Resource Conscious Reuse-Driven Tiling for GPUs

Prashant Singh Rawat, Changwan Hong
Computer Science and Engineering
The Ohio State University
{rawatp,hongc}@cse.ohio-state.edu

Mahesh Ravishankar, Vinod Grover
Nvidia Corporation
Redmond, Washington
{mravishankar,vgrover}@nvidia.com

Louis-Noël Pouchet, Atanas Rountev, P. Sadayappan
Computer Science and Engineering
The Ohio State University
{pouchet,rountev,saday}@cse.ohio-state.edu

ABSTRACT

Computations involving successive application of 3D stencil operators are widely used in many application domains, such as image processing, computational electromagnetics, seismic processing, and climate modeling. Enhancement of temporal and spatial locality via tiling is generally required in order to overcome performance bottlenecks due to limited bandwidth to global memory on GPUs. However, the low shared memory capacity on current GPU architectures makes effective tiling for 3D stencils very challenging – several previous domain-specific compilers for stencils have demonstrated very high performance for 2D stencils, but much lower performance on 3D stencils.

In this paper, we develop an effective resource-constraint-driven approach for automated GPU code generation for stencils. We present a fusion technique that judiciously fuses stencil computations to minimize data movement, while controlling computational redundancy and maximizing resource usage. The fusion model subsumes time tiling of iterated stencils, and can be easily adapted to different GPU architectures. We integrate the fusion model into a code generator that makes effective use of scarce shared memory and registers to achieve high performance. The effectiveness of the automated model-driven code generator is demonstrated through experimental results on a number of benchmarks, comparing against various previously developed GPU code generators.

Keywords

Stencil Computations; GPU; Fusion; Code Generation

1. INTRODUCTION

Stencil computations are central to numerous computational methods, ranging from image processing to the numerical solution of partial differential equations. Many previous efforts for compiler optimization of stencil computations have focused on time-iterated stencils involving repeated application of a stencil operator [23, 4, 5, 8, 9]. In contrast, multi-statement stencils and image processing pipelines apply a sequence of stencil operators on a set of input domains [16, 19]. Such stencil computations can be represented by a directed acyclic graph (DAG) in which nodes represent stencil operators and incoming edges represent inputs to these operators. These stencil pipelines are often bandwidth-bound.

Time-tiling is a key transformation to enhance temporal reuse in bandwidth-bound stencils. In a time-tiled execution, operations from several consecutive time steps of a stencil computation are combined to form a tile, which is executed atomically to exploit data reuse. Previous efforts towards optimizing 2D stencils on GPUs have demonstrated the effectiveness of time-tiling, along with the utilization of shared memory [4, 9]. However, tiling 3D stencils has proved to be a challenge for all previously reported code generators. The difficulty arises from the cubic versus quadratic increase in shared memory requirement for the 3D case, and the very limited (usually under 64 Kbytes) shared memory available per streaming multiprocessor (SM) on GPUs. Consequently, the use of overlapped tiles [9] with 3D stencils results in very rapid increase in redundant computations, even for very small values of the time-tile size.

In this paper, we present a GPU code generator that optimizes a DAG of stencil operators, making effective use of available shared memory and registers. This enables handling of multi-statement stencils and image processing pipelines, as well as time iterated stencils (as arising in multigrid smoothers, etc.) via explicit unrolling of a few time iterations. Thus, a single framework can be used to generate high-performance GPU code, both for time iterated stencils and multi-statement stencil pipelines.

A key issue in optimizing the execution of 3D stencils is the determination of tile shape/size and the amount of fusion across multiple stencils, while making effective use of a combination of shared memory and registers to enable a high degree of reuse. We use a number of strategies to overcome the difficulties faced by previous GPU code generators with 3D stencils:

- Instead of using symmetric 3D tiles, we use a sliding window along one spatial dimension and symmetric overlapped 2D tiling in the other two spatial dimensions to enable significantly greater data reuse than 3D overlapped tiling.
- We exploit the flexibility of associative reordering of stencil contributions to each data element, thereby reducing the demand on the very scarce shared memory via increased use of the more plentiful registers on modern GPUs.
- We develop a model-driven approach to determine which stencil operators to fuse, since excessive fusion leads to increased redundant computation, increased data traffic to/from global memory, and increase in overhead from register spilling.

The paper makes the following contributions:

- It develops a resource-directed model-driven approach to gen-

PACT '16, September 11-15, 2016, Haifa, Israel
© 2016 ACM. ISBN 978-1-4503-4121-9/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2967938.2967967

erate high-performance GPU code for stencil computations.

- It develops a practically effective stencil fusion algorithm for GPU code generation.
- It presents compiler algorithms for an automated code generator that takes a DAG of stencil computations as input, and generates GPU code which makes effective use of registers, shared memory, and streamed tile execution to achieve high warp occupancy.
- It presents a case study highlighting the factors that can constrain performance, and the impact of effective fusion on a resource-intensive stencil from a partial differential equation solver application.
- It evaluates a set of 3D benchmarks on Kepler and Maxwell devices, demonstrating significantly higher performance than other existing GPU code generators for stencils.

2. OVERVIEW OF APPROACH

2.1 DSL Input to Code Generator

Listing 1 shows a 7-point Jacobi stencil expressed in a domain specific language (DSL). The input language used by our code generator is similar to Forma [19], and is designed to provide a high-level description of the stencil computation. Each function prefixed by *stencil* describes the operations to execute at a point of the computation domain. Array accesses within the stencil function are described in terms of offsets to be used to access an array element. For example, $A[1, 0, -1]$ refers to access to element $A[i+1, j, k-1]$ when executing iteration point (i, j, k).

Even though we can write many operations within a stencil function, each individual operation must be written in a generic form with a view that a separate kernel call may be generated for it. The code generator decides which stencil operations to fuse, how to shift and align the offsets for array accesses when fusing operations with dependences, and how many output stencil functions the input stencil function is split into. To simplify the presentation, in the examples we omit the explicit specification of the domains (array extents) over which the stencils are applied.

The type and size of the arrays used in the computation are explicitly defined (line 10 of Listing 1). Arrays that represent the result of the computation are specified as *return*-ed (line 13). An array listed as *temporary* need not be explicitly stored in memory, freeing the compiler to either store it in temporary scratch memory (such as shared memory buffers on the GPU) or in registers. We note that such information could also be extracted from code in other languages such as C or Fortran, but the required code analyses are outside the scope of this paper.

2.2 Code Generation Challenges for 3D Stencils

For 3D stencils, existing GPU stencil code generators [4, 5, 9, 19, 24] suffer from limitations on one or more of the following issues: (a) effective management of limited GPU resources for tiled execution; (b) optimization of the degree of fusion across multiple stencils (or equivalently, the time-tile size) to achieve reduction in data movement to/from global memory; and (c) exploitation of associativity/commutativity properties of stencil operations to optimize resource usage. Next, we discuss how the approach presented in this paper addresses these issues.

a) Time Tiling and Computational Redundancy.

Time-tiling a stencil computation involves executing operations from multiple time steps of the computation atomically. For example, the stencil of Listing 1 computes two time steps (at lines 2 and

Listing 1: The stencil representation for two time steps of an order-1 7-point Jacobi computation

```
 1  stencil jacobi-def (A, B, C, a, b, c) {
 2   B[0,0,0] =                  a*(A[1,0,0]) +
 3    b*(A[0,-1,0]+A[0,0,-1]+A[0,0,0]+A[0,0,1]+A[0,1,0]) +
 4                               c*(A[-1,0,0]);
 5   C[0,0,0] =                  a*(B[1,0,0] +
 6    b*(B[0,-1,0]+B[0,0,-1]+B[0,0,0]+B[0,0,1]+B[0,1,0]) +
 7                               c*(B[-1,0,0]);
 8  }
 9  parameter L, M, N;
10  float A[L,M,N], B[L,M,N], C[L,M,N], a, b, c;
11  temporary B;
12  jacobi-def (A, B, C, a, b, c);
13  return C;
```

5) of a 7-point Jacobi computation. Tiling the *jacobi-def* stencil on a GPU requires the partitioning of the computational domain into smaller sub-domains, and assigning these sub-domains to thread blocks. However, simple rectangular partitioning of the iteration space into tiles is not feasible, since at time step t, the threads at the boundaries of adjacent tiles each need values at time step $t-1$ from each other. The overlapped boundary region is called the *ghost zone* or *halo region*. Overlapped tiling [9, 12] overcomes this problem of cyclic inter-tile dependences by having the thread blocks redundantly compute values at the halo region, thereby eliminating the mutual dependence between adjacent tiles.

For 2D stencils, the fractional volume of redundant operations with overlapped tiling can be effectively controlled [20]. Let us consider a $B \times B$ thread block size, and a time-tile size of 4. If the intermediate results are stored in shared memory, the amount of memory required is $O(4B^2)$. The total amount of shared memory available on GPUs is typically 48 Kbytes. For time-tile size of 4, it is possible to use a block size of 32×32 while keeping all the intermediates in shared memory. For an order-1 stencil, an overlapped tile will involve reading in a 32×32 block from global memory, and computation of intermediate result blocks of sizes 30×30, 28×28, 26×26, and 24×24, respectively for the 4 time steps in the tile. This represents a total work of $30^2+28^2+26^2+24^2 = 2936$ stencil operations, where the non-redundant work is $4\times24^2 = 2304$ stencil operations, resulting in a fractional redundant work volume of $632/2304 = 27\%$.

