GPU Optimization for Stencil-Based Hemodynamics Simulation

Cindy Ouyang, Rex Ying, Ellie Zheng

1 Introduction

With the advent of the National Strategic Computing Initiative (NSCI), improved high-performance computing will allow researchers to perform more massive simulations than ever before (Tom, 2015). Preparing for the next generation of GPU-based supercomputers means understanding how to optimize GPU's now. We will look at Dr. Amanda Randles's computational model of blood flow, which uses the Lattice Boltzmann Method, a particularly beneficial for computational modeling because it is stencil-based. Stencil computations involve continually updating a point on a multi-dimensional grid using the values of its neighbors at each time step. In this paper, we will explore how to best optimize this pattern using OpenMP, Cilk, CUDA in tandem with data reorganization.

2 Background

HARVEY (Amanda, 2013; Amanda, 2015) is a massively parallel code to simulate image-based fluid dynamics based on the lattice Boltzmann method (LBM) in three dimensions. Currently, the code runs on MPI and OpenMP. Our aim is to extend the efficiency of HARVEY by porting this code to GPU-based architecture and exploring other levels of parallelism.

LBM is used for this massive computational work, because it is simple, time-explicit, and accurate. Moreover, the stencil-based nature as described below also makes LBM well-suited for massively parallel simulations. In LBM, the evolution of the distribution function fi(x, t) is described as follows:

$$f_i(\vec{x} + \vec{c}_i \Delta t, t + \Delta t) = f_i(\vec{x}, t) - \omega \Delta t \left[f_i(\vec{x}, t) - f_i^{eq}(\vec{x}, t) \right] \quad (1)$$

The quantity $f_i(\vec{x}, t)$ describes the probability of finding a particle at grid point x and time t, with implicit discretized velocity ci and grid spacing Δx. In our study, a 19-point cubic stencil is used for the velocity, where a grid point connects to its first and second neighbors (referred to as D3Q19). This algorithm contains two key components: streaming and collision, given by the first and second terms in the right hand side of Eq. (1). The streaming step propagates the fluid particles to adjacent lattice points along the velocity trajectory defined by D3Q19. The collision step is calculated through a relaxation toward equilibrium, where the local equilibrium $f_i^{eq}(\vec{x}, t)$ is defined in terms of the density, the average fluid speed and parameters determined by D3Q19 and the lattice structure.

The input of the HARVEY code is a 3D model converted from patient-specific data. The geometry is then segmented by crosscutting, and the segmented pieces are assigned to each processor via MPI. The boundary conditions imposed at the top boundary is a full bounce-back such that the fluid particles will bounce back in the same direction in which they hit the wall. Those imposed to the left and right walls are periodic. The local MPI task deals with a full cardiac cycle, which iterates through each time grid containing three steps: collision, streaming (advection), and a boundary condition check. These steps are iterated for each fluid node in the three-dimensional space. Detailed implementations are introduced in Section 4.1.

For simplicity, we will use a toy code written in C instead of the full HARVEY code, where only the local MPI tasks are included. Additionally, the input data is a simplified vessel model with much smaller dimensions. Our local changes to the toy code, which enhance the performance by a large scale, could be implemented in the HARVEY code in future studies.

