Computing and Informatics, Vol. 37, 2018, 533–552, doi: 10.4149/cai_2018_3_533

ACCELERATING STENCIL COMPUTATION ON GPGPU BY NOVEL MAPPING METHOD BETWEEN THE GLOBAL MEMORY AND THE SHARED MEMORY

Tieqiang Mo, Renfa Li

College of Information Science and Engineering
Hunan University
Changsha, Hunan, 410082, China
e-mail: 852020926@qq.com, renfali@vip.sina.com

Abstract. Acceleration of stencil computation can be effectively improved by utilizing the memory resource. In this paper, in order to reduce the branch divergence of traditional mapping method between the global memory and the shared memory, we devise a new mapping mechanism in which the conditional statements loading the boundary stencil computation points in every XY-tile are removed by aligning ghost zone to reduce the synchronization overhead. In addition, we make full use of single XY-tile loaded into registers in every stencil computation point, common sub-expression elimination and software prefetching to reduce overhead. At last detailed performance evaluation demonstrates our optimized policies are close to optimal in terms of memory bandwidth utilization and achieve higher performance of stencil computation.

Keywords: Memory mapping, GPGPU, stencil computation, ghost zones

Mathematics Subject Classification 2010: 65Y05

1 INTRODUCTION

Stencil computations mean repeated updating of values associated with points on a multi-dimensional grid, using only values in a set of neighboring points. GPUs can effectively accelerate this type of application such as computational electrodynamics [1], the solution of partial differential equations (PDEs) which applies the finite

difference [2], and image processing for CT or MRI imaging [3]. But for a highly tuned implementation how to manage memory resources remains a critical problem. The challenge is to design and implement optimized algorithms that make full use of memory bandwidth and/or arithmetic units and to reduce inefficiencies due to excessive memory traffic and unnecessary computations.

GPUs just like CPUs are subject to memory bandwidth bottleneck. For example, on NVIDIA C2050, a peak double precision performance of 515 GFLOP/s and a peak memory bandwidth of 144 GB/s can be achieved, that is to say, byte-per-flop ratio indicating the balance of the memory bandwidth versus float-point operations per second (FLOP/s) is 0.28. Similar performance can nowadays be achieved by the CPU-based shared memory system, e.g., a 4-way system with a total of 482.2 GHz Opteron cores can have a peak performance of 450 GFLOP/s and a peaked bandwidth of 170 GB/s (byte-per-flop is 0.38). For all these systems the CPU delivers more bandwidth relative to floating point computation than the GPU. What is more important is the fact that the byte-per-flop is relatively low in both systems. This implies that optimizing memory access and decreasing redundant memory operations is a must even for compute bound application.

In this paper, stencil computation algorithms running on NVIDIA GPUs are optimized. Firstly, we design a novel memory mapping mechanism between global memory and shared memory of XY-tiles which extends the classical XY-tile and ghost zone to include the aligned data points of 32 bytes (8 words) of neighboring XY-tile shown in Figure 2. In this way, control flow divergence due to the conditional statement in Listing 2 can be eliminated. Secondly, in every iteration only one XY-tile's data are loaded into shared memory and further are copied into registers in the next iteration. All stencil computations about this tile are made available for these registers. Data and the partial sub-sum are stored in the temporary registers. By this way, the saved shared memory can be used to launch more other threads. Moreover, FLOPs have been further reduced by developing *common sub-expression elimination* [4] policy for symmetric 27 points stencil computations. Finally, in our implementation the software pre-fetching is used to overlap arithmetic and memory instructions to the benefit of hiding memory latency.

Comparing with inter-tile communication mechanism, policy of local computation reduces the communicating overhead of stencil computation in the neighboring tiles. Nevertheless, overlapped memory access named ghost zone overhead in the neighboring tiles may be produced in this policy. This is because of the fact that the data in the boundary points of XY-tile must be reloaded twice for the adjacent XY-tiles. We analyze the efficiency of our stencil algorithms by utilizing a model of memory traffic which takes into account the size of ghost zone due to minimum data reloading from global memory to shared memory. From this model we conclude that overhead due to ghost zone may be largely alleviated by correct reuse of data onto ghost zone in global memory or texture caches. Our implementation of the 7-point stencil is bound by memory bandwidth for both single and double precision and close to optimal on Tesla C2050, i.e., it runs only 13 % slower than a memory loading routine without considering the ghost zone overheads.

The rest of the paper is organized as follows. Section 2 deals with the CUDA programming model and algorithms behind stencil computation. In Section 3 we detail algorithm framework of stencil computation based on registers and shared memory. Section 4 deals with new memory mapping mechanism in which ghost zone overheads are adequately considered to reduce memory access traffic on the GPU. In Section 5 software prefetching mechanism is analyzed. In Section 6 we evaluate the performance of our new memory mapping mechanism. In the last section all new ideas are concluded in this paper.

2 STENCIL COMPUTATIONS ON GPUS

A stencil computation is characterized by updating each point in a structured grid by an expression depending on values on a fixed geometrical structure of neighboring grid points. The simplest example is the 7-point stencil, approximation of the 3D Laplacian operator, as shown in Figure 1 a). The update of a data point depends on the current position and its neighbors on the left, right, front, back, above and below. A more complex example is 27-point stencils shown in Figure 1 b), where an update of point (i, j, k) depends on the weighted sum of point (i, j, k) and its 26 neighbors.

