Effective Resource Management for Enhancing Performance of 2D and 3D Stencils on GPUs

Prashant Singh Rawat
The Ohio State University
Columbus, OH 43210
rawat.15@osu.edu

Changwan Hong
The Ohio State University
Columbus, OH 43210
hong.589@osu.edu

Mahesh Ravishankar
NVidia Corporation
Redmond, Washington
mravishankar@nvidia.com

Vinod Grover
NVidia Corporation
Redmond, Washington
vgrover@nvidia.com

Louis-Noël Pouchet
The Ohio State University
Columbus, OH 43210
pouchet.2@osu.edu

P. Sadayappan
The Ohio State University
Columbus, OH 43210
sadayappan.1@osu.edu

ABSTRACT

GPUs are an attractive target for data parallel stencil computations prevalent in scientific computing and image processing applications. Many tiling schemes, such as overlapped tiling and split tiling, have been proposed in past to improve the performance of stencil computations. While effective for 2D stencils, these techniques do not achieve the desired improvements for 3D stencils due to the hardware constraints of GPU.

A major challenge in optimizing stencil computations is to effectively utilize all resources available on the GPU. In this paper we develop a tiling strategy that makes better use of resources like shared memory and register file available on the hardware. We present a systematic methodology to reason about which strategy should be employed for a given stencil and also discuss implementation choices that have a significant effect on the achieved performance. Applying these techniques to various 2D and 3D stencils gives a performance improvement of 200-400% over existing tools that target such computations.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Optimization

Keywords

Resource management, Stencil computations, Tiling, GPGPU

1. INTRODUCTION

Stencil computations form the compute-intensive core of scientific applications in many domains. Many recent efforts target optimization of stencils, including several domain-specific languages and frameworks. Stencil computations exhibit a high degree of data parallelism, thereby making them well suited for execution on GPUs. However, stencil computations are often memory-bandwidth limited unless temporal reuse is exploited across a sequence of stencils or across multiple time steps for time iterated stencils.

Tiling is a key transformation to enhance temporal reuse of data and thus reduce the amount of data transfer from/to global memory on the GPU. Time-tiled execution of stencil code using the GPU's shared memory has been pursued by several efforts [1, 3, 4, 5, 11]. While good performance has been reported by many efforts for 2D stencils, high performance on 3D stencils has generally been much more challenging. As elaborated later in the paper, a primary reason for the lower performance on 3D stencils is that the shared memory capacity required for effective time-tiling of 3D stencils is much higher than that required for 2D stencils.

The total amount of shared memory in each SM (Streaming Multiprocessor) of a GPU is very limited, typically $48KB$. The low shared-memory capacity limits the maximum number of time steps in a time-tile and also limits the number of concurrently active thread blocks in each SM. The low occupancy in turn leads to lower performance due to inability to effectively overlap memory access latency with sufficient operations to execute across the active warps.

However, judicious exploitation of the data access pattern of stencil computations, combined with associative reordering of the stencil operations where appropriate, can enable the use of GPU registers to offload data from shared memory, thereby enhancing data reuse and/or occupancy. Since the register file size per SM is larger than the shared memory size on most modern GPUs (a trend that is expected to continue), this alleviates the constraints on occupancy imposed by shared memory limits. The low access latency of registers further improves the performance of the code. The optimization of register and shared-memory resources for stencil computations is a primary focus of this work. In combination with streamed execution (elaborated later) along an arbitrarily long tile dimension, we achieve significantly higher GPU performance than prior reported efforts.

The paper makes the following contributions:

- We present a systematic analysis that accounts for the constraints on various hardware resources, such as registers and shared memory on modern GPUs. This analysis is used to find an ideal configuration of the grid and block sizes for time-tiling schemes for stencil

GPGPU-9, March 12-16, 2016, Barcelona, Spain

 ISBN 978-1-4503-4195-0/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2884045.2884047

computations.

- We present an algorithm for streamed time-tile execution of stencil computations that overcomes these resource bottlenecks by effectively managing the shared memory and registers that are available on the GPU.
- We evaluate the effect of using associative reordering of the stencil updates to enhance the performance achieved.
- We validate the approach by manual implementation of different time-tiling schemes for 2D and 3D stencils. Comparisons with existing tiling compiler frameworks show that the approach developed in this paper can significantly boost the performance of stencil computations on modern GPUs, especially for 3D stencils.

2. GPU CONSTRAINTS FOR STENCIL COMPUTATION

The basic computational unit of a GPU is a thread. Going up the hierarchy, threads are grouped into blocks, and blocks are grouped into a grid. A program explicitly specifies this hierarchy of grid and blocks as kernel launch parameters. Threads within a block are executed on the same *streaming multiprocessor* (SM), can exchange data via shared memory, and synchronize among themselves.

In order to be performance efficient, apart from reducing accesses to global memory through spatial and temporal reuse, tiling algorithms in GPU must also incorporate the following.

- Access global memory in a coalesced manner.
- Maximize concurrency to keep all the execution units busy.
- Account for the limited amount of shared memory available on each SM (typically $48KB$).
- Reduce register usage per thread. Increased register usage results in either lower occupancy or expensive spills.

Some of these factors are tightly coupled to the underlying hardware, so the kernel launch parameters need to be tuned for the specific GPU architecture on which the program is executed. In this section, we present some general constraints on block and grid size for spatial tiling of d dimensional stencil computation, over an N^d input domain. Typically, each point of the iteration space is executed by a thread in the GPU. We assume that the stencil is of order k along each dimension. We also assume that for stencil computations, the amount of memory accessed per iteration point of the computation can be amortized to ≈ 1 per point.

2.1 Constraints on block and grid size

Every GPU has a hardware limit on maximum number of threads per SM (T_{sm}), maximum number of threads per block (T_b), maximum shared memory per SM (M_{sm}), maximum number of concurrently active blocks per SM (B_{sm}), and register file size per SM (R_{sm}). For instance, $T_{sm} = 2048$, $T_b = 1024$, $M_{sm} = 48KB$, and $B_{sm} = 16$, and $R_{sm} = 65536$ for Tesla K20c. The threads in an SM can be grouped in various ways (from 2 blocks of 1024 threads, to 16 blocks of 128 threads in K20c). The threads in a block are scheduled in an atomic unit of 32, called *warp*.

When a thread in a warp accesses global memory, the warp stalls for the next hundreds of clock cycles due to high memory latency. In order to hide the latency, the SM can switch to another warp in the ready state. There must be enough warps to keep the SM busy till the memory request of the stalled warp is served. The best one can do is to ensure that all the threads in an SM are utilized. If we reach the hardware limit of $\frac{T_{sm}}{32}$ warps per SM, then we achieve maximum *occupancy*, the ratio of active warps to maximum theoretical active warps per SM.

Stencil computations can benefit from spatial reuse if the data is cached in shared memory. The access latency of shared memory is orders of magnitude lower than global memory, but it puts additional constraints on the number of blocks that can be active concurrently on an SM. If each block of size sz_b uses $c.sz_b$ bytes of shared memory, then we can have no more than $\frac{M_{sm}}{c.sz_b}$ active blocks per SM. If c is large, then fewer blocks can be active per SM, which can potentially hurt occupancy.

