This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

INVITED PAPER

Domain-Specific Optimization and Generation of High-Performance GPU Code for Stencil Computations

By Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Mahesh Ravishankar, Vinod Grover, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan, *Fellow IEEE*

ABSTRACT | Stencil computations arise in a number of computational domains. They exhibit significant data parallelism and are thus well suited for execution on graphical processing units (GPUs), but can be memory-bandwidth limited unless temporal locality is utilized via tiling. This paper describes how effective tiled code can be generated for GPUs from a domain-specific language (DSL) for stencils. Experimental results demonstrate the benefits of such a domain-specific optimization approach over state-of-the-art general-purpose compiler optimizations.

KEYWORDS | General-purpose graphical processing unit (GPGPU); resource optimization; stencil computations; streaming; tiling

I. INTRODUCTION

Stencil computations arise in scientific applications in many domains. Due to the large number of variants of the stencil operators that arise in applications, it is not feasible to construct efficient libraries for stencil computations. Hence, stencils have been the focus of a number of domain-specific languages and frameworks [1]–[15]. Graphical processing units (GPUs) are an attractive architectural target for stencil computations because of the high degree of data parallelism. However, achievable performance for many stencil computations is constrained by bandwidth to global memory. If the number of floating-point operations per point is not sufficiently high, reuse across multiple time steps for an iterated stencil or across a sequence of stencils is essential to achieve high performance.

Tiling is a fundamental technique for data locality enhancement, and can enable significant reduction in the amount of data movement from/to global memory. However, the nature of data dependences that arise with stencils does not permit simple rectangular tiling across multiple time steps of an iterative stencil computation. A number of research efforts have therefore addressed the topic of effective tiling of stencil computations [2]–[8], [12]–[14], [16]–[29].

In this paper, we present a detailed analysis of constraints in achieving high performance with stencils on GPUs and describe domain-specific and GPU-target-specific optimization strategies to generate high-performance GPU code for stencils. While the GPU algorithms discussed in this paper use CUDA terminology and the implementations use NVIDIA GPUs as reference hardware, the algorithmic strategies are not limited to CUDA or NVIDIA GPUs.

Manuscript received January 31, 2018; revised July 11, 2018; accepted July 11, 2018. 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. *(Corresponding author: P. Sadayappan.)*

P. S. Rawat, **M. Vaidya**, **A. Sukumaran-Rajam**, **A. Rountev**, and **P. Sadayappan** are with The Ohio State University, Columbus, OH 43210 USA (e-mail: rawat.15@osu.edu; vaidya.56@osu.edu; sukumaranrajam.1@osu.edu; rountev.1@osu.edu; sadayappan.1@osu.edu).

M. Ravishankar and **V. Grover** are with NVIDIA Corporation, Redmond, WA 98052 USA (e-mail: mravishankar@nvidia.com; vgrover@nvidia.com).

L.-N. Pouchet is with Colorado State University, Fort Collins, CO 80523 USA (e-mail: pouchet@colostate.edu).

Digital Object Identifier 10.1109/JPROC.2018.2862896

0018-9219 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

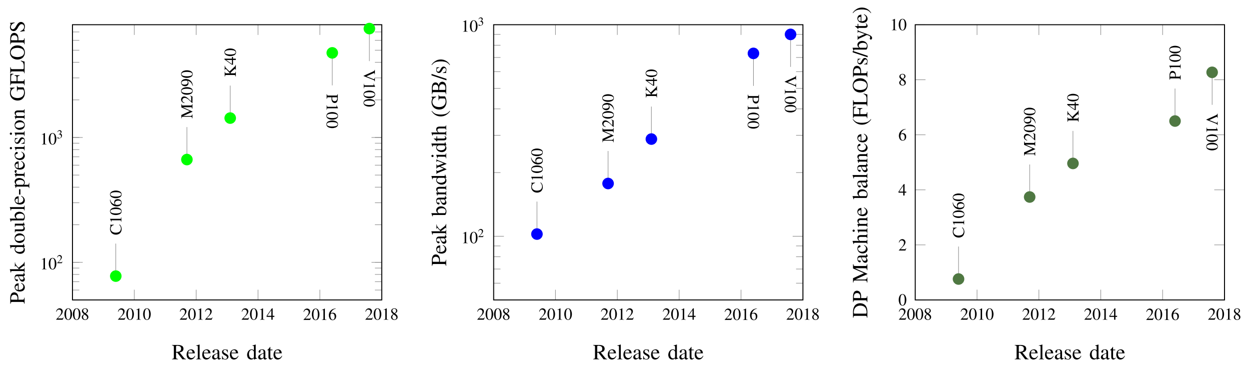


Fig. 1. ***Theoretical peak double-precision compute performance, machine balance, and theoretical peak bandwidth for NVIDIA devices over the last decade.***

These optimizations can be implemented using any language that allows mapping computations to GPU threads, e.g., OpenCL, and are applicable to GPUs from other vendors, e.g., AMD.

Several considerations are important in achieving high performance for any computation on GPUs: movement of contiguous chunks of data (coalesced accesses) from/to global memory; reuse of data in registers and on-chip scratch-pad memory (shared memory in CUDA terminology; called local memory in OpenCL); sufficient concurrency, orders of magnitude larger than the number of physical cores; and minimization of control-flow divergence among threads. The capacity of on-chip scratch-pad memory in a streaming multiprocessor (SM) is quite low, typically under 100 kB. As elaborated on later, the low shared memory capacity limits the maximum tile size. This further limits both the amount of reuse achievable for data in shared memory, and the maximum number of simultaneously active thread blocks in an SM. Low occupancy can have a deleterious effect on performance, because the number of instructions across the active warps that can be issued without stalling on dependent data might be insufficient to effectively overlap and mask the high global memory access latency. The total capacity in the register file per SM on GPUs is almost an order of magnitude larger than shared memory capacity. Further, access latency from registers is lower than shared memory. In this paper, we present details on a code generation strategy for efficient execution of stencils on GPUs, which effectively utilizes available register and shared memory resources to achieve high data reuse and thereby realize high performance.

The paper is organized as follows. Section II provides an overview of the main issues in achieving high performance with stencil computations. Section III describes a domain-specific language (DSL) for specifying stencil computations: the StencilGen language. Section IV details the impact of the GPU hardware constraints on the stencil optimization strategy. GPU code generation algorithms are described in Section V. Section VI discusses fusion across multiple stencils. Section VII discusses related work. Section VIII presents an experimental evaluation of the described stencil code generation approach, comparing it with several current general-purpose and special-purpose code generators and optimizers for multicore processors, manycore processors, and GPUs. Section IX presents our conclusions.

II. TILING OF STENCIL COMPUTATIONS

In this section, we review the nature of the data dependences and potential for data reuse in executing stencil computations. Many stencil computations are fundamentally limited by memory bandwidth unless temporal locality is exploited across a sequence of stencils in a processing pipeline, or across repeated application of a stencil in an iterated computation.

A. Machine Balance and Operational Intensity

Fig. 1 provides data on the peak double-precision floating-point performance [Fig. 1(a)] and peak global-memory bandwidth [Fig. 1(b)] for GPUs from NVIDIA. Five generations of GPUs are shown: pre-Fermi (C1060), Fermi (M2090), Kepler (K40), Pascal (P100), and Volta (V100). The peak machine double-precision floating-point performance ($GFLOPS_{mc}$) has increased by over two orders of magnitude, from under 100 GFLOPS for the C1060 to around 10 TFLOPS for the Volta V100. The peak global memory bandwidth (BW_{mc}) has also increased across successive GPU generations, but not to the same extent: from 100 GB/s to nearly 1000 GB/s. Therefore, as seen in Fig. 1(c), there has been a rise in the machine balance parameter, $MB = (GFLOPS_{mc})/(BW_{mc})$, from about 0.5 FLOPs per byte for the C1060 to around 8 FLOPs per byte for the Volta V100. As elaborated on below, in order to achieve close to peak performance, the operational intensity (OI) of a computation (ratio of the number of arithmetic/logic operations to the number of bytes of data movement from/to memory) must be higher than the machine balance parameter.

```
for(i=1; i<N-1; i++)
    b[i] = c1*(a[i-1]+a[i]+a[i+1]);
```

(a)

```
for(i=1; i<N-1; i++)
    for(j=1; j<N-1; j++)
        b[i][j] = c2*(a[i-1][j]+a[i+1][j]+
                      a[i][j-1]+a[i][j+1]+
                      a[i][j]);
```

(b)

```
for(i=1; i<N-1; i++)
    for(j=1; j<N-1; j++)
        for(k=1; k<N-1; k++)
            b[i][j][k]=c3*(a[i-1][j][k]+a[i+1][j][k]+
                           a[i][j-1][k]+a[i][j+1][k]+
                           a[i][j][k-1]+a[i][j][k+1]+
                           a[i][j][k]);
```

(c)

Fig. 2. ***Examples of 1-D, 2-D, and 3-D stencils. (a) 1-D stencil. (b) 2-D stencil. (c) 3-D stencil.***

Consider the examples of 1-D, 2-D, and 3-D stencils in Fig. 2. For the 1-D stencil, $3(N-2)$ floating-point operations are executed and a total number of $2N-2$ distinct array elements are accessed (N elements of a and $N-2$ elements of b). If full reuse in registers/cache is achieved for each accessed array element, the achieved operational intensity would be $(3(N-2))/(8(2N-2))$, i.e., approximately 3/16 FLOPs per byte for double-precision computation (8 B per data element). For the 1-D stencil computation, the maximal possible operational intensity of 3/16 FLOPs per byte is considerably lower than the machine balance parameter of modern GPUs. If an implementation of some algorithm has an upper bound of OI_{alg} for achievable operational intensity, and the peak memory bandwidth of the machine is BW_{mc}, the maximum achievable computational performance is $\text{OI}_{alg} \times \text{BW}_{mc}$. Equivalently, the maximum fraction of the machine's peak performance GFLOPS_{mc} that an implementation of an algorithm can achieve is $\text{OI}_{alg}/\text{MB}$.

As the dimensionality of the grid increases, the maximum achievable operational intensity also increases. For the 2-D stencil in Fig. 2(b), the number of floating-point operations is $5(N-2)^2$, and the number of accessed array elements is $N^2+(N-2)^2$, with an OI upper bound of around 5/16 FLOPs per byte for double-precision computation. For the 3-D stencil shown in Fig. 2(c), the OI upper bound rises to 7/16 FLOPs per byte, but is still an order of magnitude lower than the machine balance parameter for current GPUs.

B. Time Tiling of Stencils

In order to increase performance with such stencils, it is essential to exploit reuse across time steps in an iterated stencil computation. With time-iterated stencils, the output array produced by the application of the stencil operation on the input array is in turn used as the input array for the next application of the stencil. Fig. 3 illustrates a 1-D three-point stencil iterated over three time steps. Since the arrays subject to the stencil computation are typically much larger than cache, reuse of data across the iterated application cannot be achieved without tiling, i.e., computing for a number of time steps for a subset of the array elements within a tile before accessing other array elements for the first time step. However, as seen in Fig. 3(a), a simple rectangular 2-D tiling strategy is not feasible because dependences would be violated if all iteration space points within one tile were contiguously executed.