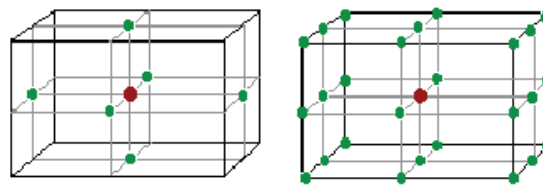

Figure 1. Stencil space structure

As an actual example, we may consider the 3D heat equation $\frac{\partial u}{\partial t} = k\nabla^2$ where ∇^2 is the Laplacian operator, and we assume a constant heat conduction coefficient and no heat sources. The following explicit finite difference scheme can solve the problem on a uniform mesh of points:

$$u_{i,j,k}^{n+1} = u_{i,j,k}^{n} + \frac{k\Delta t}{\Delta x^2}\left(u_{i,j,k-1}^{n} + u_{i,j-1,k}^{n} + u_{i-1,j,k}^{n} + u_{i+1,j,k}^{n} + u_{i,j+1,k}^{n} + u_{i,j,k+1}^{n} - 6u_{i,j,k}^{n}\right) \tag{1}$$

The superscript n denotes the discrete time step number (an iteration), the triple-subscript i, j, k denotes the spatial index. The quantity Δt is the temporal discretization (the time step) and the mesh spacing is equal in all directions. Note that in reality the formula in (1) is a 7-point computation stencil applicable only to inner grid points on a tile, and for simplicity we have omitted treatment of the boundary points.

For parallel computations on GPUs there exist thousands of threads to execute concurrently to hide memory and instruction latency: once active threads stall, warp scheduler chooses the ready threads in round robin mode. On the fine-grained level threads are arranged into completely synchronous groups of 32 threads named as one warp. Different warps are scheduled for execution independently of each other by the Streaming Multiprocessors (SM). Individual thread within the same warp is allowed to take a different execution path if conditional statements happen. However, since threads in a warp execute common instructions at a time, one execution path in one branch of that warp is disabled until different execution paths merge again. Consequently, control flow divergence significantly increases the number of the executing instructions to the sum of the instruction counts of all execution paths taken. In this situation, it is more beneficial to take branch-free implementation into consideration, especially when optimizing compute bound software.

Warps may be further grouped into 1D, 2D or 3D regular thread blocks. Warps belonging to the same thread block are executed on the same SM. The order in which different thread blocks are executed is random. Threads within a thread block can quickly synchronize and exchange data through a common shared memory. The position of a thread inside a block is described by a set of local threadIdx.x, threadIdx.y and threadIdx.z indices. Each thread block has a similar set of indices *(blockIdx.x, blockIdx.y, blockIdx.z)* locating the position of the block in the global data grid. In one policy of 3D stencil computations the grid is processed using 3D thread blocks [5]. Based on blockIdx and threadIdx coordinates threads compute global indices (i, j, k) of the processed data points.

A great many stencil optimization techniques have been previously suggested. Micikevicous [6] gave an implementation of 3D stencil (FDTD3d) in which the shared memory is used to load XY-tiles and registers are made available to save shared memory space. Phillips and Fatica [7] utilized the removing logic to reduce the number of conditional statements and made 4 texture caches available. So its memory access latency overheads increased. Recently Zhang et al. [8] statically allocated ghost zone data to threads during initialization of the thread blocks. In addition, the method avoids conditional statements, but its ghost zone in the Y-dimension are loaded in non-coalesced way. This might also lead to degraded performance.

In our memory mapping mechanism all updated stencil data in every computed point must be written into global memory. But in some applications the intermediate updated data require no writing into the global memory immediately in the defined stencil (e.g. Wave propagation solver, Jacobi solver). In this case, the temporal block optimization [5, 11, 13] could make time-dimension data available to speed up stencil computations for less memory access.

Many researchers have proposed automatic stencil code generation and performance tuning for modern multi-core heterogeneous architectures. The auto-tuning method searches through the space of parameters which may degrade performance, such as loop unrolling factor, types of memory and thread block size. This method demonstrates a portable high performance across different architectures and stencil types. Datta et al. [4] developed an auto-tuning framework for stencil computa-

tions, targeting multi-core systems, NVIDIA GPUs or Cell SPUs. Zhang et al. [8] and Christen [10] also developed the tuning framework to optimize performance on the GPU. Tang et al. [14] projected the Pochoir stencil compiler which uses a domain specific language embedded in C++ to produce high performance code for stencil computations using cache-oblivious policy for parallelism. Unat et al. [15] suggested a compiler framework named Mint using annotated C as the front-end and converting stencil computation into C code by utilizing pragmas with several levels of optimization.

In this paper a more efficient and widely used 2.5D blocking policy [6] is applied. Two-dimensional data points are used to tile the grid in the XY-plane and provide threads with the (i, j) indices of the grid points. A loop is then used to traverse the grid in the Z-dimension, providing additional k index. In this way there are no Z-dimension ghost zone in the 2.5D policy since data are processed by the thread block plane by plane. This guarantees data reuse in the neighboring XY-tile along Z-dimension and reduces plenty of memory bandwidth requirements. So this policy is superior to using 3D thread blocks because of data reuse and reduced shared memory requirements in 2.5D blocking policy. What is more, in 2.5D policy the initialization cost of the thread blocks (computing thread and grid indices, setting boundary conditions, etc.) is decreased over a larger number of grid points processed by every thread.

3 MORE EXPLOITATION OF REGISTERS IN STENCIL COMPUTATION

A novel pseudo-code implementation of stencil computation is presented in Listing 1. In registers *r1*, *r2*, ..., *r9*, values of stencil points of the same XY-plane are stored. Sub-routine load_block_ghost loads the whole XY-plane tile including ghost zone into shared memory space: sh_m from global memory; in sub-routine sh_m_regs, 9 values around stencil computation point (i, j, k) in the same XY-tile plane are copied to the registers from shared memory; in the function stencil_compute1, only the partial sub-sum is computed by adding the weighted values stored in current registers *r1*, *r2*, ..., *r9*, namely, the partial stencil result of the single XY-tile is produced by making all coefficients of the stencil kernel available to the data in one k-loop iteration.

