Modeling the Performance of 2.5D Blocking of 3D Stencil Code on GPUs

Guangwei Zhang
Department of Computer Science
Xi'an Jiaotong University
Key Laboratory of Modern Teaching Technology
Shaanxi Normal University Xi'an, Shaanxi, China
Email: zgw1982@gmail.com

Yinliang Zhao
Department of Computer Science
Xi'an Jiaotong University
Xi'an, Shaanxi, China
Email: zhaoy@xjtu.edu.cn

***Abstract*—The performance of stencil computations can be improved significantly by using GPUs. In particular, 3D stencils are known to benefit from the 2.5D blocking optimization, which reduces the required global memory bandwidth of the stencils and is critical to attaining high performance on GPU. Using four different GPU implementations of a 3D stencil, this paper studies the performance implications of combining 2.5D blocking with different memory placement strategies, including using global memory only, shared memory only, register files only, and a hybrid strategy that uses all layers of the memories. Based on static analysis of the stencil data access patterns, we additionally develop heuristics to reduce tuning time of thread configurations of the various implementations to attain the highest performance.**

***Index Terms*—Stencil, Data placement, Tuning, 2.5D blocking, GPUs**

I. Introduction

Stencil computations represent one of the most important classes of kernels and are widely used in application areas such as image processing, data mining, and physical simulation. For example, Figure 1 shows a 3D stencil kernel that computes the following heat equation,

$$p^{new}_{i,j,k} = \alpha * p^{old}_{i,j,k} + (p^{old}_{i\pm1,j,k} + p^{old}_{i,j\pm1,k} + p^{old}_{i,j,k\pm1})$$
$$(0 < i < nx - 1, 0 < j < ny - 1, 0 < k < nz - 1)$$

$p^{new}_{i,j,k}$ is the computed value of grid point (i, j, k) and $p^{old}_{i,j,k}$ is its original value. nx, ny and nz are the size of the X, Y and Z dimension of the stencil grid respectively. Each update of every node requires 8 data accesses (7 loading + 1 writing) and 7 floating point operations according to the equation. The ratio between flops and data access is $7/(8 * 8) = 0.11$ for single precision floating point data. Such a low ratio indicates that a very high memory bandwidth must be supported by the hardware platforms for the stencil to attain good performance.

Because of the high demand on memory bandwidth, stencil computations such as the one in Figure 1 are good candidates to be evaluated on modern GPU architectures, which provide higher memory bandwidth than traditional CPUs. However, to attain high performance on GPUs, the stencil code must be able to sufficiently exploit the compute power of GPUs, by carefully managing the stencil data to efficiently utilize the hierarchy of GPU resources including the global memory, shared memory, and the registers. Optimizations on reducing the bandwidth requirement help improve the performance of stencil computation. In particular, 2.5D blocking [1] is a well known strategy that can significantly reduce the memory bandwidth demand of 3D stencils.

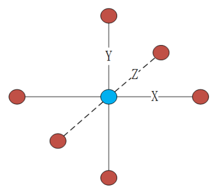

Fig. 1. Visualization of the 3D 7-point stencil used in this work.

In order to efficiently utilize the memory hierarchy of GPUs, stencil data need to be placed in the most suitable memories. In particular, the input and output data of a GPU computation must be initialized and saved in the global memory. During computation, these data typically need to moved among the global memory, the shared memory, and the registers to ensure that data are more quickly available when needed by the GPU stream processors. In particular, registers are the fastest memory in a GPU, and the bandwidth and latency of the shared memory is comparable to registers if bank conflicts can be avoided. However, when bank conflicts are prevalent, accessing the shared memory can be as slow as the global memory. Further, data in the shared memory can be shared among all the threads in a thread block, but the register can only be owned by one thread. Often both need to be utilized for stencils to achieve high performance.

Besides decisions over data placement, different thread block configuration can also influence the overall GPU performance. In particular, when using CUDA, a thread block configuration needs to be specified by the developer explicitly before executing a kernel on the GPU, and the performance attained by using different thread configurations can differ by orders of magnitude. However, the best thread configuration may vary greatly for different stencil implementations and when operating on different sizes of stencils. Automated empirical tuning is often used to find the best configuration. Auto-tuning is effective but is often also time consuming due to the large search space, especially when using exhaustive search without considering the varying data access patterns of stencil kernels.

