increase in the fusion degree, the computation becomes less bandwidth-bound at DRAM and texture cache, and the bound shifts onto shared memory. The 4×1 version is almost compute-bound at both DRAM and texture cache; at this point, ARTEMIS observes no performance benefit of any further fusion through profiling and code differencing, and stops exploring higher fusion degree.

Circled in pink in Figure 4 is the *tipping point*, which is the time tile size at which the stencil stops benefitting from fusion. The tipping point coincides with the first time tile size where better performance can be achieved by performing fission. For *27pt-smoother*, versions $(2 \times 1 + 1 \times 1)$ or (1×3) cannot achieve better performance than version (3×1). However, version (2×2) does achieve better performance than version (4×1). The tipping point was under 4 time steps for all the evaluated iterative stencils.

ARTEMIS uses the results of deep tuning to optimize these iterative stencils for arbitrary time iterations.

C. Optimizing Spatial Stencils

The last seven stencils in Table I are spatial, high-order stencils. Table III tabulates the theoretical OI, and nvprof-based OI_{dram} and OI_{tex} for their tuned global memory version. From Table III, it is clear that the global memory version for all the stencils is severely bandwidth-bound at texture cache. Since time tiling is not feasible for these spatial stencils, one can only improve performance by reducing the data transfers to global memory and texture cache by using shared memory and increasing reuse in registers. Unfortunately, all of these stencils also exhibit very low occupancy (between 12.5% – 25%) because of the high per-thread registers required for spill-free compilation. This makes the remedial loop unrolling, which improves ILP and/or register-level reuse, impossible without incurring expensive spills.

After initial profiling, ARTEMIS classifies *miniflux* kernel as bandwidth-bound at both texture cache and DRAM (Table III). Since using shared memory may be beneficial based on the profiling guidelines of Section IV, ARTEMIS then tunes the shared memory versions of the kernel. With shared memory, both OI_{dram} and OI_{tex} increase to 1.06 and 1.33, respectively, indicating a performance improvement. On the other hand, the total DRAM transactions remain unchanged for *hypterm* despite using shared memory, and code differencing indicates that the kernel is DRAM bandwidth-bound despite using shared memory. Based on the insights from Section IV, ARTEMIS tunes the global memory version for *hypterm*. Both *rhs4center* and *rhs4sgcurv* are not severely DRAM bandwidth-bound to begin with, but texture cache bandwidth-bound, and therefore ARTEMIS tunes their shared memory versions.

Bench.	OI_T	FLOP	$Byte_{dram}$	OI_{dram}	$Byte_{tex}$	OI_{tex}
miniflux	0.67	3.53e+9	6.5e+9	0.54	1.56e+10	0.22
		9.77e+8	6.92e+9	0.14	9.15e+9	0.10
hypterm	3.44	1.08e+10	5.27e+9	2.06	3.58e+10	0.30
diffterm	4.71	3.28e+9	3.73e+9	0.87	1.79e+10	0.18
		9.02e+9	6.75e+9	1.33	3.92e+10	0.23
addsgd4	4.66	9.37e+9	4.48e+9	2.08	2.63e+10	0.35
addsgd6	7.82	1.67e+10	5.32e+9	3.13	3.81e+10	0.43
rhs4center	10.4	1.93e+10	3.39e+9	5.69	4.19e+10	0.46
rhs4sgcurv	20.4	2.44e+10	4.65e+9	5.26	4.88e+10	0.50
		2.47e+10	5.81e+9	4.25	4.88e+10	0.50
		1.99e+10	4.82e+9	4.14	3.86e+10	0.51

OI_T: theoretical operational intensity

TABLE III: nvprof metrics and OI for the spatial stencils. Each entry corresponds to a distinct kernel.

D. Exploring Fission Candidates

rhs4sgcurv routine is implemented as a monolithic (*maxfuse*) kernel in SW4lite, which incurs register spills even when compiled with 255 registers per thread. The *trivial-fission* version generated by ARTEMIS splits the monolithic kernel into three spill-free sub-kernels. The *trivial-fission* outperforms the *maxfuse* version significantly (1.048 TFLOPS vs. 0.48 TFLOPS), demonstrating the importance of kernel fission.