The algorithm shown in Listing 1 runs a series of steps to finish all stencil computations. In line 9 and line 12 the input data of the first XY-tile and the second XY-tile are loaded into registers from shared memory, respectively. The updated value of every computed point is the sum of the weighted value of its neighboring points which belong to the three different planes being adjacent to each other, likewise, this value is equal to the sum of 3 partial sub-sums which can be computed by their corresponding stencil expressions in three different adjacent planes. So in line 11 and line 14 two partial sub-sums are computed for the first XY-tile plane and the second XY-tile plane, respectively. In line 15 the sum of

these partial sub-sums for the present two planes is stored in the temporary variable *inter*1 (the corresponding PTX code is the register). In line 18 the input data of the third XY-tile of the current iteration are loaded into registers from shared memory. In line 19 the input data of the third XY-tile of the next iteration are loaded into shared memory from global memory or texture cache. In line 21 the loaded data of the third XY-tile in line 18 are used to compute the partial sub-sum of the third XY-tile and the first stencil computation result is obtained by further adding the value of inter1. Lines 22 and 23 initialize the subsequent iterations. In lines 25 and 26 the updated stencil value of the last XY-tile is computed. In the loop code beginning from line 17 only the data of the third XY-tile are loaded into the registers and can be used to compute the updated stencil value of the current iteration while the partial sub-sum of the other two XY-tiles comes from the temporary value inter1 of the last iteration. So the registers are fully utilized by our stencil computation and the saved shared memory can be allocated to more threads.

```
1  _global_ void stencil_compute(in, out, nx, ny, nz)
2{
    // shared memory for data point
3  extern _shared_ float sh_m[ ];
4  const uint tile_x=threadIdx.x;
5  const uint tile_y=threadIdx.y;
    // the temporary stored data in registers for XY-tile
6  float r1~r9;
    // registers for partial stencil sub-sum
7  float inter1,inter2;
    //load data into the shared memory for the first XY-plane
8  load_block_ghost(in, first,tile_x,tile_y,sh_m);
9  sh_m_regs(sh_m,tile_x,tile_y);
    // load data into the shared
10  load_block_ghost(in,second,tile_x,tile_y,sh_m);
    // memory for the second XY-plane
    //partial stencil sub-sum for the first XY-plane
11  inter1=stencil_compute1( r1~r9 );
12  sh_m_regs(sh_m,tile_x,tile_y);
   //load data into shared memory for the third XY-plane
13  load_block_ghost(in,third,tile_x,tile_y,sh_m);
   //partial stencil sub-sum for second XY-plane
14  inter2=stencil_compute1(r1~r9);
    //partial stencil result obtained by the sum of
    // the variables  inter1 and inter2
15  inter1+=inter2;
16  out+=ix+iy*nx;
17  for(uint k=3;k<nz-1;k++){
    // load data into registers from the shared
    // memory for the third XY-plane
18     sh_m_regs(sh_m,tile_x,tile_y);
```

```
    //initialize the next iteration
19      load_block_ghost(in,k,tile_x,tile_y,sh_m);
20      out+=nx*ny;
    // update current XY-tile
21      out[0]=inter1+stencil_compute1(r1~r9);
    // initialize the next iteration
22      inter1=inter2+stencil_compute1(r1~r9);
23      inter2=stencil_compute1(r1~r9)
24      }
25  sh_m_regs(sh_m,tile_x,tile_y);
     // update the last XY-tile
26  out[0]=inter1+stencil_compute1(r1~r9);
27 }
```

Listing 1. Algorithm framework of stencil computations

The popular implementation of stencil solution is made by a holistic stencil computation, that is to say, for every stencil computation tile k, all adjacent stencil data in $k-1$, k and $k+1$ tiles are loaded into shared memory or registers of the thread block and are computed by stencil expression. Reusing the k and $k-1$ input tiles in the production of the next $k+1$ tile relies on circular queue [7]. In contrast, our implementation of stencil solution is achieved by the accumulation of multiple partial stencil results. In every partial stencil calculation in a sub-routine *stencil_compute1* only one XY-tile data are loaded into registers. The acquired partial stencil results are accumulated in registers inter1 and inter2. Once stencil result accumulated from the adjacent three XY-tiles has finished this result is stored in the output array.

4 MEMORY MAPPING MECHANISM AND OPTIMIZATION

4.1 The Classical Memory Mapping Mechanism

The classical memory mapping algorithm from global memory to shared memory is illustrated with Listing 2, in which one XY-tile data are loaded into shared memory. The homogeneous implementation can be found in [6, 7]. In Listing 2 the adjacent indices of the computed stencil point are shown here only for comprehensibility. Actually, all indices are pre-computed before the k-loop at the beginning of the routine *stencil_compute*. Since the single thread computes all stencils points along the Z-axis, only the k-index changes and needs to be updated in the k-loop.

In line 6 data in a stencil point corresponding to one thread's ID is loaded into shared memory. The ghost zone is loaded into shared memory by boundary threads, which are shown from line 7 till line 11. From this figure we can find that there are many lines of conditional statements which can effectively aggregate control flow divergence of threads in one warp and increase the total number of executed

instructions. By utilizing texture cache as shown in [7] the conditional statements have been removed and redundant loads of global memory have been prevented. But the texture cache is not designed to reduce latency, thus texture loads have similar cost of global loads regardless of whether or not there is a cache hit. In order to shun the conditional statements Zhang and Mueller [8] statically allocated ghost zone points to the individual threads at the start of the stencil routine. The threads first load the interior data points of XY-tile into shared memory then load ghost zone. Depending on the thread block size, some threads are loading the allocated ghost points while many other threads may be idle until all branches in one warp converge. On the other hand, the X-dimension ghost zone points are not loaded into the shared memory in a coalesced way.

```
1  load_block_ghost(in,k,tile_x,tile_y,sh_m)
2  {
3   uint bx=blockDim.x+2;
4   uint mx = blockIdx.x*blockDim.x + threadIdx.x;
5   uint my = blockIdx.y*blockDim.y + threadIdx.y;
6    sh_m[tile_x+tile_y*bx]=in[mx+my*nx+k*nx*ny];
   //top and bottom ghost zones
7  if (tile_y==1) sh_m[tile_x+(tile_y-1)*bx]=in[mx+
       (my-1)*nx+k*nx*ny];
8   if(tile_y==blockDim.y) sh_m[tile_x+(tile_y+1)*bx]
      =in[mx+(my+1)*nx+k*nx*ny];
   // left and right ghost zones
9  if(tile_x==1) sh_m[(tile_x-1)+tile_y*bx]=
        in[mx-1+my*nx+k*nx*ny];
10   if(tile_x==blockdim.x) sh_m[tile_x+1+tile_y*bx]=
       in[mx+1+my*nx+k*nx*ny];
    //corner ghost zones
11   if(tile_x==1&&tile_y==1)
    {sh_m[tile_x-1+(tile_y-1)*bx] = in[mx-1 + (my-1)*nx
            + k*nx*ny];
    sh_m[tile_x-1+(tile_y+1)*bx] = in[mx-1 + (my+1)*nx
            + k*nx*ny];
    sh_m[tile_x+1+(tile_y-1)*bx] = in[mx+1 + (my-1)*nx
            + k*nx*ny];
    sh_m[tile_x+1+(tile_y+1)*bx] = in[mx+1 + (my+1)*nx
            + k*nx*ny];
      }
12   __syncthreads();
}
```