3 Related Work

Optimizing stencil-based computations has been a topic of many recent studies. Holewinski *et al.* introduces an automatic code generation scheme for stencil computations on GPU accelerators by developing compiler algorithms, specifically using overlapped tiling as a technique to reduce data sharing requirements (Justin, 2012). Others, such as Maruyama *et al.*, have also introduced frameworks that will automatically translate user-written code into implementable code in CUDA (Naoya, 2011). In another article, Datta *et al.* investigates the optimization of many multi-core architectures, employing an autotuning environment that will search the optimizations to minimize runtime (Kaushik, 2008). They explore four broad categories of optimizations, oftentimes on the hardware level. First, problem decomposition: breaking down a node block (the full grid) into core blocks, then into thread blocks and finally register blocks. Second, data allocation: allocating the source and destination grids as one large array. Third, bandwidth optimizations: hardware prefetching, software prefetching, and multithreading in order to hide memory latency. Lastly, in-core optimizations: creating ISA-specific code generators that could output SIMD code. Perhaps the article most relevant to our project and Professor Randles's research is the study of Feichtinger *et al.* who have proposed the optimization of the LBM using multi-GPU implementations, resulting in nearly perfect weak scalability (Christian, 2011; Anthony, 2010). In addition to this, they also performed analyses on the performance of heterogeneous CPU-GPU clusters. Most of the relevant work relies heavily on the architecture on which the system is built.

In this paper, we mainly exploit parallelism on the software level, experimenting on approaches introduced in class, as well as some techniques inspired by the literature, and apply them to our specific application. Our goal is to produce parallel implementations that are both efficient and more independent of the architecture.

4 Methods

4.1 The Serial Algorithm

The resulting particle distribution functions are stored in a 4D array: *distr*[direction=19][x][y][z], where the last three indices define the position in a three-dimensional space, and the first index indicates the flow direction to 19 neighbors. Possibilities for each flow direction are read from input and stored separately.

The two steps -- collision (col) and streaming (advection, adv) are in serial (see Figure 1). In the collision step, temporary array *distr_adv* is calculated from *distr* based on Equation 1 for each grid point by iterating through three for_loops (corresponding to three dimensions). In the streaming (or advection) step, *distr* array is updated by accumulating temp results stored in *distr_adv* from 19 neighboring grids. This step is a stencil, also including three for_loops.

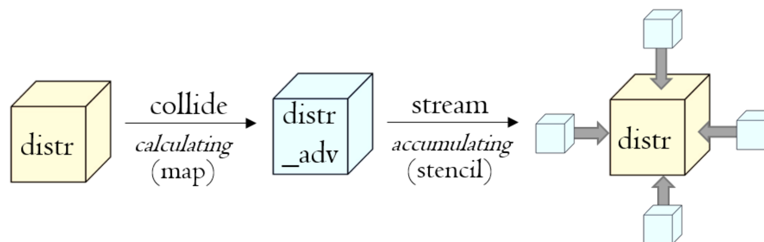

Figure 1 Serial implementation of the collision and the streaming (advection) steps.

4.2 Parallel Implementation

4.2.1 OpenMP

OpenMP involved parallelizing the nested for loops in collision and advection by testing a variety of factors: number of parallelized loops and thread count. First, we tested using one parallel_for on the outer loop for both collision and advection; then, we tested with two and three nested for_loops. In addition to this, we also varied the number of threads for each variation of nested loops, using 4, 16, 32 and 64 threads. Lastly, after reorganizing the data by combining the outer two for loops into [ix*iy][iz][direction], we also tried to implement OpenMP with this reorganization for further speedup. In the earlier OpenMP computations, we noticed that collision was fastest when serial, whereas advection noticed a significant speedup with one parallel_for loop. As a result, we tested the reorganized data using no parallel_for loops for collision and one parallel_for loop for advection.

In advection, the stencil pattern is an accumulation: *distr* += *distr_adv*[19 neighbors]. For simplicity, the toy code treats the accumulation as a random overwrite: *distr* = *distr_adv*. Therefore, currently we do not consider the concurrency issues arose from doing this stencil pattern in parallel. However, this problem could be resolved by using atomic operations supported by #pragma critical/atomic, or using locks by invoking omp_set/unset_lock in OpenMP in the HARVEY code. Locks could be used in CUDA implementations as well.

4.2.2 Cilk

Cilk involved two aspects: 1) parallelizing for_loops by turning them into recursions using cilk_spawn, 2) exploiting task parallelism as much as possible by analyzing variable dependencies inside of the three for_loops (see Figure 2).