E. Domain Expert Guided Resource Assignment

We specify resource assignments in the input specification for all the kernels from SW4lite benchmark. Essentially, all the 1D arrays, and some 3D arrays are assigned to global memory, so that ARTEMIS does not cache them in shared memory. To observe the effect of user intervention, we generate code for *addsgd4* with and without explicit resource assignment in the input specification. Without explicit resource assignment, *addsgd4* achieves 0.65 TFLOPS. With the user specifying the constraints, the performance improves to 1.05 TFLOPS.

F. Comparison With Other Code Generators

We compare the performance of ARTEMIS against PPCG-0.08 [16], and STENCILGEN [17]. PPCG is a polyhedral framework that generates CUDA/OpenCL code from a pragma-demarcated input C code. STENCILGEN is state-of-the-art CUDA code generator that outperforms several other stencil optimization frameworks [17]. It incorporates time tiling with associative reordering, and fusion for multi-statement stencils. In addition to the optimized version that appropriately uses shared memory and registers, we also present the performance results for two additional versions generated by ARTEMIS that only use global memory and no shared memory: (i) *global-stream*: the global memory version which streams along the slowest varying dimension; and (ii) *global*: the global memory version that tiles all the three dimensions. The performance results are plotted in Figure 5.

We tuned STENCILGEN for different fusion degree and block dimensions. For PPCG, we extensively autotuned for block dimensions, unrolling factors, and per-thread registers. Each benchmark was allowed to be tuned for 6 hours. For ARTEMIS, the deep tuning took 8 hours per iterative stencil. In contrast, each shared memory version of the spatial stencil took less than 20 minutes for tuning, since the resource constraints reduce the autotuning space. The global memory versions for the spatial stencils took less than 2 hours to tune.

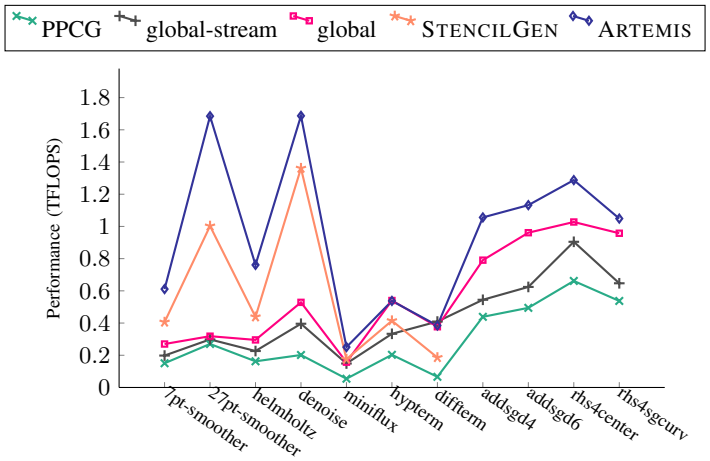


Fig. 5: Performance of benchmarks on Pascal P100

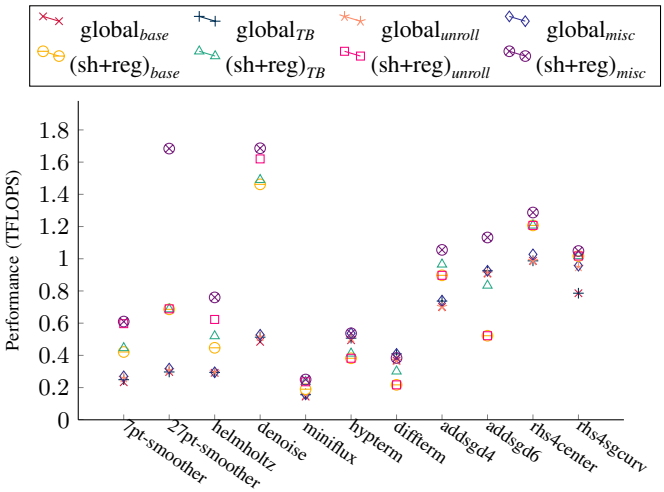


Fig. 6: Performance breakdown of various versions generated with ARTEMIS on Pascal P100