Registers are the fastest, but limited storage resource available to the thread. The maximum number of registers that a thread could use is constrained by the hardware. If all the threads in an SM are to be active, the number of threads must be bounded by $\frac{R_{sm}}{T_{sm}}$. In general, if a thread uses t_{reg} registers, then the maximum active threads per SM is $min\left(T_{sm}, \frac{R_{sm}}{t_{reg}}\right)$.

Taking all these factors into account, the minimum number of active blocks per SM, min_b can be computed as $min_b = min\left(B_{sm}, \frac{T_{sm}}{sz_b}, \frac{M_{sm}}{c.sz_b}, \frac{R_{sm}}{t_{reg}.sz_b}\right)$. Unless one intends to coarsen the computation, the block size sz_b must be chosen to maximize occupancy, i.e., $sz_b \times min_b$ must be as close to T_{sm} as possible. If there are K SMs in the GPU, then the grid size must be at least $K \times min_b$ to maximize concurrency.

Coalescing of global memory accesses is important since it reduces the total number of memory transactions, thereby minimizing DRAM bandwidth. To benefit from coalescing, the fastest varying dimension of the thread block is aligned to the fastest varying dimension of the input domain, and is chosen to be a multiple of warp size, i.e. $mod(sz_b, 32) = 0$.

2.2 Partitioning threads in each dimension

Given a 2D block of constant size sz_b, we can vary the number of threads along x and y dimensions to get different block configurations. The performance of a stencil computation varies depending on the configuration we choose. To illustrate, let $sz_b = b_x \times b_y$. Two possible configurations for $sz_b = 1024$ are shown in Figure 1. For configuration (a), $b_x = b_y = 32$. For configuration (b), $b_x = 64, b_y = 16$. For both the configurations, b_x is a multiple of warp size to benefit from global memory coalescing.

If an order-k stencil over N^2 domain uses configuration (a), total global reads are $\frac{N^2}{1024}(32+2k)^2$. For configuration (b), the total global reads are $\frac{N^2}{1024}(64+2k)(16+2k)$. For any value of k, we see that configuration (a) maximizes the computation area while minimizing the global read transactions.

To optimally partition the threads for a d-dimensional block, one can fix the number of threads along the fastest

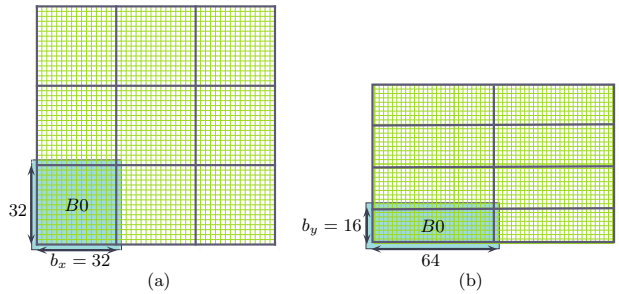


Figure 1: Different block configurations for a fixed block size

varying dimension, $t_f = max(32, \left\lfloor \frac{sz_b{}^{1/d}}{32} \right\rfloor .32)$, and set the initial number of threads in other dimensions to $\geq \left(\frac{sz_b}{t_f}\right)^{\frac{1}{d-1}}$. The numbers along non-fastest varying dimensions can then be adjusted depending on the shared memory limitations. If the numbers obtained for the non-fastest varying dimensions are too low, we recompute t_f by replacing 32 by 16 (half-warp). For example, a 3D block of size $(16, 8, 8)$ can be more favorable than a block of size $(32, 5, 6)$, since it utilizes all the available threads.

3. OVERLAPPED TILING WITH STREAMING FOR STENCIL COMPUTATION

Most stencil computations are bandwidth bound. Both the input values, and values computed at each time step are used multiple times in computation at subsequent time steps. Caching these values in faster shared memory and registers, and reusing them instead of repeated reads and writes to slower global memory, is critical to achieve high performance. Tiling is a key transformation to address this issue. It aims to exploit parallelism while maximizing spatial and temporal locality in bandwidth bound stencil computations. Tiling the input domain and assigning the tiles to different thread blocks does not guarantee concurrent execution in time-tiled stencil computations. For every tile, computing the boundary values at time step t requires data at time step $t-1$ from the neighboring tiles. This data is referred to as the *halo region* or the *ghost zone*. The dependence of a tile on the data computed in neighboring tiles introduces the need for a global synchronization after each time step.

3.1 Overlapped Tiling

Efficient tiling algorithms for GPU has been a topic of intense research. A tiling technique that has been shown to be effective for GPUs is overlapped tiling [6, 5, 12]. It eliminates the dependence between tiles by introducing redundancy in the initial data loads from global memory and intermediate computation. The extent of input data read per tile is increased such that no communication is required to compute the boundary values at each time step. It means that same point in the iteration space can be computed by neighboring tiles redundantly, as shown in Figure 2a.

The extent of redundant computation depends on the block size, number of overlapping dimensions, the order of the stencil, and the time tile size. For example, blocks TB_0 and TB_1 in Figure 2a compute three output values. There are four redundant reads and two redundant computations. These redundant reads and computations could be eliminated if the block size was increased to compute six output values. If the time tile size was increased to 3, then the number of redundant reads and computations would both increase to six. The redundancy increases with increase in stencil order as well. Due to the lack of explicit communication between tiles, the generated code is simple. Since GPUs have high a floating-point throughput for 2D stencils, the increase in computation is sometimes tolerable for low-order stencils.

On the other hand, the amount of redundant computation for a 3D stencil can be overwhelming for some stencils. Consider a block that computes an $8 \times 8 \times 8 = 512$ block of the output for a 3D first-order stencil using a time tile size of 2. Each block would need to read a $10 \times 10 \times 10 = 1000$ block of the input domain, out of which almost half are read redundantly by another thread block as well. The number of points at the intermediate time step computed by the block would be $9 \times 9 \times 9 = 729$. The amount of redundant computation would be $729 - 512 = 217$, i.e., a third of the computation performed by each block is performed by another block as well. This problem of redundancy for 3D stencils get worse with stencil order and time tile size.

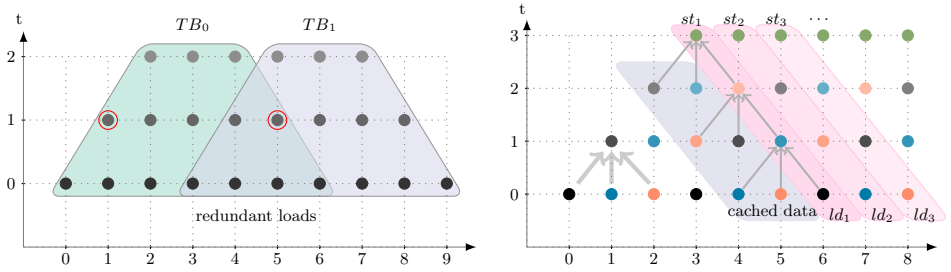
3.2 Streaming

Since overlapped tiling for 3D stencils is not an effective approach, instead of tiling along all three dimensions, we *stream* along one-dimension of the computation. We will first describe what we mean by streaming along a dimension of the computation and then discuss how it is used in combination with overlapped tiling to address the problem of redundancy for 3D stencils.