1. The base case for the recursion is: the grain size for x and y dimensions is 1, for z dimension it is varied from 2 to 10 in order to determine an optimized base case size. Bisection is used to split the tasks until the base case is matched. This transformation is done to both collision and advection.
2. Computations of the variables are parallelized as long as they are independent. Data dependencies for collision and advection are analyzed in Figure 2.

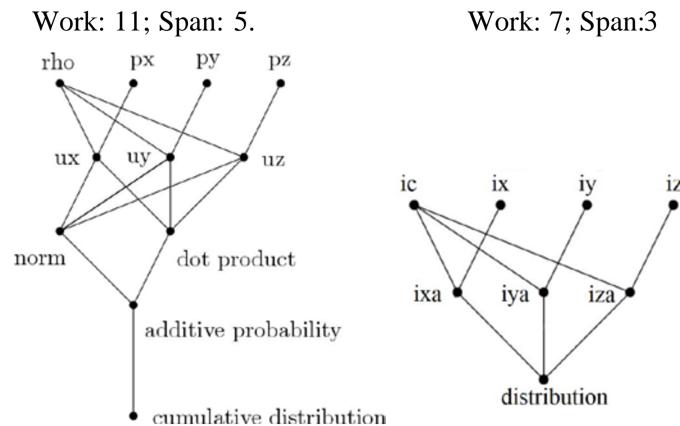

Figure 2 The data dependency graph for collision and streaming (advection), respectively, inside of the three for_loops.

a. For collision, calculations of variables *rho, px, py, pz* are done in parallel by cilk_spawn, each requires looping through 19 directions. Other independent calculations are not spawned, as they are one-step atomic operations and not rate-determining.
b. For advection, *ixa, iya, iza* are not dependent on *distr_adv*, therefore, they are spawned to be calculated in parallel with collision steps.

4.2.3 CUDA

As demonstrated by the abundant literature on GPU implementation of stencil pattern, such as Paulious *et al*. (2009), Antony *et al*. (2010) and Christian *et al*. (2013), we decide to try implementation of the stencil computation on GPU using CUDA.

The first version of our CUDA implementation uses two kernel functions for collision and streaming respectively. Firstly, due to constraint of CUDA global memory, we collapse the 4D array representing the distribution function into 1D. We use a 3D block of size adjustable using parameters. The device copy of the distribution function array is allocated and initialized first. The kernel function collision computes the lattice Boltzmann stencil and stores values in another output array in global memory. The second kernel function advection reads values from the output array of collision, computes the result and writes back to the original distribution function array in global memory. After that, boundary values are updated based on the given boundary conditions. The implementation executes the above steps for the required number of iterations. Finally, the resulting distribution function is copied to host memory.

We then consider several modifications that could potentially improve its efficiency further. Further speedup might be achieved by combining processes together in CUDA implementation. When a CUDA kernel function is invoked, there will be overhead that stems from the following operations (John, *et al*., 2008): (1) the kernel call dynamically creates a new grid with the right number of thread blocks and threads for that application step; (2) it allocates device memory (3) indices are computed in each call to locate the memory that should be accessed by each GPU thread. For program with multiple kernel calls, The overhead (1) and (3) can be reduced by combining the kernel calls, in our case, the collision and advection. This is not always possible to do efficiently. For example, 2 consecutive n-by-n stencils take $2n^2$ operations for each point, but if combined into one function, the convolution matrix becomes $2n$-by-$2n$, resulting in $4n^2$ operations needed for each point. The extra computation will outweigh the benefit of combining kernel calls in most cases.

However, in our case, the first kernel call for collision uses a neighborhood of velocity trajectories around the current direction, whereas the advection uses a neighborhood of spatial locations, the *x, y,* and *z* components. The property of separation of dimensions in both stencil computations allows us to combine the two processes, without overhead. This results in the following implementation.
For each GPU thread, the implementation first computes collision using the same algorithm. It then identifies all target cells, which are defined as the cells whose advection stencil computation would make use of the collision result at current cell. For each of the target cell, the algorithm uses the appropriate cumulative function to incorporate the value into the target cell and update the result in an output array. A schematic visualization is shown in the following figure.