In all cases, we report the best performance. For example, all the results for *rhs4sgcurv* kernel with ARTEMIS are reported on the *trivial-split* kernel instead of the *maxfuse* routine. ARTEMIS profiling is nvprof-based, and therefore the profiling time depends on the execution time of the kernel, and the number of metrics collected. Since we collect less than 10 metrics at present, and the execution time of each kernel is in milliseconds, the current profiling overhead in our runs is less than a minute for each benchmark.

PPCG is significantly outperformed by the tuned global memory versions, especially for the spatial high-order stencils. This is due to the poor fusion/fission choices, and the complex conditionals in the PPCG-generated code. Streaming is often assumed to result in better cache utilization [15]. Surprisingly, the *global-stream* version incurs much higher DRAM transactions and memory stalls than *global* version for all the benchmarks. Though streaming is crucial to reduce the shared memory usage per kernel, it results in poor L2 locality when shared memory is not used. ARTEMIS outperforms STENCILGEN for all the iterative stencils, highlighting the importance of optimizations like loop unrolling, load/compute adjustment, concurrent streaming, and prefetching, which are not implemented in STENCILGEN.

For spatial stencils, we edited the code generation pass in STENCILGEN to allow a higher degree of fusion. However, as discussed earlier, shared memory version cannot improve performance for the DRAM bandwidth-bound *hyperm*. With ARTEMIS, we were able to generate a shared memory version for *hyperm* that uses thread block load/compute adjustment to match the performance of the global memory version. We could not generate code for the kernels from SW4lite with STENCILGEN, since it does not support domains with different dimensions within the same stencil function.

ARTEMIS can generate plug-and-replace functions for the manually optimized kernels in SW4lite, and achieve similar or higher performance. For example, the ARTEMIS-optimized *rhs4center* kernel achieves 1.29 TFLOPS, compared to 1.13 TFLOPS achieved by the manually optimized version in SW4lite. We plan to integrate ARTEMIS-optimized kernels into SW4lite in the future.

G. Interaction Between Optimizations and Autotuning

Figure 6 plots the interaction between autotuning and optimizations with ARTEMIS. In the figure, *global* corresponds to generated versions with just global memory, and *sh+reg* corresponds to versions that use both shared memory and registers. The baseline versions global$_{base}$ and (sh+reg)$_{base}$ have no optimizations enabled, and the thread block size is fixed to (x=32,y=16) for iterative 3D stencils with streaming, (x=16,y=16) for register-constrained spatial stencils with streaming, and (x=16,y=4,z=4) for the non-streaming versions. On top of this baseline, we apply one optimization at a time, and plot the performance with the standalone optimization. For example, with autotuning, we explore versions global$_{TB}$ and (sh+reg)$_{TB}$ by just varying the thread block size. Many iterative and spatial stencils benefit from thread block size variation, but the impact is more pronounced for the shared memory versions of high-order stencils like *helmholtz*, *diffterm* and *addsgd6*. For versions global$_{unroll}$ and (sh+reg)$_{unroll}$, we fix the thread block size to the baseline, and autotune the unrolling factors along different dimensions. Unrolling helps the shared memory versions of the iterative stencils where register pressure is not a performance limiter. This justifies the decisions made in ARTEMIS by the profiling component to suppress loop unrolling in hierarchical autotuning for spatial stencils. Also, the unrolling factors leading to higher performance vary significantly across benchmarks, highlighting the need for autotuning. Versions global$_{misc}$ and (sh+reg)$_{misc}$ plot the performance improvement with all the optimizations like unrolling, thread block size variation, prefetching, retiming, computation folding, and load/compute adjustment simultaneously enabled. These sundry optimizations lead to significant performance improvements across all stencils. For example, retiming is the key to achieving high performance in *27pt-smoother*, load/compute adjustment leads to significant performance improvement for the shared memory version of *hyperm*, and computation folding is beneficial for *addsgd6*. On a higher level, these trends imply that (a) a single optimization cannot uniformly benefit all the stencils,

various optimizations must interact synergistically to improve the performance of stencils with varying order and complexity; and (b) autotuning and profiling are crucial to steer the code generation towards versions that achieve higher performance.