Listing 2. Ordinary memory mapping algorithm

4.2 Modeling Memory Mapping Mechanism

The purpose of modeling the memory mapping mechanism is to evaluate the efficiency of our stencil implementation in terms of memory bandwidth usage and memory overhead due to redundant data access to ghost zone. We will calculate the theoretical minimum and the algorithm constrained minimum amount of memory traffic in a single stencil computation, respectively.

Theoretically, for the global stencil memory access operations, the value of data points is loaded into shared memory once and computed, and then the result is written back to global memory once. If 4 bytes of data points are assumed for single precision floating point (SPFP) the number of bytes of memory traffic per data point is 8 bytes (4 bytes for loading and 4 bytes for writing).

For GPUs, different thread blocks executed on the stream multi-processors run independently and cannot access each other. Therefore, in stencil computation every thread block has to load the interior data of XY-tile and a layer of adjacent values targeting to calculate stencil value of peripheral points of XY-tile. The values of this peripheral layer of every XY-tile are often referred to as the ghost zone or halo which simultaneously are interior values of adjacent thread blocks. Thus, all in all, (blockDim.x + 2) ∗ (blockDim.y + 2) values of every XY-tile must be loaded into shared memory. Consequently, memory access is increased since some of the data are loaded more than once: first as interior data of XY-tile, again as the ghost zone of adjacent XY-tile. Of course, this repetitive access overhead can be lessened by enlarging the XY-tile perimeter, which produces a smaller ratio between the number of data points of ghost zone and that of interior data points in one XY-tile. Since the threads in different thread blocks cannot access each other, overhead of loading ghost zone cannot be explicitly eliminated and should be added to memory traffic of per data point.

Memory access must be aligned by 32, 64 or 128 bytes on GPUs. So loading data from global memory and writing data into global memory should be performed in a coalesced manner, in other words, threads in a warp should access the consecutive addresses within the aligned data segments. If the size of the thread block and the most frequently changing dimension of the tile is a multiple of the warp size full memory coalescence can be reached [9]. Generally, loading and writing of interior data of block can be fully coalesced so long as the tiles are properly partitioned. Since the values of the top and below Y-dimension ghost zone can also be correctly aligned in the memory accesses they are also loaded in a coalesced manner. But loading left or right X-dimension ghost zone becomes more challenging from the point of view of performance because of their non-consecutive memory access. If one tile has a thread block size of *bx*by* and ghost zone width of 1 the minimum number of the loading data values from the global memory is $bx*by+2bx+2(by+2)*32/\text{sizeof(float\_point\_number)}$, where sizeof(float_point_number) is size of the data value in bytes: 4 and 8 bytes for single and double precision number, respectively. The number of points respectively in the interior tile and the top and bottom ghost zone is $bx*by$ and $2\,bx$ in a tile. Since the number of the minimum transaction bytes is

the aligned 32 bytes, the minimum number of the loading data values for loading left and right ghost zone is $2(by+2)*32/\text{sizeof(float\_point\_number)}$. Further for single precision floating point numbers the loading number of one tile can be simplified as

$$bx*by+2bx+2(by+2)*8=bx*by+2bx+16by+32. \tag{2}$$

The expression (2) accounts for the minimum loading number of SPFP values including the ghost zone overhead produced by loading all data of XY-tile. One way to lessen the memory overhead is to enlarge the thread block. On the contrary, for some GPUs we can make use of the global memory caches to lessen the memory overhead, which can guarantee that a fewer number of data points in one XY-tile are directly loaded from global memory than the expression (2). In our tests on the Fermi-based Tesla C2050 with L1 and L2 global memory caches the global memory overhead brought by ghost zone can be partly or entirely cleared up if the data among different XY-tiles are reused appropriately. But for this maximized reuse the scheduling order of warps must be controlled and cannot be implemented in the current GPUs.

4.3 New Memory Mapping Mechanism

For expression (2) it must be divisible by 32 for more coalesced memory accesses in 32 threads of one warp, that is to say, the value of expression $(bx*by+2*bx+16*by+32)$ mod 32 is zero. So bx and by must be equal to a multiple of 16 and 2, respectively. Further, in our test on Tesla C2050 with CUDA toolkit Visual Profiler if the size of XY-tile is 32×6, all warps show higher occupancy (defined as Active Warps/Maximum Active Warps). Then the picked XY-tile size is 32×6. On the other hand, for the ghost zone in every XY-tile Y-dimension width is 2 lines (top and bottom) and X-dimension aligned overhead is 16 words or 64 bytes (for SPFPs 8 words or 32 bytes on left and right side, respectively). So the size of loaded tile is 48×8, as shown in Figure 2. In order to eliminate threads divergence brought by conditional statements in Listing 2 all threads may be mapped onto two threads blocks of size 48×4. The detailed pseudo-code implementation is shown in Listing 3. In this way, the number of bytes of memory traffic per data point is 12 bytes (4 bytes in writing and 8 bytes in loading which consists of 4 bytes data and 4 bytes ghost zone overhead).