Note that the same issue of concurrent writes need to be dealt with in this CUDA implementation. According to the CUDA C programming guide (Nvidia, 2011), one can use atomic operations such as atomicAdd to make sure that the output values at the target cell is accumulated correctly.

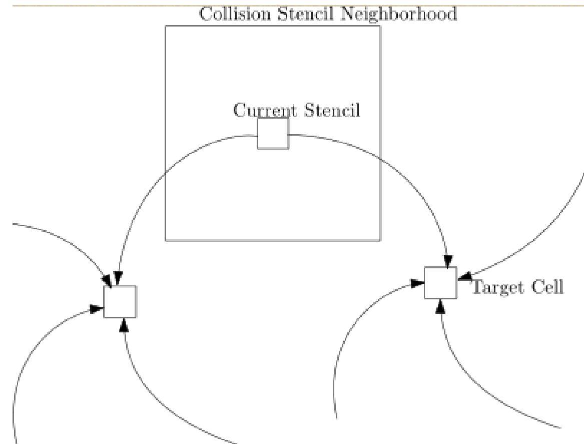

Figure 3 Illustration of merging collision and advection stencil computations. The current stencil point is identified by the GPU block index and thread index.

4.2.4 CUDA with in-place update

The basic idea of this potential optimization is to make use of the fast on-chip shared memory with low latency during stencil computation. As we can see from Figure 1, each cell of the output array needs to be updated by multiple GPU threads in the advection process. However, since access to global memory is slower compared to access to shared memory (David, 2007), we expect an improvement if in-place update is used.

Our implementation is based on the assumption that due to the small time step in the high precision simulation, the discretized velocity is bounded above by a small constant k compared to the size of the grid (in our case, the velocity is less than 4 in all directions). In another word, if we denote the current cell for a GPU thread as p, and the target cells as $p_{t1}, p_{t2}, \ldots$, we can ensure that

$$|p_{ti}.x - p.x| \le k; |p_{ti}.y - p.y| \le k; |p_{ti}.z - p.z| \le k; \quad (2)$$

for all its target cells p_{ti}. Therefore the target cells of all threads in a block of size $b.x \times b.y \times b.z$ are within the region containing the block itself as well as a halo of size k.
Based on the above formulation, we first copied the block and its halo into shared memory, perform computation the same way as introduced in the previous CUDA section, but store the resulting values in the shared memory instead of global memory. Finally, after a "__syncthreads()" call, we transfer the resulting accumulated value back to global memory.

4.2.5 Data reorganization

As mentioned above, the distribution function *distr* is stored as a 4D array originally, but it is converted to 3D array for OpenMP implementation, and to 1D array for CUDA implementations. Therefore, the serial codes with different data segmented methods (4D, 3D and 1D) are tested as a baseline for comparison. A mutation of the indices from *distr*[direction][ix][iy][iz] to *distr*[ix][iy][iz][direction] is also carried out, as operations on the 19 directions are done within the three for_loops over *x, y* and *z* axis, and thus the 19 directions should be consecutive in memory.

4.3 Methods of Evaluation

Timing results are collected for the collision step, the streaming (advection) step, and the total iterations through the cycle of blood flow for each implementation. Speedup are calculated on the basis of the original code. Scalability is not explicitly evaluated as we only work on the toy code, but we address possible issues related to large data sizes in the following section.