IX. Related Work

Many prior research efforts have explored the acceleration of stencil computations on GPUs [1], [6]–[13], [16], [18], [19], [21], [39], [43]. Verdoolaege et al. develop PPCG [16], a source-to-source compiler that generates time-tiled OpenCL/CUDA code from an annotated sequential program. Holewinski et al. develop Overtile [1], Ravishankar et al. develop Forma [11], and Rawat et al. develop StencilGen [9], [17], all of which use overlapped tiling. However, of the three, only StencilGen automates the spatial/temporal streaming approach that was used by Micikevicius [14] and Nguyen et al. [15] to optimize 3D stencils. Grosser et al. [7] present a code generator that adapts split tiling from [26] to GPUs. Grosser et al. [6] also implement hexagonal tiling [6] in PPCG. These tiling schemes differ in the communication and recomputation trade-off. ChiLL [44] is a composable loop transformation framework which allows the user to script loop transformations for stencil computations. Basu et al. [45] use ChiLL to generate, and autotune the miniGMG benchmark on GPUs. Hagedorn et al. [21] extend the Lift language with primitives to support stencil computations. Unlike the aforementioned code generators that can perform time tiling, several code generators perform only spatial tiling on the input stencil. Mint [19] is a pragma-based source-to-source translator implemented in the ROSE compiler [46] that generates a spatially tiled CUDA code from traditional C code. Physis [18] translates user-written stencil code into CUDA+MPI code for GPU-equipped clusters. Artemis optimizes both spatial and time-iterated stencils, gives the user some control over the resource allocation, and incorporates a plethora of optimizations. Another novel aspect of Artemis is its ability to perform kernel fission in order to alleviate register pressure for resource-constrained complex stencils. Our experimental results demonstrate that Artemis consistently outperforms StencilGen, which outperforms PPCG, Overtile, Forma, and Halide autoscheduler [9], [17].

Since the performance of the generated code is affected by the interaction of various code generation parameters and hardware, autotuning of stencil applications has gained popularity in recent years. Zhang and Mueller [10] evaluate a code generator and autotuner for 3D stencils on GPU clusters. They tune for unrolling factor along different dimensions, but the code generation is restricted to spatial tiling. Halide decouples algorithm specification from schedule; the schedule can be either written by a domain expert [2], auto-generated [43], or autotuned independently by OpenTuner [24]. Prajapati et al. [32] develop an analytical model that predicts the execution time of the code generated with hexagonal tiling. Artemis incorporates a profiling and autotuning strategy that is simple to automate, and uses sample-based profiling via nvprof instead of relying on analytical models which can be imprecise for register-constrained stencils [30], [32]. The hierarchical autotuning also allows for a faster tuning than OpenTuner.

Applications in image processing are often expressed as stencil DAGs. Individual stencils in the DAG are often bandwidth-bound, therefore fusion across the nodes in the DAG improves performance. Recent research has focused on integration of analytical models into code generators to guide kernel fusion [30], [32]–[35]. A common limitation of such analytical models is the precision in prediction, stemming from the complexity of the underlying hardware. Wahib and Maruyama [13], and Gysi et al. [12] use an analytical performance model to find a near-optimal fusion configuration. Wahib and Maruyama [13] further prune the search space using a search heuristic based on a hybrid grouping genetic algorithm. Unlike these approaches, Artemis uses a fission-driven approach that is driven by the resource constraints of the underlying GPU device.

X. Conclusion

High-order, multi-statement stencils are becoming commonplace in scientific computations. Such stencils present optimization challenges that were not observed with simpler stencils. This paper identifies abstractions that are imperative to performance for high-order, complex stencil computations: user-guided optimizations, and integration of code generation and tuning with bottleneck profiling. The developed abstractions are implemented into a prototype code generation framework, Artemis. Experimental evaluation of Artemis on several stencil kernels demonstrates consistent performance improvement over other state-of-the-art code generators.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and suggestions that helped improve the paper. This work was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration; in part by the U.S. National Science Foundation (NSF) under Awards 1440749 and 1513120; and in part by an award for use of computing resources at the Ohio Supercomputer Center.

