Efficient Particle-Mesh Spreading on GPUs

*Xiangyu Guo, †Xing Liu, *Peng Xu, *Zhihui Du and †Edmond Chow
*Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University, 100084, Beijing, China
E-mail: {csgxy123,bly930725}@gmail.com, duzh@tsinghua.edu.cn
†School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, Georgia, 30332, USA
Email: xing.liu@gatech.edu, echow@cc.gatech.edu

***Abstract*—The particle-mesh spreading operation maps a value at an arbitrary particle position to contributions at regular positions on a mesh. This operation is often used when a calculation involving irregular positions is to be performed in Fourier space. We study several approaches for particle-mesh spreading on GPUs. A central concern is the use of atomic operations. We are also concerned with the case where spreading is performed multiple times using the same particle configuration, which opens the possibility of preprocessing to accelerate the overall computation time. Experimental tests show which algorithms are best under which circumstances.**

***Keywords*-particle-mesh; spreading; interpolation; sparse matrices; GPU; warp shuffle**

I. Introduction

Many scientific applications involve both a set of particles that can reside at arbitrary locations in space, and a Cartesian mesh with regularly-spaced mesh points. Given a set of values, such as velocities, on the mesh points, it may be desired to find the interpolated values at the arbitrary particle locations. This is called the particle-mesh *interpolation* operation. Mesh points nearby the particle are used to interpolate the value of the quantity at that particle. The inverse operation takes values at particle positions and contributes them to values at nearby mesh points. This is called the particle-mesh *spreading* operation. The topic of this paper is particle-mesh spreading. The operation is a key step in the non-equispaced fast Fourier transform [1], [2], with applications including tomography [3], magnetic resonance imaging [4] and ultrasound [5]. Particle-mesh spreading is also used in the particle-mesh Ewald summation (PME) method [6], widely used in molecular dynamics [7] and other types of simulations [8], [9] to evaluate long-range interactions between particles.

In various particle-mesh applications, given quantities located at particle positions, such as velocities, forces or charges, are mapped onto a 3D regular mesh. The spreading contributes to a $p \times p \times p$ region of the mesh roughly centered at the particle. The value of p is related to the order of the (inverse) interpolation function. Figure 1 illustrates the particle-mesh spreading of two particles onto a 2D mesh using $p = 4$.

While both particle-mesh interpolation and spreading are important, we focus on the latter because it is much more challenging to obtain high performance for the spreading operation. The reason why these operations have very different performance characteristics is because data structures

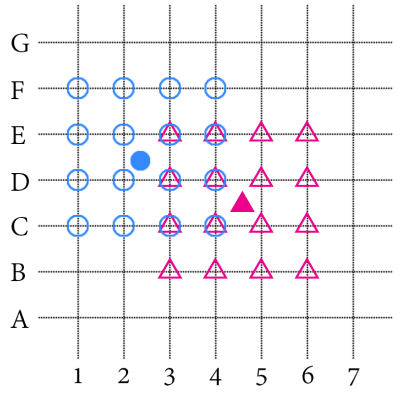


Figure 1: Particle-mesh spreading onto a 2D mesh with $p = 4$. The solid green circle and red triangle represent two particles. The mesh points receiving contributions from these particles are shown with green circles and red triangles, respectively.

are usually particle based rather than mesh based. This is due to the fact that it is easy to determine the neighboring mesh points of a particle, but not easy to efficiently determine the neighboring particles of a mesh point, especially when particles can move. For the spreading operation, a natural parallelization across particles means that the mesh variables are shared, and locking/waiting is needed to control access to these variables. For the interpolation operation, the quantity at each particle is simply computed by reading the values at nearby mesh points. This paper focuses on the particle-mesh spreading operation on GPUs, where large numbers of threads may be contending for writes on mesh variables.

The simple method of parallelizing particle-mesh spreading on GPUs is to use one thread to perform the spreading operation for each particle. As mentioned, this requires using expensive atomic operations as multiple threads might attempt to update the same mesh location simultaneously. Additional challenges arise from the sparse and irregular nature of spreading, making it hard to achieve load balance and coalesced memory access, leading to poor performance on GPU hardware.

