Optimizing Stencil Computations for NVIDIA Kepler GPUs

Naoya Maruyama
RIKEN Advanced Institute for Computational Science
Kobe, Japan
nmaruyama@riken.jp

Takayuki Aoki
Tokyo Institute of Technology
Tokyo, Japan
taoki@gsic.titech.ac.jp

ABSTRACT

We present a series of optimization techniques for stencil computations on NVIDIA Kepler GPUs. Stencil computations with regular grids had been ported to the older generations of NVIDIA GPUs with significant performance improvements thanks to the higher memory bandwidth than conventional CPU-only systems. However, because of the architectural changes introduced with the latest generation of the GPU architecture, Kepler, we show that existing implementation strategies used for such older GPUs are not as effective on Kepler as before. To fully exploit the potential performance of the latest generation of the GPU architecture, our implementation method uses shared memory for better data locality combined with warp specialization for higher instruction throughput. Our method achieves approximately 80% of the estimated peak performance by the roofline model, and even higher performance with temporal blocking.

1. INTRODUCTION

GPU accelerators are increasingly used in various fields of scientific simulations because of superior performance and power efficiency. The peak theoretical memory bandwidth of a single NVIDIA Kepler K20X GPU reaches up to 250 GB/s. Such high memory system performances are particularly important for many of stencil kernels, which are typically memory intensive. In fact, past studies reported that stencil computations with regular Cartesian grids can be ported to GPU with significant performance acceleration compared to conventional CPU-only systems [10, 11].

Effective optimization techniques for stencils include locality optimization by loop blocking. On GPUs, there are several different on-chip memories that could be used for blocking. Shared memory is a programmable memory that can be accessed with very low latency, and has been used for blocking in many past studies [1, 6]. It was particularly important to optimize GPU programs with shared memory blocking when using older generations of NVIDIA GPUs; however, we have also observed that such manual optimizations are not always necessary with recent GPU architecture called Fermi, where DRAM loads are cached at L1 cache. Although GPU cache size is typically very small, it is often very effective for regular data access patterns in stencils with Cartesian grids, allowing good performance even without manual shared memory blocking [11].

One of the major architectural changes in Kepler, the most recently released GPU architecture by NVIDIA, is that L1 cache is not used for DRAM load caching, but only used for register spilling [9]. Therefore, stencil kernels written for Fermi GPUs with an assumption that redundant localized data accesses are automatically cached at L1 cache may not perform as efficiently on Kepler as on Fermi. Since the new architecture also introduced a number of major and minor changes in hardware configurations, it is important to examine stencil kernel performance on Kepler GPUs with particular focus on memory access optimizations.

This paper evaluates the performance of a 7-point 3-D stencil on Fermi and Kepler with a series of memory access optimizations.[1] We first begin with a baseline program that is written in a straightforward way, and apply loop blocking at registers and shared memory. We also extend the stencil with temporal blocking for further saving DRAM accesses [4, 7, 12, 14, 15]. While these manual blocking optimizations with shared memory should in theory improve performance in memory-bound stencil computations, its effectiveness may by limited by the constraints imposed by the GPU compute architecture and limited on-chip memory capacity. Although stencils with Cartesian grids mostly consist of regular computation patterns, irregular processing of halo regions can have large performance impacts due to the wide-vector execution model. In fact, several past studies reported that temporal blocking is not effective for 3-D stencil problems [4, 15]. We study several implementation techniques for shard memory blocking, including those using texture memory and warp specialization, and attempt to answer the following questions:

- How does stencil programs optimized for Fermi perform on Kepler?
- How much performance gain can be obtained with shared memory blocking?
- How close to peak performance can be obtained in stencil computations on GPUs?