Figure 2b describes streaming through x dimension for a 1D Jacobi stencil with time tile of 3 time steps. $t = 0$ is the initial state from which the input values are read at $t = 1$. At each time step, intermediate output values are computed using the data from the previous time step. The final output is computed at $t = 3$. Every time step requires a different set of buffers. In the figure, different colored dots at time step $0 - 2$ represent distinct shared memory buffers. The global memory is represented by green dots. For a 1D Jacobi computation, three distinct memory buffers are required at each time step, shown by different shades of black, blue, and orange dots. In general, an order-k stencil needs $2k + 1$ distinct buffers per time step.

After an initial prologue, an iteration of the computation loads a value from global memory at $t = 0$, computes intermediate results at each time step, and finally writes out an output value to global memory. To observe an instance of this computation, let us assume that the input values at $x = \{4, 5, 6\}$ are cached in shared memory buffers $\{blue_0, orange_0, black_0\}$ respectively before their use (each buffer is labeled as $color_t$). At time step 1, $x = 5$ can be computed by reading from these buffers, and stored in the $blue_1$. Assuming that the values computed at $x = \{3, 4\}$ are available in $orange_1$ and $black_1$, we can compute $x = 4$ at time step 2, and store the result in $orange_2$ buffer. Once again, assuming that the values computed at $x = \{2, 3\}$ are available in $black_2$ and $blue_2$, we can compute $x = 3$ at time step 3, and write out the result to global memory.

In the subsequent iteration, the input values at $x = \{5, 6\}$ are already buffered due to the data overlap from the stencil point on the left. However, the value $x = 4$ is no longer re-



(a) Overlapped tiling (b) Streaming

Figure 2: Different tiling schemes for 3-point 1D Jacobi computation

quired in the computation, and its buffer can be reused. So we store the next input value from $x = 7$ in $blue_0$. This reuse holds for each time step. We make two quick observations from the computation: (1) For each input point x_i read in iteration i, we can compute the output point x_{i-3}. (2) We only need 3 buffers per time step for the entire computation, irrespective of the memory footprint of the tile size. In general, for an order-k stencil, we need $2k + 1$ buffers per time step. The buffer storing the oldest value in iteration i can always be reused to store the newest value at iteration $i + 1$. The scheme is called streaming since we use a sliding window to minimize the shared memory footprint, and the consumption of input is moderated at one point per iteration.

When a thread block computes all the points along x axis serially in a streaming fashion, we call it *serial-streaming*. An alternative would be block the iteration space along x, and assign each partition to different thread blocks. Each thread block will still serially stream through its share of iteration space, but all the blocks can concurrently stream through their share of iteration space. We call such streaming *concurrent-streaming*.

Streaming can be extended to 2D or 3D Jacobi stencils by interpreting a point in Figure 2b as a line or a plane respectively. For large problem size, fitting $2k + 1$ lines (or planes) in entirety in shared memory may be infeasible. The line (or plane) then needs to be blocked, with the block size carefully chosen depending on the time tile size and hardware constraints. If the time tile size is 1, then we only exploit spatial reuse. For an N^3 domain, we can interpret a point in Figure 2b as a blocked plane of size B^2, and then serial-stream through N blocked planes in the third dimension. Such spatial blocking is known as 2.5-D blocking, and the corresponding time-tiled blocking is known as 3.5-D blocking [9].

Since streaming does not perform any redundant computation and uses very limited amount of shared memory, it is an attractive option to combine with overlapped tiling. This is done by using streaming along one dimension and overlapped tiling along the others. For example, for a 3D computation, using overlapped tiling along two dimensions and streaming along the third would reduce the amount of redundant operations (loads and computations). The downside of streaming is that it serializes the computation along one dimension of the problem. If the problem size is not big enough to keep all the SMs busy, this would result in a performance degradation. In such cases concurrent streaming can be used to increase the degree of concurrency in the problem to keep all the SMs busy. The next section describes a systematic approach to decide the best tiling strategy for a given problem.

4. RESOURCE OPTIMIZATION FOR STENCIL COMPUTATION

In this section, we walk through the different factors to be considered while converging upon a tiling strategy that is best suited for a given stencil computation. Section 4.1 describes our time-tiled implementations of overlapped tiling for 2D stencils. Section 4.2 characterizes 3D stencils based on their access patterns, and describes optimizing techniques for them. We also discuss implementation details that might not be directly related to the tiling scheme, but are necessary to achieve good performance. Further, some of these techniques use explicit registers for storage, and a high per thread register pressure can result in poor occupancy and inferior performance.The NVCC compiler provides a compile-time flag `-maxrregcount=n` which limits the number of registers per thread to n. This forced reduction may cause register spills, but since we can always choose n so that $\frac{R_{sm}}{n} = T_b$, we will not consider register pressure as a limiting factor while computing occupancy in this section.

For convenience of description, we fix the order of stencils to k and the time tile size to T, and assume all the operations to be single precision. An optimized tiling code must be tuned for different architectures; we use Tesla K20c as the underlying GPU.

4.1 Overlapped Tiling for 2D stencils

Overlapped tiling described in Section 3.1 can be extended to any d-dimensional domain. Figure 3a shows overlapped tiling for a 2D thread block along x dimension. Most stencil compilers that generate naïve overlapped time-tiled code start by partitioning the d-dimensional output domain at time step T into blocks of size B'^d and then backtrack, growing the block size at previous time steps to include the points comprising the halo region [5, 12]. The resultant block size for the input domain at $t = 0$ is B^d, where $B = B' + 2Tk$. The thread block size is chosen to be B^d instead of B'^d to avoid serialization of computation.

An inherent performance limitation of such code stems from the partitioning scheme; an issue that is orthogonal to overlapped tiling. The block size of input domain is same as the thread block size, which itself is constrained by the

hardware. The total redundant computation for overlapped tiling of an N^d domain is

$$V = \left(\frac{N - 2Tk}{B'}\right)^d \sum_{i=1}^{T-1} (B - 2ik)^d - B'^d$$

Increasing B' will reduce the volume of redundant computation. But since B'^d cannot increase beyond T_b, the volume of redundant computation can be overwhelming for higher values of d.

Overlapped tiling + streaming. Given an N^2 input domain, we can eliminate redundant computations along y through streaming, and perform overlapped tiling along x to achieve concurrency in execution. The input is partitioned into overlapping strips of size $B \times N$ ($B_y = N$ in Figure 3a). A single thread-block is in charge of computing all the N points along the y dimension. A thread block reads b_y input lines along y axis per iteration (b_y is the block-size along y-dimension), to execute b_y points of the iteration space. Here each thread executes the iteration space at a stride of b_y in the y direction. In this mode, each thread block has to store $T.(b_y + 2k)$ lines in the shared memory. With this information, we can navigate through the choices for block dimensionality. To simplify the computations, we fix $k = 1$ and $T = 4$.