Previous research on particle-mesh spreading on GPUs attempt to enhance coalesced memory access and partially avoid the use of atomic operations [10], [11]. In these studies, a preprocessing step is used to create a mesh based data structure. Each mesh point can store a single particle [10] or a list of particles [11]. No atomic operations are needed to perform the actual spreading operation because a single thread sums the contributions for a mesh point using

the mesh based data structure.

A number of issues can be raised with the above mesh based approach. Performance is highly dependent on the number of particles per grid point. (The relationship between the number of particles and the number of grid points is chosen by balancing accuracy and cost.) For fewer than 1 particle per grid point on average, the mesh based approach may be inefficient because of the large number of mesh points not associated with particles. Also, while avoiding atomic operations is a good optimization guideline, on recent GPU microarchitectures, e. g., the Kepler GK110, the atomic operation throughput has been substantially improved, making particle based approaches more competitive.

In this paper, we are particularly interested in a new use case for particle-mesh spreading, making it worthwhile to reinvestigate these algorithms. In traditional uses of particle-mesh spreading, the operation is performed once for a given configuration of particles, where a configuration is a set of particle locations. The new use case is to perform the spreading operation multiple times for the same particle configuration. This is necessary when the spreading operation is performed inside an iterative method, for example, inside the Lanczos algorithm, to compute Brownian displacements in Brownian dynamics simulations for the given particle configuration [9]. This use case means that it may be profitable to perform some preprocessing, such as construction of mesh based data structures, to speed up the overall computation.

The main contribution of this paper is two-fold: 1) propose a new algorithm for computing a mesh based data structure on GPUs that is useful when the spreading operation is performed multiple times, and 2) propose a technique of using GPU warp shuffle operations to optimize the spreading operation with the mesh based structure. It is unlikely that one single spreading method achieves the best performance for all applications, with different densities of particles relative to mesh points. To fully understand when to use what algorithms, we compare several spreading algorithms using well-selected test cases. For example, we will show that particle based approaches are now very fast on GPUs, given improvements in the speed of atomic operations.

II. Critique of Existing Approaches

A. Particle Based Approach

The simple particle based approach assigns one thread per particle to perform the spreading operations. Because multiple threads working on nearby particles may need to update the same mesh points concurrently, the use of atomic operations is generally necessary. While this approach may work well on CPUs, it is traditionally thought to be inefficient on GPUs where atomic operations are relatively more expensive.

A major advantage of the particle based approach is that it only needs a simple data structure, consisting of the list of particles and their coordinates. The (inverse) interpolation coefficients are computed "on-the-fly" using the particle coordinates.

B. Mesh Based Approach

The mesh based approach, in contrast to the particle based approach, assigns threads to mesh points. This is the approach in the clever work of Harvey and Fabritiis [10] on NVIDIA's Tesla microarchitecture. The basic idea is to use a "gather" for each mesh point rather than a "spread" for each particle. The algorithm consists of three steps. In the first step, each particle is placed at the nearest mesh point. Atomic operations are still needed in this step, but they are much fewer than in the particle based approach (by a factor of p^3 because particles rather than spreading contributions are collected at the mesh points). Each mesh point can hold at most one particle, so any additional particles are placed on an overflow list. In the second step, the actual spreading operation is performed at each mesh point by gathering contributions from particles placed in the surrounding p^3 mesh points. Since each thread only updates one mesh point, the use of atomic operations is not needed in this step. As designed, memory access is coalesced in this step as adjacent threads update adjacent mesh points. In the third step, particles on the overflow list are processed using the particle based approach. This algorithm follows the paradigm of dividing the computation into a regular part and an irregular part. The regular part can be computed quickly on GPU hardware and hopefully dominates the irregular part. In this paper, we refer to this specific mesh based algorithm as the "Gather algorithm."

When the number of particles is smaller than the number of mesh points, the Gather algorithm has much more memory transactions than the particle based approach. This may be an acceptable cost if it is lower than the penalty of using atomic operations. This was the case for the Tesla microarchitecture used by Harvey and Fabritiis [10], but on NVIDIA's Kepler microarchitecture where atomic operations can be as fast as global memory load operations, the extra memory transactions may outweigh the gain of avoiding atomic operations.