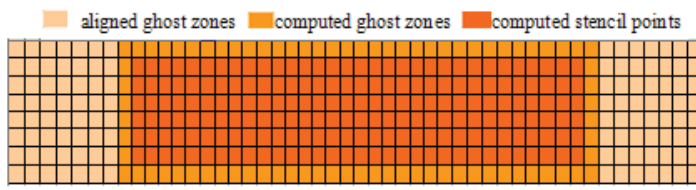

Figure 2. New-mapped XY-tile

```
1  load_block_ghost_new(in,k,sh_m)
2  { // change into one dimension index
3   uint  index= (threadIdx.y * blockDim.x) + threadIdx.x;
4   uint  tile_y1=index/48;
5   uint  tile_y2=tile_y1+4;
6   uint  tile_x=index%48;
7   int ix = blockIdx.x*blockDim.x + tile_x - 8;
8   int iy1 = blockIdx.y*blockDim.y + tile_y1 - 1;
9   int iy2 = blockIdx.y*blockDim.y + tile_y2 - 1;
10  // loading the data into the shared memory
11  sh_m[tile_x + tile_y1*48] = in[ix + iy1*nx + k*nx*ny];
12  sh_m[tile_x + tile_y2*48] = in[ix + iy2*nx + k*nx*ny];
13  __syncthreads();
}
```

Listing 3. New memory mapping algorithm

In global memory writing full coalescence can be achieved for the proper block size. If all interior data points of tile have an appropriate alignment in the memory, that is to say, for every thread block, every interior data value always starts a 128-byte aligned global memory address, writes of every warp can be coalesced into one 128-byte access on the GPU Compute Capability 2.0 architecture or into two 64-byte accesses on the GPU Compute Capability 1.x architecture. On the contrary, loading data in one warp may demand two shared memory accesses for some warps. The mapping to a thread block size of 48×4 produces misalignment, since 48 is not divisible by 32, namely, some warps have to access non-contiguous memory areas. Further the accessed data on the left ghost zone starts with a 32-byte offset from the 128-byte aligned interior data. In this way, one loading operation may be performed by the hardware as a combination of a separate 32-byte access and at least one other access. But it does not reveal significant penalty by our performance tests if the data is accessed by textures.

5 SOFTWARE PREFETCHING

Software prefetching is a well-known technique to overlap memory access latency with computation. In our prefeching mechanism the prefetched data in a k-iteration is used for stencil computation in $(k + 1)$-iteration. Concretely, this prefetching mechanism is implemented in the following sequence of Listing 1.

1. *load_block_ghost* – initialize the stencil computation by loading the first two XY-tiles from the global memory or texture cache to the shared memory (in line 10 and line 13).
2. *sh_m_regs* – copy the data loaded from the shared memory in the previous iteration to register for the current partial stencil computation sub-sum (in line 18).

3. *load_block_ghost* – load data for the next iteration from global memory or texture cache to the shared memory (in line 19).
4. *stencil_compute1* – compute the weighted sub-sum value for one XY-tile in the current iteration by utilizing data in registers (in line 21–23)
5. go to step 2 (iteration of the k-loop).

The data loaded from the global memory in step 3 is not promptly used in the current iteration. But the computations in step 4 are dependent on the data of being copied into registers in step 2, namely in the current iteration, which is loaded into the shared memory in the previous k-loop iteration. Therefore, operations of steps 3 and 4 can be overlapped by the hardware as they have no common operands. In this way, the global memory access latency can be hidden by performing the arithmetic instructions in step 4. While the warp scheduling mechanism can overlap arithmetic operations and memory access operations our prefetching method has more opportunity for the scheduled warps and leads to a higher occupancy.

6 PERFORMANCE EVALUATION

In our performance evaluation performance constraints of the available memory bandwidth are evaluated by memory bandwidth benchmarks and the memory mapping mechanism above. The computation constraints are evaluated by counting the number of instructions in the assembly PTX code produced by the compiler.

Architecture	Clock Cycle GHz	Peak Memory Bandwidth GB/s	SPFP Performance GFLOP/s	DPFP Performance GFLOP/s
Tesla c2050	1.115	144	1 030	515

Table 1. Performance characteristics of Tesla C2050

First, we run a memory transit routine with zero-overhead shown in Listing 4 for 3D cube grids of single precision data whose memory access traffic per data point is 8 bytes (4 bytes for loading and writing respectively). Since there is no communication among different XY-tile threads and no ghost zone overhead for this transit routine, we can get the theoretical lower bound memory access throughput and an absolutely idealized constraint on stencil performance. Secondly, we run the routine transmitting all the data which has a thread block size of 32×6, Y-dimension bottom and top ghost zone width of 1, respectively, and aligned X-dimension left and right ghost zone size of 8 words, respectively. Then we can evaluate the efficiency and memory overhead of our new memory mapping mechanism by comparing the results of two benchmarks. Thirdly, we estimate how close the single precision float stencil computation by our novel memory mapping mechanism is to the theoretical lower value of 8 bytes per data point in memory access without the overhead of ghost zone. Finally, we compare the performance among different implementations of stencil computation.