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

[1]All program code is available for download at http://github.com/naoyam.

```
 1  // trip count of the time loop
 2  int count;
 3  // 3-D arrays of size nx*ny*nz
 4  float f1[nx,ny,nz];
 5  float f2[nx,ny,nz];
 6  // coefficient variables
 7  float cc, cw, ce, cs, cn, cb, ct;
 8
 9  // Time loop executing for count iterations
10  repeat count times
11    // update all grid points from f1 to f2
12    for (x,y,z) in nx*ny*nz
13      // 7-point stencil where accessing exterior
14      // points is replaced with nearby boundary points
15      int w = (x == 0)    ? 0 : -1;
16      int e = (x == nx-1) ? 0 : 1;
17      int n = (y == 0)    ? 0 : -1;
18      int s = (y == ny-1) ? 0 : 1;
19      int b = (z == 0)    ? 0 : -1;
20      int t = (z == nz-1) ? 0 : 1;
21      f2[x,y,z] = cc*f1[x,y,z] + cw*f1[x+w,y,z]
22          + ce*f1[x+e,y,z] + cs*f1[x,y+s,z]
23          + cn*f1[x,y,z+n] + cb*f1[x,y,z+b]
24          + ct*f1[x,y,z+t];
25    swap f1, f2;
26    end for
27  end repeat
```

Figure 1: Pseudo code for the 7-point diffusion stencil.

Our performance studies using both Fermi and Kepler GPUs reveal that the shared memory blocking with warp specialization is highly effective in achieving optimal performance on Kepler, but not on Fermi. Common implementation techniques to use the shared memory, such as [6] and [10], however, is shown to be less effective. We also show that temporal blocking with warp specialization can further improve the performance. Overall, our optimized implementations achieve approximately 80% of the estimated peak performance by the roofline model, and even higher performance with temporal blocking on Kepler. While the studies in this paper are limited to a 7-point stencil code, we believe that our findings would give a useful guideline for optimizing other stencils for NVIDIA GPUs.

2. TARGET STENCIL

As an example of stencil, we use a simple 7-point stencil that computes a diffusion equation on 3-D domains of single-precision floating-point data. Figure 1 shows its pseudo code. Notice that each point is updated by accessing six nearest neighbor points as well as its own previous value (i.e., `f1[x,y,z]`) with an exception for boundary points. For each boundary point, neighbor access that falls outside the defined grid area is replaced by its own value (lines 15-20). In this paper, we evaluate and optimize the performance of this stencil on a single GPU device.

The performance of the stencil for a given architecture can be estimated based on the roofline model [13]. As shown in the pseudo code, an update of a single point requires 13 single-precision FP operations, so each iteration performs $13 \times \texttt{nx} \times \texttt{ny} \times \texttt{nz}$ operations. To simplify modeling of DRAM accesses, we assume that for each iteration once array points are loaded from memory, they remain on cache for that iteration. This is an optimistic assumption for problem sizes larger than the last-level cache size, which is typically the case with reasonable problem sizes for realistic computational fluid simulations. Based on the assumption, the size of data loaded and stored per iteration is modeled as $\texttt{nx} \times \texttt{ny} \times \texttt{nz} \times \texttt{sizeof(float)} * 2$ bytes. Thus, the compute intensity, defined as flop per DRAM byte, is calculated as $\frac{13 \times \texttt{nx} \times \texttt{ny} \times \texttt{nz}}{\texttt{nx} \times \texttt{ny} \times \texttt{nz} \times \texttt{sizeof(float)} * 2} = 1.625$.

The intensity clearly indicates that the performance of the stencil on a GPU is bounded by its DRAM bandwidth. For example, the performance of the latest NVIDIA GPU Kepler is as high as 4 TFLOPS (single precision) [9], whereas the theoretical peak memory bandwidth is 250 GB/s. Optimizing memory accesses, therefore, is the most important strategy in improving the performance of the stencil. In particular, although the above modeling assumes the perfect data locality that all grid points in array `f1` are loaded from DRAM at most once and that subsequent accesses are from the cache, the constraints imposed by the GPU architecture require non-trivial tuning as presented in this paper.

3. OVERVIEW OF KEPLER GPU ARCHITECTURE

This section gives a brief overview of the NVIDIA Tesla Kepler GPU [9], which is the main target of our study in this paper. We particularly focus on the differences between Kepler and the previous generation of NVIDIA GPU architecture called Fermi. Since the specific Kepler GPU used in this paper is Tesla K20X, we first present a brief overview of K20X, and explain how memory accesses in CUDA are mapped to actual data movements in the architecture.