This paper combines analytical modeling and auto-tuning techniques to investigate the performance implications of placing stencil data differently in the shared memory and register files of GPUs, when the stencil code is blocked with varying thread configurations.

Our main contributions are:

- We study the performance implications of four different data placement strategies of the 2.5D blocking optimization: including blocking only with shared memory, with both shared memory and register files and with only register files. The register only blocking version performs the best on Kepler and Maxwell GPUs;
- By analyzing the data access patterns of stencils, we develop auto-tuning heuristics to quickly find the best thread block configurations for different implementations;
- We present an analytical model to help choose the best data placement strategies for blocked 3D stencils on varying GPU platforms.

The rest of the paper is organized as follows. Section II introduces the 2.5D blocking optimization of stencil codes; Section III demonstrates the relationship of different data placement strategies and their performance implications; Section IV analyzes the main factors affecting the performance of stencil codes on GPUs based on their data access patterns.

II. 2.5D Blocking of 3D stencil computations

Because of the limitation of the on-chip memory (i.e. shared memory and registers), stencil codes on GPUs typically block the input data grid into multiple smaller grids that can be stored in on-chip memory (a.k.a. spatial bocking). The sub-grids could either be mapped directly to 3D thread blocks or 2D thread blocks on modern GPUs, and the latter usually performs better than the former [1][2][3]. The latter is also called 2.5D blocking, in which the original stencil grid is blocked in XY-plane. Each thread in a thread block computes a grid point in the XY-plane and sweeps through the Z dimension, so that the grid points computed by a thread block is a cuboid with the shape $bx \times by \times nz$. bx and by are the size of the sub-grid of XY-plane on X and Y dimension respectively. Listing 1 is a basic implementation of 2.5D blocking without exploiting the data reuse between the iterations along Z dimension. Though it is easier to map the stencil grid directly onto a set of 3D thread blocks than onto 2D thread blocks, the 3D mapping usually needs more shared memory or registers, and as the result it often performs worse than baseline version of 2.5D blocking shown in Listing 1.

Listing 1. The baseline version of 2.5D blocking of 3D 7-point stencil
```
__global__ kernel(in, out, bx, by, fac){
  tx_b = threadIdx.x;
  ty_b = threadIdx.y;
  if (!isboundary){
      for (k = 1; k < nz - 1; k++){
        c_g = tx_b + ty_b * nx + k * xy;
        c_s = input[c_g];
        l = c_g - 1;
        r = c_g + 1;
        t = c_g + nx;
        d = c_g - nx;
        z = c_g + xy;
        b = c_g - xy;
        l_s = in[l];
        r_s = in[r];
        u_s = in[u];
        d_s = in[d];
        f_s = in[f];
        b_s = in[b];
        out[c_g] = l_s + r_s + u_s + d_s + f_s
                 + b_s - c_s * fac;
      }
    }
}
```

By enhancing data reuse, 2.5D blocking helps reduce the bandwidth requirement of stencil computations, thereby resulting in better performance. When the kernel compute the values of points in a sub-plane in the XY-plane (we name it 'current' sub-plane), two neighbor sub-planes need to be loaded (we name 'top' and 'down' sub-planes respectively) as well as itself. Two of the three loaded sub-planes can be reused at the next iteration, that is the 'current' sub-plane becomes the 'down' one and the 'top' sub-plane becomes the 'current' one when the iteration is from bottom to top. The overlapped and split blocking methods have been studied to improve stencil code, and 2.5D blocking could be implemented using either blocking method on XY-plane. The overlapped blocking of XY-plane is used in this paper.

Besides spatial blocking, existing research also explored blocking in the temporal dimension, by reusing the data already loaded into the GPU's on-chip memory across multiple time steps. Temporal blocking increases the computation intensity of stencil code. However, not all stencil computations with temporal blocking perform better, due to the need to load multiple time iterations of halo data and the complex control flow introduced [4]. In this paper, We focus on 2.5D spatial blocking only without considering temporal blocking.