Fig. 3(b) shows a valid tile shape for the 1-D stencil example. All data dependences to a tile are from iteration-space points in a tile to the left. As long as the

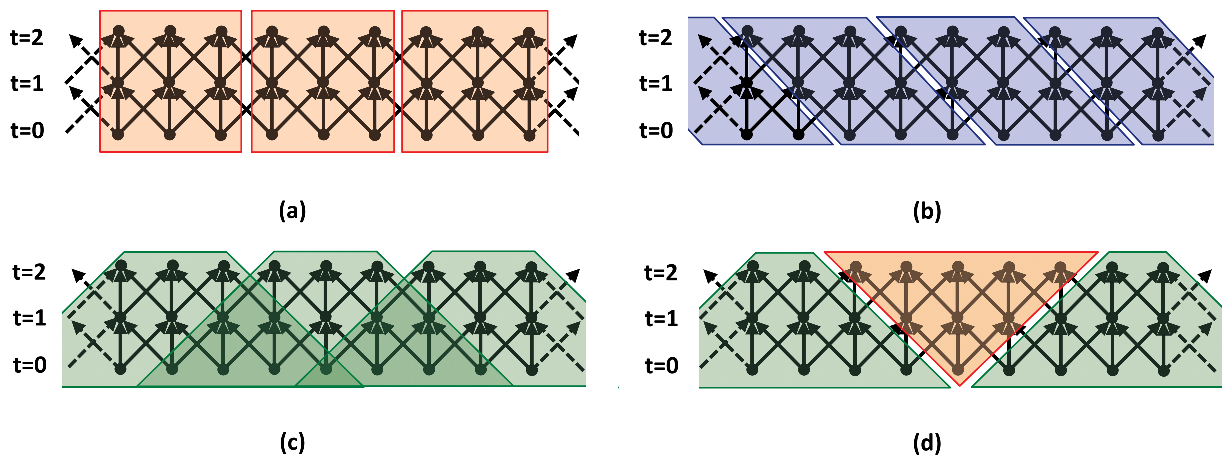


Fig. 3. ***Time tiling to enhance data reuse with stencil computations. (a) Rectangular tiling is invalid since dependencies are violated. (b) Parallelogram tiles preserve dependences but intertile dependences inhibit parallelism across tiles. (c) Overlapped tiling enables concurrent execution of tiles at the expense of redundant computation. (d) Split tiling allows concurrent execution without redundant computation but tiles are less regular.***

tiles are executed from left to right, all dependences will be preserved. However, the serializing intertile dependences are undesirable. Fig. 3(c) shows overlapped trapezoidal tiles. The iteration-space points within a tile depend only on other points included within the same tile. Thus, concurrent execution of tiles is feasible. However, some redundant computation is necessary, as can be seen in Fig. 3(c) with the iteration-space points that belong in multiple tiles. Fig. 3(d) shows another strategy called split tiling that also enables concurrent tile execution. With this strategy, a tile belongs in one of multiple groups (the number of groups grows with the dimensionality of the arrays and stencil operator). Tiles within the same group can execute concurrently, and the different groups are scheduled in order. In Fig. 3(d), all green-colored tiles execute first, followed by the orange-colored tiles. Split-tiling has no redundant computational overhead. However, the tiles exhibit greater irregularity than with overlapped tiling, making it more challenging to achieve high performance on GPUs. In this paper, we present details on a stencil optimizer for GPUs that uses the overlapped tiling strategy.

C. Combining Overlapped Tiling With Streaming

The intertile concurrency enabled by overlapped-tiling comes at the price of redundant computation (and redundant data movement). While the fraction of redundant computation can be made very small for 1-D overlapped tiling, the overhead can be very significant for 3-D stencils. Consider a thread block of $8 \times 8 \times 8$ threads that computes two time steps of a 3-D order-1 stencil. The thread block generates output after two time steps for $8 \times 8 \times 8$ elements, which means that the output after the first time step must be generated for a domain of size $10 \times 10 \times 10$, which in turn requires that input over a $12 \times 12 \times 12$ domain must be read. Thus, 12^3 data elements must be read in to produce results for 8^3 elements, almost half of them being also read by neighboring thread blocks. The amount of computation for the intermediate time step is proportional to $10^3 = 1000$, out of which $1000 - 512 = 488$ are redundant. This problem of redundant computation overhead for 3-D stencils gets worse with an increase in stencil order and time tile size. Increasing the tile size can lower the overhead, but the maximum limits of thread-block size and shared memory on an SM prohibit tiles much larger than $8 \times 8 \times 8$.

Since overlapped tiling for all the dimensions of a 3-D stencil incurs high overhead on GPUs, an alternative is to use overlapped tiling along two of the three spatial dimensions, and sequentially stream along the third. We first describe overlapped tiling with streaming for a 2-D stencil. Fig. 4(a) shows a 2-D stencil where overlapped tiling is applied along both x- and y-dimensions with 2-D blocks; the figure only shows overlap along the x-dimension. Next, Fig. 4(b) shows streamed execution along the y-dimension with 1-D block. We can imagine the 2-D domain sliced into 1-D lines along the streaming dimension. We observe that due to the dependence pattern, only three consecutive lines need to be read from the input domain to compute one line of the output domain. In general, an order-k 2-D stencil will need to read $2k + 1$ consecutive input lines to compute one output line. The lines are cached in three distinct shared memory buffers per time steps, represented by different colors in Fig. 4(b). The number over the buffers indicates the logical timestamp (LT) at which the particular buffer is populated.

In the prolog, three input lines are loaded into the buffers at $t = 0$, so that an intermediate result can be computed at $t = 1$ (LT $= 4$). After LT $= 4$, the data held in the buffer populated at LT $= 1$ are no longer needed, and this buffer can be subsequently used to cache a new input line. After this initial prolog, we can compute one intermediate result at $t = 1$ for every new input line read at $t = 0$. When we have three intermediate result lines available at $t = 1$, we can compute one line of the final output at $t = 2$ at LT $= 9$. In the steady state, we compute one output line at $t = 2$ by reading one input line at $t = 0$, and computing one intermediate line at $t = 1$.

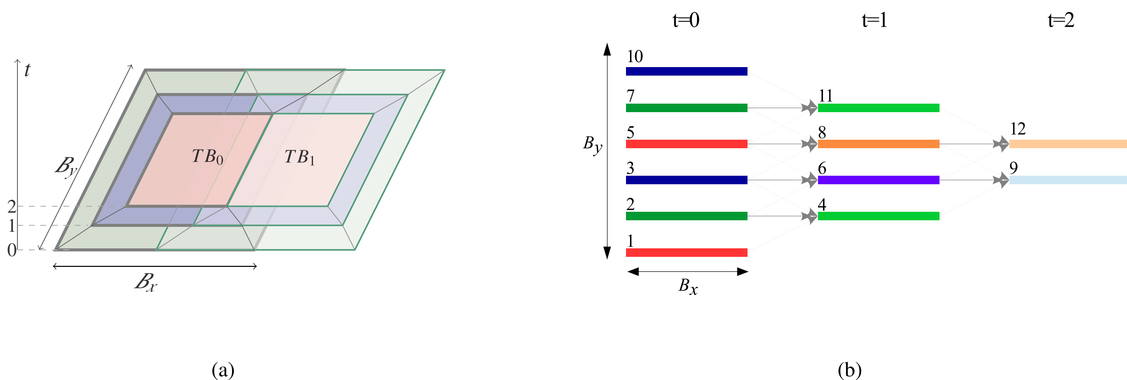


Fig. 4. ***Overlapped tiling and streaming for 2-D stencil. (a) Overlapping tiling with 2-D blocks. (b) Streaming along dimension y with 1-D block.***

This scheme requires only three buffers per time step, irrespective of the number of lines. In general, for an order-k stencil, $2k + 1$ buffers will be required per time step. The buffer that holds the oldest value at $\mathrm{LT} = r$ can be reused to store the newest value at $\mathrm{LT} > r$. If the different lines along y are computed sequentially, one after the other, we refer to it as serial streaming. However, the iteration space could also be partitioned along the y-axis, with the different partitions being assigned across thread blocks. Each thread block serially streams through its portion of the iteration space, but different blocks can concurrently stream through their partition of iteration space. We refer to this as concurrent streaming.

Streaming can be extended to 3-D Jacobi stencils by interpreting the line in Fig. 4(b) as a plane. Since streaming eliminates redundant computations along the streaming dimension and reduces the amount of shared memory, it is attractive to combine it with overlapped tiling. For a 3-D stencil, overlapped tiling along two of the dimensions and streaming along the remaining dimension enables sufficient intertile concurrency, and significantly reduces the amount of redundant operations (computations and global-memory loads) since the required shared memory capacity per thread block corresponds to a 2-D slice and not a 3-D slice.

If the time tile size is 1, only spatial reuse is exploited. Such spatial blocking, also known as 2.5-D blocking, was used by Micikevicius [17] to optimize 3-D finite difference computation. Nguyen *et al.* [16] used the corresponding time tiled blocking, also known as 3.5-D blocking, to accelerate LBM and seven-point 3-D Jacobi stencil on multicore CPU and GPU. Section V describes a systematic approach to determine the best tiling strategy for a given problem.

III. OVERVIEW OF A STENCIL DSL

To allow specification of stencil computations, we define StencilGen, a DSL used as input to the code generation techniques described in subsequent sections. The use of a DSL enables the easy identification of the stencil patterns so that stencil-specific optimizations may be performed. A code generator can easily determine the high-level semantic properties of a DSL code region. In principle, such properties could also be inferred from an equivalent code region written in a general-purpose language. However, in such a scenario, it is necessary that a variety of precise compiler analyses establish important properties such as lack of aliasing, values of induction variables, etc. The use of a DSL helps avoid these problems. The StencilGen language describes:

- the point operation for each data element, as well as boundary conditions and time iterations;
- the region of data on which a stencil is applied.

Note that general-purpose programming is allowed outside the stencil regions, i.e., StencilGen can be used both as standalone and embedded DSL.

```
1  parameter M, N;
2  iterator i, j;
3  double in[M,N], out[M,N];
4  copy-in in;
5  stencil five_point_avg (A, B) {
6    double ONE_FIFTH = 0.2;
7    B[i][j] = ONE_FIFTH*(A[i-1][j]
8      + A[i][j-1]+A[i][j]+A[i][j+1]+A[i+1][j]);
9  }
10 stencil boundary (A, B) {
11   B[i][j] = A[i][j];
12 }
13 iterate (1:10) {
14   [0 : M-1][0 : N-1] : five_point_avg (in, out);
15   [1 : M-2][1 : N-2] : five_point_avg (out, in);
16 }
17 [0   : 1  ][0   : N-1] : boundary (in, out);
18 [M-2 : M-1][0   : N-1] : boundary (in, out);
19 [0   : M-1][0   : 1  ] : boundary (in, out);
20 [0   : M-1][N-2 : N-1] : boundary (in, out);
21 copy-out out;
```

Listing 1. ***A representative stencil in* StencilGen *language.***

The code example in Listing 1 highlights some of the main features of StencilGen. The read-only integer parameters `M` and `N` are used to describe the dimensions of the input and output arrays (line 3). Line 2 declares iterators, each of which will be mapped to a unique dimension of the computational loop nest. The iterators must be immutable within the body of the loop nest, and are assumed to be incremented in unit steps by the increment condition of the loop. All declared arrays and scalars (e.g., `in` and `out` in line 3) will be passed as arguments to the host function that will be generated from the StencilGen input. Line 4 specifies the arrays and scalars that need to be copied from host to device.

Lines 5–9 define a stencil `five_point_avg`. The stencil definition has as arguments the input and output arrays/scalars used in the computation. This stencil averages the five neighboring data elements from array `A` and writes the result to an element of array `B`. All memory accesses in the stencil function must be scalars or array elements. Lines 10–12 define a stencil `boundary` that performs pointwise copy of elements from array `A` to array `B` at the boundary. Lines 13–20 invoke the stencil functions over subsets of the problem domain. The `iterate` construct at line 13 defines a time loop over the stencil calls. We explicitly unroll the call to stencil `five_point_avg` by a minimum number of iterations to obviate the need for exchanging the input and output after each time step. For such time-iterated stencils, the StencilGen-based code generator automatically determines the optimal degree of unrolling for the time loop, fuses the execution of two or more time steps, an optimization termed *time tiling*, to exploit the temporal reuse exposed by the producer–consumer relationship between the time steps. Finally, line 21 defines the arrays and scalars that need to be copied back from device to host.

A. Embedding StencilGen in C/C++ Code

Every embedded StencilGen region is delineated by #pragma annotations. The starting #pragma specifies additional code generation parameters, including the size of the GPU thread block, the dimension along which spatial

streaming is to be performed, and the size of shared memory available on the target GPU.

When embedding STENCILGEN regions, the data arrays are assumed to be allocated by the outside code, and to be stored contiguously in the C/C++ memory space. The right-hand-side expressions of the assignment statements in the stencil body are assumed to be side-effect free. Those expressions are allowed to contain standard side-effect-free math functions such as `sin`, `sqrt`, etc.

Additional details about STENCILGEN, including its grammar, are available in [30].