The K20X GPU consists of a GK110 processor equipped with 6 GB of GDDR5 memory. The GK110 processor in K20X consists of 14 streaming multiprocessors (SM), each of which consists of 192 CUDA cores clocked at 732 MHz, achieving 3.95 TFLOPS in single-precision peak performance. Each SM has an on-chip memory area of 64 KB that can be accessed both explicitly as shared memory and implicitly as L1 cache. It also has a register file of 256 KB, which is doubled from Fermi. In addition to these memories, the GK110 processor is equipped with a read-only cache of 48 KB per SM, which is an extension of the texture cache available in the older generations of GPUs, but becomes more easily accessible in Kepler. Furthermore, 1.5 MB of L2 cache is shared by all 14 SMs.

Each memory access to CUDA global memory is first serviced by the off-chip GDDR memory. The same coalescing rule as enforced in Fermi is still applicable in Kepler [8]. However, a significant change in memory accesses is that global memory loads and stores are not cached in the L1 cache on Kepler, whereas on Fermi load accesses are cached by default. We have observed that the Fermi L1 cache is often very effective for reducing the DRAM bandwidth pressure in stencil computations with regular grids, making explicit blocking with the shared memory almost unnecessary in such computations. In fact, our optimized CUDA implementation of a phase-field simulation application [11], which won a 2011 Gordon Bell Award, did not use shared memory since we saw no performance benefit on Fermi GPUs. The same implementation, however, would not perform as efficiently on Kepler as on Fermi since global memory loads are no longer cached.

Instead of relying on automatic caching with the L1 cache, the shared memory and read-only cache can be used on Kepler, both of which require code modifications. In CUDA,

```
__global__ void baseline(float *f1, float *f2,
                         int nx, int ny, int nz,
                         float ce, float cw, float cn,
                         float cs, float ct, float cb,
                         float cc) {
  int xy = nx * ny;
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  int j = blockDim.y * blockIdx.y + threadIdx.y;
  const int block_z = nz / gridDim.z;
  int k = block_z * blockIdx.z;
  const int k_end = k + block_z;
  int c = i + j * nx + k * xy;
#pragma unroll
  for (; k < k_end; ++k) {
    int w = (i == 0)        ? c : c - 1;
    int e = (i == nx-1)     ? c : c + 1;
    int n = (j == 0)        ? c : c - nx;
    int s = (j == ny-1)     ? c : c + nx;
    int b = (k == 0)        ? c : c - xy;
    int t = (k == nz-1)     ? c : c + xy;
    f2[c] = cc * f1[c] + cw * f1[w] + ce * f1[e] + cs * f1[s]
        + cn * f1[n] + cb * f1[b] + ct * f1[t];
    c += xy;
  }
  return;
}
```

Figure 2: Baseline stencil kernel.

the shared memory can be explicitly used as a scratchpad memory by annotating arrays with the `__shared__` attribute. The read-only cache can be explicitly used by the `__ldg` intrinsic or the CUDA compiler automatically uses the read-only cache when arrays are declared with the `const` and `__restrict__` annotations. We present our optimization techniques using those memories in Section 5.

4. BASELINE IMPLEMENTATION AND PERFORMANCE

A straightforward CUDA implementation of the stencil is shown in Figure 2. We use this code as the baseline for subsequent performance evaluations and optimizations. It partitions the 3-D domains of size $\texttt{nx} \times \texttt{ny} \times \texttt{nz}$ into sub domains of $\texttt{blockDim.x} \times \texttt{blockDim.y} \times \texttt{nz/gridDim.z}$, each of which is computed by a CUDA thread block. Note that we assume that the z dimension is the slowest varying dimension. As shown in the code, the grid points in the x-y planes are computed in parallel by the threads in a thread block, whereas the computation over the z-direction is swept sequentially.