The performance improvement from 2.5D blocking is related to the data placement of the sub-grids. Unlike CPUs, GPUs have many more registers. In particular, the capacity of the register is even larger than shared memory, therefore the sub-grid could be loaded into shared memory, or be directly loaded into registers, or be loaded partly into shared memory and partly into registers. Each type of memory has its own requirements if we want to efficiently access the data, for example, coalesced global memory accesses and accessing shared memory without bank conflicts would be benefit to the performance; but too many shared memory and register files allocation would decrease the occupancy of GPUs and may result in performance degradation.

III. Data placement strategies

The placement of data in GPU's on-chip memory plays an important role in the achieved performance of 3D stencil codes with 2.5D blocking. For example, consider the 3D 7-point stencil in Fig 1. Here for each grid point being modified, four neighboring points located in the same XY-plane (p_1) are needed, plus two neighboring points located in the XY-planes located above and below (p_0 and p_2). All the three XY-planes can be stored in the shared memory, and p_1 and p_2 be reused in a circular queue pattern [5]. An alternative strategy is to only place the middle plane in the shared memory, with the p_0 and p_2 planes in register files [2]. We implemented both these stategies to study their performance implications. Additionally, we implemented a third version, which places all the three planes, p_0, p_1, and p_2, in register files only. Performance statistics are collected for all three versions using the three metrics, gld_throughput, gst_thoughput, and achieved_occupancy, supported by the NVIDIA *nvprof* command tool. Figure 2 shows the relative performance of the three data placement strategies with the input stencil grid of 512x512x512 on GTX 780. The following subsections discuss the implications of each data placement strategy from the results collected.

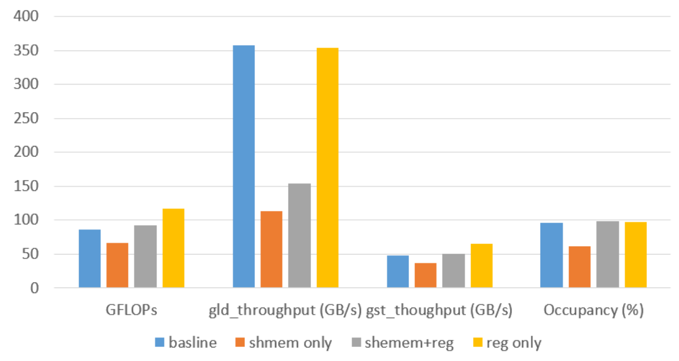

Fig. 2. Performance and the metrics

A. *Placing data in shared memory*

The GPU's shared memory is used to store data shared among different threads of a thread block. Because it has a much higher bandwidth and a much lower latency than the global memory, it acts as a user-managed cache reducing the required global memory bandwidth for memory-bound GPU applications [2]. However, certain patterns of data access, e.g., the 3D 7-point stencil in Figure 1, can induce severe bank conflicts and thereby greatly degrade the performance of shared memory. The kernel code of the shared memory version is in Listing 2. The performance metrics of this version is the worst as shown in Figure 2.

Listing 2. Stencil kernel (shared memory)

```
bx = blockDim.x;
by = blockDim.y;
x_s  = bx + 2;
y_s  = by + 2;
xy_s = x_s * y_s;
CURRENT_G = ix + iy*nx + nx*ny;
CURRENT_S = tx + ty*x_s + xy_s;
__shared__ float s_data[];
//down
s_data[CURRENT_S-xy_s] = in[CURRENT_G-nxy];
//curr
s_data[CURRENT_S] = in[CURRENT_G];
//top
s_data[CURRENT_S+xy_s] = in[CURRENT_G+nxy];
// halo region
s_data[CURRENT_S-1] = in[CURRENT_G-1];
s_data[CURRENT_S+1] = in[CURRENT_G+1];
s_data[CURRENT_S-x_s] = in[CURRENT_G-nx];
s_data[CURRENT_S+x_s] = in[CURRENT_G+nx];
__syncthreads();
if(!isboundary){
  curr  = s_data[CURRENT_S];
  east  = s_data[CURRENT_S+1];
  west  = s_data[CURRENT_S-1];
  north = s_data[CURRENT_S-bx];
  south = s_data[CURRENT_S+bx];
  __syncthreads();
  temp = east + west + north + south
       + top + down - curr * fac;
  out[CURRENT_G] = temp;
}
```