- $b_y = 1$, *i.e. 1D block*: To achieve maximum occupancy, the block size must be $\frac{2048}{16} = 128$. In each iteration, a block reads one line from input at $t = 0$ to compute one line of output at $t = 4$. For this, it needs 6144 bytes of shared memory, which constrains the actual number of active blocks per SM. From Section 2.1, an SM can only have $min\left(16, \frac{2048}{128}, \frac{48KB}{6144B}\right) = 8$ blocks, a 50% loss of occupancy.
- $b_y = 32$, *i.e. 2D block*: Two blocks of size 32×32 can theoretically be active per SM. Since a block now operates on a chunk of 32 lines instead of 1, it needs $17KB$ of shared memory. Each SM can practically have $min\left(16, \frac{2048}{1024}, \frac{48KB}{18KB}\right) = 2$ active blocks, which implies maximum occupancy.

Clearly, we achieve better occupancy with 2D blocks. For a $b_x \times b_y$ block, the computation proceeds as shown in Figure 2b; the point being computed is interpreted as a set of contiguous b_y lines. In each iteration after prologue, b_y lines are computed at time step t by reading $b_y + 2k$ lines from the shared memory buffer at time step $t - 1$. A *sliding-window* approach is used where after each iteration, b_y oldest lines in the buffer can be reused to cache the new lines. The buffer itself can be implemented as a circular array, and row performs modulo operations to find the top and bottom k rows.

Optimized streaming. Modulo operations are costly on the GPU since they compile to multiple instructions in the assembly code. If $b_y + 2k$ is a power of 2, then we can replace the modulo operator by bitwise operator instead, which has a very high throughput. ($a \bmod b \equiv a \,\&\, (b-1)$ if b is a power of 2). Since k is zero only for point-wise operations, b_y may not be a power of 2 for most stencils. This would result in 2048 not being a multiple of $b_x \times b_y$. No configuration would then maximize the occupancy of an SM.

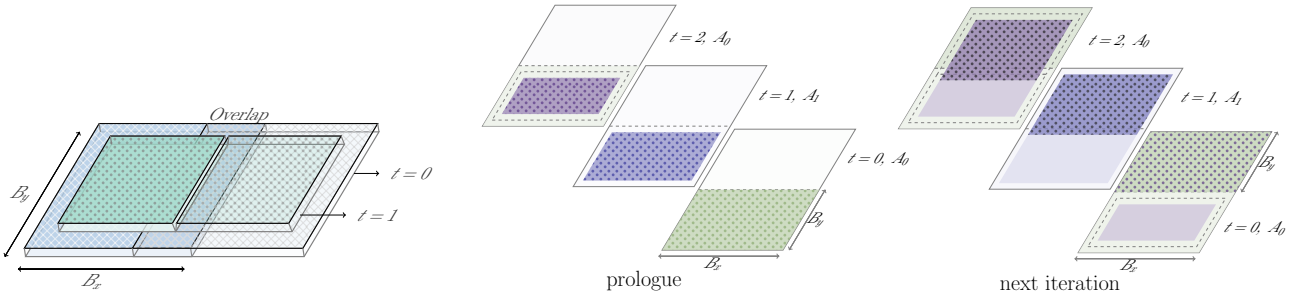
Instead of a *sliding-window* approach, we use an *ping-pong buffer* approach. We allocated two shared memory buffers A_0 and A_1. All even time steps read from buffer A_0 and write to buffer A_1, and all odd time steps read from A_1 and write to A_0. In Figure 3b, for each iteration at $t = 0$, a block reads b_y lines to fill half of the buffer A_0. The even iterations use the first half of A_0, and the odd iterations use the second half. The prologue computes $b_y - 2tk$ lines at each time step $t > 0$. In the subsequent iterations, each time step computes b_y lines using $b_y + 2k$ lines from the previous time step.

Here b_y can be chosen to be a power of 2. This approach only avoids the use of expensive modulus operation, the shared memory requirement for a $b_x \times b_y$ block decreases to two buffers of size $2b_yb_x$ (from $T.(b_y + 2k).b_x$ for the sliding-window approach). Further, the amount of shared memory is independent of the time-tile size. If we create thread blocks of size (32×16), then each block will use 8192 bytes of shared memory. Each SM can concurrently schedule at most $min\left(16, \frac{2048}{512}, \frac{48KB}{8192B}\right) = 4$ blocks, utilizing all the available threads per SM. Assuming that there is enough concurrency to keep all SMs busy, this optimized version outperforms the traditional tiling.

Algorithm 1 presents an implementation sketch of the optimized version. Line 2 computes the partition of input domain that a block reads. Line 8-11 identify the iteration space of the output domain. Line 13 computes the indices of the input rows in the circular buffer that will be used to compute an output row. The function *apply_stencil ()* in Line 19 applies the stencil operator on the input buffer to compute values in the output buffer.

Overlapped tiling + registers + concurrent streaming. Overlapped tiling + streaming with 1D thread blocks was not optimal because the high shared memory usage lowered the occupancy. We call a stencil *diagonal-access free* if the access pattern (x_0, y_0, z_0) from one plane along z axis to other plane is strictly of the form $(0, 0, z_0)$. If the stencil is *diagonal-access free*, then the shared memory requirement can be lowered by using registers to cache the $2k$ accesses to lines [7]. For a thread block of 128 threads, $k = 1$ and $T = 4$, a block now needs 8 registers and 2048 bytes of shared memory. With this optimization, the number of feasible active blocks per SM is $min\left(16, \frac{2048}{128}, \frac{48KB}{2048B}\right) = 16$ blocks. While the occupancy is maximized, this strategy suffers from low concurrency. To keep all the 13 SMs of K20c busy, we need at least $13 \times 16 = 208$ thread blocks. However, even a large input domain of size 8192^2 can only be partitioned into $\frac{8192}{128} = 64$ blocks. This was not a problem with the overlapped tiling + optimized streaming algorithm since it needed only $13 \times 4 = 52$ blocks to achieve full concurrency, and the input domain for it was partitioned into 256 blocks.

The simplest way to increase concurrency is to block the input domain in y dimension as well. We only need to partition the domain into $\left\lceil \frac{208}{64} \right\rceil = 4$ blocks along the y dimension. For correctness, we perform overlapped tiling along y dimension as well. Since the block is 1D and the input grid is 2D, we perform streaming within a tile.