We measure the performance on Tesla C2050 with error correction turned off whose main hardware performance parameters are enumerated in Table 1. During the tests we guarantee the grid size nx and ny is multiples of the thread block size. All the codes were compiled by utilizing NVIDIA CUDA 5.0 for the 2.0 GPU architecture.

```
1 _global_ void cube_transit(source, target, nx, ny, nz)
2  {
3    const int ix=blockIdx.x*blockDim.x +threadIdx.x;
4    const int iy=blockIdx.y*blockDim.y+threadIdx.y;
5    For (int k=0;k<nz;k++)
6      target[ix+iy*nx+k*nx*ny]=source[ix+iy*nx+k*nx*ny];
7   }
```

Listing 4. Zero-overhead memory transit routine

6.1 Measure Memory Bandwidth of Traffic Without Ghost Zone Overhead

The global memory bandwidth is estimated by running the benchmark in Listing 4. The k-loop structure in *cube_transit* routine represents the loading and writing of 2.5D-blocking implementation. We measure the latency of transmitting a 3D data cube of SPFP values inside the global memory. The effective memory bandwidth can be calculated by the expression

$$\frac{nx \cdot ny \cdot nz \cdot (bytes\_per\_point)}{t \cdot 2^{30}} \,(\text{GB/s}) \tag{3}$$

where t is the latency in seconds of one *cube_transit* call and can be obtained by n executions of *cube_transit* and the *bytes_per_point* is 8 bytes. In Table 2 we give the calculated bandwidth *bytes_per_point* (BPP), iteration time t and the throughput in data points per second gained by cube size of $256 \times 256 \times 256$. Similarly, the throughput in the selected two other cube sizes of $192 \times 192 \times 192$ and $512 \times 512 \times 512$ can also be calculated easily. So in the last column of Table 2, the average throughput is produced by the three different cube sizes. The average memory bandwidth of *cube_transit* is approximately 75 % of the theoretical memory bandwidth on Tesla C2050 (comparing with Table 1).

Architecture	cube_transit			
	$256 \times 256 \times 256$			Avg. Throughput
	BPP [GB/s]	[GP/s]	t [s]	[GP/s]
Tesla C2050	112.7	14.1	1.20E−03	1.35E+01

Table 2. Memory bandwidth tests using benchmark cube_transit

6.2 Measure Memory Bandwidth of New Memory Mapping Mechanism

The efficiency of our new memory mapping mechanism can be analyzed by the routine *cube_ghost_transit* shown in Listing 5. What is more, by simply configuring data type and parameters of texture cache, this mapping policy shows increasing performance improvements in texture cache. Table 3 accounts for this result for cube $256 \times 256 \times 256$, where the memory bandwidth BPP is calculated by the expression (3) but the value of *bytes_per_point* is 12 bytes.

In comparison with the peak memory bandwidth (144 GB/s) given in Table 1 the throughput ($12.8 \times 12 = 153.6$ GB/s) of our texture access implementation given in Table 3 shows much advantage on Tesla C2050. It means there is much reuse of data in the ghost zone in the texture caches. In fact, we have proved it by using tools of Visual Profiler. On the other hand, It is significantly slower ($8.7 \times 12 = 104.4$ GB/s) to have access to the global memory for reduced data reuse. All these results demonstrate the texture caches can effectively eliminate the performance downside of the misalignment brought by ghost zone.

```
1  _global_ void cube_ghost_transit(in, out ,nx,ny,nz)
2  {
3   const int ix=blockIdx.x*blockDim.x+threadIdx.x;
4   const int iy=blockIdx.y*blockDim.y+threadIdx.y;
5   const uint tile_x=threadIdx.x;
6   const uint tile_y=threadIdx.y;
    // calculate the global and shared memory indices t1,
    //t1,t2, i1 and i2 acts like Listing~3
7   out+=ix+iy*nx;
8   for (int k=0;k<nz;k++) {
9     sh_m[t1]=in[i1];
10     sh_m[t2]=in[i2];
11     i1+=nx*ny;
12     i2+=nx*ny;
13     __syncthreads();
14     out[0]=sh_m[tile_x+8+(tile_y+1)*48];
15     out += nx*ny;
16     __syncthreads();
17    }
18  }
```

Listing 5. Bandwidth measure routine with ghost zones

The average transferred number of points of cube in our new global memory mapping routine *cube_ghost_transit* is computed as 12.8 GP/s in Table 3 in contrast to 13.5 GP/s in routine *cube_transit* shown in Table 2. Explicitly there is no ghost zone overhead in the running of the former. But ghost zone overhead (only reading the ghost zone and no writing the ghost zone) exists in the running of the latter.

So the average throughput of the latter is less than the former, that is to say, the latter needs more bytes of memory accesses traffic than the former in processing a data point of XY-tile. Further since the memory access traffic of the former is 8 bytes/point (4 bytes in reading and writing respectively) the effective memory access traffic of the latter can be computed as $13.5/12.8 \times 8 = 8.4$ bytes per data point in every XY-tile which is approximately 5 % more than the idealized memory access traffic lower bound value of 8 bytes per data point (without the ghost zone overhead). From Table 3 evidently the average performance is inferior to that achieved by $256 \times 256 \times 256$ cube, which is due to a lower cache reuse as can be revealed by the Visual Profiler of NVIDIA performance tools.

Architecture	Memory Type	cube_ghost_transit			
		$256 \times 256 \times 256$			Avg. Throughput
		BPP [GB/s]	[GP/s]	t [s]	[GP/s]
Tesla C2050	Global	126	10.5	1.61E−03	8.7
	texture	161	13.4	1.26E−03	12.8

Table 3. Memory bandwidth tests using benchmark cube_ghost_transit

6.3 Performance of Single Precision Floating Point Stencil Computation

There are 53 FLOPs (27 multiply and 26 add operations in all) in the PTX assembly code of the general 27-point stencil. By utilizing Common Sub-expression Elimination the number of FLOPs can be reduced to 18. On Tesla C2050 successive multiplication and addition operations are combined into a single hardware instruction: Fused Multiply Add (FMA). We summed up the assembly of the stencil routines compiled for Tesla C2050 GPU by using cuobjdump. There are 53 instructions in an iteration of the k-loop, including 27 floating point instructions (26 FMA and 1 FMUL) and other auxiliary instructions for loop branch, memory access, data copying and thread synchronization. If all instructions are executed with the peak floating point instruction number of 515 billion instructions per second, the computed maximized throughput for the general 27-point stencil is $515/53 = 9.7$ GP/s.