IV. GPU CONSTRAINTS FOR STENCIL COMPUTATION

This section discusses architectural constraints to be considered when optimizing stencil computations on GPUs. The basic computational unit of a GPU is a thread. Going up the hierarchy, threads are grouped into a warp, warps are grouped into thread blocks, and thread blocks are grouped into a grid. The number of threads per warp is architecture specific; 32 and 64 threads form a warp in NVIDIA and AMD devices, respectively. The GPU host code explicitly specifies the grid and thread block dimensions when the kernel is launched. All threads in a warp execute instructions in a synchronized, lock-step fashion. Warps within a thread block are mapped to the same streaming multiprocessor (SM), and can synchronize among themselves using synchronization primitives. Each thread has a fixed number of registers that can be varied at compile time. Threads in a thread block can exchange data via shared memory. In recent GPU architectures, threads within the same warp can also exchange data held in registers via shuffle intrinsics.

Efficient tiling schemes for GPUs must

- perform coalesced access to global memory;
- have sufficient parallelism to tolerate memory access latencies;
- judiciously use faster storage, such as shared memory and registers, for caching data.

Shared memory has lower access latency than global memory, and is well suited to cache data that are accessed by multiple threads in a thread block. Registers are local to a thread, and therefore well suited to cache data that are accessed by a single thread. Registers are the fastest, and are more plentiful than shared memory in most GPU architectures. Therefore, it is often beneficial to offload some storage from shared memory to registers. However, excessive register usage may result in lower occupancy or expensive register spills.

In this section, we discuss GPU resource considerations in determining size/shape of thread blocks and grid for spatial tiling of stencil computations.

A. Constraints on Thread Block and Grid Size

GPUs have hardware limits on 1) the maximum number of concurrently loaded threads per SM (T_{sm}); 2) the maximum number of threads in a thread block (T_b); 3) total shared memory per SM (M_{sm}); 4) the maximum number of concurrently loaded thread blocks per SM (B_{sm}); and 5) the register file size per SM (R_{sm}). For instance, $T_{sm} = 2^{11}$, $T_b = 2^{10}$, $M_{sm} = 48$ kB, and $B_{sm} = 16$, and $R_{sm} = 2^{16}$ for an NVIDIA Tesla K20c GPU. The threads in an SM can be grouped in various ways, e.g., two blocks of 1024 threads, 16 blocks of 128 threads, etc.

The typical approach to data-parallel execution on GPUs is to assign one thread for computation of one element of the result array. In order to tolerate the very high latency to global memory (several hundred clock cycles), massive parallelism must be utilized. If a sufficient number of independent instructions are ready to be issued among the collection of active warps in an SM, the memory access latency can be fully overlapped. But even if memory access latency can be fully overlapped, the peak memory bandwidth can limit performance, as discussed earlier. The only way to overcome the performance limitation from global memory bandwidth is to exploit temporal reuse on the accessed data, by buffering it in faster shared memory or registers. However, the shared memory and/or registers used for such buffering may result in a reduction in warp occupancy because fewer thread blocks may now be concurrently schedulable on an SM due to the register/shared memory usage per thread block.

Shared memory allows significantly faster access than global memory, and unlike registers, allows sharing of data across threads in a thread block. However, the more shared memory a thread block uses, the fewer thread blocks can be simultaneously scheduled on an SM. If m_b is the bytes of shared memory used by a thread block, the maximum number of concurrently active thread blocks cannot exceed M_{sm}/m_b.

Registers are the fastest storage resource available to a thread. The maximum number of registers per thread can be controlled via compiler flags for GPUs. If the maximum possible number of threads are to be active on an SM, the number of registers used per thread must be $\leq R_{sm}/T_{sm}$. If a thread uses t_{reg} registers, the maximum number of active threads per SM can be no more than $\min(T_{sm}, R_{sm}/t_{\text{reg}})$.

If the size of a thread block (number of threads) is sz_b, the maximum number of concurrently schedulable thread blocks per SM, $\max_b$, is upper bounded by $\max_b \leq \min(B_{sm}, T_{sm}/sz_b, M_{sm}/m_b, R_{sm}/t_{\text{reg}}.sz_b)$. For bandwidth-bound computations, unless thread coarsening is utilized to increase instruction level parallelism (ILP), the block size sz_b is often chosen to maximize occupancy, and consequently the thread level parallelism (TLP), i.e., $sz_b \times \max_b$ is set as close to T_{sm} as possible.

Coalesced access to global memory is important since it minimizes the number of memory transactions for accessing a given number of data elements. This requires that the fastest varying dimension of a multidimensional thread block be aligned with respect to data access with the fastest varying dimension of the accessed multidimensional array. The fastest varying dimension of the thread block is usually

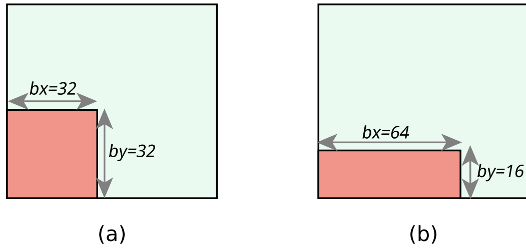


Fig. 5. ***Different thread block configurations for a fixed thread block size of 1024.***

chosen to be a multiple (or factor) of the warp size to benefit from coalescence.

B. Partitioning Threads Across Thread Block Dimensions

For a 2-D thread block of size sz_b, many choices exist for the extents along the x- and y-dimensions. The performance of a stencil computation can be quite sensitive to this choice. Let $sz_b = b_x \times b_y$. Two possible configurations for $sz_b = 1024$ are shown in Fig. 5. For an order-k stencil over an N^2 domain, the number of global memory loads is $N^2(32+2k)^2/1024$ for configuration (a) and $N^2(64+2k)(16+2k)/1024$ for configuration (b). For any value of k, configuration (a) requires fewer global load transactions for performing the same number of arithmetic operations. Consider, for example, a 3-D single-precision seven-point Jacobi stencil executed on a GP100 device. The version with block size 32×32 incurs 5.91E+07 global load transactions, achieving 1.2 TFLOPS. In contrast, the version with block size 64×16 incurs much higher 7.59E+07 global load transactions, achieving only 989 GFLOPS.

V. RESOURCE OPTIMIZATION STRATEGIES

In this section, we discuss various factors that affect the best choice of tiling strategy for a given stencil computation. Section V-A describes considerations for the 2-D case, for streamed overlapped tiling. Section V-B characterizes 3-D stencils based on their access patterns, and describes optimization techniques for them. We consider a stencil of order $k = 1$, with time tile size $T = 4$, and all operations in single precision. Tiled code must be appropriately tuned for different GPUs; for the discussion below, we use hardware parameters of an NVIDIA Tesla K20c as the target GPU.

A. Overlapped Tiling for 2-D Stencils

Many prior code generators that optimize 2-D stencils perform overlapped tiling along both the dimensions [3], [4]. In contrast, we partition the N^2 input domain into overlapping strips of size $B_x \times N$: we stream along the y[1]-dimension to eliminate redundant computations, performing overlapped tiling along x for intertile concurrency. Fig. 4(a) illustrates this strategy. The required amount of per-thread-block shared memory and registers depend on the thread block geometry.

1) Streaming With 2-D Thread Blocks: In the steady state, at each step, a $B_x \times B_y$ thread block reads B_y input lines along the y-dimension to compute B_y points of the iteration space. Each thread traverses the iteration space at a stride of B_y along the y-dimension. Each thread block stores $T(B_y + 2k)$ lines in shared memory. Consider two possible choices for B_y.

- $B_y = 1$ (1-D block): For maximum occupancy, each thread block must have $2048/16 = 128$ threads. In each iteration, a thread block loads one line from input at $t = 0$ and generates one line of output at $t = 4$. For this, it needs 6 kB of shared memory. From Section IV-A, it follows that an SM can have at most $\min(16, 2048/128, 48\text{ kB}/6\text{ kB}) = 8$ blocks, a 50% loss of occupancy.
- $B_y = 32$ (2-D block): Two thread blocks of size 32×32 can theoretically be active per SM. Since a thread block now operates on a chunk of 32 lines instead of 1, it needs 16 kB of shared memory. Each SM can have $\min(16, 2048/1024, 48\text{ kB}/16\text{ kB}) = 2$ active blocks, which implies maximum occupancy.

Clearly, better occupancy is achieved with 2-D blocks. For a $B_x \times B_y$ block, the computation proceeds as shown in Fig. 4(b): in the steady state, $B_y - 2k$ output lines are generated at time step t, using B_y lines from the shared memory buffer corresponding to time step $t - 1$. With a sliding-window approach, the B_y oldest lines in the buffer can be reused to cache the new lines. The buffer can be implemented as a circular array, with modulo operations to find the top and bottom k rows. Since modulo operations are costly on GPUs as they compile to multiple instructions, it is beneficial to make B_y a power of 2, so that they can be implemented by bitwise operators[2] which has a very high throughput.

2) Concurrent Streaming With 1-D Thread Block and Using Registers for Storage: As demonstrated above, serial streaming with 1-D thread blocks suffered lowered occupancy due to high usage of shared memory. If z is the streaming dimension, then a stencil is termed diagonal-access free along z-dimension if all stencil offsets (x_0, y_0, z_0) for access to points on different planes along the z-dimension are strictly of the form $(0, 0, z_0)$. For such diagonal-access-free stencils, the shared memory requirement can be decreased by using registers instead of shared memory to hold the $2k$ accesses to lines [17]. For a thread block of 128 threads, $k = 1$ and $T = 4$, a thread block now

[1]Streaming along the x-dimension is usually not beneficial because it entails noncoalesced accesses while loading the input data.

[2]$a \bmod b \equiv a$ and $(b-1)$ if b is a power of 2.

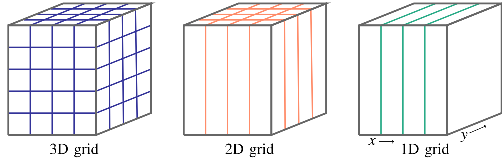


Fig. 6. *Various grid dimensionalities for a 3-D input.*

needs eight registers and 2 kB of shared memory. Thus, the number of concurrently loadable thread blocks per SM is min(16, 2048/128, 48 kB/2 kB) = 16. While occupancy is maximized, this strategy suffers from a low overall number of thread blocks. To keep all 13 SMs of the K20c GPU busy, at least 13 × 16 = 208 thread blocks are needed. However, even a large input domain of size 8192^2 can only be partitioned into 8192/128 = 64 blocks.

The number of thread blocks can be increased by using overlapped tiling along the y-dimension as well. However, it is only necessary to partition the space into $\lceil 208/64 \rceil = 4$ blocks along the y-dimension. Since the thread block is 1-D and each tile in the input grid is 2-D, streaming is performed within a tile.

This version does incur redundant computation and data access along the y-dimension, but this is negligible compared to the reduction in redundant computation and data access along the x-dimension due to increase in block size from 32 to 128. The lower access latency of registers, compared to shared memory is an additional benefit.

B. Overlapped Tiling for 3-D Stencils

Fig. 6 depicts different possible choices of grid dimensionality for 3-D stencils over a 3-D domain: 3-D, 2-D, or 1-D grid. A 3-D grid implies a halo region on all six sides of a 3-D interior data cube handled by a thread block, and suffers from significant redundant computation and data movement, as discussed earlier. At the other end, a 1-D grid incurs the lowest overhead from redundant computation and data movement, but the total number of thread blocks may be insufficient to keep all SMs of the GPU busy. A 2-D grid represents a good tradeoff and offers moderate overhead from redundant computation and data movement along with adequate number of thread blocks for the SMs on the GPU.

Streamed time tiling of a 3-D stencil requires $T(2k+1)$ planes to be cached in shared memory. For $T=4$ and $k=1$, a 32 × 32 thread block needs 48 kB of shared memory, the maximum per SM on a K20 GPU. This means that only one thread block can be scheduled at a time on an SM, limiting occupancy to 50%. For $k=2$, the required shared memory exceeds the hardware limit, forcing a smaller thread block size. Micikevicius [17] and Nguyen *et al.* [16] use registers to offload the caching of some planes from shared memory to registers for spatial and time tiling. Details regarding implementation of time tiling code with a combination of shared memory and registers are discussed next.