Another potential disadvantage of the Gather algorithm is that the interpolation weights must be computed multiple times, once for every particle contributing to a mesh point, rather than simply once for every particle in the particle based approach. This is because the interpolation weights for a particle depends on a particle's position. In essence, the interpolation weights are computed p^3 times rather than once. In this paper, we use cardinal B-spline interpolation (used in the smooth PME method [12]). For high order B-spline interpolation, e.g., $p > 7$, the interpolation weights are often computed via a recursive process, making this cost significant.

We note that when the Gather algorithm for spreading must be performed many times for the same particle configuration, the result of the first placement step of the Gather algorithm can be saved and reused. In this paper, we have implemented and optimized the Gather algorithm in order to study its performance for various test cases.

C. Multicoloring Approach

In our previous work on Intel Xeon Phi, we parallelized the spreading operation by using a particle based approach that does not need atomic operations [9]. Multicoloring is used to partition the particles into sets called "colors." Spreading is performed in stages, each corresponding to a color. In each stage, a thread is assigned a subset of the

particles of the current color such that each thread can update mesh locations without conflict from other threads. This algorithm, however, is not appropriate for GPUs because of limited parallelism, due to the fact that each thread must be assigned the spreading operation for many particles. In essence, the particles assigned to a thread must be processed sequentially, otherwise conflicts would occur. We will not discuss the multicoloring approach further in this paper.

III. Proposed Mesh Based Approaches

A. Relation to Sparse Matrix Techniques

When the spreading operation is performed multiple times for the same particle configuration, it may be worthwhile to separately consider a preprocessing step and a spreading step such that the spreading step is as fast as possible, and the cost of the preprocessing step can be amortized over the multiple spreading operations. To avoid needing atomic operations in the spreading step, the preprocessing step generally needs to compute a mesh based data structure. The mesh based data structure computed by the Gather algorithm, however, has two main issues: 1) it requires performing gather operations on every mesh point even for mesh points that do not have particles spreading onto them, and 2) it requires recomputing the interpolation coefficients many times.

In order to make the spreading step as fast as possible, it is tempting to use a different mesh based data structure where the interpolation coefficients are stored and not recomputed. This addresses the second problem above, but introduces the drawback that DRAM reads would be needed for the interpolation coefficients. Although these reads can be coalesced, the tradeoff between storage and recomputation of interpolation coefficients must be studied. To address the first problem above, we can explicitly store a list of contributions at each mesh point. This also avoids the need for an overflow list in the Gather algorithm.

The above ideas can be implemented using a sparse matrix. Each row of the sparse matrix is stored contiguously, and the elements in a row represent the interpolation weights for a given mesh point. Applying the spreading operation consists of performing a sparse matrix-vector product (SpMV), where the vector is the quantities at the particle locations to be spread.

In this section, we describe three mesh based approaches, which we call *single mesh*, *group mesh*, and *hybrid mesh*. The single mesh approach uses a data structure identical to the compressed sparse row (CSR) data structure used in sparse matrix computations. We describe a new fast algorithm for GPUs for computing the spreading operator as a sparse matrix in CSR format. To further improve performance, we describe a new group mesh approach. Finally, for completeness, we describe a hybrid mesh approach, which is analogous to using the hybrid sparse matrix format in cuSPARSE for representing the spreading operator.

B. Construction of Single Mesh and Group Mesh Data Structures

The spreading approach that we describe here involves precomputing mesh based data structures. The efficiency of this preprocessing step must not be ignored, because it itself is repeated for every particle configuration, and the number of spreading steps over which it is amortized may not necessarily be very large. In this section, we focus on the efficient construction of the single and group mesh based data structures.

We first consider a simple data structure to set the stage for a more complex approach. As mentioned, the single mesh data structure corresponds to a sparse matrix in CSR format. The data structure is mesh based because rows, which are stored contiguously, correspond to mesh points, and columns correspond to particles. Constructing such a sparse matrix on GPUs is straightforward, and is illustrated in Figure 2.