Listing 2 reveals the throughput of general 27-point stencil computation on Tesla C2050, along with memory bandwidth by the *cube_ghost_transit* routine and the computed constraint (9.7 GP/s) above. For the general 27-point stencil the constraint of the maximized instruction throughput is lower than the streaming memory bandwidth of *cube_ghost_transit* routine, therefore the 27-point stencil implementation is compute bound. In Figure 3, the 27-point stencil performance verges on the computed constraint above. This demonstrates that the time to access memory largely overlaps with the time to execute the instructions through GPU hardware. There are two reasons for it. Firstly, the implementation has a high occupancy, namely, there is a large number of resident warps per stream multi-processor. Consequently, when stencil data is transited there is a high likelihood that the stream

multi-processors schedule instruction to execute for some warps. Secondly, the software prefetching leads to an explicit overlap between the memory transit and computations.

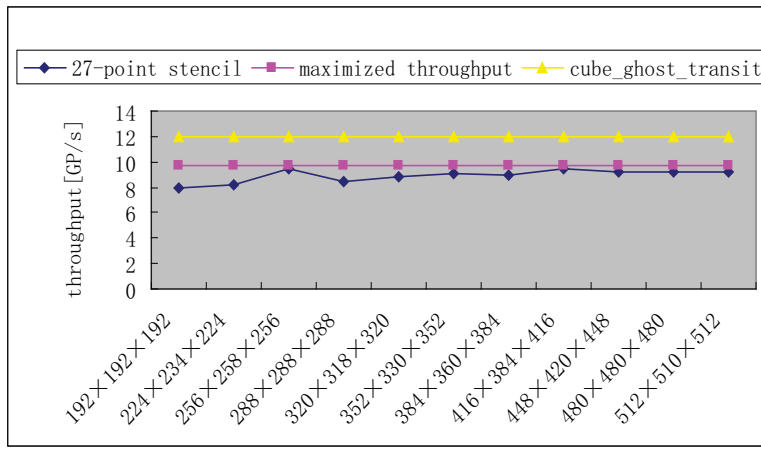

Figure 3. Bandwidth comparison among 27-point stencil, cube_ghost_transit routine and the computed constraint

In Table 4 the average performance of our stencil computation is given. The percent fraction of the peak Instruction per Cycle (IPC) is acquired by Visual Profiler. On average, the stencil routine is executed at 93 % of peak IPC and acquires approximately 46 % of the peak FLOP/s performance. In the last column, the percent fraction of the *cube_ghost_transit* routine throughput is listed as 71 % and achieves relatively lower fraction of the streaming memory bandwidth. This indicates that 27-point stencil is compute bound for Tesla C2050.

Architecture	27-Point Stencil			
	IPC [%]	[GP/s]	GFLOP/s	cube_ghost_transit [%]
Tesla C2050	93	9.1	472	71

Table 4. Average performance of single precision 27-point stencil

We ran the tests ourselves in the best thread block size of FDTD3d 7-point stencil [6] and listed the result in Table 5. From this table our new mapping implementation is almost twice faster than FDTD3d on Tesla C2050. It demonstrates that the loading policy of the shared memory has a great impact on performance. From the generated PTX code in the FDTD3d implementation 100 instructions are approximately generated in one iteration of the k-loop like Listing 1. In this NVIDIA's implementation, traditional loading policy is made available. However, the control flow divergence in traditional policy leads to non-synchronization of

32 threads in one warp, namely, two different groups' sequential execution. In this way, NVIDIA's implementation brings more instructions. The performance of the tuning framework devised by Zhang et al. [8] and Christen [10] listed in Table 5 is inferior to ours. It is because of the fact that the control flow divergence of warps brought by conditional statement is removed and higher occupancy holds due to our problem-specific hand-tuned optimization. But for another problem domain in which the intermediately processed data requires no writing into the global memory immediately in a stencil computation, a higher performance demonstrates with temporal blocking pipeline on the GTX 285 in Table 5 by Nguyen et al. [11], which is approximately 1.4 higher than ours. Recently in Naoya et al. [16] their stencil original data in ghost zone are approximately replaced by the data of nearby boundary points in the same computed XY-tile. So the communication overheads among different thread blocks require no consideration. Its optimized performance can reach 131.4 GFLOP/s shown in Table 5.

6.4 Performance of Double Precision Floating Point Stencil Computation

Table 5 also summarizes the performance of different kinds of stencil implementations. In the last two columns results of some double precision stencils are listed. Since processed data of double precision computation are twice as large as those of single precision memory bandwidth bound double precision implementation should perform roughly at 50 % of the single precision throughput. This case can be revealed by our implementation of 7-point stencil shown in Table 5. However, for the compute-bound 27-point stencil performance of double precision is only 32 % performance of single precision. Further by Nvidia's cuobjdump tool the total number of instructions of double precision implementation is about 91 more than that of single precision implementation. It is because of the fact that the coefficients of the stencil kernel are stored in the constant memory of Tesla C2050. For the single precision case the instructions get stencil coefficients directly from the constant memory, but for the double precision case the compiler remarkably produces additional instructions which load the coefficients from the constant memory to the registers. In this way, the occupancy decreases due to a relatively larger register usage. In reverse, the double precision implementation of 7-point stencil stores the coefficients in registers and achieves a relative 50 % more float point than its single precision version.

7 CONCLUSIONS

In this paper, we have devised a new memory mapping mechanism between shared memory and global memory to remove the conditional statement of the surrounding XY-tile stencil computation points by combining coalesced memory accesses of the GPU with aligned ghost zone overhead. In addition, in our stencil computation only one XY-tile is loaded into registers and the other two XY-tiles utilize the last

Policy	Type	Single Precision		Double Precision	
		GP/s	GFLOP/s	GP/s	GFLOP/s
Our implementation	7 points	12.3	97	6.5	51
	27 points	8.9	472	3.1	153
FDTD3d (Nvidia)	7 points	6.7	54		
Holewinski et al. [13]	7 points	5.9	48	3.2	26
Kamil et al. [12] GTX280	7 points			1.6	13
Nguyen et al. [11]	7 points	9.2	74	4.6	37
	7-p time steps	17	136		
Christen et al. [10]	7 points	3	24		
Zhang and Mueller [8]	7 points	10.9	87	5.7	46
Naoya et al. [16]	7 points	16.4	131		

Table 5. Performance of many kinds of different stencil computation implementation

results in the temporary registers. In this way a great deal of shared memory is saved for storing more values of XY-tiles and reduce shared memory clashes. We make full use of a thread block size of 32×6 to guarantee only two coalesced memory loading operations and no conditional statements. So it is significantly important for many-threaded GPUs to alleviate control flow divergence of a warp. On Tesla C2050 the acquired memory traffic for single precision data is 8.9 bytes per stencil computation point, only 11 % worse than the idealized value of 8 bytes about which only reading from and writing into global memory is considered and ghost zone overhead may be omitted. In comparison with previous implementation execution of our code has greater bandwidth than all other stencil executions in which the intermediate results must be written into global memory. At last our Common Sub-expression Elimination decreases the number of FLOPs in the 27-point stencil from 53 to 18.