For a 3D stencil, a thread block size of $10\times10\times10$ can be used with all the intermediates in shared memory. The work performed for 4 time steps with overlapped tiling would be $8\times8\times8$, $6\times6\times6$, $4\times4\times4$, and $2\times2\times2$, respectively. The total work is $8^3+6^3+4^3+2^3 = 800$ stencil operations, of which only $4\times2^3 = 32$ operations are non-redundant, with a fractional overhead of $768/32 = 2400\%$. Even with a time-tile size of 2, the total work done is $8^3+6^3 = 728$ stencil operations, of which $2\times6^3 = 432$ are non-redundant, resulting in a high fractional overhead volume of $294/432 = 68\%$.

This indicates that when using overlapped tiling, going from 2D stencils to 3D stencils significantly increases the computational redundancy. The resulting performance degradation makes overlapped 3D tiling ineffective for 3D stencils.

To overcome this problem, we perform overlapped tiling along two spatial dimensions of the 3D domain, and launch 2D thread blocks that *stream* through the unpartitioned third dimension at each time step [15, 20]. This scheme, referred to as *sliding window* time-tiling, provides full reuse along the unpartitioned dimension. The size of the sliding window at time step t equals the number of input planes read to compute *one* output plane at time step $t+1$. For a 3D order-k single input/output Jacobi stencil, the sliding window size will be $2k+1$ at each time step. For example, since Listing 1 is an order-1 ($k = 1$) stencil, the computation at lines 2–4 needs to

Listing 2: Fusion profitability for multi-statement stencils

```
stencil fusion (A, B, C, D, E, d) {
  s1 : B[0,0,0] = A[1,0,0]+A[0,0,0]+A[-1,0,0];
  s2 : C[0,0,0] = A[1,0,0]+A[0,0,0]+A[-1,0,0]+B[-1,0,0];
  s3 : d += A[0,0,0];
  s4 : D[0,0,0] += E[0,0,0] - E[0,-1,0];
}
parameter L, M, N;
float A[L,M,N], B[L,M,N], C[L,M,N], D[L,M,N], E[L,M,N];
float d;
temporary B;
fusion (A, B, C, D, E, d);
return C, D, d;
```

read only three input planes ($A[1,*,*]$, $A[0,*,*]$, and $A[-1,*,*]$), to compute output plane $B[0,*,*]$ using a 2D grid/thread configuration that spans the extent of the inner dimensions of B. If the planes of A are stored in shared memory, then after computing $B[0,*,*]$, to next compute $B[1,*,*]$ would require loading of just one additional plane of A (i.e., $A[2,*,*]$).

The amount of shared memory also constraints the extent of time-tiling. Consider the case where the shared memory available per SM is 48 Kbytes. If each of the $2k+1$ input planes read by a block is mapped to a distinct shared memory buffer, then for single precision computation with $k = 1$ and a maximum of 2048 threads per SM (as is the case for the Nvidia Kepler and Maxwell GPUs used for experiments in this paper), the maximum number of shared memory buffers per thread block is ≤ 6 (i.e., 48 Kbyte/(4×2048)). Since each time step requires 3 shared memory buffers, the time-tile size is constrained to 2 time steps. For $k = 2$, we can only perform spatial tiling with a 32×32 thread block. For higher values of k, we must either perform global memory read transactions at each time step, or sacrifice occupancy (and hence thread-level parallelism) to perform time-tiling. Our modeling accounts for such resource constraints in order to generate appropriate code for different GPU devices.

b) Fusion of Stencil Operators.

GPUs have a significant gap between their compute and memory throughput. Many stencil applications are bandwidth-bound, making it extremely difficult to achieve near-peak computational performance. Performance for such stencils can be improved by reducing global memory transactions via caching or use of shared memory and registers. Time-tiling and fusion are two key transformations used to enhance data locality. However, as shown later in Sections 3 and 4, an indiscriminate attempt to time-tile or fuse stencil operators can be counter-productive.

Consider the computation in Listing 2. If s_1 is fused with s_2, array B does not have to be written to global memory since it is marked *temporary*. The output of s_1 can be stored in a register or shared memory buffer before being read by s_2. Thus, global read and write transactions for an entire array can be eliminated after fusion. However, since there is a RAW dependence between s_1 and s_2 due to B, atomic execution of a tile will require computation of a halo region. Fusing s_3 with either s_1 or s_2 will also lead to savings in global read transactions for array A. The savings in global memory transactions are less in this case, but no halo region needs to be computed. Since none of the arrays from s_4 appear in any other statement, fusing s_4 with any other statement will not enhance data reuse.

While fusion can significantly reduce the amount of data moved from/to global memory, it increases demands on shared memory and/or registers. For complex stencils that arise in many applications, the number of possible fusion choices can be large. In this paper, we develop a *fusion profitability* metric and a greedy heuristic to select good fusion configurations that satisfy all resource constraints.

Listing 3: Leveraging associativity to rewrite Listing 1 stencil

```
 1  stencil jacobi (A, B, C, a, b, c) {
 2  s1  b1: B[1,0,0]   = c*(A[0,0,0]);
 3      b2: B[0,0,0]  += b*(A[0,-1,0] + A[0,0,-1] + A[0,0,0]
 4                          + A[0, 0,1] + A[0,1, 0]);
 5      b3: B[-1,0,0] += a*(A[0,0,0]);
 6  s2  b4: C[1,0,0]   = c*(B[0,0,0]);
 7      b5: C[0,0,0]  += b*(B[0,-1,0] + B[0,0,-1] + B[0,0,0]
 8                          + B[0, 0,1] + B[0,1, 0]);
 9      b6: C[-1,0,0] += a*(B[0,0,0]);
10  }
11  parameter L, M, N;
12  float A[L,M,N], B[L,M,N], C[L,M,N], a, b, c;
13  temporary B;
14  jacobi (A, B, C, a, b, c);
15  return C;
```

c) Associative Reordering for Resource Management.

As noted above, complex stencil computations can place excessive demands on shared memory. An approach that has previously been developed to reduce register pressure with high-order stencils, in the context of multi-core processors, is to perform reordering of operations by exploiting the associativity of additive contributions in stencil computations [21]. We use associative reordering to balance resource usage by trading off register use for shared memory. For example, the additive contributions to the statement at lines 2–4 in Listing 1 can be reordered without changing the semantics of the computation. Although floating-point additions are not strictly associative, it is generally acceptable to perform associative reordering of accumulations in stencil computations. DSLs such as Forma [19] and Halide [18] currently do not recognize and exploit associative operations. Our approach uses operator associativity to reduce the shared memory demand by a simple restructuring of the computation. Listing 3 shows the resulting DSL code after restructuring the stencil from Listing 1, which is done automatically by the DSL compiler.

Statement s_1 (s_2) at lines 2–4 (lines 5–7) of Listing 1 is split into sub-statements b_1 through b_3 (b_4 through b_6) at lines 2–5 (lines 6–9) in Listing 3. For the same plane of the input array, these sub-statements update different points of the output domain, i.e., the plane $A[0,*,*]$ is read once to update the values of $B[-1,*,*]$, $B[0,*,*]$, and $B[1,*,*]$. Since the computations per plane are executed in parallel by each thread in a kernel, neighboring threads will access the same element of array A. To reduce the number of global memory accesses, the plane of array A can be loaded into shared memory, with each thread loading a single element of the plane. The threads can then access all the values of $A[0,*,*]$ needed to perform the computation specified in statement s_1 from shared memory.

After executing s_1, the plane corresponding to $B[-1,*,*]$ has received all the updates needed from statement s_1 and can be used to update the value of array C in statement s_2. The plane corresponding to $B[-1,*,*]$ in s_1 is used as the generic plane $B[0,*,*]$ in statement s_2. Again, neighboring threads of the kernel access common elements of the plane $B[0,*,*]$ in statement s_2. It is therefore beneficial to store this plane in shared memory as well for the same reason as above.

Note that the planes corresponding to the writes represented by $B[0,*,*]$, and $B[1,*,*]$ in statement s_1 are not needed in statement s_2 yet. These can be stored in registers. As the sliding window moves across the planes of array A, the values in these registers can be updated. With the execution model described above, only two shared memory buffers are needed, one for the plane corresponding to $A[0, *, *]$ in statement s_1, and the other for the plane corresponding to $B[-1, *, *]$ in statement s_1 (which is the same as the plane corresponding to $B[0, *, *]$ in statement s_2).

In general, for a single statement that executes an order-k stencil, instead of $2k + 1$ shared memory buffers per time step, the optimized kernel code uses $2k$ registers and 1 shared memory buffer per time step with reassociation. The advantage of using registers for storage is twofold: (1) in current GPU architectures, registers are more plentiful than shared memory, and (2) the access latency of registers is lower than that of shared memory, and so using them as cache speeds up the computation. We will use the *jacobi* stencil of Listing 3 as a running example throughout the rest of the paper.

3. RESOURCE-CONSTRAINED FUSION

To maximize performance with complex stencil DAGs, we must judiciously use registers and the memory hierarchy of GPUs to reduce expensive data movement. Fusion is a classical loop transformation that can enhance data locality, thereby minimizing data movement. The key idea is to fuse the nodes in the DAG with common predecessors (i.e., stencil operators that access common input arrays), or a chain of nodes in the DAG (i.e., stencil operators that have producer-consumer dependences) so that temporal locality and the per-load data reuse is maximized.