Algorithm 1: Streaming and Using Registers for Storage for a 3D Order-1 Stencil

Input : in : input array, T : time tile size
Output: out : output array

1. A_v^t : shared memory buffer for plane p_v at time step t;
2. r_{v-1}^t, r_{v+1}^t : registers for planes p_{v+1}, p_{v-1} at time step t;
   `// Initial reads`
3. $r_{v-1}^0 \leftarrow$ *load_plane* $(in[0][\ldots][\ldots])$;
4. $A_v^0 \leftarrow$ *load_plane* $(in[1][\ldots][\ldots])$;
5. **for** *each* z *from* 1 *to* $N-1$ **do**
6. $r_{v+1}^0 \leftarrow$ *load_plane* $(in[z+1][\ldots][\ldots])$;
7.   *__syncthreads ()*;
     `// Perform the computation per time step`
8.   **for** s *from 1 to* T **do**
9. $r_{v+1}^s =$ *compute_stencil* $(r_{v-1}^{s-1}, A_v^{s-1}, r_{v+1}^{s-1})$;
10.   *__syncthreads ()*;
11. $out[z-kT][\ldots][\ldots] = r_{v+1}^T$;
     `// Shift data per time step`
12.   **for** s *from 0 to* T **do**
13. $r_{v-1}^s \leftarrow A_v^s \leftarrow r_{v+1}^s$

1) Streaming and Using Registers for Storage: In the scenario above with $k=1$ and block size 32 × 32, if the stencil is diagonal-access free along the streaming dimension, then some storage can be offset to registers, increasing the per thread register pressure by $2Tk$, and simultaneously reducing the shared memory requirement to 16 kB. With this tradeoff, an SM can have two active blocks, achieving maximum occupancy. If there are register spills due to the increased register pressure, then the value of T can be reduced to alleviate the register pressure.

Fig. 7 illustrates this scheme applied to time tile the seven-point 3-D Jacobi stencil. Each output point at time step t needs to read data from three input planes at time step $t-1$. The invariant maintained in this scheme is that the data needed to compute plane p_v^t at time step t comes from registers that hold the values of planes p_{v-1}^{t-1} and p_{v+1}^{t-1}, and a shared memory buffer that holds the value of plane p_k^{v-1} at time step $t-1$. This is illustrated in Fig. 7(a), where the red (black) edges indicate that the data is being read from register (shared memory). The number on the edges indicates the sequence in which the contributions are made. Before the plane p_{v+1}^{t-1} makes contribution to the plane p_{v+1}^t, we perform a shift, where the data from the shared memory buffer holding the value of plane p_v^{t-1} is moved into the register that held the value of plane p_{v-1}^{t-1}, and the data from the register holding the value of p_{v+1}^{t-1} is moved into the shared memory buffer. The shifts are represented as the blue edges in Fig. 7(b). The shift helps maintain the invariant as the computation streams through the z-dimension. In the steady state, we can compute one plane at each time step, as shown in Fig. 7(c).

An implementation sketch of this scheme is presented in Algorithm 1. The algorithm is presented independently

```
for (i=1; i<N-1; i++) {
   for (j=1; j<N-1; j++) {
       b[i][j] = c0*(a[i-1][j-2] + a[i-1][j+2]) +
              c1*(a[i][j-1] + a[i][j] + a[i][j+1]) +
                c2*(a[i+1][j-2] + a[i+1][j+2]);
   }
}
```

Listing 2. ***A seven-point 2-D associative stencil that accesses two points from planes along y-dimension.***

```
for (i=0; i<N; i++) {
   for (j=0; j<N; j++) {
       b[i+1][j] += c0*(a[i][j-2] + a[i][j+2]);
       b[i][j] += c1*(a[i][j-1] + a[i][j] + a[i][j+1]);
       b[i-1][j] += c2*(a[i][j-2] + a[i][j+2]);
   }
}
```

Listing 3. ***Reading one plane from input at a time, and accumulating its contribution at different output points.***

of the tiling scheme for the other two dimensions. The initializations at line 1 depend on the tiling scheme used for the nonstreaming dimensions. Algorithm 1 can be generalized to all stencils where each plane accesses only one point per plane from other planes along z-dimension. If such a stencil accesses a diagonal point (x_0, y_0, z_0), then appropriate shuffle intrinsic will have to be inserted in the code, so that each thread maintains a correct central value in registers.

2) Storage Optimization for Associative Stencils: For stencils that access more than one point per plane from other planes along z-dimension (i.e., stencils that are not diagonal-access free along the streaming dimension), the optimization scheme described above may require too many registers to be beneficial. For example, a 27-point 3-D stencil with $T = 2$ will require 36 explicit registers per thread just for storage. If such a stencil is associative, we can use an optimization strategy that leverages contribution reordering to reduce the number of registers required at each time step to just $2k + 1$. A similar optimization strategy is used by Stock *et al.* [31] in the context of high-order stencils on multicore CPUs. Listing 2 shows an example of a stencil that is associative, and hence is amenable to our optimization. We leverage the associativity of addition and multiplication to rewrite it as an accumulation stencil of Listing 3.

Fig. 8 shows the application of this scheme to time tile the seven-point 3-D Jacobi stencil. An input plane at time step $t-1$ contributes to three points belonging to distinct output planes at time step t. There are three invariants maintained: 1) the plane p_v^{t-1} making the contribution from time step $t-1$ is cached in shared memory; 2) contributions to the planes p_{v+1}^t, p_{v-1}^t and p_v^t at time step t are accumulated in registers; and 3) after the accumulation, the contribution to the plane p_{v-1}^t is written out into a shared memory buffer. This is illustrated in Fig. 8(a), where the red (black) edges indicate that after accumulating the contribution, the final result is in register (shared memory). The number on the edges indicates the sequence in which the contributions are made. One can make the following observations from Fig. 8(b): 1) at $t = 0$, all the contributions from a plane are made before starting with the next plane, and therefore one can reuse the same shared memory buffer for all the planes; 2) at any time step, at most one shared memory plane is required, and at most $2k+1$ registers simultaneously receive contributions. Thus, we only need $2k + 1$ registers, and one shared memory buffer per time step. In a steady state [Fig. 8(c)], a plane that is in shared memory at time step t can start making contributions to the accumulation registers at time step $t + 1$. Algorithm 2 presents an implementation sketch for this approach.

VI. FUSION HEURISTICS

More complex stencil computations, as arising in image processing pipelines and multistatement stencils, can be

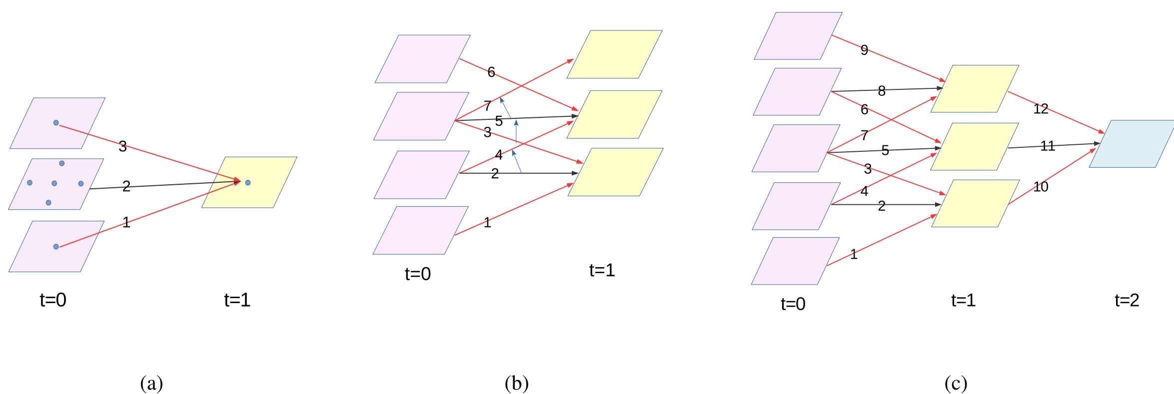
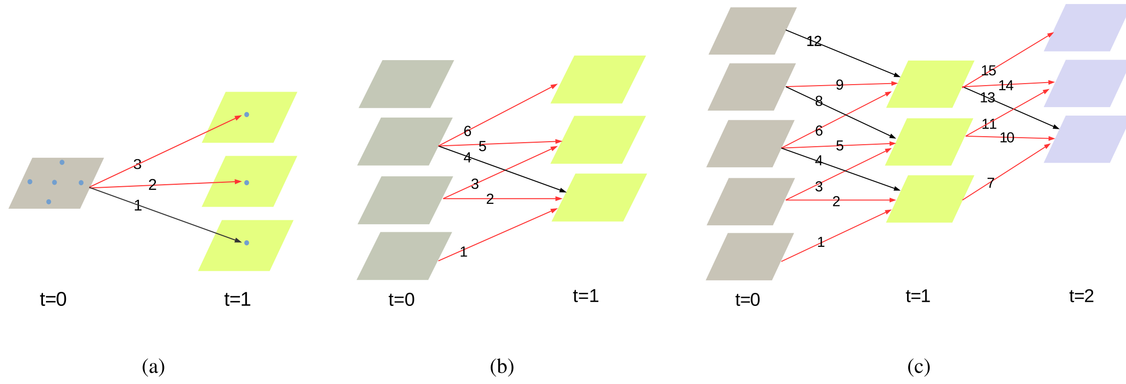


Fig. 7. ***Streaming for 2-D overlapped blocks of a 3-D seven-point Jacobi stencil. Red (black) arrows indicate that the value from the previous time step comes from register (shared memory). (a) Contributions from different planes at time step t−1 to an output point at time step t. (b) Prolog and data rotation. (c) Steady-state computation for two time steps.***

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.



(a) (b) (c)

Fig. 8. ***Streaming with associative reordering for 2-D overlapped blocks of a 3-D seven-point Jacobi stencil. Red (black) arrows indicate that the contributions from the previous time step are accumulated and written into a register (shared memory buffer). (a) Contributions from a single plane at time step $t-1$ to three output points at time step t. (b) Prolog. (c) Steady-state computation for two time steps.***

Algorithm 2: Storage Optimization for an Associative 3D Order-1 Stencil

Input : *in* : input array, T : time tile size
Output: *out* : output array

1 A^t_{v-1} : shared memory for plane p_{v-1} at time step t;
2 $r^t_{v-1}, r^t_v, r^t_{v+1}$: registers for planes p_{v-1}, p_v, p_{v+1} at time step t;
3 **for** *each z from* 0 *to* $N-1$ **do**
4 $\quad A^0_{v-1} \leftarrow$ *load_plane* ($in[z][\ldots][\ldots]$);
5 $\quad$ *__syncthreads*();
$\quad$ `// Perform the computation per time step`
6 $\quad$ **for** *s from 1 to T* **do**
7 $\quad\quad r^s_{v+1} \leftarrow$ *bottom_plane_contribution* (A^{s-1}_{v-1});
8 $\quad\quad r^s_v$ += *mid_plane_contribution* (A^{s-1}_{v-1});
9 $\quad\quad r^s_{v-1}$ += *top_plane_contribution* (A^{s-1}_{v-1});
10 $\quad\quad A^s_{v-1} = r^s_{v-1}$;
11 $\quad\quad$ *__syncthreads*();
12 $\quad out[z-kT][\ldots][\ldots] = A^T_{v-1}$;
$\quad$ `// Shift data per time step`
13 $\quad$ **for** *s from 1 to T* **do**
14 $\quad\quad r^s_{v-1} \leftarrow r^s_v \leftarrow r^s_{v+1}$;

represented by a directed acyclic graph (DAG) where each vertex represents a single stencil operation identified by the stencil statements on a set of input domains [3], [32], and edges represent data produced/consumed by these operations. Stencils that are iterated a fixed number of times can also be written in a DAG form, by unrolling the time loop to expose different nodes for different time steps.

Fusion of stencil operators in a DAG can be essential to improve reuse of data between operators. For example, fusing together nodes which read the same data can enable better temporal locality, and reduce memory traffic. But on the other hand, fusing nodes increase resource pressure (e.g., registers) as the kernel would implement multiple stencil operators with the same per-kernel resources budget. In this section, we present a simple resource-aware greedy heuristic to fuse stencil operators in a DAG.