As illustrated in the above sections, the previous optimization of stencil computation places stress on use of registers, more locality of texture cache, less control flow divergence and more effective coalesced accesses method. In our implementation, the new mapping mechanism guarantees more coalesced memory accesses and completely eliminates control flow divergence of loading boundary data points which constitutes ghost zone overhead in tiles. On the other hand, control flow divergence is weakness and bottleneck in GPU. Once it happens, two different thread groups of 32 threads in warps have to execute in sequence rather than in parallel mode. In this way, despite some other overhead, such as calculation in mapping addresses, the performance of stencil computation is improved.

Acknowledgements

The authors would like to thank to the anonymous reviewers for their valuable comments and suggestions. This work was partially supported by the National Natural Science Foundation of China (No. 61173036).

REFERENCES

[1] Taflove, A.—Hagness, S. C.: Computational Electrodynamics: The Finite-Difference Time-Domain Method. Artech House Publishers, Boston, 2005.

[2] Smith, G. D.: Numerical Solution of Partial Differential Equations: Finite Difference Methods. Oxford University Press, Philadelphia, 2004.

[3] Cong, J.—Huang, M.—Zou, Y.: Accelerating Fluid Registration Algorithm on Multi-FPGA Platforms. 2011 International Conference on Field Programmable Logic and Application, Chania, USA, September 2011, pp. 50–57, doi: 10.1109/FPL.2011.20.

[4] Datta, K.—Williams, S.—Volkov, V.—Carter, J.—Oliker, L.—Shalf, J.—Yelick, K.: Auto-Tuning the 27-Point Stencil for Multicore. 4th International Workshop on Automatic Performance Tuning (iWAPT 2009), 2009, pp. 75–84.

[5] Meng, J.—Skadron, K.: A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations. International Journal of Parallel Programming, Vol. 39, 2011, No. 1, pp. 115–142, doi: 10.1007/s10766-010-0142-5.

[6] Micikevicius, P.: 3D Finite Difference Computation on GPUs Using CUDA. Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2), ACM, New York, NY, USA, 2009, pp. 79–84, doi: 10.1145/1513895.1513905.

[7] Everett, H. P.—Massimiliano, F.: Implementing the Himeno Benchmark with CUDA on GPU Clusters. IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2010), April 19–23, 2010, IEEE Computer Society Atlanta, GA USA, pp. 1–10.

[8] Zhang, Y.—Mueller, F.: Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters. IEEE Transactions on Parallel and Distributed Systems, Vol. 24, 2013, No. 3, pp. 417–427, doi: 10.1109/TPDS.2012.160.

[9] NVIDIA Corporation: CUDA C Programming Guide. Version 5.0. 2012.

[10] Christen, M.—Schenk, O.—Burkhart, H.: PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2011), May 16–20, 2011, Anchorage, AK, pp. 676–687, doi: 10.1109/IPDPS.2011.70.

[11] Nguyen, A.—Satish, N.—Chhugani, J.—Kim, C.—Dubey, P.: 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10), November 13–19, 2010, IEEE Computer Society New Orleans, LA USA, 2010, pp. 1–13.

[12] Kamil, S.—Chan, C.—Oliker, L.—Shalf, J.—Williams, S.: An Auto-Tuning Framework for Parallel Multicore Stencil Computations. 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2010), April 19–23, 2010, IEEE Computer Society Atlanta, GA USA, pp. 1–12.

[13] Holewinski, J.—Pouchet, L.-N.—Sadayappan, P.: High-Performance Code Generation for Stencil Computations on GPU Architectures. Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12), ACM, New York, NY, USA, 2012, pp. 311–320, doi: 10.1145/2304576.2304619.

[14] Tang, Y.—Chowdhury, R. A.—Kuszmaul, B. C.—Luk, C.-K.—Leiserson, C. E.: The Pochoir Stencil Compiler. Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11), 2011, pp. 117–128, doi: 10.1145/1989493.1989508.

[15] Unat, D.—Cai, X.—Baden, S. B.: Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. Proceedings of the 25th International Conference on Supercomputing, May 31–June 4, 2011, ACM, Tuscon, Arizona, USA, pp. 214–224, doi: 10.1145/1995896.1995932.

[16] Maruyama, N.—Aoki, T.: Optimizing Stencil Computations for NVIDIA Kepler GPUs. First International Workshop on High-Performance Stencil Computations (HiStencils '14), January 21, 2014, Vienna, Austria.

Tieqiang Mo is currently Assistant Professor in computer science and electronic engineering at Hunan Univerisity. He received his Ph.D. degree from Hunan University, China in 2016. Currently, he is engaged in the research of parallel and distributed algorithms, especially skilled in heterogeneous algorithms and automatic code generation.

Renfa Li received his Ph.D. degree in electronic engineering from Huazhong University of Science and Technology, Wuhan, China, in 2002. He is currently Professor of computer science and electronic engineering with Hunan University, Changsha, China. He is the Director with the Key Laboratory for Embedded and Network Computing of Hunan Province, Changsha, China. He is also an expert committee member of the National Supercomputing Center in Changsha, China. His major interests include computer architectures, distributed computing systems and code optimization. He is a member of the Council of CCF and a Senior Member of ACM.