Without violating dependences, different nodes in a DAG can be fused to get different valid schedules. The performance of each schedule can vary depending on the profitability of fusion. The space of all valid execution schedules can be vast. To avoid exploring the entire space to find the best fusion schedule, we use a greedy algorithm which fuses nodes in the DAG with the intent to minimize a chosen objective function. From the initial stencil DAG $D_g = (V, E)$, our aim is to create a resultant *fused* directed 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 of nodes forms clusters of "macro-nodes' with no dependence chain leaving and re-entering any macro-node. Just as convex-shaped tiles of an iteration space can execute atomically, the fused macro-nodes computed by the fusion algorithm ensure that no dependences are violated upon their atomic execution. The convex partitioning therefore guarantees that D_f is acyclic. Only nodes whose fusion is deemed profitable by the objective function are combined together in v_f. A single GPU kernel is generated for each node in v_f. No global memory transactions are needed for domains whose definition and use do not cross partition boundaries.

Overview of the Greedy Fusion Algorithm.

The rest of this section details a flexible greedy algorithm for fusing nodes in a directed stencil graph, designed for easy incorporation of multiple objective metrics. The fusion algorithm consists of the following steps:

Step 1: Compute the resource usage of each individual statement in the stencil DAG (Section 3.1).

Step 2: Identify pairs of statements that can be legally fused, as well as some profitability metrics that quantify their fusion benefits (Section 3.2).

Step 3: For each statement pair, compute the resource usage of the fused node based on the resource usage of individual statements obtained in Step 1 (Section 3.3).

Step 4: From the resource usage of the fused node, compute the profitability metrics identified in Step 2 (Section 3.4).

Step 5: Define a custom sort to order the profitability metrics of statement pairs, and choose to fuse the most profitable pair that satisfies additional hardware-imposed constraints (discussed in Section 3.6 and Section 4). Update the stencil DAG and the dependence graph, and repeat again from Step 1, until no more statement pairs can be fused (Section 3.5).

Once the fusion algorithm creates the fused stencil DAG D_f, the code generator described in Section 5 generates a CUDA kernel for each node of D_f.

3.1 Modeling Resource Requirements

Efficient management of GPU resources is crucial for performance. In the reordering optimization of Listing 3, each statement uses some registers and shared memory buffers to cache some input and output values. Since fusion of different statements may result in an increase in the resource requirement, it is important to quantify the hardware resources used by each statement. We use the access pattern for each array in the statement for this estimation. The per-statement resource estimates are then used to estimate the resource requirements after fusing any two statements (Section 3.2). Without loss of generality, let us assume that we stream through the outermost dimension of a 3D domain. To determine the resource requirement of a stencil statement that is order-k along the streaming dimension, we need to reason about the storage type of the $2k + 1$ accessed planes. We follow a simple chain of reasoning:

(a) If the computation of an output element at (z_0, y_0, x_0) only uses a single input element at (z_r, y_0, x_0) from an input plane z_r, then that input element is stored in an explicit register (i.e., storage type of z_r is register). Otherwise, z_r is cached in shared memory

(b) An output element is written to an explicit register (i.e., its storage type is register) if it is an accumulation statement, or it is not the last assignment to a non-temporary array. For the last assignment to any non-temporary array, the storage is type is global memory.

Here, an *explicit* register refers to a scalar temporary variable created to hold an array element instead of using shared memory. It is expected that the compiler will place such scalars in registers. We distinguish these from other *implicit* registers that are used internally by NVCC, to hold elements of arrays and intermediate results in computing expressions. We note that these terms are used just for the purpose of explanation; in the final code, the explicit registers just appear as thread scalars.

From rule (a), we can infer that statement s_1 of Listing 3 should use a shared memory buffer to store the input plane $A[0, *, *]$. This is because five different values are read from that plane and contribute to different output points. Similarly, from rule (b), it follows that statement s_1 should use three explicit registers to store the output values written to $B[1, 0, 0]$, $B[0, 0, 0]$ and $B[-1, 0, 0]$.

We represent the resource requirement for each statement by a 3-tuple $(N_{reg}, N_{shm}, M_{acc \to res})$, where N_{reg}, N_{shm} are the number of explicit registers and shared memory buffers used by the statement, and $M_{acc \to res}$ is a resource map from each accessed plane to a handle that represents the storage used for that plane. For example, the 3-tuple for s_1 is $(3, 1, M_1)$, where $M_1 = \{B[-1, 0, 0] \to b_m, B[0, 0, 0] \to b_c, B[1, 0, 0] \to b_p, A[0, *, *] \to shm_a\}$; the 3-tuple for s_2 is $(3, 1, M_2)$, where $M_2 = \{C[-1, 0, 0] \to c_m, C[0, 0, 0] \to c_c, C[1, 0, 0] \to c_p, B[0, *, *] \to shm_b\}$.

3.2 Modeling Fusion Profitability

An objective function is used to choose among multiple valid fusion choices. The impact of fusion on performance is governed by an intricate interplay of many factors, some of which are hardware dependent. Fusion of two nodes can decrease data movement, but may lead to an increase in the shared memory/register requirement. Beyond a certain point, any further increase in resources can result in reduced occupancy and/or excessive register spills, both negating the benefits of fusion. Let us revisit the computation of Listing

2. Clearly, data movement can be significantly reduced by fusion across the first three stencil statements. The data movement savings and total resource requirement after fusing any pair of statements will depend on the data dependences between the statements. Assuming that all the arrays are of the same size, the data movement cost is modeled simply as the number of arrays read from or written into during the computation. Consider the following fusion choices for the code in Listing 2.

- Fusing s_1 and s_2: the data movement cost reduces by 3 (1 load saving for A, and 1 store + 1 load saving for B), but the resources used increase by 2 over s_1 (a register for $B[-1,0,0]$ and $C[0,0,0]$ each).
- Fusing s_1 and s_3: the data movement cost reduces by 1 (1 load saving for A), while the resource requirement increases by 1 over s_1 (a register for d).

If the fusion of any two statements is guaranteed not to exceed resource constraints, then we would prefer to fuse statements s_1 and s_2 if minimizing data movement is the objective, but prefer to fuse s_1 and s_3 if minimizing combined use of resources is the objective. Since there are multiple resources to be managed, and many complex interacting factors that affect overall performance, we do not attempt to develop a single aggregated objective function to guide the choice of fusion. Instead, we use a multi-dimensional vector, whose components quantitatively reflect various considerations of importance, and a lexicographic ordering among the vectors as a basis for choice among valid fusion configurations. Such an approach allows great flexibility in experimenting with different permutations of the "objective-vector" to change the relative priority of the different components in choosing between alternatives.

Components of the Fusion Profitability Metric

The greedy fusion algorithm begins by computing the 3-tuple defined in Section 3.1 for each stencil statement to model its resource requirements. The dependences in the stencil DAG are captured by the dependence graph $G_{dep} = (V, E_{dep})$. From G_{dep}, we compute a transitive dependence graph $G_{trans} = (V, E_{trans})$, such that

$$\begin{aligned} s_i \rightarrow s_j \in E_{trans} \text{ iff } \{\ & \exists s_k \mid s_k \in V \wedge s_k \neq s_i \wedge s_k \neq s_j \wedge \\ & \text{there exists a path from } s_i \text{ to } s_k \text{ (denoted as } s_i \rightsquigarrow s_k\text{)} \wedge \\ & \text{there exists a path from } s_k \text{ to } s_j \text{ i.e., } s_k \rightsquigarrow s_j\ \} \end{aligned} \quad (1)$$

We use G_{trans} to ensure the validity of fusion, i.e., that all partitions generated by fusion are convex. If $s_i \rightarrow s_j \in E_{trans}$, then fusing s_i and s_j will violate convexity unless all the nodes along any path $s_i \rightsquigarrow s_j$ are also included in the fused macro-node. The definition above does not preclude fusion of a statement with its immediate predecessor as long as there are no intervening statements along all paths from s_i to s_j.

Since the benefits from fusing different statement pairs may vary, we construct a *fusion profitability* metric for each distinct stencil statement pair (s_i, s_j). This metric is a 7-tuple $(D_m, S_{reg}, S_{shm}, E_{reg}, E_{shm}, T_{reg}, T_{shm})$, where

- D_m represents the savings in data movement after fusing s_i and s_j
- S_{reg} represents the savings in explicit registers, modeled as the number of common registers between s_i or s_j
- S_{shm} represents the savings in shared memory, modeled as the number of common shared memory buffers between s_i and s_j
- E_{reg} represents the excess of explicit registers, modeled as the minimum extra registers required in the fused node over the amount used by s_i or s_j before fusion.
- E_{shm} represents the excess of shared memory, modeled as the minimum extra shared memory buffers required in the fused node over the amount used by s_i or s_j before fusion.
- T_{reg} represents the total number of explicit registers in the fused node
- T_{shm} represents the total number of shared memory buffers used in the fused node

3.3 Computing the Resource Map

In order to assess the impact of fusing two statements, we need to find the resources that would be needed to execute the fused node. Let M_i and M_j be the maps from accessed array planes to GPU resources ($M_{acc \rightarrow res}$) for the fusion candidates s_i and s_j, respectively. Let M_{fused} be the resource map for the fused node. This map is computed as shown in Algorithm 1. If there is no dependence between s_i and s_j, the resource map of the fused node is a union of the resource map of the individual nodes, where the rules of union are as defined in this section.