The performance of the code on Fermi M2075 and Kepler K20X GPUs are shown in Figure 3. The size of the grid is 256^3. The CUDA version used in our experiments is version 5.0 on Linux kernel 2.6.32 (CentOS 6.4). We only evaluate the kernel execution time excluding the PCI data transfer cost. We configure the L1 cache and shared memory partitioning to 48 KB and 16 KB, respectively. The measurement is done five times and the fastest performance is reported. In addition to the baseline performance, the figure shows the upper bound performance that is estimated with the roofline performance model. The measured bandwidth of the Fermi and Kepler GPUs are 103 GB/s and 170 GB/s, respectively. As discussed in Section 2, the compute intensity of the kernel is 1.625, indicating the peak attainable performances are 167 GLFOPS and 276 GFLOPS, respectively.

As shown in the graph, the actual performances of the baseline version has relatively large gap compared to the

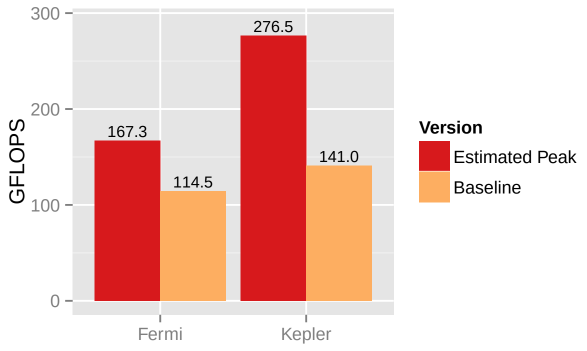

Figure 3: Performance of the baseline implementation on Fermi M2075 and Kepler K20X.

model-estimated peak performances especially on Kepler. The ratios against the peak for Fermi and Kepler are 66.7% and 49.7%, respectively.

5. OPTIMIZATIONS

The large gap with the peak performance indicates that the baseline kernel still has a large room for improvements. This section presents a series of optimizations focused on minimizing off-chip memory accesses. Specifically, we first explain a set of basic optimizations, including loop peeling and register blocking. Next, we apply spatial and temporal blocking to exploit the on-chip memories available on the GPU.

5.1 Basic Optimizations

We optimize the computation by moving the conditional operations out of the loop. Notice that the conditional evaluations on variables `i` and `j` are loop independent, so they can be safely moved before the loop. The conditional operations on variable `k` are loop dependent, but peeling the first and last iterations allows us to remove the conditional operations from the loop inside. It also uses registers to cache the points along the Z dimension since they are reused locally by each thread [2].

5.2 Spatial Blocking with Read-Only Cache

To use the read-only cache on Kepler, we add `const` and `__restrict__` annotations to the input grid parameter declaration and `__restrict__` to the output parameter. This change should not affect actual native code for Fermi, but when compiled for Kepler the compiler should generate code that uses the read-only cache for all loads from the input grid.

5.3 Spatial Blocking with Shared Memory

Another well-known opportunity for reducing the DRAM pressure is explicit blocking using the shared memory. As previously presented by, e.g., [6, 10], each thread block allocates a 2-D chunk of shared memory for its corresponding sub domain and its halo data. Every thread stores its point value to the shared memory, and the boundary threads that are located on the boundary of the sub domain are also responsible for loading the halo data from the global memory to the shared memory. Since the data reuse between threads in a thread block only exists for the single X-Y plane with the z offset being zero, we only need one 2D plane of $(\mathtt{bx}+2)\times(\mathtt{by}+2)$ in the shared memory, where `bx` and `by` represent the dimension of a sub domain. We also change the configuration of shared memory and L1 cache sizes as 48 KB and 16 KB, respectively.

A potential problem with this implementation is that loading halo data needs conditional operations, which are executed every iteration. In particular, the accesses to the Y-direction halo points cause branch divergence since only the two boundary threads among their warps are involved. Therefore, although it appears that only those boundary threads issue the global memory load and shared memory store instructions, the execution overhead can be significant because every thread in the same warp also issue the same instructions. In fact, as shown in Section 6, this version turned out to be less efficient than the above versions.