The shared memory version decreases the performance than the baseline version, because its access pattern can not avoid bank conflicts as shown in Figure 3. The "up" (denoted as "UP") and "down" ("D") nodes of the computed node ("C") of each thread are located in the same bank, resulting in the bank conflict. If the bank conflict degree is high, the latency of shared memory could be even higher than global memory [6]. Besides that, the achieved occupancy is also low compared to other implementations as shown in Figure 2, that is because it requires a larger amount of shared memory than the other versions. The shared memory is divided among all the thread blocks running on a SM. If the shared memory allocated for each thread block is too large, the number of concurrent active thread blocks is limited, resulting in not enough parallelism. This also contributes to the poor performance.

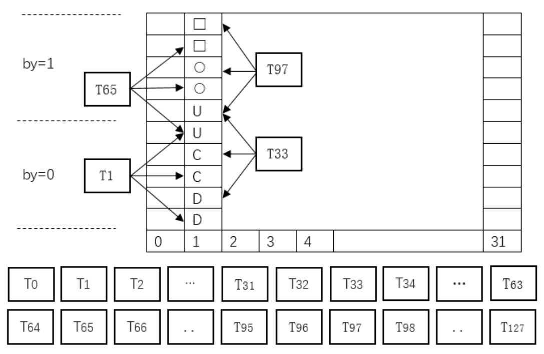

Fig. 3. The bank conflict of shared memory version

B. *Placing data in shared memory and register files*

Placing data only in shared memory performs worse than the baseline. However, it can be improved by storing the

”current” plane in the shared memory and the ”top” and ”down” planes in register files as shown in Listing 3. We name it the hybrid version. The serious bank conflicts in shared memory only version are avoided and the amount of shared memory allocated for each thread block is also reduced. The performance metrics of the hybrid version is better than both the shared memory only version and the baseline version, as shown in Figure 2. Because of the mixed use of the shared memory and register files, the size of shared memory of each thread block is reduced than the shared memory only blocking version and the size of register files of each thread is reduced than register blocking version. Therefore, the achieved occupancy of the hybrid blocking version is the best, but high occupancy does not always mean high performance.

Listing 3. Stencil kernel (hybrid)

```
down    = in[CURRENT_G − nx∗ny];
curr   = in[CURRENT_G];
s_data[CURRENT_S] = curr;
top = in[CURRENT_G + nx∗ny];
if(!isboundary){
   curr  = s_data[CURRENT_S];
   east  = s_data[CURRENT_S + 1];
   west  = s_data[CURRENT_S − 1];
   north = s_data[CURRENT_S − bx];
   south = s_data[CURRENT_S + bx];
   __syncthreads();
   temp = east + west + north + south
        + top + down − curr ∗ fac;
   out[CURRENT_G] = temp;
}
```

C. Placing data in register files

Placing data in register files achieves the highest global data access throughput and occupancy, as shown in Figure 2. For different grid size, they share the same trend among the three blocking methods. __syncthreads() is an overhead for shared memory blocking versions. Because there is no data sharing between threads, the register blocking version involves no synchronization as needed in the blocking of shared memory. Besides that, the register blocking version is similar to the shared memory blocking version in structure and it also keeps the data reuse in the iterations along Z dimension. Therefore it overtakes the codes blocking with the shared memory. The code of register blocking version is in Listing 4. Vizitiu et al. showed the similar result in [2] and Vasily also gave the similar analysis [7]. Besides the performance gain, the programming is also easier than the shared memory version.

Listing 4. Stencil kernel (register)

```
if(!isboundary){
   down   = in[CURRENT_G − nx∗ny];
   curr   = in[CURRENT_G];
   top    = in[CURRENT_G + nx∗ny];
   east   = d_in[CURRENT_G + 1];
   west   = d_in[CURRENT_G − 1];
   north  = d_in[CURRENT_G − nx];
   south  = d_in[CURRENT_G + nx];
   temp   = east + west + north + south
          + top + down − curr ∗ fac;
   out[CURRENT_G] = temp;
}
```

The loading data from global memory directly to register files utilizes the cache better, resulting much higher global memory throughput as shown in Figure 2, which is denoted in the Formula 1 on page 5. On Kepler GPUs, the size of register files is 64KB per SM as is even larger than the shared memory, and the maximum registers per thread is 255 make the register spilling not that easy. The shared memory and L1 cache share the same on-chip memory on Kepler GPUs, such that there exist interference between the accesses to shared memory and L1 cache of the hybrid blocking version. But in Maxwell GPUs, the L1 cache and shared memory are separate, such that the interference disappears.