If there is a dependence between the two statements, we need first compute the *schedule_id* of the target statement. This id represents the statement's execution schedule. Two statements have the same execution schedule if their instances can be perfectly interleaved. The array accesses in the target statement are shifted by its *schedule_id* along the streaming dimension. The shift is to ensure that the dependences are satisfied within the fused node. For example, in Listing 3, s_2 can read the values of array B at a plane i only after s_1 had read the plane $i+1$ of array A and updated the values of plane i of B. Therefore, the *schedule_id* of s_2 has to be one less than the *schedule_id* of s_1. Function *min_plane_offset()* at line 5 iterates over all the sub-statements of the source statement, and returns the minimum LHS access offset along the streaming dimension. If we stream along the outermost dimension, then the value returned by *min_plane_offset()* for s_1 of Listing 3 will be $min\ (1, 0, -1) = -1$. This will be the *schedule_id* for the target statement.

While the *schedule_id* is needed for the correctness of the generated code, it also affects the mapping of resources in the fused node (lines 6–27 of Algorithm 1). Once all array accesses in the target statement have been offset by *schedule_id*, there might a non-empty intersection between the planes accessed by statement s_i and s_j. For example, for s_1 in Listing 3, $B[-1,*,*]$ is mapped to registers (Section 3.1). Since the *schedule_id* for s_2 is -1, $B[0,*,*]$ in s_2 will be shifted to $B[-1,*,*]$ in the fused node. Since this plane is mapped to shared memory in M_2, this plane will be mapped to shared memory in the fused node as well (line 14 of Algorithm 1). There might also be cases where the in-plane accesses of the intersecting plane in statement s_i are different from the in-plane accesses in statement s_j. In these cases as well, the plane is mapped to shared memory in the fused node, in keeping with the approach described in Section 2-c (lines 16–21 of Algorithm 1).

For Listing 3, the fused resource map computed by Algorithm 1 would be $M_{fused} = \{C[-2,0,0]{\rightarrow}c_m, C[-1,0,0]{\rightarrow}c_c, C[0,0,0]{\rightarrow}c_p, B[-1,*,*]{\rightarrow}shm_b\ B[0,0,0]{\rightarrow}b_c, B[1,0,0]{\rightarrow}b_p, A[0,*,*]{\rightarrow}shm_a\}$.

3.4 Computing the Profitability Metric

Once the resource map of the fused node has been computed using Algorithm 1, we can quantify the profitability of fusing statements s_i and s_j using the 7-tuple described in Section 3.2. We populate the 7-tuple only if $s_i \rightarrow s_j \notin E_{trans}$, to ensure that we do not violate any dependencies. For every $s_i \rightarrow s_j \in E_{raw}$, there is an implicit reduction in data movement for the array carrying the dependence. To capture this, D_m is incremented by 2 for each RAW dependence between the two statements that are carried by arrays marked as *temporary*, once each for the write in s_i and read in s_j. If the RAW dependence is carried by an array that is not

Algorithm 1: Computing the resource map for a fused node

```
Input  : $s_i, s_j$: stencil statements
Input  : $M_1$, $M_2$: Resource map for $s_1$ and $s_2$ respectively
Output: $M_{fused}$: Resource map for the fused node
1  if $\neg$has_dependence ($s_i, s_j$) then
2      $M_{fused} \leftarrow M_1 \cup M_2$;
3  end
4  else
5      schedule_id = min_plane_offset ($s_i$);
6      $M_{fused} \leftarrow M_1$;
7      for each entry $m \in M_2$ do
8          curr_access_plane ← get_accessed_plane (m);
9          new_access_plane ← curr_access_plane + schedule_id;
10         accessed_array ← get_accessed_array (m);
11         key ← get_map_key (accessed_array, new_access_plane);
12         if key $\in M_{fused}$ then
13             if mapped_to_shared_mem ($M_2$, key) then
14                 $M_{fused}$[key] ← new_shared_mem_resources ();
15             end
16             else if mapped_to_registers ($M_{fused}$, key) then
17                 fused_offsets ← in_plane_accesses ($s_i$,
                   new_access_plane, accessed_array);
18                 curr_offsets ← in_plane_accesses ($s_j$,
                   curr_access_plane, accessed_array);
19                 if fused_offsets ≠ current_offsets then
20                     $M_{fused}$[key] ← new_shared_mem_resources ();
21                 end
22             end
23         end
24         else
25             $M_{fused}$.insert (m);
26         end
27     end
28 end
```

Algorithm 2: Creating the objective function for fusion

```
Input  : IN: input DAG with $n$ statements
Output: OUT: A sorted list of 7-tuple, $L_{tuple}$
1  $L_{tuple} \leftarrow \emptyset$;
2  compute_resource_3-tuple ();
3  $\{G_{raw}, G_{war}, G_{waw}\}$ ← create_dependence_graphs (IN);
4  $G_{dep} \leftarrow G_{raw} \cup G_{war} \cup G_{waw}$;
5  $G_{trans}$ ← transitive_closure ($G_{dep}$) $\backslash G_{dep}$;
6  for each distinct stmt pair $(i, j)$, $i, j \le n$ do
7      $D_m \leftarrow 0$;
8      if ($s_i \rightarrow s_j \notin G_{trans}$) then
9          $M_i$ ← resource_usage ($s_i$);
10         $M_j$ ← resource_usage ($s_j$);
           // Call Algorithm 1
11         $M_{fused}$ ← fused_resource_map ($s_i, s_j, M_i, M_j$);
12         $D_m$ += 2*RAW_dependence_count($s_i, s_j$,
           TEMPORARY_ARRAYS);
13         $D_m$ += RAW_dependence_count ($s_i, s_j$,
           NON_TEMPORARY_ARRAYS);
14         $D_m$ += WAW_dependence_count ($s_i, s_j$);
15         $D_m$ += RAR_dependence_count ($s_i, s_j$);
16         $S_{reg}$ ← num_registers ($M_i$) + num_registers ($M_j$) -
           num_registers ($M_{fused}$);
17         $S_{shm}$ ← num_shared ($M_i$) + num_shared ($M_j$) - num_shared
           ($M_{fused}$);
18         $\{E_{shm}, E_{reg}, T_{shm}, T_{reg}\}$ ← compute ($M_i, M_j, S_{shm}, S_{reg}$);
19         $L_{tuple}$.append (($D_m, S_{reg}, S_{shm}, E_{reg}, E_{shm}, T_{reg}, T_{shm}$));
20     end
21 end
22 sort ($L_{tuple}$, minimize_data_movement);
```

marked *temporary*, D_m is incremented by 1 since we do not have to reload the data written to global memory. There is no explicit data movement saving for arrays carrying WAR dependence. If there is WAW dependence between s_i and s_j but no RAW dependence, D_m is incremented by 1 for each WAW dependence to capture the reduction in multiple writes to the same location in global memory. The savings due to RAR dependences can be computed by finding the common arrays accessed among s_i and s_j, and incrementing D_m accordingly.

The other tuple items are computed as follows:

- $S_{reg} = num\_registers(M_i) + num\_registers(M_j) - num\_registers(M_{fused})$
- $S_{shm} = num\_shared(M_i) + num\_shared(M_j) - num\_shared(M_{fused})$
- $E_{reg} = min(num\_registers(M_i), num\_registers(M_j)) - S_{reg}$
- $E_{shm} = min(num\_shared(M_i), num\_shared(M_j)) - S_{shm}$
- $T_{reg} = num\_registers(M_i) + num\_registers(M_j) - S_{reg}$
- $T_{shm} = num\_shared(M_i) + num\_shared(M_j) - S_{shm}$

To illustrate the construction of the profitability metric, we construct the 7-tuple for statements s_1 and s_2 of Listing 3. The 3-tuples for s_1 and s_2 were computed in Section 3.1. Since there is a true dependence from s_1 to s_2, $D_m = 2$. The number of registers used is 3 for M_1, 3 for M_2, and 5 for M_{fused}. Therefore, $S_{reg} = 1$. The number of shared buffers used is 1 for M_1, 1 for M_2, and 2 for M_{fused}. Therefore, $S_{shm} = 0$. Since s_1 requires 3 registers for B and 1 shared memory buffer for A, $E_{reg} = 3 - 1 = 2$ and $E_{shm} = 1 - 0 = 1$. Post fusion, $T_{reg} = 3 + 3 - 1 = 5$, and $T_{shm} = 1 + 1 = 2$. The 7-tuple for s_1 and s_2 is therefore $\{2, 1, 0, 2, 1, 5, 2\}$.

The process described above is codified in Algorithm 2 for computation of the profitability metric for each statement pair.

3.5 Constructing the Objective Function

Once the 7-tuple has been computed for all statement pairs, we need a method to order them based on their fusion profitability. To do so, we define a custom-sort relation $\prec$ as a total order over the list of tuples (L_{tuple}). We first define a set of sorting rules that can be applied to order two 7-tuples $c_i, c_j \in L_{tuple}$.

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

where rule k is only used to ensure that a total order can be found in case of tuples with strictly identical metrics.

The order in which the sorting rules are applied to break the tie depends on the minimization objective. If the objective is to minimize the data movement, $\prec$ follows the rule sequence $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i \rightarrow j \rightarrow k$, where a is the primary sorting rule, and the ties are settled by following the remaining rules in the sequence. If the objective is to minimize the total resource requirement, $\prec$ follows the rule sequence $b \rightarrow c \rightarrow d \rightarrow a \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i \rightarrow j \rightarrow k$, where b is the primary sorting rule, and the ties are settled by following the remaining rules. A high priority to minimizing explicit register count (b and c) in both the cases is to ensure that the fused kernel does not generate register spills upon compilation. Line 22 of Algorithm 2 sorts all the 7-tuples with an objective to minimize data movement.