For the fusion of two nodes in the DAG to be valid, it must preserve data dependences. For Jacobi-like stencils in a DAG, such fusion is always valid if the fused node is atomic, i.e., there is no dependence cycle after fusion. There are many valid fusion schemes for the nodes in the DAG, but their performance may vary significantly depending on the profitability of fusion (gains in temporal locality and memory traffic versus reduction in available per-operator resources in the fused kernel). The space of all valid fusion structures can be very large [33], and exhaustively exploring this space to find the optimal solution can be prohibitively expensive. To avoid such exploration, we instead propose a greedy algorithm to determine which nodes will be fused together, to optimize a dedicated objective function. Starting from the input DAG $D_g = (V, E)$, the objective is build the fused graph $D_f = (V_f, E_f)$ such that each $v_f \in V_f$ is a convex partition of nodes from V. A convex partition is an atomic "macronode," comprising nodes from V, such that there is no dependence path going out and back in the macronode. To form a macronode v_f, we fuse node(s) $v \in V$ following a profitability function. Eventually, a single GPU kernel is generated for each node $v_f \in V_f$. Note that if a temporary array is produced in v_f by design, and all its uses are contained in v_f, then no global memory transaction is needed for such temporary array.

A. Roadmap of the Greedy Fusion Algorithm

The greedy resource-driven fusion algorithm consists of the following steps.

Step 1) For each stencil operator, compute the amount of shared memory and registers that will be needed to cache the data (Section VI-B).

Step 2) Identify pairs of stencil operators that can be fused without violating dependences (Section VI-C).

Step 3) For each pair of stencil operators, compute the shared memory and registers used when fusing them in a single kernel, based on the individual resource usage computed in Step 1) (Section VI-D).

Step 4) Based on the resource usage of the fused node computed in Step 3), create a profitability metric that encodes the impact of fusion on the GPU resources, and the data movement volume (Section VI-E).

Step 5) Define a custom sort to order the profitability metrics of various stencil operator pairs, and choose to fuse the operators of the most profitable pair into a macronode (Section VI-F). Update the stencil DAG and the dependence graph, and repeat again from Step 1), until no more stencil operator pairs can be fused.

Once a fused stencil DAG D_f is produced by the algorithm, an optimized CUDA kernel can be generated for each node in D_f using the algorithms described in Section V.

B. Computing Resource Requirements of a Stencil Operator

Without loss of generality, we assume that streaming, as presented in prior sections, is done on the outermost dimension of a 3-D domain. As we present in Section V, to determine the resource requirements of an order-k stencil operator along the streaming dimension, we must determine the storage properties for the $2k+1$ accessed planes. We proceed as follows.

1) If computing an output value at (z_0, y_0, x_0) only requires a single input value at (z_r, y_0, x_0) from an input plane z_r, then z_r is stored as an explicit register (that is, z_r has register storage type). If not, then z_r is cached into shared memory
2) If an output element is to be (over)written again, e.g., because it is an accumulator, then it is stored in an explicit register. Conversely, the last assignment to a copy-out array uses global memory for its storage type.

An explicit register above is implemented with a (scalar) temporary variable, which contains an array element. The compiler will place such temporary variables in registers. These explicit registers are distinguished from implicit registers that are used to store arrays elements and the intermediate results of computing expressions, as produced by the compiler during traditional register allocation. We use this distinction for explanation purpose only, as in the generated code the explicit registers are thread scalars.

Listing 4 shows two consecutive time steps of a 3-D seven-point Jacobi stencil defined in the StencilGen language. From rule 1) above, the statement line 7 should use a shared memory buffer, storing the input plane $A[0,*,*]$, since five different values are read from that plane and contribute to different output points. From rule 2), three explicit registers must be used to store the output values written to $B[1,*,*]$, $B[0,*,*]$ and $B[-1,*,*]$ at lines 2, 3, and 5 respectively.

```
1  stencil j3d7pt (A, B, C) {
2    B[k+1][j][i] =  (A[k][j][i]);
3    B[k][j][i]   += (A[k][j-1][i] + A[k][j][i-1] + A[k][j][i]
4                     + A[k][j][i+1] + A[k][j+1][i]);
5    B[k-1][j][i] += (A[k][j][i]);
6
7    C[k][j][i]   =  (B[k-1][j][i]);
8    C[k-1][j][i] += (B[k-1][j-1][i] + B[k-1][j][i-1] +
9                  B[k-1][j][i] + B[k-1][j][i+1] + B[k-1][j+1][i]);
10   C[k-2][j][i] += (B[k-1][j][i]);
11 }
```

Listing 4. ***Two time steps of a seven-point 3-D Jacobi stencil.***

We represent the resource requirement of a stencil operator using three attributes: N_{reg} and N_{shm} capture how many explicit registers and shared memory buffers are used by the operator, and $M_{\text{acc}\rightarrow\text{res}}$ is a map from each accessed data plane to the storage type used for that plane.

C. Identifying Fusion Candidates

The dependence graph $G_{\text{dep}} = (V, E_{\text{dep}})$ captures the dependences between stencil operators in the DAG. From G_{dep}, consider a transitive dependence graph $G_{\text{trans}} = (V, E_{\text{trans}})$, such that

$$\begin{aligned} s_i \rightarrow s_j \in E_{\text{trans}} \text{ iff } \{ &\exists s_k \mid s_k \in V \wedge s_k \neq s_i \wedge s_k \neq s_j \\ &\wedge \text{ there exists a path from } s_i \text{ to } s_k \text{ (denoted as } s_i \rightsquigarrow s_k) \\ &\wedge \text{ there exists a path from } s_k \text{ to } s_j \text{ i.e., } s_k \rightsquigarrow s_j\}. \end{aligned}$$

G_{trans} is used to ensure the validity of fusion, i.e., that all macronodes generated by fusion are convex/atomic nodes. If $s_i \rightarrow s_j \in E_{\text{trans}}$, then fusing s_i and s_j would break convexity, unless all nodes along any path $s_i \rightsquigarrow s_j$ are also included in the fused node. This definition allows the fusion of two stencils in producer–consumer relationship, as long as there are no other operators along any path from s_i to s_j.

For each unordered pairs of stencil operators (s_i, s_j) such that $s_i \rightarrow s_j \notin E_{\text{trans}}$, we first compute the total GPU resources required if s_i and s_j are fused, and then compute a fusion profitability metric that will quantify the benefit of fusing s_i and s_j in terms of GPU resource usage and data movement reduction.

D. Computing the Postfusion Resource Map

We first determine the resource usage of the fused macronode made of s_i and s_j. Let M_i and M_j be the maps from accessed array planes to GPU resources ($M_{\text{acc}\rightarrow\text{res}}$) for the operators s_i and s_j. Let M_{fused} be the resource map for the fused node. If there is no dependence between s_i and s_j, then M_{fused} is simply the union of the resource maps of the individual nodes, where the rules of union are as defined in this section.

But if there is a true dependence between s_i and s_j, to accurately compute resource usage, it is necessary to

first build the interleaving of the two operators domain that captures a fused execution schedule that preserve the dependence(s). Once this schedule is built, the resource map of the dependence sink may need adjustments. For example, in Listing 4, after the contribution at line 5, the values of plane $B[-1,*,*]$ can be used as input for the next time step (lines 7–10). There is a true dependence from the write to plane $B[-1,*,*]$ at line 5 to the subsequent reads at lines 7–10. Observe that the accesses along the streaming dimension are shifted by the dependence distance in lines 7–10 to ensure that after fusing lines 2–10, dependences are preserved. It is sufficient to shift by the dependence distance along the streaming dimension.

Once the schedules are adjusted to represent a valid fused schedule, the set of planes accessed by s_i and s_j may intersect. For example, in Listing 4, $B[-1,*,*]$ is mapped to a register in line 5, and to shared memory in lines 8–9. In the presence of such storage conflict for a plane, we always choose the storage that is lower in the memory hierarchy, i.e., shared memory in this example.

E. Computing the Profitability Metric

Once the resource map for the fusion of s_i, s_j is computed, we determine the profitability of fusing these nodes using five metrics as follows.

D_m represents the savings, in terms of data movements, after fusing s_i and s_j. For every array leading to a true dependence between s_i and s_j, data movements are reduced after fusion, as this array does not require as much global memory transfers after fusion versus without fusion. To capture this, when D_m is incremented by 2, this models a saving in a store and then subsequent load of that array. There is no explicit data movement saving for arrays carrying write-after-read dependence. If there is a write-after-write (or read-after-read) dependence between s_i and s_j but no read-after-write dependence, then D_m is incremented by 1 to represent the reduction in multiple writes (reads) to (from) the same location in global memory.

S_{reg} represents the total number of accesses that are mapped to the same explicit registers in s_i and s_j after fusion, i.e., $S_{\text{reg}} = \text{num_registers}(M_i) + \text{num_registers}(M_j) - \text{num_registers}(M_{\text{fused}})$. Similarly, S_{shm} represents the total number of accesses that are mapped to the same shared memory buffer in s_i and s_j after fusion, i.e., $S_{\text{shm}} = \text{num_shared}(M_i) + \text{num_shared}(M_j) - \text{num_shared}(M_{\text{fused}})$.

Finally, T_{reg} represents the total number of explicit registers in the fused node, and T_{shm} the total amount of shared memory used.

F. Constructing the Objective Function

Once all metrics above has been computed for all valid stencil operator pairs, they are sorted based on their profitability. We use a customized sort operation $\prec$, which models a total order of the list of candidate fusion pairs. The set of rules that are used to order two pairs $c_i, c_j \in L_{\text{tuple}}$ is as follows:

a) $(D_m)_{c_i} < (D_m)_{c_j} \Rightarrow c_i \prec c_j$;
b) $(T_{\text{reg}} + T_{\text{shm}})_{c_j} < (T_{\text{reg}} + T_{\text{shm}})_{c_i} \Rightarrow c_i \prec c_j$;
c) $(T_{\text{shm}})_{c_j} < (T_{\text{shm}})_{c_i} \Rightarrow c_i \prec c_j$;
d) $(T_{\text{reg}})_{c_j} < (T_{\text{reg}})_{c_i} \Rightarrow c_i \prec c_j$;
e) $(S_{\text{reg}} + S_{\text{shm}})_{c_i} < (S_{\text{reg}} + S_{\text{shm}})_{c_j} \Rightarrow c_i \prec c_j$;
f) $(S_{\text{shm}})_{c_i} < (S_{\text{shm}})_{c_j} \Rightarrow c_i \prec c_j$;
g) $(S_{\text{reg}})_{c_i} < (S_{\text{reg}})_{c_j} \Rightarrow c_i \prec c_j$;
h) $i < j \Rightarrow c_i \prec c_j$.

Rule h) is only used to ensure that a total order can be found in case of tuples with strictly identical metrics.

It is possible for two or more candidate pairs to have the same metric, and the order in which the sorting rules are applied determines how such case is handled. For example, if the objective is to minimize data movements, $\prec$ follows the rule sequence a) $\rightarrow$ b) $\rightarrow$ c) $\rightarrow$ d) $\rightarrow$ e) $\rightarrow$ f) $\rightarrow$ g) $\rightarrow$ h), where a) is the primary sorting rule, and ties are further sorted by implementing the remaining sorting rules [e.g., b), c), etc.].

Once sorted, the topmost tuple of L_{tuple} is the most profitable candidate for fusion. If the fusion 1) does not exceed the hardware limit on shared memory, and 2) does not result in a prohibitively high stencil order (and consequently high quantity of redundant computations) if s_i and s_j are in producer–consumer relationship, then these nodes are fused. The stencil DAG is modified by replacing the original operators by the node modeling their fusion, and the dependence graph is updated as needed. The process is then repeated on this updated DAG, until no further fusion is feasible or profitable.

VII. RELATED WORK

Automatic high-performance GPU code generation for stencils has been a topic of active research for both CPUs [2], [9]–[11], [20], [34] and GPUs [4]–[8], [12], [13], [18], [19]. The main issues discussed in this paper about developing high-performance code for stencil computations have also been addressed in similar or equivalent ways in these efforts. In this section, we provide a brief discussion of the related work.