IV. Performance analysis

Efficient use of global memory is critical to the performance of applications on GPUs, that is the data in global memory need to be loaded into on-chip memory as efficiently as possible. Accessing off-chip memory is a major performance bottleneck in microprocessors [8], and it is especially critical to the performance of the stencil code on GPUs [9]. The efficiency of data access is closely related to the data access pattern of the code. Because the access pattern of stencil computations is regular, a heuristic of finding the optimal thread block configuration and an estimation of the achieved global memory throughput are derived from its data access pattern analysis. They both help us better understand how an optimization delivers its performance improvement.

A. A heuristic for finding optimal thread block configuration

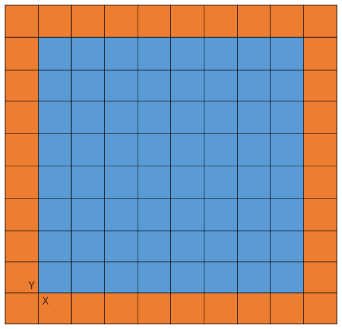

Fig. 4. A block of an XY-plane

A thread block and the data it loads are similar in shape, therefore the optimal thread block configurations can be approximately derived through analyzing its data access efficiency. The data loaded by each thread block of a XY-plane is shown in Figure 4, both the blue and orange regions. The blue region is the grid points to be computed, and the orange part is usually called the *halo* region used for computing the boundary points of the blue region. The dimension of a thread block is the same with the blue region, each small square of which represents a thread if one thread computes one node on the XY-plane. Each square in the blue region is loaded by the corresponding thread at the same position, and the orange points are usually loaded by the boundary threads in the thread block. The thread block is specified by the developers when programming, thus the blocking of the stencil data grid is also specified at the same time. The size of the loaded data is $(bx+2)\times(by+2)$ if the halo width is 1.

The *halo* region points are loaded multiple times, therefore it is a great overhead when each XY-plane is blocked into many small blocks. The abundant loading data size is related to the shape of the thread block. The points in the halo region of the thread block is also the points needed to be computed in the neighboring blocks, therefore they would have to be loaded multiple times. As stated in [1], if bx is times of 32, the loading of each row from global memory is coalesced. If bx is smaller there is more blocks along X dimension, therefore the ratio of redundant points is higher. We can infer that larger bx would provide better performance. The loading of the points of the halo in the Y dimension can not be coalesced, therefore the larger by the more un-coalesced global memory accesses. From this analysis, for the 2.5D blocking of the 3D 7-point stencil computation, a larger bx and a smaller by mean better performance, which can be used to quickly find the optimal thread block shape. For example, a stencil grid of $256\times256\times256$ points, the baseline version 128 delivers the best performance on GTX 780, while the optimal configuration of the shared memory + register version is 128×2. The results in [10] are similar to ours. Though the configurations are different from different input data size and implementations, they share the similar shape: long and flat.

Though this heuristic is consistent for the different implementations of the 3D 7-point stencil computation, the optimal thread configurations are different, because the thread block configuration not only influences the efficiency of data accessing but also impacts other performance factors such as the occupancy. Therefore, we employ an auto-tuning method to find the optimal ones. This heuristic greatly reduces the searching space and saves the tuning time.

B. *Analyzing global memory throughput*

The achieved throughput of global memory is related to the data size transferred, the number of threads in a thread block [11], as well as the shape of the thread block and the efficiency of utilizing L1 cache. The achieved bandwidth of global memory is linearly proportional to the size of the data transferred before saturating the global memory bandwidth if there is no contention. The less-thread blocks cannot provide enough parallelism and the more-thread blocks can lead to memory requests sequentially serviced. The global memory loading/storing throughput of 3D stencil under 2.5D optimizing methods is shown in Figure 5, from which we can see the throughput increases with the data size in each optimizing implementation.