Once sorted, the topmost tuple of L_{tuple} represents the most profitable candidate for fusion. However, the constraints described so far do not account for constraints imposed by hardware, such as the limits on shared memory and registers available per SM, or the trade-off between redundant computation and reduction in global memory accesses due to the use of overlapped tiling. These two constraints are treated differently than the other component metrics in the L_{tuple}. Instead of just being components in a multi-component profitability metric, they are treated as filters. We eliminate any candidate fusion choices that violate either hardware constraints (Section 3.6) or generate too much redundant computation (Section 4). Henceforth, we collectively refer to the hardware constraints and redundancy constraints as *fusibility constraints*. The highest tuple in the list that satisfies the fusibility constraints is chosen, and the nodes corresponding to it are fused. The stencil DAG is updated to include the fused node, and the original statements are removed after updating the dependence graph appropriately. Algorithm 2 is reapplied to this new DAG. This is repeated till no further fusion is feasible.

3.6 Imposing Hardware Constraints

For each node in the stencil DAG, a single kernel is generated to perform the computation represented by that node. We achieve full utilization of the GPU device when the hardware-defined maximum number of concurrent threads supported per SM, say Th_{SM}, are all active. To minimize the volume of redundant computation, we try to maximize the block size on the device, B_{max}, such that the threads are maximally utilized (i.e., maximize B_{max} such that $Th_{SM}\%B_{max}$ is minimum, ideally 0). The number of blocks per SM, say N_B, would be $\lfloor Th_{SM}/B_{max}\rfloor$.

The fusion algorithm described in Algorithm 2 has so far not accounted for the constraints of shared memory available per SM on the device (K_{max}), as well as the number of registers available for each thread (R_{max}). To do so, the sorted list generated at the end of Algorithm 2 is traversed from top to bottom to select the highest tuple that also satisfies the criteria described in the rest of this section.

The amount of shared memory used by each node can be estimated from the resource map ($M_{acc\to res}$) associated with each node, the size of the thread block used, and the data type of each buffer mapped to shared memory. The tuples in the list where the fused node would require more than the amount of shared memory available per block, i.e., K_{max}/N_B, are ignored.

While the register files on modern GPUs are quite large, a large number of registers per thread can adversely affect the occupancy achieved. We define R_{opt} as the number of registers available to each thread for maximum occupancy. R_{opt} can be computed by dividing the register file size by Th_{sm} ($R_{opt} = \lfloor 65536/2048\rfloor = 32$ for Kepler device). The occupancy decreases from usage of R_{opt} to R_{max}. High occupancy is essential in order to tolerate the access latencies to global memory. To maintain a sufficiently high occupancy, we ignore tuples in L_{tuple} for which the number of explicit registers in the fused node is higher than some threshold. Our current implementation uses a threshold of 12.

It is difficult to model the use of implicit registers as it varies with the compiler version and the optimization level specified during compilation. However, the number of implicit registers used by NVCC usually increases with an increase in the degree of fusion. Our fusion algorithm handles implicit register pressure by ignoring tuples in L_{tuple} which result in register spills upon compilation. The amount of register spills can be determined by using *"-Xptxas -v"* flag during compilation. Setting the threshold of register spills to 0 might seem too conservative. However, determining a perfect balance between spills, occupancy, and performance is beyond the scope of our framework. We currently allow the user to set the threshold on number of spills that can be tolerated.

3.7 Controlling recompilation

Since we need to compile the code generated for each fusion candidate to determine register usage/spills, excessive recompilation can add significant overhead to the code generation time for complex stencils. However, this overhead is mitigated in part due to the choice of a greedy algorithm over one that explores all valid schedules. The number of recompilations is controlled by the following choices:

- Fusion always results in an increase in the resources used by the fused cluster. We only recompile the code of the fused cluster when the fusibility constraints are not already violated by the constraints on shared memory, explicit registers, and redundant computation.
- If fusing any two nodes results in register spills, then we rule out, without recompilation, fusion of any clusters involving either of these two nodes. For example, if fusing nodes A and B results in spills, then we will not consider fusing clusters such as $\{A, C\}$, $\{B, D\}$, etc.
- We provide the user an option to specify a recompilation-bypass size, and only recompile for clusters exceeding that size. For example, if the recompilation-bypass size is 2, then we only recompile when the fusion results in a cluster of size ≥ 3.

With these choices, we prune away a large number of fusion candidates requiring recompilation, thus controlling the total code generation time.

4. MODELING COMPUTATION AND COMMUNICATION REDUNDANCY

We use overlapped tiling to achieve concurrency of tiled execution. Overlapped tiling trades off redundant computation for reduced communication/synchronization overheads. For convenience of description, we make the following assumptions:

- All the stencil operators are order-k, operating on a single N^3 input domain without boundary conditions
- Overlapped tiling applied to any dimension of the input domain tiles it in chunks of size B
- Applying the fusion strategy of Section 3.1 results in the fusion of T stencil operators, where operator O_p has a RAW dependency on operator O_{p-1}

Traditional 3D overlapped tiling partitions the input domain into blocks of size B^3. A stencil operator at fusion depth t will compute an output block of size $(B - 2tk)^3$. Since the computable output domain is of size $(N - 2Tk)^3$, the total number of thread blocks launched must be $\left(\frac{N-2Tk}{B-2Tk}\right)^3$. If we use sliding window tiling along one dimension and perform overlapped tiling along the other two dimensions, the per-block computation increases, but the total number of blocks launched reduces to $\left(\frac{N-2Tk}{B-2Tk}\right)^2$. Table 1 lists the per-block data transfer and computation volume for both the methods, assuming all global memory transactions are perfectly aligned. Multiplying them by the number of blocks launched gives us the total data transfers and computation volume.

We note that with both forms of overlapped tiling, the fractional redundant computation overhead increases with the time tile size T and stencil order k, and decreases as the block size B is increased. However, the demand on shared memory and registers grows as $O(B^3)$ for 3D overlapped tiling, but only as $O(B^2)$ for sliding-window overlapped tiling. As was illustrated by the exam-

Scheme	Load Inst.	Load Transactions	Store Inst.	Store Transactions	Total Computation	Redundant computation
Traditional 3D Tiling	B^3	$B^2\lceil \frac{B}{W} \rceil$	$(B-2Tk)^3$	$(B-2Tk)^2\lceil \frac{B-2Tk}{W} \rceil$	$\sum_{i=1}^{T}(B-2ik)^3$	$\sum_{i=1}^{T-1}((B-2ik)^3-(B-2Tk)^3)$
Sliding Window with 2D Tiling	NB^2	$NB\lceil \frac{B}{W} \rceil$	$(N-2Tk)(B-2Tk)^2$	$(N-2Tk)(B-2Tk)\lceil \frac{B-2Tk}{W} \rceil$	$\sum_{i=1}^{T}(N-2ik)(B-2ik)^2$	$\sum_{i=1}^{T-1}((N-2ik)(B-2ik)^2 - (N-2Tk)(B-2Tk)^2)$

Table 1: Per-block data transfers and computational volume for different overlapped tiling schemes. W is the warp size.

ple in Section 2, the limited size of shared-memory per SM (under 64 Kbytes) makes it infeasible to use block sizes much larger than 10 for 3D overlapped tiling, while a block size of around 32 is feasible for sliding-window overlapped tiling. Even for k=1, the redundant computation overhead explodes for 3D overlapped tiling, even for fusion over 3 time steps, making it completely ineffective for 3D stencils for the hardware parameters of current GPUs. In contrast, the redundant computation overhead for sliding-window overlapped tiling is much lower, enabling useful fusion over multiple statements or time steps.

We next characterize the redundancy in computation and the global memory transactions for sliding-window overlapped tiling as a function of the input domain and the tiling parameters. Such a characterization is essential for modeling the profitability of fusion for statements with RAW dependence between them.

For a fixed k, the data transfer and redundant computation volume depends on T, which is controlled by the extent of fusion. For illustration, assume that there are four fusion-amenable stencil operators. We can explore several fusion choices – fuse all the four to get a single operator, fuse two to get two resultant operators, or fuse none. For sliding-window tiling, Table 2 explores the profitability of different fusion choices for stencils of various orders.

k	fusion degree	Load Transactions	Store Transactions	Redundant Computation
1	1	18939904	17686800	0
	2	10786022	9364037	39978854
	4	7225344	5334336	144831456
2	1	21572044	18728073	0
	2	14450688	10668672	95227776
	4	15745024	7626496	487849728
4	1	28901376	21337344	0
	2	31490048	15252992	313916416

Table 2: Choosing degree of fusion for varying k

We seek to minimize global memory transactions via fusion, while accounting for the increase in redundant computation as more statements are fused together. The hardware constraints used to eliminate tuples of the sorted list L_{tuple} in Section 3.6 usually eliminate high degrees of fusion. The profitability of fusion for the configurations in Table 2 is determined by computing the estimated time for performing global memory transactions (T_{mem}) and the estimated time for performing the computation (T_{comp}). To compute these estimates, we use the peak memory bandwidth and the peak performance achieved by a set of synthetic stencil benchmarks on the underlying device. The comparison metric $T_{max} = max\ (T_{comp}, T_{mem})$, inspired by the roofline model [26], is used to quantify the efficiency of the generated sliding-window overlap-tiled code. If a tuple of the list L_{tuple} has a value of T_{max} for the fused node that is greater than the sum of the values of T_{max} for the unfused nodes, that fusion choice is eliminated from consideration. The configurations with lowest T_{max} for different values of k are highlighted in Table 2. For experimental validation, we plot the performance for 12 time steps of a 13-point order-2 3D Jacobi stencil sequence with varying degree of fusion. The results are shown in Figure 1. The best performance is achieved when the degree of fusion is 2, as predicted by the model.