5 Results and Discussion

5.1 Data reorganization

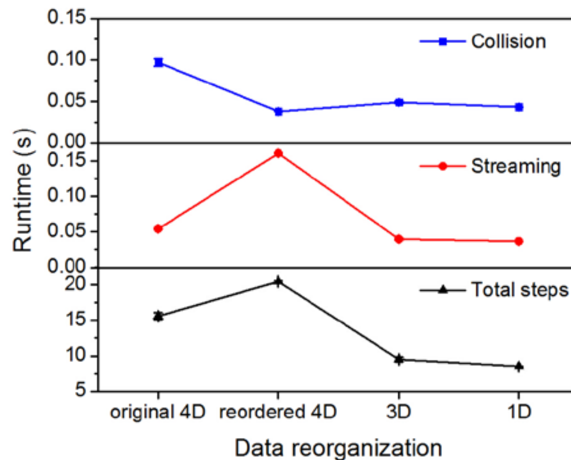

Figure 4 Runtimes for data reorganizations of the *distr* array.

As shown in Figure 4, reducing the dimensions of the distribution function array speed up the runtimes. The *distr* array is initialized by looping through *x, y, z* axes and then the 19 directions, which is the same order as in the calculation steps. Therefore, the ways of using cache by 4D, 3D and 1D arrays are literally the same, and 1D array reduces the cost of tracing indices. This also explains why reordering the indices of the 4D array leads to longer execution time, as the order of the loops for the initialization and the calculations are not consistent in this case. However, there will be potential issues associated with allocating consecutive memories for lower dimensional arrays when the data size increases significantly.

5.2 OpenMP

Our series of performance tests indicate that there is no speedup when adding additional nested parallel_for loops (Figure 5a). In fact, performance decreases. Moreover, as shown in Figure 5(c), the overall runtime seems to be correlated with the collision time. In other words, speeding up the OpenMP implementation means minimizing the collision runtime. This is also evident in the right half of Figure 5a. While it doesn't seem like the streaming time changed significantly, there is a significant difference in runtime between 4 threads and 32 threads (Figure 5b). Using 4 threads provided the best speedup, perhaps because collision is fastest using 4 threads.

Since we noticed that collision runs fastest serially, perhaps because of its intense computations, we suspected that combining a running collision serially and streaming in parallel would be the best combination for overall runtime. As expected, data reorganization *without* the collision for loop and with one advection for loop resulted in the best speedup.

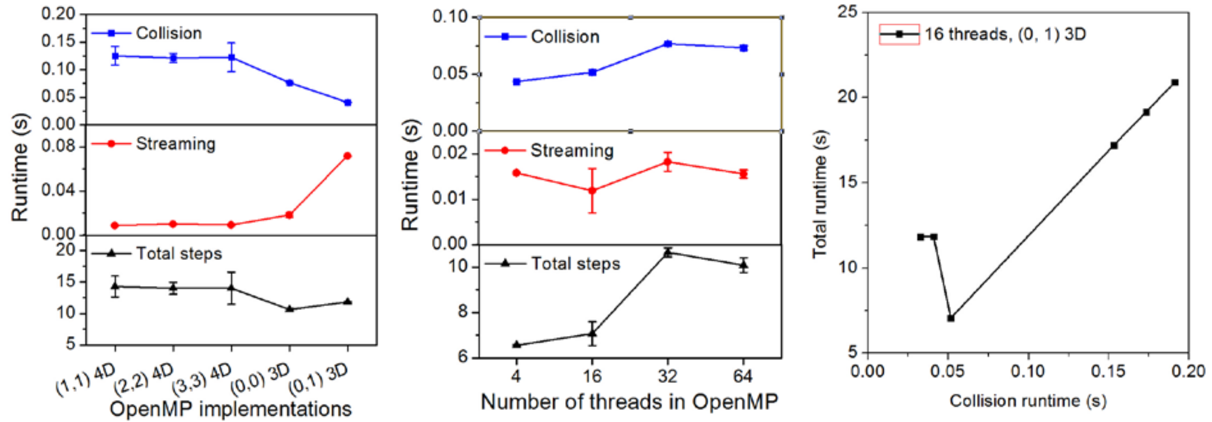

Figure 5 (a) Runtimes for different OpenMP implementations. The notation in the *x*-axis labels means "(number of parallel for loops for collision, number of parallel for loops for streaming) data_reorganization basis". (b) Runtimes as a function of the number of threads used in OpenMP. (c) Dependencies of the total runtime on the collision runtime in OpenMP implementations with different number of threads