Phillips and Fatica presented an optimization method that does not use conditional operations by exploiting the texture memory [10]. All threads in a thread block redundantly reads the global memory four times, each of which is diagonally shifted to account for the halo points. The cost of increased number of read transactions and mis-aligned accesses is minimized by using the texture memory. We also evaluate this method with our stencil on Fermi and Kepler.

5.4 Spatial Blocking with Warp Specialization

We further extend the shared memory optimization by using a programming technique called warp specialization [1]. Since warp divergence does not happen if all threads in a warp take the same control flow, the overhead by the conditional operations and branch divergence can be minimized by a careful assignment of warps based on control flows as described below.

We first divide the execution of a thread block to three control flows: one for the interior points, another for the Y-direction boundaries, and another for the X-direction boundaries. This is realized in our CUDA kernel by creating three disjoint code paths that correspond to the three flows. The conditional branch to select each path is done only once at the beginning of the kernel, so the cost of branches is eliminated. Furthermore, since the warps for the interior points are not responsible for loading the X-direction halo points, the branch divergence caused in the previous version is completely eliminated.

A drawback of the warp specialization is the increased number of threads per thread block, which has two performance implications. First, it requires a larger number of registers per thread block, which can adversely affect the number of active threads running on each SM. Since latency hiding by a large number of active threads is one of the most important optimizations for achieving high memory throughput, increase of the register usage can result in lower overall performance. Second, it could also increase the cost of thread synchronization within a thread block. Since the blocking with the shared memory needs two times of thread synchronization per iteration, the increased cost of thread synchronization can also reduce the overall performance.

Another drawback of this approach is the increased cost of code development. Even though some part of the separate warp groups perform common operations,

5.5 Temporal Blocking with Shared Memory

Finally, we study the effectiveness of temporal blocking using the shared memory [3, 4, 7, 12, 14, 15]. It has been shown to be particularly effective for bandwidth-bound computations such as stencils.

Similar to the shared memory version, we use warp specialization to minimize the cost of halo processing. In this present work, we only focus on two-way blocking; further aggressive blocking remains to be studied. Also for simplicity we use overlapped tiling [4] that incurs redundant overlapped grid points to update for saving DRAM memory accesses. We speculate that the increased compute cost due to overlapped tiling can be justified on GPUs, which has very high flops and bandwidth ratios.

Figure 4 illustrates a flow of each thread block performing two time steps with only one global memory read and one write. The colors represent the groups of warps. Filled regions are areas that are updated by the stencil, while the dotted regions are only loaded from the global memory for updating neighbor points. As indicated in the figure, each thread block uses a 2-D plane of shared memory region of size $(\mathtt{bx}+4)\times(\mathtt{by}+2)$, which is updated in place once, and then stored back to global memory after another update.

As shown in the figure, we use four groups of warps: one for the interior points (blue region), one for each of the horizontal and vertical halo regions (red and yellow regions), and another for the four diagonal points (green region). The number of actual warps for each group depends on the actual size of the sub domain, although the green region always uses just one warp. Note that since only four points are accessed in the green region, only four threads are used among the 32 threads.

The overhead due to this blocking is two-fold. First, it increases the computation cost since additional $2(\mathtt{bx}\times\mathtt{by}+\mathtt{bx}\times\mathtt{bz}+\mathtt{by}\times\mathtt{bz})$ points need to be updated when performing two time steps for each of $\mathtt{bx}\times\mathtt{by}\times\mathtt{bz}$ sub domains. We expect that this overhead would be negligible given the low compute intensity of the stencil. Second, compared to the non temporal blocking version, it needs to load additional grid points in the dotted region. Assume that the blue region is perfectly aligned to a line-size boundary and that the horizontal dimension is the stride-one dimension. In this case, the accesses to the yellow dotted regions in fact has no additional cost compared to the non temporal-blocked version, since the dotted regions should also be covered the same cache lines as the filled region. Overall, the size of the data that are additionally loaded consists of the red region ($\mathtt{bx}\times\mathtt{bz}\times 2\times\mathtt{sizeof(float)}$ bytes), the bottom and top blue regions ($\mathtt{bx}\times\mathtt{by}\times 2\times\mathtt{sizeof(float)}$ bytes), and the green regions ($\mathtt{bz}\times 4\times 128$ bytes). Note that the cache line size in NVIDIA GPUs is 128 bytes. Therefore, the reduction of DRAM accesses can be estimated as: $0.5+\dfrac{\mathtt{bx}\times\mathtt{bz}+\mathtt{bx}\times\mathtt{by}+\mathtt{bz}\times 64}{\mathtt{bx}\times\mathtt{by}\times\mathtt{bz}\times 2}$.