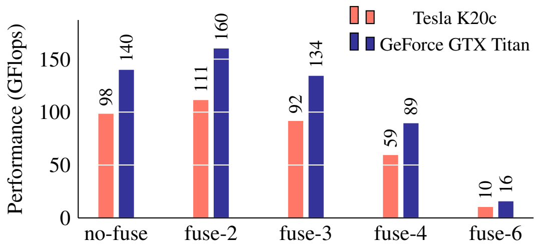

Figure 1: Performance of 12 time steps of a 13-point Jacobi stencil (k=2) for varied degree of fusion

5. CODE GENERATION

The code generation for an input stencil DAG consists of three main steps:

1. Fuse nodes in the DAG using the process described in Sections 3 and 4.
2. Generate a device kernel for each node in the fused DAG to execute the computation represented by the node.
3. Generate a host function to copy input arrays into device memory, launch the kernels generated in Step 2 in a sequence that satisfies dependencies between the fused DAG nodes, and copy the output of the computation back to the host.

Algorithm 3: Device code generation

```
Input  : IN: a node from the post-fusion stencil DAG
Output: $D_{gpu}$: device function for the node
1 $M_{IN}$ ← resource_usage(IN) ;
2 $D_{gpu}$ ← generate_allocated_resources ($M_{IN}$);
3 for each A in input_arrays (IN) do
4     $D_{gpu}$.append (generate_read_new_plane ($M_{IN}$, A));
5 end
6 $D_{gpu}$.append (generate_stencil_code (IN, $M_{IN}$));
7 for each A in (all_arrays (IN) - input_arrays (IN)) do
8     $D_{gpu}$.append (generate_rotate_data ($M_{IN}$, A));
9 end
```

Algorithm 3 shows the phases involved in the device code generation, and Listing 4 shows the output of our code generator. The output CUDA code can be divided into four distinct parts.

The first part uses the resource map M_{IN} computed in Section 3.3 to allocate GPU resources for different arrays involved in the computation. Function *generate_allocate_resources()* at line 2 declares the required shared memory buffers and registers for arrays A, B, and C at lines 10–12 of Listing 4.

The second part reads a new plane for each input array from global memory into the sliding window. Function *generate_read_new_plane()* at line 4 reads a new plane for the input array A into the shared memory buffer at line 15 of Listing 4.

The third part generates the stencil code for each statement. Function *generate_stencil_code()* at line 6 visits all the accesses in each stencil statement, and replaces it by the assigned GPU resource from M_{IN} (lines 17–28 of Listing 4).

Once the stencil computation is performed, the fourth part slides the window across the arrays cached in shared memory buffers/explicit registers for each statement freeing one resource to read from global memory in the next iteration. Function *generate_rotate_-data()* at line 8 rotates the data for arrays B and C at lines 31–32 of Listing 4, freeing a register for each array in the process.

To write the final accumulated value for the output array C to global memory, a store statement is generated after the last write to C at line 27 of Listing 4. Synchronization barriers are inserted (using *__syncthreads()*) between each part to maintain memory consistency.

Listing 4: Generated CUDA code for DSL of Listing 3

```
 1  __global__ void jacobi (float* restrict A,...) {
 2    // Determine the block indices
 3    int blockdim_i= (int)(blockDim.x);
 4    int i0 = (int)(blockIdx.x)*(blockdim_i-4);
 5    int i = i0 + (int)(threadIdx.x);
 6    int blockdim_j= (int)(blockDim.y);
 7    int j0 = (int)(blockIdx.y)*(blockdim_j-4);
 8    int j = j0 + (int)(threadIdx.y);
 9    // Resource Allocation
10    float __shared__ shA_c0[32][32];
11    float __shared__ shB_m1[32][32];
12    float rB_c0=0, rB_p1=0, rC_m2=0, rC_m1=0, rC_c0=0;
13    for (int k=0; k<=L-1; ++k) {
14      // Fetch new plane
15      shA_c0[j-j0][i-i0] = A[k*M*N + j*N + i];
16      __syncthreads ();
17      if (j >= max (j0+1,1) & ...) {
18          rB_p1 = c * shA_c0[j-j0][i-i0];
19          rB_c0 += stencil (shA_c0);
20          shB_m1[j-j0][i-i0] += b * shA_c0[j-j0][i-i0];
21      }
22      __syncthreads ();
23      if (j >= max (j0+2,1) & ...) {
24          rC_c0 = c * shB_m1[j-j0][i-i0];
25          rC_m1 += stencil (shB_m1);
26          rC_m2 += b * shB_m1[j-j0][i-i0];
27          C[max(k-2,0)*M*N + j*N + i] = rC_m2;
28      }
29      __syncthreads ();
30      // Value rotation
31      shB_m1[j-j0][i-i0] = rB_c0;
32      rB_c0 = rB_p1; rC_m2 = rC_m1; rC_m1 = rC_c0;
33      __syncthreads ();
34    }
35  }
```

Even though we limit the discussion to accumulative statements, our code generator can handle the diagonal access-free stencils as presented in [15, 17, 20] without converting them to accumulations.

6. CASE STUDY AND EVALUATION

In this section we present experimental results comparing performance achieved using other available GPU code generators with the approach described in this paper. We begin with a case study using a component stencil from a DOE mini-application, which we use to illustrate the developed fusion strategy. We follow that by a presentation of experimental results on a set of benchmarks.

6.1 Case Study: Hypterm

We evaluate the impact of applying the fusion heuristics on the hypterm routine of the ExpCNS mini-application from DoE that integrates the Compressible Navier-Stokes (CNS) equations [1]. For the discussion, we set the fusion objective to be minimization of data movement. The hypterm routine reads from a set of input arrays (*cons* and *q*) to update a set of *flux* arrays. In the stencil DAG of hypterm shown in Figure 2, the intermediate and output nodes (in shades of black) are order-4 stencil operators. There are 15 accumulation statements in the stencil computation, which are labeled in the figure. Different colored edges in Figure 2 represent 1D stencils along different dimensions. The hypterm routine is not time-iterated, and thus efficient data reuse across multiple stencils is crucial for performance. An untiled implementation of hypterm which does not use shared memory achieves only 35.65 GFlops on a Tesla K20c device.

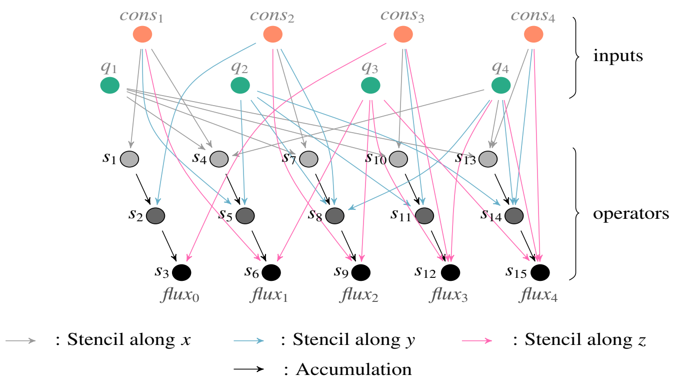

Figure 2: The stencil DAG of hypterm routine. *cons* and *q* are the input arrays, *flux* is the output array.

The fusion approach is applied to this DAG as follows:

– We begin by noting that s_{12} and s_{15} (s_6 and s_9) apply an order-4 1D stencil along z to three (two) input arrays. The number of registers used for spatial tiling would be $\geq$ 24 ($\geq$ 16), which exceeds M_{reg}. Therefore, these statements with never be fused with any other statements.
– Statements s_{13} and s_{14} accumulate data to the same array, and have two common input arrays. Since their fusion would result in maximum data movement saving among all nodes, the fusion algorithm merges them.
– The next candidates for fusion are clusters $\{s_{10},s_{11}\}$, and $\{s_4,s_5\}$. Both clusters result in a data movement saving of 2. However, the algorithm chooses to fuse s_{10} and s_{11} first, since the resulting node requires fewer total shared memory buffers. Statements s_4 and s_5 are fused in the next step.
– Fusing clusters $\{s_4,s_5\}$ and $\{s_{13},s_{14}\}$ will result in a data movement saving of 3. However, the fusion results in register spills in the generated code, violating the fusibility constraints. Hence this cluster is not fused.
– Statements s_7 and s_8 have a common input, and contribute to the same output. Hence they are fused next.
– Clusters $\{s_4,s_5\}$ and $\{s_7,s_8\}$ are fused next, since fusing them saves on data movement without register spills. Statements s_1 and s_2 are fused as they contribute to the same array.
– Statement s_3 and cluster $\{s_{10}, s_{11}\}$ are fused next, since they have a common input. Cluster s$\{s_1,s_2\}$ and $\{s_{13},s_{14}\}$ are fused to minimize the number of kernel launches.

[1]http://exactcodesign.org/proxy-app-software/

Any further fusion violates the fusibility constraints, since the shared memory usage of the fused cluster would exceed M_{shm}. The final nodes in the optimized DAG are: $\{s_1, s_2, s_{13}, s_{14}\}$, $\{s_3, s_{10}, s_{11}\}$, $\{s_4, s_5, s_7, s_8\}$, s_6, s_9, s_{12}, and s_{14}. As mentioned earlier, statements s_6, s_9, s_{12}, and s_{14} apply an order-4 stencil along the streaming dimension z on two or more input arrays.