5.3 Cilk

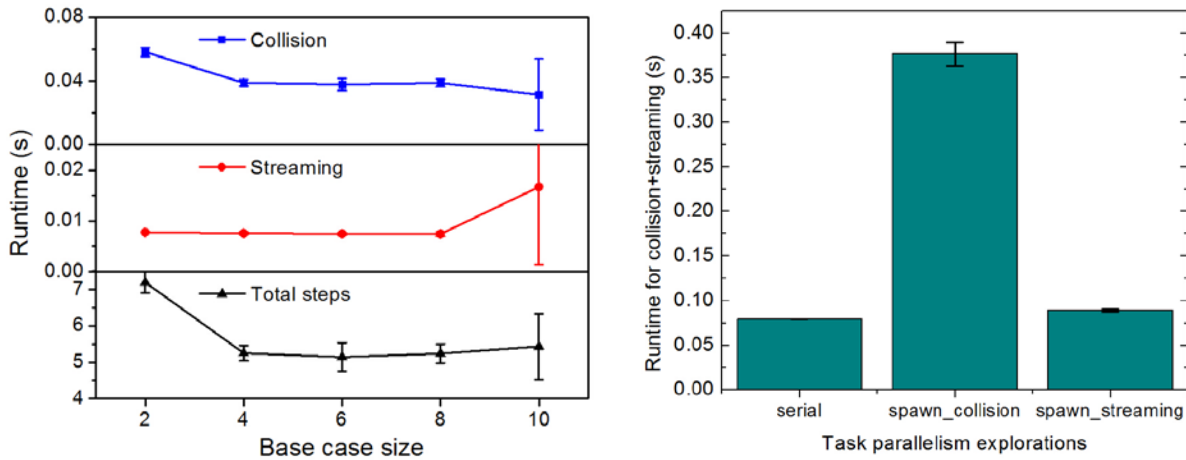

Figure 6 (a) Runtimes for converting for_loops into recursion by Cilk_spawn in collision and streaming steps under different base case sizes (grain size for *z* direction). (b) Runtimes for employing task parallelism in collision or streaming step by Cilk_spawn, compared with the serial version.

Similar to the OpenMP implementations, rewriting for_loops into recursive fork-join using cilk_spawn/cilk_sync also parallelized the map and stencil patterns in collision and streaming. However, Table 1 shows that the Cilk implementation can be more efficient (24% faster) than OpenMP with the use of an optimized base case size. This might be universal for systems where multiple loops are involved. Figure 6(a) shows that the choice of the base case is essential for this high efficiency, which should balances the scheduling overheads and the scale of parallelism.

Extra level of task parallelism is realized by cilk_spawn. As mentioned in Section 4.2.2, calculating *rho, px, py* and *pz* in parallel in the collision step should bring about a speedup of 4 theoretically. However, in

contrast, the first and second columns in Figure 6(b) shows that it slows down the collision step by more than 3 times. This should be attributed to the vast scheduling overhead compared with the size of the tasks. For similar reasons, calculating *ixa, iya* and *iza* in parallel with the collision step (third column in Figure 6b) gives similar runtime as the original serial version. We can conclude that employing extra task parallelism does not evidently speed up the collision and advection steps due to the relatively large overhead.