PPCG [19] is a polyhedral source-to-source compiler that generates classically time tiled OpenCL and CUDA code from an annotated sequential program. Patus [2] is a code generation and autotuning framework for stencil computations that can generate spatially tiled CUDA code without shared memory usage from the input DSL stencil specification. Mint [6] is a pragma-based source-to-source translator implemented in the ROSE compiler [35] that generates a spatially tiled CUDA code from traditional C code. Unlike these approaches that generate code for a single GPU device, Physis [7] translates user-written structured grid code into CUDA+MPI code for GPU-equipped clusters. Zhang and Mueller [21] develop an autotuning strategy for 3-D stencils on GPUs. Their framework uses thread coarsening along different dimensions with

Table 1 Benchmark Characteristics

Benchmark	N	T	k	FPP	Benchmark	N	T	k	FPP	Benchmark	N	T	k	FPP
j2d5pt	8192^2	4	1	10	j3d7pt	512^3	4	1	13	heat	512^3	4	1	15
j2d9pt-gol	8192^2	4	1	18	j3d13pt	512^3	4	2	25	poisson	512^3	4	1	21
j2d9pt	8192^2	4	2	18	j3d17pt	512^3	4	1	28	cheby	512^3	4	1	39
gaussian	8192^2	4	2	50	j3d27pt	512^3	4	1	54	denoise	512^3	4	2	62
gradient	8192^2	4	1	18	curl	450^3	1	1	36	hypterm	300^3	1	4	358

N: Domain Size, T: Time Tile Size, k: Stencil Order, FPP: FLOPs per Point

judicious use of shared memory and registers to improve performance of spatially tiled 3-D stencils.

Overtile [4] and Forma [3] are DSL compilers that generate time-tiled CUDA code from an input stencil DSL specification. PolyMage [32] is a DSL-based code generator for automatic optimization of image processing pipelines. All these approaches use overlapped tiling to fuse the stencil operators in the computation, with the intermediate arrays stored in shared memory. The code generated for 3-D stencils by Overtile and Forma uses overlapped tiling along all three spatial dimensions and therefore incurs higher data movement and redundant computation overhead than streamed tiling [15]. Grosser *et al.* [5] implement the split-tiling approach of [10], and hexagonal tiling [12] for temporal tiling in GPUs. ChiLL [36] is a composable loop transformation framework which allows the user to script loop transformations for stencil computations.

Micikevicius [17] presents a CUDA implementation of 3-D finite difference computation that performs spatial tiling, and uses registers to alleviate shared memory pressure. Nguyen *et al.* [16] extend this implementation to a 3.5-D blocking algorithm for 3-D stencils: streaming along one dimension, and temporal tiling along the other two dimensions. For CPUs, their approach reduces the cache footprint, and for GPUs, it reduces the shared memory required to time tile a class of cross stencils.

In context of CPUs, the Pochoir [9] stencil compiler uses a DSL approach to generate high-performance Cilk code that uses an efficient parallel cache-oblivious algorithm to exploit reuse. The Pochoir system provides a C++ template library that allows the stencil specification to be executed directly in C++ without the Pochoir compiler, which aids debugging. Henretty *et al.* [10] propose hybrid split tiling and nested split tiling, to achieve parallelism without incurring redundant computations. Halide [20] is a DSL language for image processing pipelines. It decouples algorithm specification from its execution schedule. The advantage of this separation is that one can write multiple schedules for the same computation without rewriting the entire computation. Halide schedules can express loop nesting, parallelization, loop unrolling, and vector instruction. The performance of the optimized code depends on the efficacy of the schedule. The schedule can be either written manually by a domain expert, or generated by extensive autotuning (e.g., with OpenTuner [37]); these approaches either require some degree of expertise with Halide DSL, or are time consuming. Recently, Mullapudi *et al.* [38] extended the scheduling strategy of PolyMage [32] to automatically generate schedules for Halide. Bandishti *et al.* [11] propose diamond tiling to ensure concurrent startup as well as perfect load-balance whenever possible. Yount *et al.* describe the YASK [34] framework to simplify the task of defining stencil functions, generating high-performance code targeted for Intel Xeon Phi architecture. In Section VIII, we present experimental results with several of these stencil optimizers. Olschanowsky *et al.* [39] focus on large-scale PDE benchmarks as those written using Chombo [40] framework. Chombo parallelizes the application across the nodes using MPI, where each MPI process operates over a set of boxes, and each box applies a sequence of stencil operations over its domain. They evaluate the benefits of interloop stencil optimizations like loop shifting, loop fusion, wavefront tiling, and overlapped tiling on various box sizes. Davis *et al.* [41] use modified macro dataflow graphs to represent the stencil computation, and then apply fusion, rescheduling, and tiling optimizations to reduce communication volume and storage requirements. The stencil operations in the optimized graph can then be reinterpreted as relations bounded by affine constraints, to enable automatic code generation using ISL [42]. They also propose a cost model to compare schedules based on the memory traffic.

Table 2 Benchmarking Hardware

Resource	Details
Quadro GP100 GPU	3584 CUDA cores, 717 GB/s peak global memory bandwidth, 10.6 TFLOPS peak single-precision performance, 5.3 TFLOPS peak double-precision performance
Quadro GV100 GPU	5120 CUDA cores, 870 GB/s peak global memory bandwidth, 14.8 TFLOPS peak single-precision performance, 7.4 TFLOPS peak double-precision performance
Intel Skylake CPU	Intel core i7-6700K (4 cores, 2 threads/core 4.00 GHz, 512 GFLOPS peak single-precision performance)
Intel Xeon CPU	Intel Xeon E5-2680 (14 cores, 2 threads/core 2.40 GHz, 1.075 TFLOPS peak single-precision performance)
KNL phi	Intel Xeon Phi 7250 (68 cores, 4 threads/core 1.40GHz, 6.092 TFLOPS peak single-precision performance)

Table 3 Compilation Flags for GPU, Multi-Core CPU, and Xeon Phi

Compiler	Flags
nvcc	-O3 -maxrregcount=*n* -ccbin=g++ -std=c++11 -Xcompiler ”-fPIC -fopenmp -O3 -fno-strict-aliasing” –use_fast_math -Xptxas ”-v -dlcm=ca” -arch=sm_60/70
gcc	-Ofast -fopenmp -ffast-math -fstrict-aliasing -march=core-avx2/{knl -mavx512f -mavx512er -mavx512cd -mavx512pf} -mfma -ffp-contract=fast
llvm	-Ofast -fopenmp=libomp -mfma -ffast-math -fstrict-aliasing -march=core-avx2/knl -mllvm -ffp-contract=fast
icc	-Ofast -qopenmp -no-prec-div -march=native -fma -xMIC-AVX512 -fp-model fast=2 -complex-limited-range -override-limits

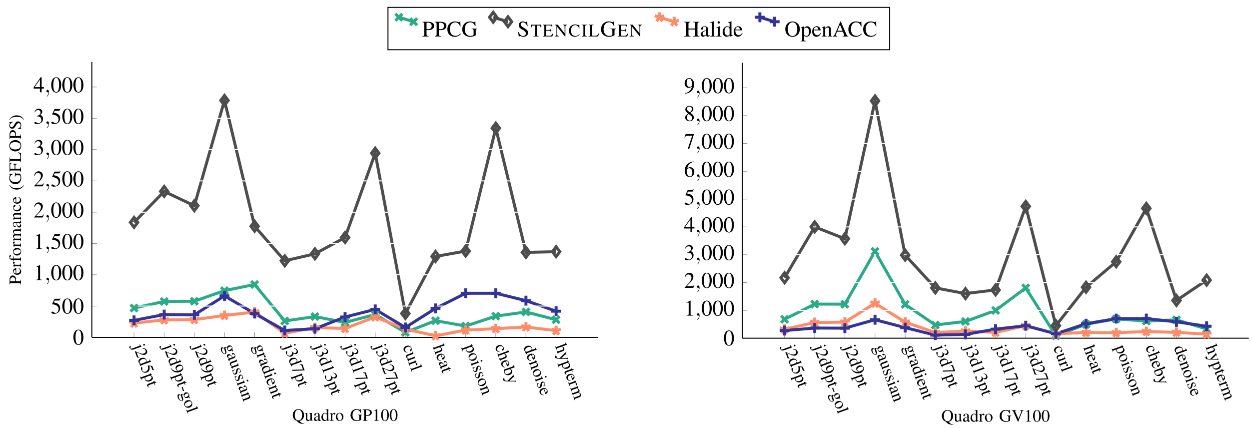


Fig. 9. ***Performance of single-precision benchmarks on Pascal and Volta GPU devices.***

Wahib and Maruyama [13] pose kernel fusion as an optimization problem, and use a codeless performance model to choose a near-optimal fusion configuration among other possible variants. The space of feasible solutions is pruned using a search heuristic based on a hybrid grouping genetic algorithm. Gysi *et al.* [8] also propose a model-driven stencil optimization approach. Like Wahib and Maruyama [13], they use a codeless performance model to find the best fusion configuration among all valid topological sorts of the stencil DAG. The model has been used to guide kernel fusion in the Stella library [43]. Prajapati *et al.* [44] propose an analytical model that predicts the execution time of the code generated with hexagonal tiling.

VIII. EXPERIMENTAL EVALUATION

A. Evaluation on GPUs

1) Experimental Setup: We have implemented the tiling schemes and fusion heuristics described in Sections V and VI into the STENCILGEN code generator, and we compare below the performance of the STENCILGEN -generated code against PPCG-0.08 [19], OpenACC-17.4 [45], and the auto-scheduler branch of Halide [38]. The benchmarks used in evaluation are listed in Table 1. We evaluate the performance of both single- and double-precision versions of the benchmarks on Pascal GP100 and Volta GV100 devices; the hardware is detailed in Table 2. For PPCG and STENCILGEN, the generated code was compiled using NVCC 9.1 [46]. The Halide generated code was compiled with LLVM 3.7. The compilation flags are listed in Table 3.

2) Code Generation: PPCG performs classical time tiling along with thread coarsening. Mapping multiple iterations to a thread exposes instruction level parallelism. Coarsening within the sustainable per thread register pressure aids register level reuse, and helps hide memory access latency by exposing instruction-level parallelism [47]. We created multiple versions by tuning the block size and unrolling factors for PPCG, and report results for the version with best performance. STENCILGEN does not have the support for thread coarsening at present. For each benchmark, we leverage operator associativity to alleviate pressure on GPU resources. GPUs allow compile-time flexibility in assigning the number of registers per thread. The perfor-

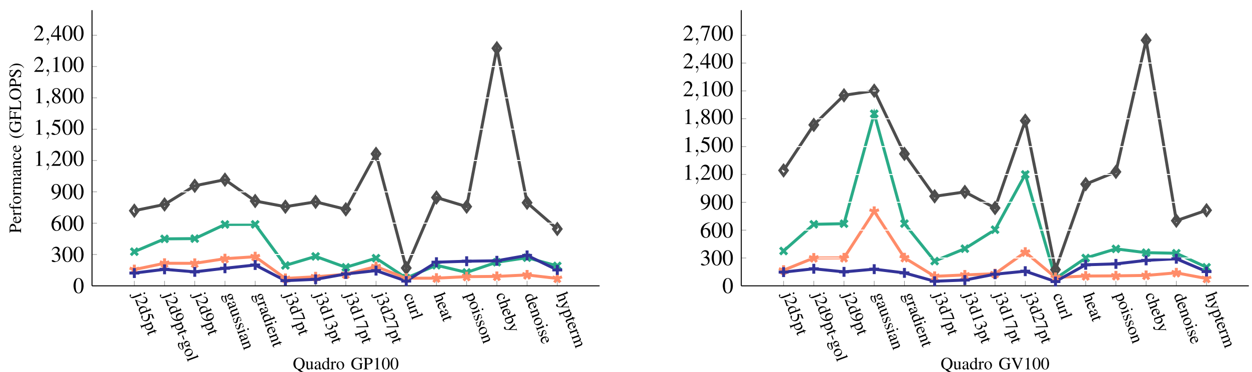


Fig. 10. ***Performance of double-precision benchmarks on Pascal and Volta GPU devices.***