6. EXPERIMENTAL EVALUATIONS

To evaluate the effectiveness of the stencil optimizations, we measure the performance of the stencil on Fermi M2075 and Kepler K20X GPUs. We use CUDA version 5.0 on Linux 2.6.32 (CentOS 6.4). Since this paper only focuses on intra-GPU performance optimizations, we only measure the GPU kernel performance, excluding the cost of PCI data transfers. The reported performance numbers are based on the measured timing that and the cost analysis presented in

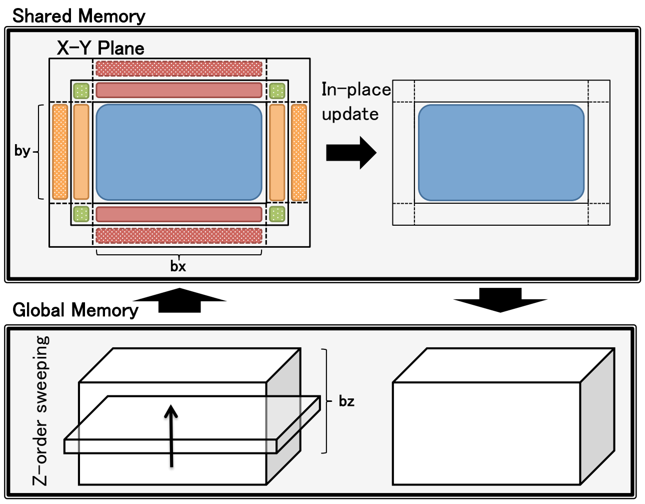

Figure 4: Temporal blocking execution flow. The filled regions are updated by the stencil, while the dotted regions are only read from global memory for updating neighbor points. Each color represent a group of warps for the same execution flow.

Table 1: Summary of employed blocking methods.

	Register	RO cache	Shared memory	Temporal
Baseline				
OPT	✓			
ROC	✓	✓		
SHM	✓		✓	
SHM-TEX	✓		✓	
SHM-WARP	✓		✓	
TEMP	✓		✓	✓

Section 2. We measure execution time of 100 time steps of each kernel for five times and use the fastest one to minimize system noise.

We evaluate five variations of optimized kernels. Specific blocking methods used in each kernel is summarized in Table 1. Note that the baseline version is the same as presented in Section 4. The Opt kernel uses the optimizations described in Section 5.1; the ROC kernel uses the read-only cache as described in Section 5.2; the SHM kernel uses the shared memory blocking described in Section 5.3; the SHM-TEX and SHM-WARP also use the shared memory blocking but uses the texture-based DRAM loads and warp specialization; the TEMP kernel is the one with the temporal blocking. The ROC kernel result is only reported for Kepler since the read-only cache is not available in Fermi GPUs. The sizes of CUDA thread block and grid of each kernel as well as its register and shared memory size are shown in Table 2. The block sizes are selected among several valid configurations by an empirical search.

Figures 5 and 6 show the performance of each version on Fermi and Kepler, respectively. The dashed line shows the possible peak performance estimated based on the roofline model discussed in Section 2. Note that it does not account for the DRAM transaction reduction by temporal blocking, it can be surpassed by the optimized version with temporal blocking. The best performances on the Fermi and Kepler GPUs are 131 GFLOPS and 287 GFLOPS, which are obtained with the OPT and TEMP kernels, respectively.

The results with the Fermi GPU shows that explicit blocking with shared memory does not yield performance improvements. Without temporal blocking, all explicit shared memory blocking kernels exhibited degraded performances by approximately 40%. The DRAM transaction saving by temporal blocking allowed the TEMP kernel to achieve much better performance than the SHM kernels; however, it was still slightly behind the kernel with no shared memory blocking. In fact, the limited effectiveness of shared memory blocking for stencils on the GPU match with our past experiences.