Due to the high register pressure with our tiling scheme, we opt for naïve tiling with no use of shared memory or explicit registers for s_{13} and s_{14} in the generated code. The optimized code for the post-fusion DAG achieves 128.98 GFlops on a K20c, a 3.6× speedup over the base unfused version.

Recompilations. If we set the recompilation-bypass size to 1, then we perform nine recompilations to detect any violation of the fusibility constraints – five for clusters of size 2, two for clusters of size 3, and two for clusters of size 4. If the recompilation-bypass size is set to 2, then only four recompilations are needed.

Effect of implicit registers on degree of fusion. Since all the statements in the hypterm stencil are accumulations, a fusion model that does not account for implicit registers will fuse too many statements together. This will create high register pressure in the generated kernel, leading to register spills. In Figure 3, we plot the performance of the generated code for various degrees of fusion. *max-fuse* refers to the version with maximum fusion. It requires a 264 byte stack frame, generating 648 bytes of spill stores and 488 bytes of spill loads. *fuse-8* and *fuse-6* also generate spills. *fuse-8* requires 72 bytes of stack frame (76 bytes spill stores, 92 bytes spill loads), and *fuse-6* requires 16 bytes of stack frame (20 bytes spill stores, 20 bytes spill load). There are no spills generated for *fuse-4* and *fuse-2*. Any advantages from reduction in data movement for a degree of fusion higher than 4 is offset by the increase in global memory traffic from register spills. Since our fusion approach checks for register spills in the fusibility constraints, we generate a code with degree of fusion 4, thereby achieving the highest performance amongst these variants.

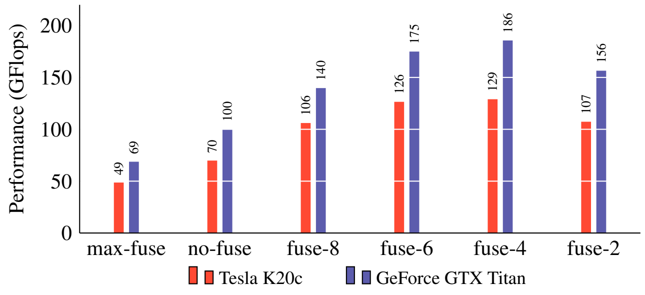

Figure 3: Performance of hypterm for varied degree of fusion

6.2 Experimental Evaluation

We evaluate our framework on a set of benchmarks listed in Table 3, and compare performance against PPCG-0.04 [24], Overtile-0.3.2 [9], and Forma [19]. PPCG and Overtile are open source compilers. All benchmarks use single-precision floating point computations. The code was compiled using NVCC 7.0 with flags *'–use_fast_math Xptxas "-v -dlcm=cg" -maxrregcount=32'*. The objective function was chosen to be minimization of data movement, since the performance of code generated with it was comparable to or better than the variants using other objective functions. The recompilation-bypass size was set to 1.

In each case, the time taken to generate the optimized code is dominated by the time for recompilations. As mentioned earlier, code versions are recompiled because we cannot effectively estimate the impact of fusion on the number of "implicit" registers used by the NVCC compiler. Since the compilation time depends on the host machine on which the code is compiled, instead of reporting absolute time for code generation, we provide information on the number of recompilations needed. The additional time to run the greedy fusion heuristic is negligible compared to the time for the NVCC recompilations. The number of recompiled variants is not excessive because of pruning from violation of any of the resource constraints, and the fact that growth of a fused cluster increases its resource usage. The most complex of the tested stencils was the previously discussed hypterm case study – any topological sort is valid for this stencil DAG, but it required only nine recompilations.

Benchmark	Domain	T	k	Flops	Arrays
Heat	512^3	4	1	15	2
Poisson	512^3	4	1	21	2
Chebyshev	512^3	4	1	39	6
Denoise	500^3	4	2	62	4
MiniGMG	480^3	4	1	27	9
Hypterm	300^3	1	4	358	13
FDTD	300^3	4	1	39	6

Table 3: Characteristics of the 3D benchmarks

Figure 4 presents performance in GFlops, for kernel execution time of the benchmarks from Table 3, on a Tesla K20c and a GeForce GTX Titan device. Figure 5 shows the performance on a GeForce GTX Titan X Maxwell device. For Kepler, the read-only arrays annotated with *__restrict__* keyword can be loaded through the cache used by texture pipeline. To exploit this feature, the input and output arrays were annotated in all the codes except for the baseline PPCG code. PPCG performs default thread coarsening, and the degree of coarsening for the Overtile compiler can be specified as part of the DSL input. Mapping multiple iterations to a thread exposes instruction level parallelism. The performance numbers for Overtile were measured by coarsening the slowest varying dimension by a factor of 2. Our code generator does not support thread coarsening at present. The tile sizes for all codes were tuned to achieve maximum occupancy.

The access offsets in Poisson and Chebyshev make them unresponsive to the optimizations proposed in [15, 17]. We are able to optimize them for storage by associative reordering of operations. For Chebyshev, which reads from four input arrays, time tiling would be impossible if all the input is stored in shared memory. With associative reordering, we are able to offload the storage of some arrays to registers, thereby allowing fusion. Denoise and MiniGMG are multi-statement stencils. Despite the high number of input arrays read, we are able to achieve maximal occupancy by optimizing them for storage. All benchmarks have RAW dependence between stencil statements. For Heat and Poisson, the degree of fusion is limited to 2 by the constraints on shared memory and redundant computations. For MiniGMG, FDTD and Denoise, the constraints on shared memory buffers and explicit registers are violated for fusion degree beyond 2. For Chebyshev, the constraint on explicit registers is violated for a fusion degree beyond 2. For the benchmarks with $T = 4$, we recompile clusters of size 2 only twice to check for a violation of the fusibility constraints.

Since PPCG does classical tiling without utilizing shared memory, the performance of its generated code is not comparable with other code generators. The Overtile compiler could not generate correct code for MiniGMG and Chebyshev. Since hypterm can

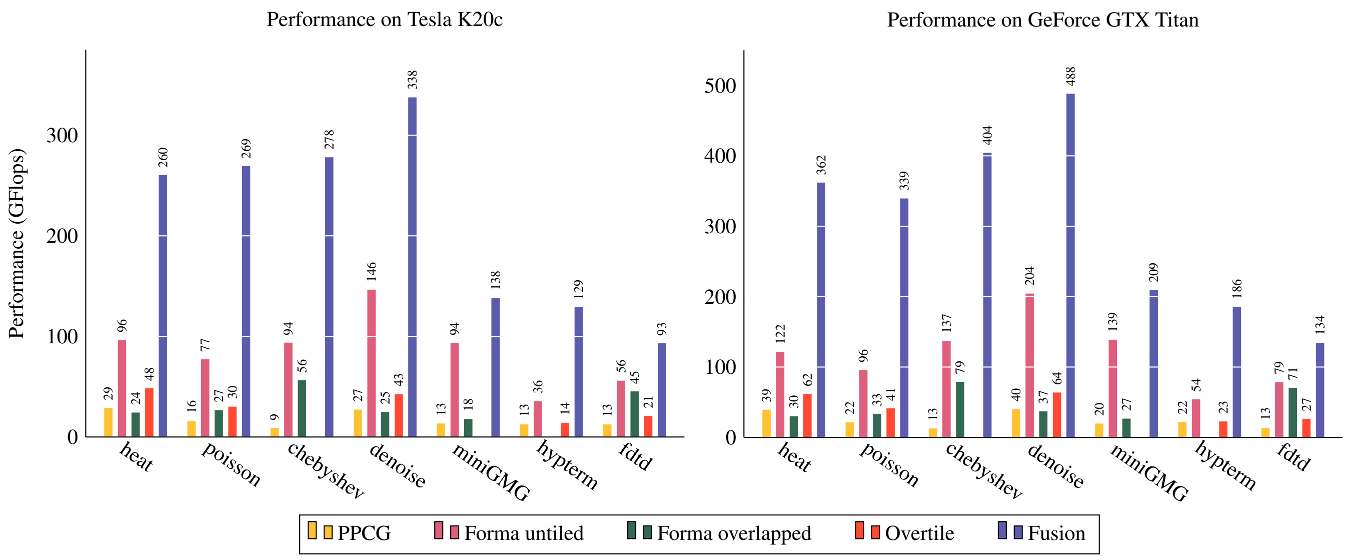

Figure 4: Performance results for 3D benchmarks on Kepler Tesla K20c and GeForce GTX Titan

only be spatially tiled, we do not present the performance numbers for Forma's overlap-tiled code for hypterm. Our optimized auto-generated code achieves 3.6× speedup for hypterm, 2.95× speedup for Chebyshev, and 2.3× speedup for Denoise. The speedup for MiniGMG is only 1.5×, due to the high volume of global transactions to read from a large number of input arrays. FDTD performs global writes and reads for six arrays at each time step. Time tiling reduces global transactions, but increases resource pressure, thereby reducing the achievable occupancy. If the shared memory size increases in future GPU architectures, we would be able to time-tile FDTD without sacrificing occupancy, further improving its performance.

The performance gap between other frameworks and our framework stems from the following factors: (a) we use a fusion heuristic to reduce data movement across stencil operators; (b) we use registers along with shared memory to cache the input data – this not only reduces access latency, but also reduces shared memory pressure, enabling higher occupancy than the other frameworks as the order of the stencil goes up; (c) we leverage associative reordering to transform the computation to a form on which the resource optimizations can be performed – this allows us to optimize a wider range of stencil computations than the other frameworks. As demonstrated by the experimental results, the proposed perspective on fusion and tiling for 3D stencils on GPUs leads to significantly better performance than what is possible with previously developed approaches.