The global memory throughput of the computation can be modeled in Formula 1:

$$Throughput_{gm} = \frac{\alpha\cdot bx\times D_{all}\times U_{cache}\times BW_{gm}^{peak}}{\beta\cdot by} \quad (1)$$

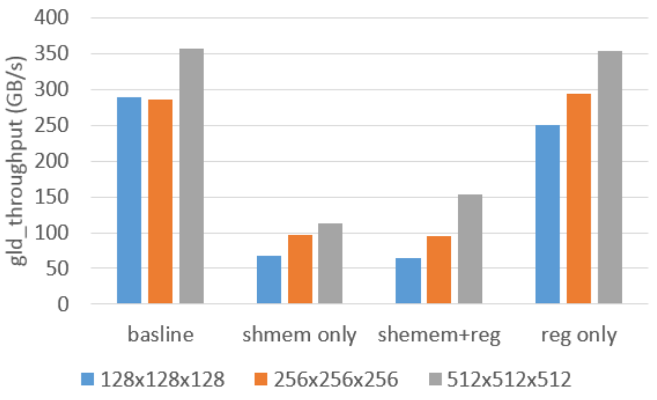

Fig. 5. Global memory loading throughput

It approximately describes the factors to the throughput of loading data from global memory. D_{all} is the data size of the input stencil grid. The throughput increases with D_{all} before the bandwidth is saturated. U_{cache} is the utilization of L1 and L2 cache, which is described in Section III. bx and by is the thread block dimension. Their influences to the performance is analyzed in IV-A. BW_{gm}^{peak} is the peak bandwidth of global memory. For memory intensive applications, higher memory throughput usually means better performance.

V. Experimental Results

All the different implementations are executed and tuned on GTX 780(Kepler) and GTX 960M(Maxwell). They share the similar results: the performance of placing data in register only versions increase 20% to 40% than the baseline version on GTX 780 and 20% on GTX 960M with different input data sizes; the performance of placing data in shared memory only versions decreases by 20% to 40% on GTX 780 and 10% to 20% on GTX 960M. The performance results are shown in Figure 6 and 7. The difference between the hybrid blocking version and the register blocking version on GTX 960M is not as much as on GTX 780 as stated in section IV.

The optimal configurations of each implementation on the same GPU with different input data size are different as listed in table I.

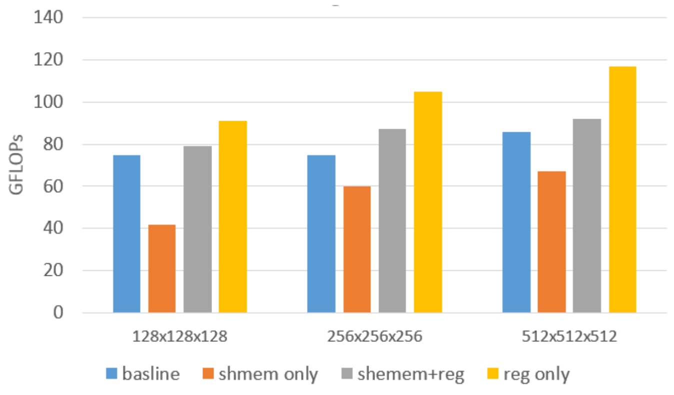

Fig. 6. Performance on GTX 780

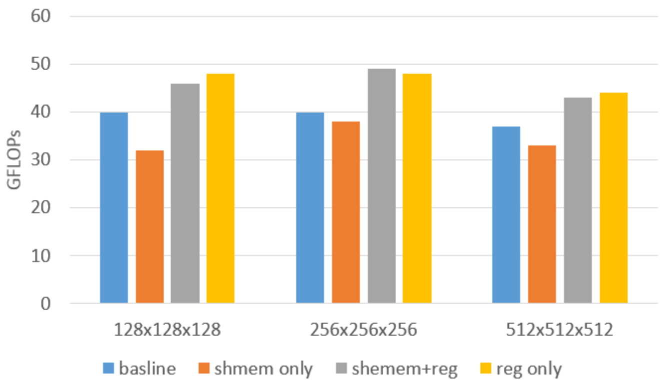

Fig. 7. Performance on GTX 960M

TABLE I
OPTIMAL THREAD BLOCK CONFIGURATIONS ON GTX 780

Data Placement	128^3	256^3	512^3
Baseline	64×2	128×1	256×4
Shmem only	128×1	128×1	256×1
Shemem+reg	128×4	128×2	256×1
Reg only	64×2	128×1	256×1