(a) Overlapped tiling along x for a 2D block (b) A block's computation in optimized version of streaming along y

Figure 3: Overlapped tiling for 2D stencils

Algorithm 1: Overlapped tiling + optimized streaming

Input : IN : input array,
T : time tile size,
k : stencil order,
A_0,A_1 : sh-mem buffers,
(B_x, B_y) : block size
Output: OUT : output array

```
1  size_y = 2.B_y;
2  i_x = blockIdx.x × (B_x − 2Tk),  i_y = 0;
3  while i_y < N do
4      u = max(0, i_y)..min(i_y + B_y, N − 1);
5      v = max(0, i_x)..min(i_x + B_x, N − 1);
6      A_0[u & (size_y − 1)][0..B_x] ← IN[u][v];
7      for t from 1 to T do
8          start_x = max (t, i_x + t);
9          start_y = max (t, i_y − t);
10         end_x = min (i_x + B_x − t, N − 2k);
11         end_y = min (i_y + B_y − t, N − 2k);
12         for l from -k to k do
13             row_{y+l} = (row_y + l) & (size_y − 1);
14         end
15         g = start_y..end_y;
16         h = start_x..end_x;
17         buf_r = (t mod 2 == 0) ? A_1 : A_0;
18         buf_w = (t mod 2 == 0) ? A_0 : A_1;
19         buf_w[g & (size_y − 1)][h] = apply_stencil (buf_r);
20         __syncthreads ();
21     end
22     OUT[g][h] ← buf_w[g & (size_y − 1)][h];
23     i_y += B_y;
24 end
```

This version incurs some volume of redundant computation and bandwidth along y axis when compared to overlapped tiling + streaming, but it is negligible compared to the reduction in redundancy and bandwidth along x axis due to larger *blockDim.x* (128 vs. 32). This implementation also benefits from the low access latency of the registers.

4.2 Tiling for 3D stencils

Efficient 3D tiling must strive to reduce the extra data transfers involved in reloading ghost region across tiles. Figure 4 shows the possible grid dimensionality for a 3D domain. 3D grid requires extra memory bandwidth for redundantly loading the same halo regions for neighboring blocks along z axis. 2D grid entails streaming through the non-partitioned dimension. For 1D grid, we stream through one non-partitioned dimension, and tile the other non-partitioned dimension using parallelogram tiling. However 1D grid might not provide enough concurrency to achieve good performance.

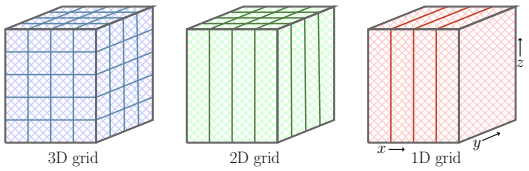


Figure 4: Various grid dimensionalities for a 3D input

For higher dimensional domain, streaming simplifies the tiling algorithm and code generation. Since the streamed dimension is not tiled, a d-dimensional domain can be tiled using the same algorithm that tiles $d-1$ dimensions. Our tiling implementation for 3D stencils partitions the input domain as a 2D grid, and assigns a 2D thread block to each partition that streams through the unpartitioned dimension. Concurrency is achieved by using 2D overlapped tiling (Section 4.1) on the partitioned dimensions. Note that we cannot stream through x axis for 3D stencils, since this would entail non coalesced accesses while loading data from a yz plane. Without loss of generality, we choose z to be the streaming dimension in this section.

Streaming in a time-tiled 3D stencil requires $T.(2k+1)$ planes to be in shared memory. When $T = 4$ and $k = 1$, a thread block of size 32×32 will need $48KB$ shared memory. Since this is the total available shared memory, an SM can have no more than one active block, resulting in 50% occupancy. If $k = 2$, then the required shared memory exceeds the hardware limit. [7] uses registers to offload the caching of some planes to registers for spatial tiling. We discuss the details of implementing a time-tiled code with shared memory + registers. The discussion will be independent of the tiling scheme used across thread blocks to achieve concurrency, as it is orthogonal to the streaming optimizations that are local to a block.

Streaming + registers. In the scenario above with $k = 1$ and block size 32×32, if the stencil is *diagonal-access free*, then we can increase the per thread register pressure by $2Tk$, and bring down the shared memory requirement to $16KB$. With this trade-off, an SM can have two active blocks, achieving maximum occupancy. If there are register spills due to the increased register pressure, then we can reduce the value of T to reduce the register pressure. Figure

5 shows one time step of the time-tiled 7-point 3D stencil using registers.

For the 7-point stencil, the resources involved in the computation are shared memory buffer $A[T]$, and registers $r_p[T]$, $r_m[T]$. In each iteration i, threads in a block read a point from input plane z_{i+1} into $r_p[0]$. The block computes plane z_i at time step 1 using $A[0], r_p[0]$, and $r_m[0]$, and stores this computed value in $r_p[1]$. Using $A[1], r_p[1]$, and $r_m[1]$, the block computes plane z_{i-1} at time step 2. Proceeding this way, at time step T, plane z_{i-T} of the output array is computed. After each iteration, we perform a data shift ($r_p \rightarrow A \rightarrow r_m$) at each time step, freeing r_p to store new values.

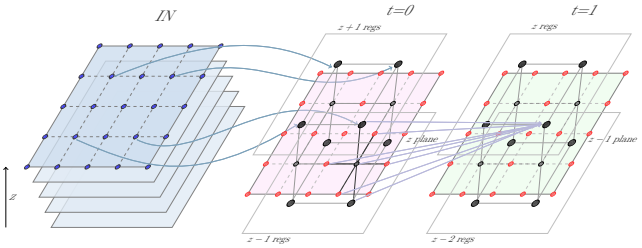


Figure 5: Streaming along z dimension for 7-point 3D Jacobi

An implementation sketch of the scheme is presented in Algorithm 2. The algorithm is presented independently of the tiling scheme for the other two dimensions. The initializations at Line 1 depend on the tiling schemes for x and y dimensions. The function *compute_stencil* returns the output of applying stencil computation to a point.

Algorithm 2: streaming + registers algorithm for a 3D order-1 stencil

Input : IN : input array,
T : time tile size,
(B_x, B_y) : block size,
(tid_x, tid_y) : thread index,
A_t : sh-mem buffer for time step t,
$r_p[t]$ & $r_m[t]$: registers storing $z+1$ and $z-1$ planes at time step t
Output: OUT : output array

1 $i_x = \ldots, i_y = \ldots, i_z = 1$;
// Initialization
2 $A_0[0..B_y][0..B_x] \leftarrow \text{IN}[0][i_y..i_y + B_y][i_x..i_x + B_x]$;
3 $r_p[0] = \text{IN}[1][i_y + tid_y][i_x + tid_x]$;
4 **for** *each* i_z *from* 1 *to* $N-2$ **do**
// Shift data
5 $r_m[0] = A_0[tid_y][tid_x]$;
6 $A_0[tid_y][tid_x] = r_p[0]$;
7 $r_p[0] = \text{IN}[i_z + 1][i_y + tid_y][i_x + tid_x]$;
8 *__syncthreads ()*;
9 **for** t *from 1 to* T **do**
// Shift data, put result in r_p[t]
10 $r_m[t] = A_t[tid_y][tid_x]$;
11 $A_t[tid_y][tid_x] = r_p[t]$;
12 $r_p[t] = compute\_stencil(A_{t-1}, r_p[t-1], r_m[t-1])$;
13 *__syncthreads ()*;
14 **end**
15 $\text{OUT}[i_z - kT][i_y + tid_y][i_x + tid_x] = r_p[T]$;
16 **end**

Generalizing shared memory + register amenable stencils. With minor modifications, we can apply Algorithm 2 to all the stencils where each plane accesses only one point per plane from other planes along z axis. If such a stencil accesses a diagonal point (x_0, y_0, z_0), the writes to r_p and r_m at Line 3, Line 5, Line 7, and Line 10 have to factor in the offset (x_0, y_0) during data access. We will also need a temporary buffer to store the output of computation at Line 12, and other reads from r_p or r_m. After the data is written into the temporary buffer, each thread can read appropriately offset value into r_p and r_m. There will be extra synchronization barriers after every write to the temporary data, but we still use only $T+1$ shared memory buffers instead of $T(2k+1)$ buffers.