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

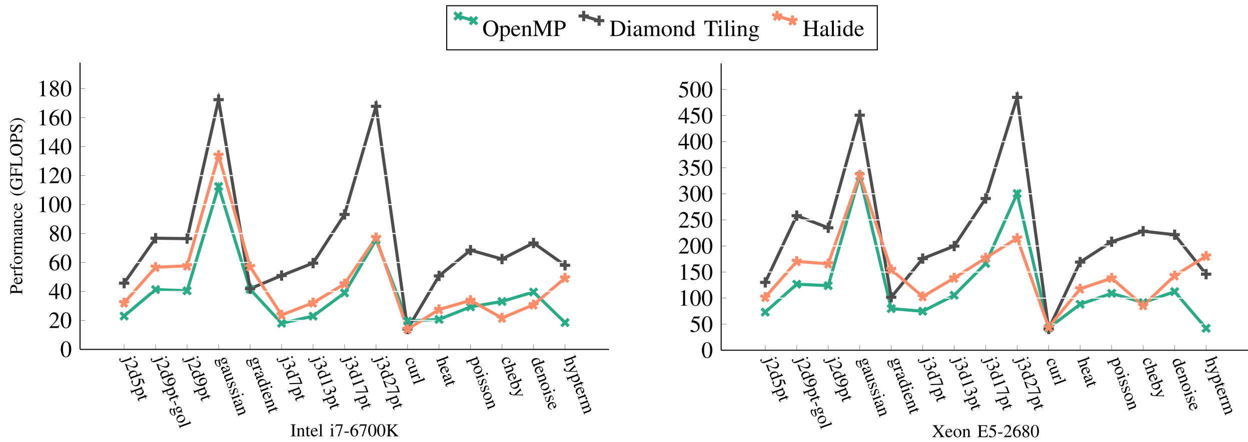


Fig. 11. ***Performance of benchmarks on Skylake and Xeon multicore CPU.***

mance of the same code with varying registers per thread can vary dramatically. This flexibility particularly affects the fusion heuristics for 3-D benchmarks. We use STENCILGEN to generate multiple versions for the 3-D benchmarks with different number of explicit registers. The versions with higher limit of explicit registers have a greater degree of fusion, and are compiled with higher registers per thread (e.g., maxrregcount= 64 for *heat* and *poisson*, maxrregcount= 255 for *hypterm*). Pascal and Volta devices can load the read-only data through the cache used by texture pipeline. To enable this feature, the read-only data in STENCILGEN -generated code is automatically annotated with the `__restrict__` keyword. Both PPCG and STENCILGEN -generated codes are compiled with different registers-per-thread settings to find the best configuration. To ensure coalescence, the fastest-varying dimension of the thread block is never chosen as a streaming dimension.

3) GPU Performance Results: Figs. 9 and 10 plot the performance of STENCILGEN-generated single- and double-precision codes, respectively, against different code generators on the two GPU devices. STENCILGEN systematically outperforms the other code generators in our experiments, by a factor up to 9×. Our optimization scheme for associative stencils helps reduce shared memory requirement for stencils like *gaussian, j3d27pt*, and *chebyshev*, allowing us to achieve higher occupancy, and consequently higher performance. We achieve 1.0 TFLOPS (2.1 TFLOPS) for double-precision *gaussian* stencil, 1.2 TFLOPS (1.7 TFLOPS) for double-precision *j3d27pt* stencil, and 2.2 TFLOPS (2.6 TFLOPS) for double-precision *chebyshev* stencil on GP100 (GV100) device. For the 2-D stencils, the best performance is achieved by overlapped tiling along both x- and y-dimensions, and concurrent-streaming along y-dimension. For the 3-D stencils, overlapped tiling along x- and y-dimensions, and serial-streaming along z-dimension gives best performance. For 3-D order-2 time-iterated stencils, restricting the fusion to two time steps yields better performance due to lower recomputation volume.

Despite applying a plethora of optimizations, we achieve only 37% and 58% (44% and 36%) of the single-precision

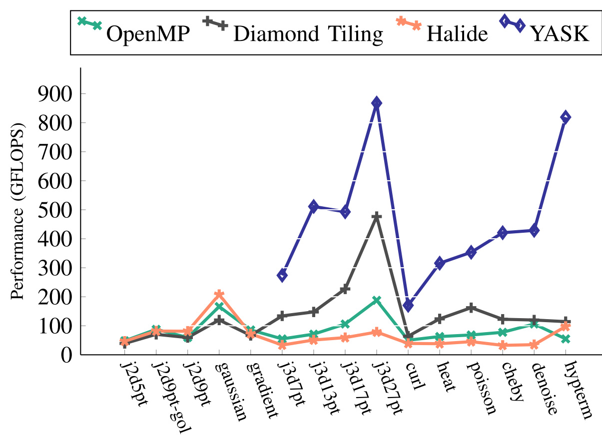


Fig. 12. ***Performance of benchmarks on Xeon Phi 7250.***

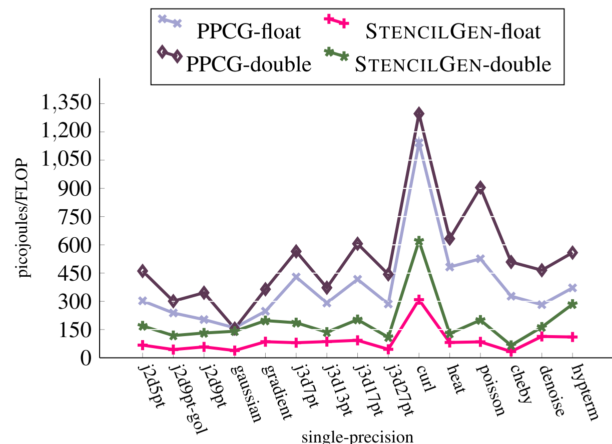


Fig. 13. ***Energy expended per FLOP on Pascal device.***

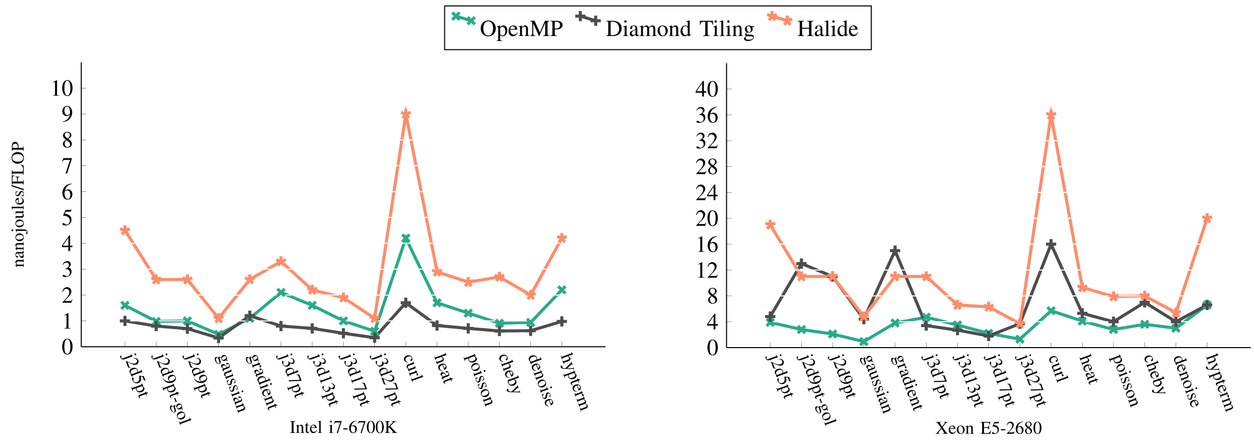


Fig. 14. ***Energy expended per FLOP on Skylake and Xeon multicore CPU.***

(double-precision) peak performance on GP100 and GV100, respectively. This serves to highlight the bandwidth-bound nature of stencil computations. For theoretically compute-bound stencils like *hypterm* and *denoise*, high register pressure lowers the achieved occupancy, making it difficult to hide the memory access latency. One can further reduce the bandwidth, and improve the ILP by exploring optimizations like thread-coarsening, vectorized loads, and hiding access latency via data prefetching.

B. Stencil Computations on CPUs

1) Experimental Setup: Many frameworks explicitly target stencil optimizations on CPU [2], [10], [11], [20], [34]. We evaluate the CPU code generated by diamond tiling [11] and Halide with autoscheduling support [38] against a baseline OpenMP code on two different multicore CPU processors, and a Xeon Phi processor. We also evaluate Intel's latest stencil optimization framework, YASK [34], on the Xeon Phi processor. Due to the limitations of the framework, we could only generate YASK code for 3-D stencils. We use three different compilers, namely ICC-17.0 [48], LLVM-5.0 [49], and GCC-7.2 [50] to compile the generated C/C++ stencil code, and report the highest performance.

2) CPU Performance Results: The performance is plotted in Figs. 11 and 12. One can observe that diamond tiling outperforms both Halide and the baseline code with its time tiling and concurrent start optimizations. On Xeon Phi, the performance of YASK is far superior to that of other frameworks. YASK performs optimizations like vector folding, cache blocking, and temporal wave-front tiling, that are specifically tuned for the Xeon Phi architecture.

For the multicore CPUs, only diamond tiling is able to achieve nearly 30%–45% of the machine peak for *j3d27pt* stencil. Despite the high performance, YASK is able to achieve only 15% of the machine peak.

C. Energy Efficiency

Figs. 13–15 plot the energy expended in a single floating-point operation for GP100, multicore CPUs, and Xeon Phi, respectively. For the stencil codes compiled with NVCC, we use the NVIDIA Management Library (NVML[3]) to measure the Joules/FLOP ratio. For multicore CPUs and Xeon Phi, we use RAPL[4] to obtain the measurement. Although typically measurements indicate that faster code versions have a lower Joules per FLOP ratio, data show situations where a faster code has a lower energy efficiency as when comparing Halide and OpenMP versions of *j2d9pt-gol* on Xeon Phi. We conjecture the use of more power-hungry instructions (e.g., FMA) and additional in-processor data traffic as possible causes for this slight decrease in energy efficiency, as unfortunately obtaining fine-grain power measurements to determine the root cause is not feasible with our energy measurement setup.

[3] https://developer.nvidia.com/nvidia-management-library-nvml
[4] http://web.eece.maine.edu/ vweaver/projects/rapl/

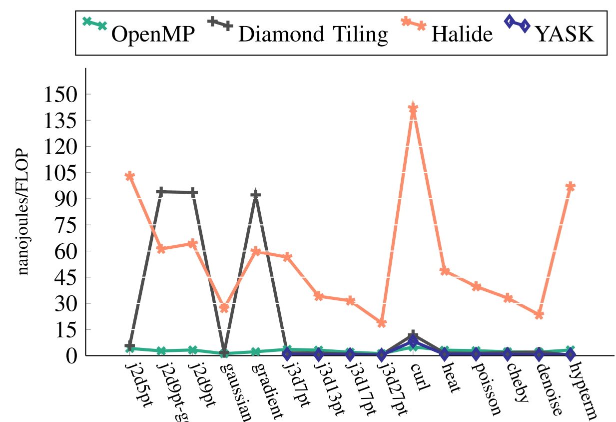


Fig. 15. ***Energy expended per FLOP on Xeon Phi 7250.***

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Note that the benchmarks running on GP100 have a much lower (1× – 0.1×) Joules/FLOP ratio that the multicore CPUs and Xeon Phi, while also achieving slightly higher peak performance. It indicates that the current generation of GPU devices appear more energy efficient than these CPU processors in executing such bandwidth-bound stencil computations.

IX. CONCLUSION

Stencil computations are at the computational core for many applications. Unlike dense linear algebra computations, where efficient libraries are widely available, the variety of manifestations of stencil patterns makes it infeasible to create libraries for stencils. However, the fundamental characteristics of the data access patterns for stencil computations can be used to devise domain-specific and target-specific optimizations in a source-to-source transformer and code generator for stencils. This paper has presented an analysis of the fundamental considerations in achieving high performance for stencil computations on GPUs, focusing on tiling strategies that make effective use of key resources like shared memory and registers. Experimental results demonstrate the significant benefits from use of domain-specific optimization over state-of-the-art general-purpose optimizers. ■

Acknowledgments

The authors would like to thank the reviewers for their feedback that helped improve this paper.