5.4 CUDA implementation

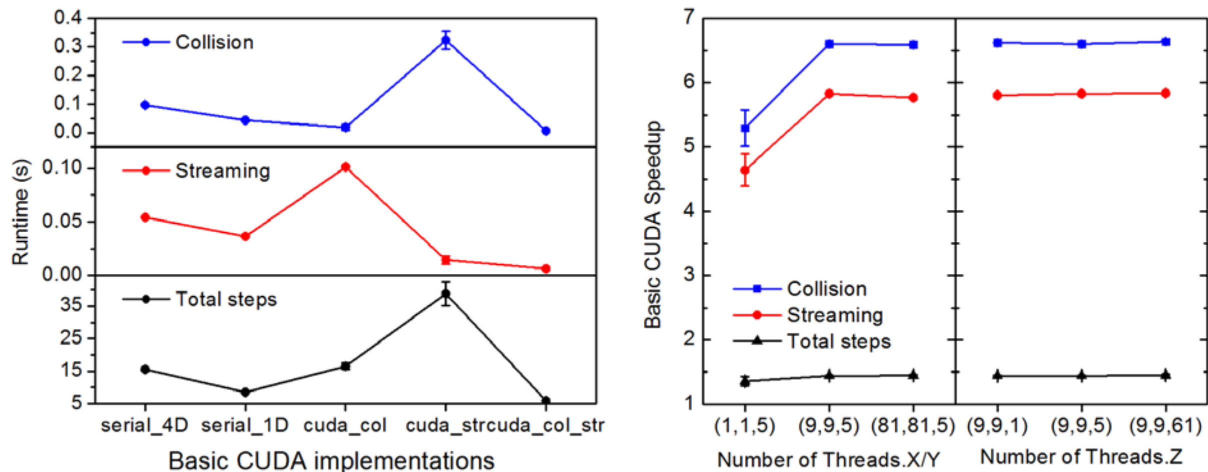

Figure 7 (a) Runtime for basic CUDA implementations of collision or/and streaming using 1D array. (b) Speedup for basic CUDA implementations using different number of threads in the 3D (9, 9, 5*19) block.

Table 1 Selected (Best) results for each implementation

	Original	Data_reorg: 1D	OpenMP	Cilk	Task Parallelism	Basic CUDA	In-place update CUDA
Runtime/s (col+adv)	0.151	0.080	0.059	0.045	0.089	0.013	0.007
Total runtime/s	15.599	8.542	6.557	5.140	10.083	5.924	0.6972
speedup	N/A	1.9	2.6	3.4	1.7	11.6	21.6

As we expected, GPU excels at iterative stencil computations. The speedup achieved from GPU (11.6) is significantly higher than all other methods that use CPU only. The reason behind is that GPU typically contains a much larger number of threads. Our stencil computation does not involve complex functions, and is mostly homogeneous for all cells. These are the important characteristics for which GPU is advantageous (John D. *et al*. 2008).

Figure 7(a) shows that implementing both collision and streaming (advection) in CUDA give the least runtime. For the in-place update CUDA implementation, these two steps are combined and involved as a single GPU kernel function, which further reduces the runtime of collision+advection by two (Table 1). After exploring different number of threads utilized for each dimension (x, y and z) within a 3D block, we found that the speedup reaches a plateau when the number of threads for x, y and z is increased (9, 9, 5). This value is set by default in further improvements on the basic CUDA implementations.

As shown in Table 1, the CUDA implementation with in-place update produces the best results, having a clear advantage compared to our previous CUDA implementations. The result demonstrates the advantage of the shared memory in situations where it is updated by multiple GPU threads, a fact that is also demonstrated theoretically in Shane *et al*.

In our implementation, the size of halo is set as precisely the same as k, the upper-bound on the maximum discretized velocity in the simulation system. A halo of larger size could enable multiple in-place updates, before a global synchronization, thus potentially improving the runtime further. However, we note that there is a space-time tradeoff when varying the size of halo. As halo gets larger, the benefit of less global synchronization comes at the expense of larger number of ghost cells added into the shared memory. This increase in the number of ghost cells is especially salient in higher dimensions. In our case, we need to consider all 3 spatial dimensions, x, y and z. Let n denote the block side length in all three dimensions (for simplicity we assume a cubic block here), then the total number of ghost cells should be:

$$(n+k)^3 - n^3 = 3kn^2 + 3k^2n + k^3 \quad (3)$$

Even for constant k, we can see that the number of ghost cells grows quadratically with respect to the side length of the block. In practice, we observe that the block size and the number of blocks that can be stored in one machine decrease dramatically for a halo of size $2k$. Considering the memory constraint, we determine that the most appropriate size of halo is k.