Optimization for associative stencils. For stencils that access more than one point per plane from other planes along z axis, the streaming + registers version will incur high register pressure. There is also some redundancy in the values stored in the registers of neighboring threads. For a 27-point 3D stencil with $T = 2$, the number of registers needed per thread is 36, which will inadvertently be spilled if one intends to maximize occupancy.

If such a stencil is associative, we propose an optimization that reduces the number of registers required at each time step to just 1 for each plane accessed, bringing down the total number of registers per thread to $2Tk+1$. An example of such a stencil is shown in Listing 1. We exploit the associativity of addition and multiplication to convert the stencil into an accumulation stencil (Listing 2).

Listing 1: A 7-point 2D associative stencil that accesses 2 points from planes along y axis

```
1 for (y=1; y<N-1; y++) {
2   for (x=1; x<N-1; x++) {
3     B(y,x) = c0*(A(y-1,x-2) + A(y-1,x+2)) +
4              c1*(A(y,x-1) + A(y,x) + A(y,x+1)) +
5                c2*(A(y+1,x-2) + A(y+1,x+2));
6   }
7 }
```

Listing 2: Reading one plane from input at a time, and accumulating its contribution at different output points

```
1 for (y=0; y<N; y++) {
2   for (x=0; x<N; x++) {
3     B(y+1,x) += c0*(A(y,x-2) + A(y,x+2));
4     B(y,x) += c1*(A(y,x-1) + A(y,x) + A(y,x+1));
5     B(y-1,x) += c2*(A(y,x-2) + A(y,x+2));
6   }
7 }
```

Figure 6 shows the time tiling using accumulative registers for associative stencils. Unlike the tiling scheme of 5, that used register to cache input values, the tiling scheme shown in Figure 6 uses registers to accumulate the output values. The output value for planes z_0+1 and z_0 are accumulated in registers $r_p[T]$ and $r_c[T]$, and the output value for plane z_0-1 is accumulated in shared memory buffer $A[T]$. In each iteration i, the input plane z_i is read into $A[0]$. From it, the contributions to output plane z_{i-1}, z_i, and z_{i+1} are accumulated in $r_p[1], r_c[1]$, and $A[1]$ at time step 1. At all the remaining time steps, we repeat the same process of accumulating output values using $A[t-1]$ as the input plane. After T time steps, all the contributions to plane z_{i-T} would

have been accumulated in $A[T]$. After each iteration, we shift the data ($r_p \rightarrow r_c \rightarrow A$), freeing r_p to accumulate the contributions for next output plane.

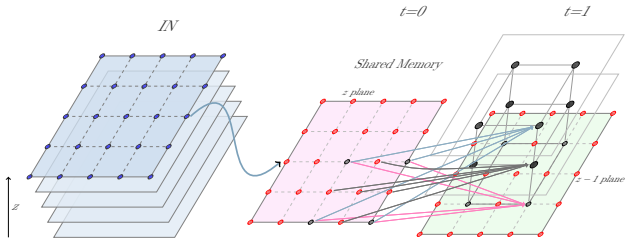


Figure 6: Minimizing register pressure for associative stencils

Algorithm 3 presents an implementation sketch for this method. An optimized implementation of the algorithm uses only $2Tk+1$ registers and T shared memory buffers.

Algorithm 3: streaming + registers algorithm for a 3D order-1 associative stencil

Input : IN : input array,
T : time tile size,
(B_x, B_y) : block size,
(tid_x, tid_y) : thread index,
A_t : sh-mem buffer for time step t,
$r_p[t]$ & $r_c[t]$: registers storing $z+1$ and z planes at time step t
Output: OUT : output array

$i_x = \ldots, i_y = \ldots, i_z = 1$;
// Initialization
for *each* t *from* 1 *in* T **do**
  $A_t[0..B_y][0..B_x] \leftarrow 0$;
  $r_c[t] = 0$;
end
for *each* i_z *from* 0 *to* $N-1$ **do**
  $A_0[0..B_y][0..B_x] \leftarrow \text{IN}[i_z][i_y..i_y+B_y][i_x..i_x+B_x]$;
  __syncthreads ();
  for t *from 1 to* T **do**
    // Scatter contributions from A_{t-1}
    $r_p[t]$ = *bottom_plane_contrib* (A_{t-1});
    $r_c[t]$ += *mid_plane_contrib* (A_{t-1});
    $A_t[tid_y][tid_x]$ += *top_plane_contrib* (A_{t-1});
    __syncthreads ();
  end
  $\text{OUT}[i_z - kT][i_y + tid_y][i_x + tid_x] = A_t[idy][idx]$;
  __syncthreads ();
  // Shift data
  for t *from 1 to* T **do**
    $A_t[[0..B_y][0..B_x] = r_c[t]$;
    $r_c[t] = r_p[t]$;
  end
end

5. EXPERIMENTAL EVALUATION

Experimental setup. To evaluate the performance of the tiling schemes discussed in Section 4, we compare the performance of their CUDA implementation against PPCG-0.04 [13], Overtile-0.3.2 [5], and Forma [11]. Results have been generated using Tesla K20c GPU (SP peak of 3.52 TF/s, all results below use SP floating point computations) for the benchmarks summarized in Table 1. All the generated code was compiled using NVCC 7.0[1] with compilation flags '`-use_fast_math Xptxas "-v -dlcm=cg" -maxrregcount=32`'. In our implementations, we fuse all the conditional statements, and replace logical operators with bitwise operators to avoid thread divergence.

Benchmark	Domain	T	Loads/ point	Flops
jacobi-2d-5pt *(j2d5pt)*	8192^2	4	5	10
GoL-2d-9pt *(GoL-9pt)*	8192^2	4	9	18
jacobi-2d-9pt *(j2d9pt)*	8192^2	4	9	18
gaussian-2d-25pt *(gaussian)*	8192^2	4	25	50
gradient-2d-5pt *(gradient)*	8192^2	4	5	18
jacobi-3d-7pt *(j3d7pt)*	512^3	4	7	12
jacobi-3d-13pt *(j3d13pt)*	512^3	4	13	25
jacobi-3d-17pt *(j3d17pt)*	512^3	4	19	28
jacobi-3d-27pt *(j3d27pt)*	512^3	4	27	54
curl-3d *(curl)*	450^3	2	16	30

Table 1: List of 2D and 3D benchmarks, and their features

Impact of register optimization. For a (128×1) 1D block, Table 2 highlights the interplay of hardware constraints and tiling strategies, as discussed in Section 4.1. With 1D overlapped tiling using just shared memory as storage, the occupancy with serial-streaming is particularly low due to high shared memory pressure, and low concurrency.