We can estimate the optimal thread block configuration according to the thread block configuration heuristic stated in section IV for the three stencil codes. The optimal shape of blocking configurations for the 3D 7-point stencil is long and flat as shown in Table I, and it meets the heuristic. The trend is still agree with the prediction of our performance model: with the fixed $block.x$, the performance increases when $block.y$ decreases; with the fixed $block.y$, the performance increases when $block.x$ decreases.

VI. Related work

The model in [12] is similar to ours and they also focus on 3D low order stencil computations, comparing a baseline and a z-blocked version, but not the more efficient 2.5D blocking with shared memory and registers. S. Tabik et al. [10] proposed a method for finding the optimal thread block configurations efficiently through the analysis of the memory access transactions of different levels. Though the optimal configuration is not unique, the wider and shorter blocking usually performs better, because they can reduce the number of memory access transactions. Better performance does not always means higher occupancy [13], the register blocking version delivers the best performance though its achieved occupancy is not the highest. An analytical performance prediction model is proposed in [14] and empirical tuning is used as an effective to solve the complex problem.

VII. Conclusion

We analyze how the data placement would influence the performance of the 2.5D blocking optimization of 3D 7-point stencil computation and the result can be used as a guide when implementing the stencil code on GPUs. We employ the empirical tuning method to find the best performance according to the heuristic based on data access pattern analysis, which can significantly reduce the tuning time. However, when an application is ported to a different GPU, the tuning needs to be done again. Though different GPU architectures perform differently, there exist common places in the relationship between hardware characteristics and performance. We are going to study using machine learning techniques and make the code adaptive to new architectures based on the tuning results of existing hardware.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work is supported by fund of key laboratory of modern teaching technology, ministry of education, P.R. China. Guangwei Zhang is supported by the China Scholarship Council for 1 year study under the direction of Dr. Qing Yi at the University of Colorado at Colorado Springs.

References

[1] M. Krotkiewski and M. Dabrowski, "Efficient 3d stencil computations using CUDA," vol. 39, no. 10, pp. 533–548.

[2] A. Vizitiu, L. Itu, C. Nita, and C. Suciu, "Optimized three-dimensional stencil computation on fermi and kepler GPUs," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6.

[3] N. Maruyama and T. Aoki, "Optimizing stencil computations for nvidia kepler gpus," in *Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna*, 2014, pp. 89–95.

[4] 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 *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov. 2010, pp. 1–13.

[5] 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*. IEEE Press, p. 4.

[6] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *arXiv:1509.02308 [cs]*, Sep. 2015, arXiv: 1509.02308.

[7] V. Volkov, "Programming inverse memory hierarchy: case of stencils on GPUs," in *GPU Workshop for Scientific Computing, International Conference on Parallel Computational Fluid Dynamics (ParCFD)*.

[8] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988.

[9] N. Fauzia, L.-N. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on GPUs," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. IEEE Computer Society, pp. 12–22.

[10] S. Tabik, M. Peemen, N. Guil, and H. Corporaal, "Demystifying the 16x16 thread-block for stencils on the gpu," *Concurrency and Computation: Practice and Experience*, 2015.

[11] A. Resios, "GPU performance prediction using parametrized models," the Netherlands, 2011.

[12] H. Su, X. Cai, M. Wen, and C. Zhang, "An analytical gpu performance model for 3d stencil computations from the angle of data traffic," *The Journal of Supercomputing*, pp. 1–21, 2015.

[13] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU Technology Conference, GTC*, vol. 10. San Jose, CA.

[14] S. S. Baghsorkhi, M. Delahaye, W. D. Gropp, and W. H. Wen-mei, "Analytical performance prediction for evaluation and tuning of GPGPU applications," in *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM), In conjunction with The International Symposium on Code Generation and Optimization (CGO) 2009*.