In this paper, we used an empirical approach to determine the size of the halo. Meng *et al*. (2009) proposed a new algorithm that could automatically determine the size of the halo for stencil computation. When the memory constraint is not so tight, this technique could be used to effectively determine the optimal size of halo in this application.

7 Conclusions

In this paper, we presented several alternatives that exploit parallelism in the stencil-based hemodynamics simulation. We first utilized data reorganization to achieve higher rate of cache hits. OpenMP improves the runtime by running for loops of stencil computation in parallel. In addition to that, based on our analysis of data dependence graph, we use a Cilk implementation to exploit extra level of parallelism by computing variables that are independent in parallel via cilk_spawn and cilk_sync functions. However, the overhead greatly limits the speedup of task parallelism. Massive parallelism can be achieved if the application is run on GPU. We showed that using CUDA, it is possible to achieve a speedup of 11.6. Ultimately, we explored various optimizations based on the characteristics of the simulation algorithm, namely merging CUDA kernel functions and in-place update with halo. These techniques further boost speedup to above 20. The resulting implementation can then be run on a large computing cluster, with each computer being in charge of simulating one part of the body. Using the MPI for message passing, we can achieve our goal of running the hemodynamics simulation on the entire human body in real time.

References

1. Datta, Kaushik, *et al*. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures." *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008.
2. Feichtinger, Christian, *et al*. "A flexible patch-based lattice Boltzmann parallelization approach for heterogeneous GPU–CPU clusters." *Parallel Computing* 37.9 (2011): 536-549.
3. Harris, Mark. "Optimizing parallel reduction in CUDA." *NVIDIA Developer Technology* 2.4 (2007).
4. Holewinski, Justin, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. "High-performance code generation for stencil computations on GPU architectures."*Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012.
5. Kalil, Tom, and Jason Miller. "Advancing U.S. Leadership in High-Performance Computing." *The White House*. The White House, 29 July 2015. Web. 4 Dec. 2015.
6. Kirk, David. "NVIDIA CUDA software and GPU parallel computing architecture."*ISMM*. Vol. 7. 2007.
7. Maruyama, Naoya, *et al*. "Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers." *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE, 2011.
8. Meng, Jiayuan, and Kevin Skadron. "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs." *Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009.
9. Micikevicius, Paulius. "3D finite difference computation on GPUs using CUDA."*Proceedings of 2nd workshop on general purpose processing on graphics processing units*. ACM, 2009.
10. Nickolls, John, *et al*. "Scalable parallel programming with CUDA." *Queue* 6.2 (2008): 40-53.
11. Nguyen, Anthony, *et al*. "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*. IEEE Computer Society, 2010.
12. Nvidia, "Nvidia cuda c programming guide." *NVIDIA Corporation*120 (2011): 18.
13. Obrecht, Christian, *et al*. "Multi-GPU implementation of the lattice Boltzmann method." *Computers & Mathematics with Applications* 65.2 (2013): 252-261.
14. Ryoo, Shane, *et al*. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA." *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008.
15. Owens, John D., *et al*. "GPU computing." *Proceedings of the IEEE* 96.5 (2008): 879-899.
16. Randles, Amanda, *et al*. "Massively parallel simulations of hemodynamics in the primary large arteries of the human vasculature." *Journal of Computational Science* 9 (2015): 70-75.
17. Randles, Amanda, *et al*. "Performance analysis of the lattice Boltzmann model beyond Navier-Stokes." *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013.