Using 2D overlapped tiling and concurrent-streaming helps increase the concurrency, but the shared memory pressure still limits the occupancy to below 50%. For *j2d5pt* stencil, each block consumes $6KB$ of shared memory. So irrespective of concurrency, an SM can only have at most 8 active blocks, resulting in occupancy of 0.48. The problem is exacerbated for *j2d9pt*, since it needs $10KB$ shared memory per block. Each SM can schedule only 4 blocks concurrently, resulting in 25% occupancy.

	only shared memory		shared memory+registers	
	1D overlap serial-stream	2D overlap concurrent-stream	1D overlap serial-stream	2D overlap concurrent-stream
j2d5pt	17.7 (0.33)	14.2 (0.48)	12.3 (0.34)	6.7 (0.92)
GoL-9pt	22.7 (0.33)	18.7 (0.48)	14.8 (0.32)	8.1 (0.92)
j2d9pt	43.1 (0.18)	31.4 (0.24)	17.3 (0.35)	10.0 (0.95)
gaussian	65.7 (0.19)	52.1 (0.24)	16.0 (0.34)	12.1 (0.96)
gradient	18.1 (0.32)	14.1 (0.48)	13.7 (0.35)	7.07 (0.91)

Table 2: Time in ms, and occupancy (displayed within parenthesis) for overlapped tiling with 1D blocks of size 128 on different 2D stencils

Once we alleviate shared memory pressure using registers, we see immediate benefits for order-2 stencils, since the shared memory per block reduces by $8KB$. The occupancy for all the stencils now is only limited by concurrency, so we opt for 2D overlapped tiling with concurrent-streaming. Using 2D overlapped tiling + registers along with shared memory + concurrent-streaming helps overcome both concurrency and occupancy constraints.

[1]http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc

To analyze the benefit of using registers for storage in 3D stencils, we compare the performance of a version that uses only shared memory for storage against one that uses both shared memory and registers. The tile size is fixed to 32×32 wherever feasible. The results are presented in Table 3.

	only shared memory		shared memory + registers	
	T=4	T=2	T=4	T=2
j3d7pt	75.1 (0.49)	43.5 (0.99)	29.7 (0.99)	28.9 (0.99)
j3d13pt	617 (0.28)	101 (0.49)	86.0 (0.99)	55.1 (0.99)
j3d17pt	114 (0.49)	73.7 (0.99)	51.1 (0.99)	50.7 (0.99)
j3d27pt	153 (0.49)	101 (0.99)	52.7 (0.99)	52.2 (0.99)

Table 3: Time in ms, and occupancy (displayed within parenthesis) for tiling 3D stencils with different resource management schemes

For all the order-1 stencils, $3T$ planes must be in cache for tiling across T time steps. If we only use shared memory, then one block will need $48KB$ shared memory for $T = 4$, reducing the occupancy to 50%. For $T = 2$, exactly two blocks can be active per SM, achieving 100% occupancy. However, using registers brings down the per block shared memory requirement to T planes. For *j3d7pt*, even with 100% occupancy, the version that uses registers along with shared memory achieves a 1.5x speed-up over the version that only uses shared memory. This benefit comes from the fast access to registers during the computation. For order-2 *j3d13pt*, the number of planes that need to be cached is $5T$. When using only registers, a single block requires $40KB$ for $T = 2$. This effectively means a 50% drop in occupancy. The shared memory requirement for $T = 4$ forces the tile size to be reduced to 24×24. The number of active threads per SM will be $\frac{24 \times 24}{2048} = 0.28$, which is the achieved occupancy. The performance of the tiled code deteriorates to $617ms$ due to low occupancy, and a high volume of redundant computation resulting from a smaller block size.

From the results in Table 3, we conclude that apart from the performance advantage of register accesses over shared memory accesses, the additional level of storage provided by registers is crucial for an efficient 3D tiling algorithm.

Comparative performance results. Forma untiled is the naïve untiled code generated by Forma compiler. K20c can load the read-only data through the cache used by texture pipeline. This feature can be enabled by simply annotating the read-only data with `__restrict__` keyword. We annotate the naïve code generated by Forma to get the Forma untiled + annotated version. Reading through texture cache proves beneficial for tiling on Kepler device, so we annotated the input and output arrays with keyword `__restrict__` for all the benchmarks except the baseline code (PPCG and Forma untiled).

For our tiling implementation, overlapped+stream+shmem-opt refers to 1D overlapped tiling that just uses shared memory along with optimized serial-streaming described in Section 4.1. The version overlapped+stream+shmem+regs refers to the 2D overlapped tiling version that uses both registers and shared memory as buffers along with streaming (serial-stream for 3D benchmarks, concurrent-stream for 2D benchmarks).

PPCG performs classical time tiling along with default 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 [14]. For Overtile compiler, the degree of coarsening has to be specified as a part of the DSL code. The performance numbers for Overtile compiler in Figure 7 were measured by coarsening the slowest varying dimension by a factor of 2, which consistently performed better than the generated code with no coarsening. With coarsening, Overtile code outperforms Forma's overlapped tiling code for all benchmarks. We could not generate the correct version of 3D curl benchmark using Overtile compiler.

From the performance numbers, we draw the following conclusions: (1) One has to choose a combination of varying optimizations that overcome the performance-limiting hardware constraints depending on the dimensionality of the problem. (2) Using registers along with shared memory as buffers is crucial to performance irrespective of the dimensionality. (3) Serial-streaming for 3D stencils and concurrent-streaming for 2D stencils reduces the extra bandwidth consumption without constraining concurrency.

Our optimization strategy for associative stencils helps achieve peak performance (837 GFlops for *gaussian-2d*, and 604 GFlops for 27-point 3D stencil). For 2D stencils, best performance was achieved by overlapped tiling along x and y, and concurrent-streaming along y. For 3D stencils, the favorable choice was overlapped tiling along x and y, and serial-streaming along z.

6. RELATED WORK

Automated high-performance GPU code generation for stencils is a topic of active research [3, 5, 4, 12]. These recent efforts cover a range of tiling approaches: overlapped tiling [5, 12], split tiling [4], and hexagonal tiling [3], all enabling concurrent execution of tiles, which is essential for maximum utilization of the GPU resources. PPCG [13] is a polyhedral-model based source-to-source compiler that generates classically time tiled OpenCL and CUDA code from an annotated sequential program. In contrast to our approach presented in this paper, none of these code generators utilize streaming to reduce bandwidth.