References

[1] 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. ACM.

[2] J. Ragan-Kelley, 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," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. ACM.

[3] "ExaCT: Center for Exascale Simulation of Combustion in Turbulence: Proxy App Software," https://exactcodesign.org/proxy-app-software/.

[4] "Seismic Wave Modelling (SW4) - Computational Infrastructure for Geodynamics," https://geodynamics.org/cig/software/sw4/.

[5] E. C. Davis, M. M. Strout, and C. Olschanowsky, "Transforming loop chains via macro dataflow graphs," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO '18. ACM.

[6] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for GPUs," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. ACM.

[7] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for GPUs: Automatic parallelization using trapezoidal tiles," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU '13. ACM.

[8] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, and P. Sadayappan, "Effective resource management for enhancing performance of 2D and 3D stencils on GPUs," in *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, ser. GPGPU '16. ACM.

[9] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Resource conscious reuse-driven tiling for GPUs," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. ACM.

[10] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. ACM.

[11] M. Ravishankar, J. Holewinski, and V. Grover, "Forma: A DSL for image processing applications to target GPUs and multi-core CPUs," in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU '12. ACM.

[12] T. Gysi, T. Grosser, and T. Hoefler, "MODESTO: data-centric analytic optimization of complex stencil programs on heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. ACM.

[13] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound GPU applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE Press.

[14] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU '09. ACM.

[15] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. IEEE Computer Society.

[16] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM TACO*, 2013.

[17] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L. Pouchet, and P. Sadayappan, "Domain-specific optimization and generation of high-performance GPU code for stencil computations," *Proceedings of the IEEE*, 2018.

[18] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM.

[19] D. Unat, X. Cai, and S. B. Baden, "Mint: Realizing CUDA performance in 3D stencil methods with annotated C," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. ACM.

[20] 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. IEEE Computer Society.

[21] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO '18. ACM.

[22] P. S. Rawat, A. Sukumaran-Rajam, A. Rountev, F. Rastello, L.-N. Pouchet, and P. Sadayappan, "Register optimizations for stencils on GPUs," in *Proceedings of the 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. ACM.

[23] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[24] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. ACM.

[25] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. ACM.

[26] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector SIMD architectures," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. ACM.

[27] 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*, ser. SPAA '11. ACM.

[28] "High-Performance Geometric Multigrid," https://hpgmg.org/.

[29] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, "Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. ACM, 2015.

[30] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, New York, NY, USA.

[31] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of SGEMM on fermi and kepler GPUs," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '13, Washington, DC, USA.

[32] N. Prajapati, W. Ranasinghe, S. Rajopadhye, R. Andonov, H. Djidjev, and T. Grosser, "Simple, accurate, analytical time modeling and optimal tile size selection for GPGPU stencils," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. ACM.

[33] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. ACM.

[34] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11. IEEE Computer Society.

[35] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. ACM.

[36] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*.

[37] "NVIDIA Profiler," http://docs.nvidia.com/cuda/profiler-users-guide.

[38] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *2010 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '10.

[39] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. IEEE Press.

[40] "Center for Domain-Specific Computing," https://code.google.com/archive/p/cdsc-image-processing-pipeline/.

[41] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *CoRR*, vol. abs/1804.06826.

[42] "NVCC," docs.nvidia.com/cuda/cuda-compiler-driver-nvcc.

[43] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Trans. Graph.*

[44] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC '09.

[45] P. Basu, S. Williams, B. V. Straalen, L. Oliker, P. Colella, and M. Hall, "Compiler-based code generation and autotuning for geometric multigrid on GPU-accelerated supercomputers," *Parallel Computing*.

[46] D. J. Quinlan, "ROSE: Compiler support for object-oriented frameworks." *Parallel Processing Letters*.