7. RELATED WORK

Loop fusion and time tiling for stencil computations have been studied for CPUs. Ahmed et al. [1] present a polyhedral model based approach for synthesizing transformations to enhance data locality in imperfectly-nested loops. Li et al. [14] present a compiler framework for automatic tiling of iterative stencils, with a cost model to minimize cache misses. Mullapudi et al. present PolyMage [16], a domain-specific compiler for image processing pipelines on CPUs. It uses the polyhedral model to develop model-driven transformations for fusion, tiling, and storage optimization for image processing pipelines. However, due to architectural differences, the performance of tiling approaches on CPUs does not necessarily translate well to GPUs, especially for higher-dimensional stencils. High-performance GPU code generation for stencil computations has been a topic of active research [2, 4, 5, 6, 7, 9, 12, 15, 17, 19, 24, 25]. These research efforts can be grouped into four categories.

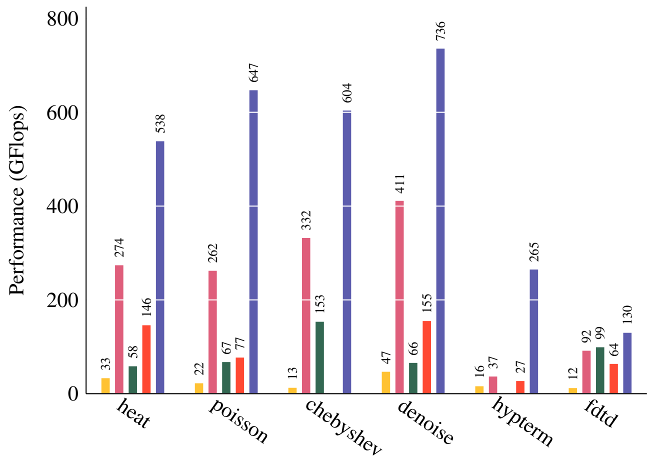

Figure 5: Performance results for 3D benchmarks on Maxwell GeForce GTX Titan X device

The first category comprises *automated code generators*, mainly targeting time-iterated stencils [2, 4, 5, 9, 23, 19, 24]. To achieve concurrent execution of tiles with time-tiling, these code generators use some form of non-rectangular tiling, such as overlapped tiling [9, 19, 16], split tiling [23, 5], or hexagonal tiling [4]. PPCG [24] is a polyhedral source-to-source compiler that generates classically time-tiled CUDA code from an annotated sequential program. Patus [2] generates time-tiled code with varying degree of thread coarsening, but without any use of shared memory. Overtile [9] and Forma [19] use shared memory, but do not utilize streaming. The split tile and hexagonal tile approaches of Grosser et al. [4, 5] also do not use streaming; the codes are not publicly available for evaluation. In contrast to our code generator, none of these code generators use fusion across stencil operators to maximize reuse, or streaming to reduce the tile footprint. PolyMage [16] performs fusion across stencils, but does not use streaming and does not address GPU code generation. Pochoir [23] is a stencil compiler that uses a cache-oblivious approach to generate split-tiled code for multi-core CPUs, but does not consider sequences of multiple stencil statements. With prior GPU code generators [2, 4, 5, 9, 19, 24], while good performance was demonstrated on a variety of 2D stencils, very few 3D stencils were evaluated, and performance on any evaluated 3D stencils for large problem sizes was considerably lower than performance achieved with large 2D stencil computations.

The second category of research comprises optimized *hand-tuned implementations* that target specific computations. Micikevicius [15] implemented spatial tiling with registers to alleviate shared memory pressure for a 3D finite difference computation. However, the optimizations in the implementation are limited to cross-stencils with a specific access pattern. Nguyen et al. [17] use the techniques presented in [15] to time-tile a 7-point 3D Jacobi computation that streams through one dimension. Our prior work [20] demonstrated high-performance for many 2D and 3D stencils through manual implementation of overlapped tiling with sliding windows, along with associative reordering for effective use of shared memory and registers.

The third category of research focuses on *auto-tuning* of certain parameters to optimize stencil computations. Datta et al. [3] analyze the performance of a 7-point 3D stencil (as implemented by Nguyen et al. [17]) on an Nvidia GTX280. Kamil et al. [11] develop a code generator and an auto-tuning framework, but their code generator does not utilize shared memory for stencil computations. Zhang and Mueller [27] evaluate a code generator and auto-tuner for 3D stencils on GPU clusters, but do not incorporate time-tiling. For many bandwidth-bound kernels, just spatial tiling is insufficient to achieve high performance. Halide [18] decouples algorithm specification from schedule. Halide schedules can express spatial tiling and sliding-window optimizations. However, the performance depends on either manually writing an efficient schedule, or exploring a vast schedule space with auto-tuning. Instead of auto-tuning, our code generator employs a model-driven approach to choose between fusion configurations.

The fourth category of research addresses the development of *analytical models* for the performance of stencil computations on GPUs. These models are intended for integration into code generators to guide kernel optimizations. Hong and Kim [10] propose an analytical model to predict performance of GPU kernels. However, their model relies on manual gleaning of low-level information about the kernel. Lai et al. [13] analyze a performance upper bound for SGEMM on GPUs, and use register blocking assuming maximum register reuse. Their model only provides performance upper-bounds for compute-bound kernels. Su et al. [22] propose a model that estimates the execution time based on the data traffic between different memory hierarchies. A common limitation of all these models is the precision in prediction, stemming from the complexity of the underlying hardware. Wahib and Maruyama [25] extend the analytical model in [13]. They pose kernel fusion as an optimization problem, and use a code-less performance model to choose a near-optimal fusion configuration amongst other possible variants. The space of feasible solutions is pruned using a search heuristic based on a hybrid grouping genetic algorithm. Gysi et al. [6] also propose a model-driven stencil optimization approach. Like [25], they use a code-less 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 [7]. While these works do model the shared memory constraints for kernel fusion, their models cannot accurately account for the register usage of the final kernel generated by NVCC, since the register count is highly influenced by the domain-specific optimizations first performed on the original code. We address this issue by splitting the register usage into two classes, *explicit* and *implicit* registers. While the explicit registers are modeled accurately, the implicit register usage is accounted for by limited compilation of a small number of fused configurations.

It is worth noting that the main focus of this work was not to develop a model that could accurately predict kernel performance for different fusion variants. Instead, our goal was to develop a practically effective model that incorporated fusion, sliding-window overlapped tiling, and associative reordering to enable automated generation of high-performance GPU kernels for 3D stencils. The modeling approach was validated by the performance achieved for many stencil benchmarks. Indeed, if a better model is available that can more accurately predict the performance of the generated CUDA kernels, it can be easily incorporated into the compilation tool-chain.

8. CONCLUSIONS

Several factors are important in enhancing the performance of a 3D stencil computation on a GPU: (a) maximizing use of the processing resources; (b) minimizing global memory accesses; (c) effective use of shared memory and registers; (d) accounting for the computational redundancy introduced by the tiling scheme. We present an automated code generator that manages the allocation of GPU resources, using fusion along with 2D overlapped tiling and streaming through one unpartitioned spatial dimension. A heuristic for fusion choice seeks to optimize the use of shared memory and registers. Experimental results on two GPU devices on a set of resource-intensive benchmarks demonstrate significant performance improvement over other available code generators.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback and constructive comments. This work was supported in part by the National Science Foundation through award ACI-1440749, 1513120, and 1524127, and the Department of Energy through award DE-SC0008844.

10. REFERENCES

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *Int. J. Parallel Program.*, 29(5):493–544, Oct. 2001.

[2] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687. IEEE Computer Society, 2011.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12. IEEE Press, 2008.

[4] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International*

Symposium on Code Generation and Optimization, CGO '14, pages 66:66–66:75. ACM, 2014.

[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 *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 24–31. ACM, 2013.

[6] T. Gysi, T. Grosser, and T. Hoefler. MODESTO: data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 177–186. ACM, 2015.

[7] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 41:1–41:12. ACM, 2015.

[8] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector SIMD architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24. ACM, 2013.

[9] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320. ACM, 2012.

[10] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[11] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[12] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 235–244. ACM, 2007.

[13] J. Lai and A. Seznec. Performance upper bound analysis and optimization of SGEMM on fermi and kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–10, Washington, DC, USA, 2013. IEEE Computer Society.

[14] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.*, 26(6):975–1028, Nov. 2004.

[15] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84. ACM, 2009.

[16] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443. ACM, 2015.

[17] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13. IEEE Computer Society, 2010.

[18] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530. ACM, 2013.

[19] M. Ravishankar, J. Holewinski, and V. Grover. Forma: A DSL for image processing applications to target GPUs and multi-core CPUs. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, pages 109–120. ACM, 2015.

[20] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, and P. Sadayappan. Effective resource management for enhancing performance of 2D and 3D stencils on GPUs. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, GPGPU '16, pages 92–102. ACM, 2016.

[21] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 65–76, New York, NY, USA, 2014. ACM.

[22] H. Su, X. Cai, M. Wen, and C. Zhang. An analytical GPU performance model for 3D stencil computations from the angle of data traffic. *J. Supercomput.*, 71(7):2433–2453, July 2015.

[23] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[24] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM TACO*, 9(4):54:1–54:23, Jan. 2013.

[25] M. Wahib and N. Maruyama. Scalable kernel fusion for memory-bound GPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 191–202. IEEE Press, 2014.

[26] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[27] Y. Zhang and F. Mueller. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 155–164. ACM, 2012.