Another direction of research is autotuning of stencil applications. Datta et al. [2] investigate the performance of a 7-point 3D stencil on NVIDIA GTX280. Their implementation uses streaming along the slowest varying dimension. Zhang and Mueller [15] evaluate an autotuner for 3D stencils on GPU clusters. Patus [1] and Halide [10] decouple algorithm specification from schedule, and then rely on autotuning to achieve efficiency. Halide schedules can express spatial tiling and sliding-window optimizations to reduce memory footprint. However, writing an efficient schedule manually is tedious, and stochastically exploring the schedule space while autotuning can be time consuming. Additionally, explicit use of registers for storage cannot be expressed in Halide.

Micikevicius [7] developed a CUDA implementation of 3D finite difference computation. This implementation performs spatial tiling, and uses registers to alleviate shared memory pressure. In contrast to the limited "cross" stencils from 3D finite difference computations where registers could be used, our approach to use of GPU registers for reducing shared-memory requirements applies to a much broader

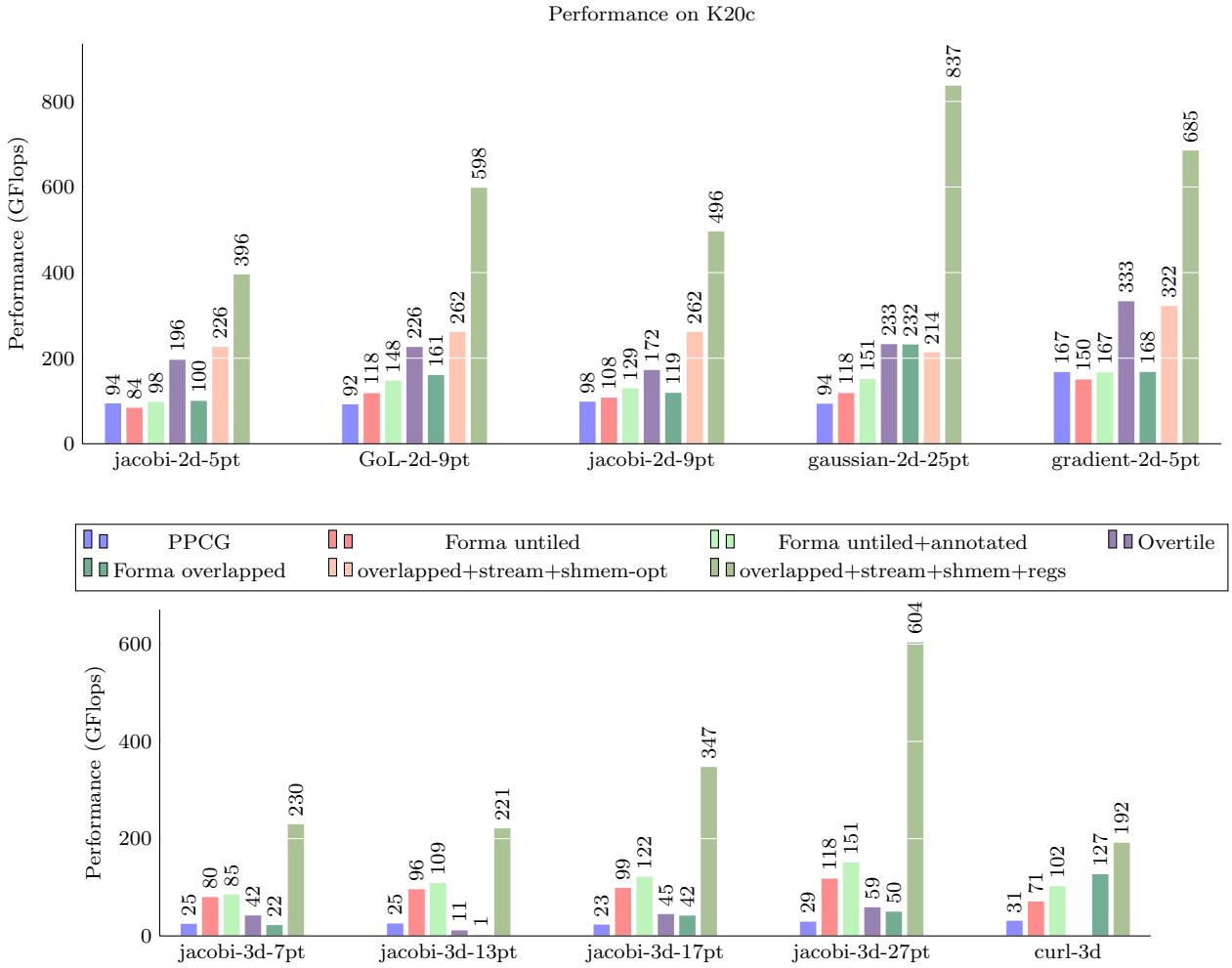


Figure 7: Performance results for 2D and 3D stencils

class of stencils. Nguyen et al. [9] use streaming with time-tiling for the Intel Xeon Phi and NVIDIA GTX 285. While the approach we present in this paper also uses streaming with time tiling, our work focuses extensively on a broader range of stencil computations on GPUs with different optimization constraints. We also use concurrent streaming in conjunction with overlapped tiling along a streaming dimension when the available parallelism from other dimensions is insufficient for the number of SMs in the GPU (often the case for 2D stencils). Our approach to optimizing register usage via associative reordering of stencil operations is also very different from their approach to tiling.

In this paper, we have presented direct performance comparisons with Overtile [5], Forma [11] and PPCG [13]. We have not been able to directly compare performance with the hybrid hexagonal tiling approach of Grosser et al. [3] since the system is not publicly available. However, we can perform some indirect comparisons through reported performance improvements on common benchmarks over Overtile [5]. For Jacobi-2d-5pt (called Laplacian-2d by Grosser et al. [3]), a speedup of 1.4 over Overtile is reported on a GTX 470 GPU, while we show a speedup of 2.0. For Jacobi-2d-9pt (called Heat-2D by Grosser et al. [3]), their speedup over Overtile is 2.17, while ours is 2.88; for Jacobi-3d-7pt, their speedup over Overtile is 1.3, while our speedup over Overtile is 5.0; for Jacobi-3d-27pt, their speedup is 1.5, while ours is over 10.

7. CONCLUSION AND FUTURE WORK

In this paper, we evaluate a combination of overlapped tiling and streaming to better utilize GPU resources, specifically registers and shared memory. By systematically working through the various constraints of the modern GPU hardware, we demonstrated that a combination of these approaches, along with some practical implementation level optimizations can deliver significant performance improvements for 3D stencil computations. Such computations have been a stumbling block for many existing tools. The techniques discussed in this work can be used by application developers interested in 3D stencil computations, or can be integrated into DSL compilers like Forma and PolyMage [8] to auto-generate efficient time tiling code.

8. 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, and the Department of Energy through award DE-SC0008844.

9. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] M. Ravishankar, P. Micikevicius, and V. Grover. Fusing convolution kernels through tiling. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 43–48. ACM, 2015.

[13] 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.

[14] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11. IEEE Press, 2008.

[15] 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.