REFERENCES

[1] *Advanced Stencil-Code Engineering (ExaStencils)*. [Online]. Available: http://www.exastencils.org/
[2] M. Christen, O. Schenk, and H. Burkhart, "PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2011, pp. 676–687.
[3] M. Ravishankar, J. Holewinski, and V. Grover, "Forma: A DSL for image processing applications to target GPUs and multi-core CPUs," in *Proc. 8th Workshop General Purpose Process. Using (GPUs)*, 2015, pp. 109–120.
[4] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *Proc. 26th ACM Int. Conf. Supercomput. (ICS)*, 2012, pp. 311–320.
[5] 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 *Proc. 6th Workshop Gen. Purpose Processor Using Graph. Process. Units (GPGPU)*, 2013, pp. 24–31.
[6] D. Unat, X. Cai, and S. B. Baden, "Mint: Realizing CUDA performance in 3D stencil methods with annotated C," in *Proc. Int. Conf. Supercomput. (ICS)*. New York, NY, USA: ACM, 2011, pp. 214–224.
[7] 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 *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*. New York, NY, USA: ACM, 2011, pp. 11-1–11-12.
[8] T. Gysi, T. Grosser, and T. Hoefler, "MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures," in *Proc. 29th ACM Int. Conf. Supercomput. (ICS)*. New York, NY, USA: ACM, 2015, pp. 177–186.
[9] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir stencil compiler," in *Proc. 23rd Annu. ACM Symp. Parallelism Algorithms Archit. (SPAA)*. New York, NY, USA: ACM, 2011, pp. 117–128.
[10] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector SIMD architectures," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput. (ICS)*. New York, NY, USA: ACM, 2013, pp. 13–24.
[11] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2012, pp. 1–11.
[12] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for GPUs," in *Proc. Annu. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2014, pp. 66-66–66-75.
[13] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound GPU applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*. Piscataway, NJ, USA: IEEE Press, 2014, pp. 191–202.
[14] P. S. Rawat *et al.*, "Resource conscious reuse-driven tiling for GPUs," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, 2016, pp. 99–111.
[15] 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 *Proc. 9th Annu. Workshop Gen. Purpose Process. Using Graph. Process. Unit (GPGPU)*, 2016, pp. 92–102.
[16] 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 *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2010, pp. 1–13.
[17] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proc. 2nd Workshop Gen. Purpose Process. Graph. Process. Units (GPGPU)*, 2009, pp. 79–84.
[18] M. Ravishankar, P. Micikevicius, and V. Grover, "Fusing convolution kernels through tiling," in *Proc. 2nd ACM SIGPLAN Int. Workshop Libraries, Lang., Compil. Array Program. (ARRAY)*, 2015, pp. 43–48.
[19] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM TACO*, vol. 9, no. 4, pp. 54-1–54-23, Jan. 2013.
[20] 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 *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2013, pp. 519–530.
[21] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters," in *Proc. Tenth Int. Symp. Code Gener. Optim. (CGO)*. New York, NY, USA: ACM, 2012, pp. 155–164.
[22] K. Datta *et al.*, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*. Piscataway, NJ, USA: IEEE Press, Nov. 2008, pp. 4:1–4:12.
[23] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs," in *Proc. 23rd Int. Conf. Supercomput. (ICS)*. New York, NY, USA: ACM, 2009, pp. 256–265.
[24] A. Vizitiu, L. Itu, C. Niţă, and C. Suciu, "Optimized three-dimensional stencil computation on Fermi and Kepler GPUs," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2014, pp. 1–6.
[25] N. Maruyama and T. Aoki, "Optimizing stencil computations for NVIDIA Kepler GPUs," Tech. Rep., 2014.
[26] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart, "Parallel data-locality aware stencil computations on modern micro-architectures," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–10.
[27] A. D. Pereira, M. Castro, M. A. R. Dantas, R. C. O. Rocha, and L. F. W. Góes, "Extending OpenACC for efficient stencil code generation and execution by skeleton frameworks," in *Proc. Int. Conf. High Perform. Comput. Simulation (HPCS)*, Jul. 2017, pp. 719–726.
[28] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua, "Hierarchical overlapped tiling," in *Proc. 10th Int. Symp. Code Gener. Optim. (CGO)*. New York, NY, USA: ACM, 2012, pp. 207–218.
[29] P. Basu, S. Williams, B. Van Straalen, L. Oliker, P. Colella, and M. Hall, "Compiler-based code generation and autotuning for geometric multigrid on GPU-accelerated supercomputers," *Parallel Comput.*, vol. 64, pp. 50–64, May 2017.
[30] P. S. Rawat, "Optimization of stencil computations on GPUs," Ph.D. dissertation, Ohio State Univ., Columbus, OH, USA, 2018, pp. 1–200.
[31] K. Stock *et al.*, "A framework for enhancing data reuse via associative reordering," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI)*. New York, NY, USA: ACM, 2014, pp. 65–76.
[32] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic optimization for image processing pipelines," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2015, pp. 429–443.
[33] L.-N. Pouchet *et al.*, "Loop transformations: Convexity, pruning and optimization," in *Proc. 38th Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 2011, pp. 549–562.
[34] C. Yount, J. Tobin, A. Breuer, and A. Duran, "YASK-yet another stencil kernel: A framework for HPC stencil code-generation and tuning," in *Proc. 6th Int. Workshop Domain-Specific Lang. High-Level Frameworks HPC (WOLFHPC)*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 30–39.
[35] D. J. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Process. Lett.*, vol. 10, nos. 2–3, pp. 215–226, 2000.
[36] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance CUDA code," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 31:1–31:25, Jan. 2013.
[37] 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 *Proc. 23rd Int. Conf. Parallel Archit. Compilation (PACT)*. New York, NY, USA: ACM, 2014, pp. 303–316.
[38] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling Halide image processing pipelines,"

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

ACM Trans. Graph., vol. 35, no. 4, pp. 83:1–83:11, Jul. 2016.

[39] C. Olschanowsky C. Olschanowsky, M. M. Strout, S. Guzik, J. Loffeld, and J. Hittinger, "A study on balancing parallelism, data locality, and recomputation in existing PDE solvers," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2014, pp. 793–804.

[40] M. Adams *et al.*, "Chombo software package for AMR applications—Design document," Lawrence Berkeley Nat. Lab., Berkeley, CA, USA, Tech. Rep. LBNL-6616E, 2009.

[41] E. C. Davis, M. M. Strout, and C. Olschanowsky, "Transforming loop chains via macro dataflow graphs," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*. New York, NY, USA: ACM, 2018, pp. 265–277.

[42] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Mathematical Software—ICMS* (Lecture Notes in Computer Science), vol. 6327. Springer, 2010, pp. 299–302.

[43] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, "STELLA: A domain-specific tool for structured grid methods in weather and climate models," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2015, pp. 41:1–41:12.

[44] 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 *Proc. 22nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*. New York, NY, USA: ACM, 2017, pp. 163–177.

[45] (2017). *OpenACC Compiler*. https://www.pgroup.com/doc/openacc_gs.pdf

[46] (2017). *NVCC*. [Online]. Available: http://www.docs.nvidia.com/cuda/cuda-compiler-driver-nvcc

[47] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*. Piscataway, NJ, USA: IEEE Press, Nov. 2008, pp. 31:1–31:11.

[48] (2017). *Intel C++ Compiler*. [Online]. Available: https://software.intel.com/en-us

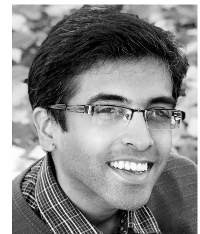
[49] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Feedback-Directed Runtime Optim. (CGO)*. Washington, DC, USA: IEEE Computer Society, 2004, p. 75.

[50] R. M. Stallman and G. DeveloperCommunity, *Using the GNU Compiler Collection: A GNU Manual for GCC Version 4.3.3*. Paramount, CA, USA: CreateSpace, 2009.

ABOUT THE AUTHORS

Prashant Singh Rawat received the B.Tech. degree in computer science from Mumbai University, Mumbai, India, in 2007, the M.Tech degree from the Indian Institute of Technology Bombay, Mumbai, in 2012, and the Ph.D. degree from The Ohio State University, Columbus, OH, USA, in 2018. His Ph.D. dissertation focused on optimizing bandwidth-bound and register-constrained stencil computations on GPUs.

His research interests lie in the domain of optimizing compilers, especially the aspects of data flow analysis, and optimization techniques for high-performance computing.



Mahesh Ravishankar received the Ph.D. degree from the Department of Computer Science and Engineering at The Ohio State University, Columbus, OH, USA.

He is currently a Senior Compiler Engineer at NVIDIA, Redmond, WA, USA. Previously, he was in the CUDA compiler team. His research interests include developing compiler tools tailored for specific application domains. He has worked on developing DSLs for image processing, as well as automatic distributed memory parallelization of codes from the scientific computing domain.

Miheer Vaidya is currently working toward the Ph.D. degree at the Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA, advised by Professor Sadayappan.

He is beginning his research in optimizing code generation for stencil computations. He is interested in compiler optimizations for parallel architectures. Before starting graduate school, he was a System Software Engineer (2013–2017) at NVIDIA Pune India. As a GPU backend compiler developer, his work involved performance analysis, developing and enhancing machine dependent and independent optimizations.

Vinod Grover received the M.S. degree in computer science from Syracuse University, Syracuse, NY, USA, in 1982, and the M.S. degree in physics from the Indian Institute of Technology, New Delhi, India, in 1979.

He is the Chief Architect for CUDA C++ and Director of Compilers at NVIDIA, Redmond, WA, USA. Over the course of last 10 years, he and his team helped define and build the CUDA C++ programming model and compilers for heterogenous programming. He is now helping build the next generation of compilers and tools for machine learning models such as deep learning and probabilistic programming. He is also interested in applications of program synthesis to this goal. Before coming to NVIDIA, he worked in various research, engineering, and management roles at Microsoft Research and at Sun Microsystems.

Aravind Sukumaran-Rajam received the M.E. degree from the Indian Institute of Science (IISc), New Delhi, India, and the Ph.D. degree from the University of Strasbourg (UDS), Strasbourg , France, in collaboration with INRIA and iCUBE.

He is a Senior Research Associate with Professor Sadayappan at the HPC Research Laboratory (HPCRL), The Ohio State University, Columbus, OH, USA. His research targets high-performance computing, machine learning, compilers, dynamic optimizations, DSLs, and algorithms for various targets such as multi/many cores, GPUs, and FPGAs.

Atanas (Nasko) Rountev is a Professor in the Department of Computer Science and Engineering at the Ohio State University, Columbus, OH, USA. His research interests are in static and dynamic program analysis, software for Android devices, software understanding and testing, and high-performance computing. He has served on the program committees of several conferences in the areas of software engineering and programming languages. He currently serves on the Editorial Board of the *ACM Transactions* on *Software Engineering and Methodology*.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Louis-Noël Pouchet is an Assistant Professor of Computer Science at Colorado State University, Fort Collins, CO, USA, with a joint appointment in the Electrical and Computer Engineering Department. He is working on pattern-specific languages and compilers for scientific computing, and has designed numerous approaches using optimizing compilation to effectively map applications to CPUs, GPUs, FPGAs, and System-on-Chips. His work spans a variety of domains including compiler optimization design especially in the polyhedral compilation framework, high-level synthesis for FPGAs and SoCs, and distributed computing. Previously, he has been a Visiting Assistant Professor (2012–2014) at the University of California Los Angeles, where he was a member of the NSF Center for Domain-Specific Computing, working on both software and hardware customization, and a Research Assistant Professor at the Ohio State University until 2016. He is the author of the PolyOpt and PoCC compilers, and of the PolyBench benchmarking suite.

P. (Saday) Sadayappan (Fellow, IEEE) received the B.Tech. degree from the Indian Institute of Technology Madras, Chennai, India, and the M.S. and Ph.D. degrees from Stony Brook University, Stony Brook, NY, USA.

He is a Professor of Computer Science and Engineering at The Ohio State University, Columbus, OH, USA. His research interests include compiler optimization for parallel and heterogeneous systems, domain/pattern-specific compiler optimization, and analysis/characterization of data movement complexity of algorithms.