In contrast to the Fermi results, the shared memory optimizations were able to achieve performance improvements with varying degrees. The performance of the SHM-WARP and TEMP kernels were 232 GFLOPS and 287 GFLOPS, achieving the speedup of 1.65 and 2.05 times compared to the baseline. The reason of the better effectiveness of the SHM-WARP on Kepler still remains to be studied; however, we speculate that other architectural changes such as the increased number of registers and twice the size of L2 cache would allow for minimizing the cost of warp specialization. Further detailed studies will be reported in the future.

Although the temporal blocking version did achieve the better performance than the non temporal blocking versions, its improvement turned out not to be as large as expected. More specifically, the speedup of the TEMP over SHM-WARP kernels is only 1.23 times. Our preliminary analysis with CUDA Performance Profiler indicates that the cost of thread synchronization is the largest performance bottleneck. More detailed analysis is a subject of our ongoing study.

Another interesting finding is that the performance gap between the baseline and most optimized versions is much larger on Kepler than on Fermi, which implies that the importance of code optimizations for stencils is more profound. While achieving comparable performance as the TEMP kernel would be fairly challenging for more complex kernels than the simple stencil studied in this paper, the optimization techniques used in the ROC kernel are relatively straightforward to apply. In fact, our stencil code generation framework is already able to apply most of the ROC optimizations automatically [5].

7. RELATED WORK

Stencil computations with regular grids have been one of the extensively studied types of computations on GPUs because many of such computations can be improved by taking advantage of the GPU DRAM bandwidth. Micikevicius reported an implementation method of finite difference stencils in CUDA with shared memory blocking [6]. Phillips and Fatica reported a CUDA implementation of Jacobi kernel for multiple GPUs with MPI. They also presented blocking methods with the shared memory and texture memory. Datta et al. [2] studied stencil performances and optimizations on various multi-core and accelerator architectures. Shimokawabe et al. presented a highly scalable phase-field simulation based on stencil computations [11].

The most common optimizations used in these past studies are blocking at registers and shared memory. Shared memory blocking was particularly important in pre-Fermi generations of NVIDIA GPUs since they were not equipped with hardware cache memories. However, since the Fermi

Table 2: Details of kernel configurations and resource usage. The values of the TB and Z block columns show the empirically-found optimal 2-D thread-block size and the Z dimension of CUDA grid for each kernel. The number of registers and shared memory are per-thread and per-block resource usage, respectively.

Kernel	Fermi (SM 2.0)				Kepler (SM 3.5)			
	TB size	Z block	# registers	Shared memory (bytes)	TB size	Z block	# registers	Shared memory (bytes)
Baseline	128x1	4	23	0	128x1	16	28	0
OPT	128x1	8	28	0	128x1	8	30	0
ROC					128x1	8	30	0
SHM	128x2	32	27	2080	128x2	16	30	2080
SHM-TEX	64x2	256	32	1056	64x2	1	33	1056
SHM-WARP	64x4	128	27	1056	32x4	1	32	544
TEMP	32x8	16	24	1440	32x8	8	29	1440

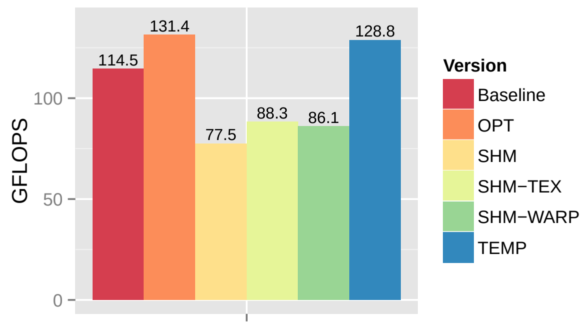

Figure 5: Performance of the baseline and optimized versions on Fermi M2075. The dashed line shows the peak attainable performance estimated based on the roofline model with no temporal blocking.

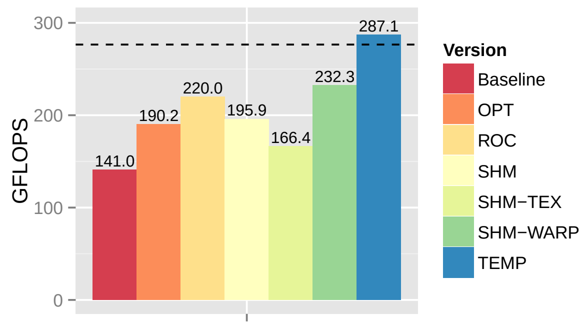

Figure 6: Performance of the baseline and optimized versions on Kepler K20X. The dashed line shows the peak attainable performance estimated based on the roofline model with no temporal blocking.

GPU can automatically cache global memory loads at L1 cache, we have observed that explicit blocking with shared memory is not always beneficial. Our experimental results reported in this paper also exhibit similar performance behavior. The latest generation of NVIDIA GPUs, Kepler, again changed the memory architecture and now global memory loads are not cached at L1, so different blocking methods need to be employed to efficiently run on the new architecture. This paper looked at several optimization methods and showed that simple implicit blocking with the read-only cache is indeed very effective, while explicit blocking with shared memory can further improve the performance.

Temporal blocking has also been extensively studied with various scientific computations such as stencils [12, 14]. It has also recently been evaluated on GPUs [3, 4, 7, 15], although mixed performance benefits have been reported. For example, for 3-D stencils, Zumbusch reported that it did not improve the performance on both Fermi and Kepler when compared to the version only with register blocking and no temporal blocking [15]. Holewinski et al. also reported that temporal blocking was not beneficial for 3-D Jacobi stencil [4]. In contrast to those previous results, our temporal blocking achieved an improvement by 20% with the 3-D stencil problem on the Kepler GPU.

Efficient halo accesses with warp specialization was first reported by Bauer et al. [1]. Our experiments in this paper confirm similar performance results on Kepler and show that further speedups can be attainable by combining warp specialization with temporal blocking.

8. CONCLUSION

This paper presented performance studies of a 7-point 3-D stencil on the recent NVIDIA GPU architectures. Our experimental evaluations show that the blocking with shared memory is essential for the stencil to achieve optimal performance on Kepler. This used to be the case for older pre-Fermi generations of NVIDIA GPUs, but had been considered unnecessary for stencils with regular grids since the hardware L1 cache of the Fermi GPU often works quite effectively. Overall, we achieved approximately 80% of the estimated peak performance by the roofline model, and even higher performance with temporal blocking on Kepler. While our current experiments are limited to the 7-point stencil, we expect that the results reported here will be applicable to other 3-D stencils as well.

The shared memory blocking optimizations, however, incurred non-trivial cost of code transformation. For example, while the original stencil can be written with less then 20 lines of code in non optimized CUDA, the fully optimized kernel takes more than 300 lines of code, resulting in 10-times increase of code size. Since manual applications to full-scale stencil applications that consist of, e.g., tens of much larger stencil kernels would be prohibitively costly, automated transformation techniques for these optimizations need to be developed. We plan to extend our high-level stencil framework, Physis, to automatically generate CUDA codes with the optimizations shown to be effective in this study [5].

Acknowledgments

This project was partially supported by a JST, CREST program: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing."

References

[1] M. Bauer, H. Cook, and B. Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM, 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 autotuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, Piscataway, NJ, USA, 2008. IEEE Press.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] NVIDIA. CUDA C Programming Guide version 5.0, 2013.

[9] NVIDIA. NVIDIA Kepler GK110 Architecture Whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2013.

[10] E. Phillips and M. Fatica. Implementing the Himeno Benchmark with CUDA on GPU Clusters. In *IEEE International Parallel & Distributed Processing Symposium*, pages 1–10, Apr. 2010.

[11] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM, 2011.

[12] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C. K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128. ACM, 2011.

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

[14] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 171–180. IEEE, 2000.

[15] G. Zumbusch. Vectorized Higher Order Finite Difference Kernels. In *State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2012.