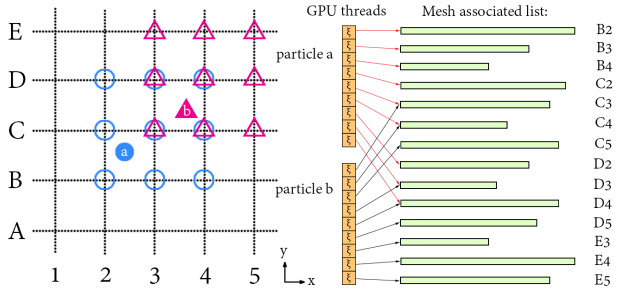


Figure 2: Constructing the spreading operator in the single mesh data structure with $p = 3$. The solid blue dot and solid red triangle represent two particles. For each particle, 9 GPU threads are used to compute its spreading contributions as well as to update the rows of the data structure.

Specifically, constructing the spreading operator in CSR format consists of three steps: *Count*, *Scan* and *Collect*. First, the *Count* step traverses all the particles to count the number of spreading contributions to each mesh point. Next, the *Scan* step performs a prefix sum on the counts to obtain the starting positions of each mesh point in the CSR matrix. Finally, the *Collect* step computes the spreading contributions from each particle and inserts them into the rows of the data structure. Since multiple threads may attempt to update the same row of the matrix simultaneously, atomic operations are used. (We rely, in some sense, on the fact that the atomic operations are not too expensive, as will be shown later.) In all three steps, we assign p^3 threads rather one thread to each particle to maximize use of parallel resources.

An inefficiency with the above procedure, however, is that in the Collect step, the threads assigned to each particle are in one warp, but will update different rows of the CSR matrix. The Collect step will have low performance on GPUs because of non-coalesced memory access. To promote coalesced memory access, we group sets of grid points in the single mesh data structure. Within each group, a stored interpolation weight must identify which mesh point it is associated with. This gives the group mesh data structure. It is similar it spirit to various multirow sparse matrix storage formats for GPUs [13], [14], [15]. Figure 3 illustrates the group mesh data structure.

The main advantage of the group mesh data structure, compared to the single mesh data structure, is that it has better memory access locality and tends to have coalesced memory accesses. It is expected that constructing and using

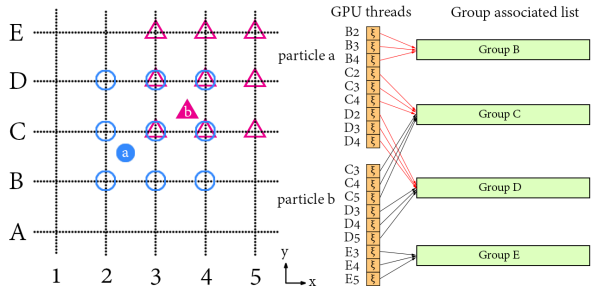


Figure 3: Constructing the spreading operator in the group mesh data structure with $p = 3$. The solid blue dot and red triangle represent two particles. For each particle, 9 GPU threads are used to compute its spreading contributions and update the rows of the data structure.

the group mesh data structure is more efficient than for the single mesh data structure. An issue with the group mesh data structure is that the spreading step once again needs to use atomic operations. Experimentally, we find that the improved memory access patterns can make the group mesh approach better than the single mesh approach, despite its use of atomics.

C. Spreading Optimization

With the intermediate results of spreading stored in the sparse matrix format, the spreading step can be efficiently computed using sparse matrix-vector multiplications (SpMV). While the optimization techniques for SpMV have been intensively studied on GPUs [16], [17], we apply special techniques to accelerate the spreading step on GPUs that utilizes the hardware features introduced in the Kepler microarchitecture.

We define a *compute unit* (CU) as a group of threads used to collect the spreading contributions at a mesh point. By using more than one thread for a mesh point, thread divergence is reduced and coalesced memory access is promoted. This is analogous to why more than one thread is used to multiply a row in GPU implementations of SpMV [16].

Using multiple threads for a mesh point or row, however, requires the use of atomic operations because multiple threads within a CU will update the same mesh point simultaneously. To avoid the use of atomic operations, we let a specific thread in the CU collect the sum using an intra-CU reduction operation. On the Kepler microarchitecture, the reduction operation can be efficiently implemented by using a hardware feature called *warp shuffle*. Warp shuffle is a new set of instructions that allows threads of a warp to read each other's registers, providing a new way to communicate values between parallel threads besides shared memory. Compared to shared memory communication, warp shuffle is much more efficient. The throughput of warp shuffle instructions is 32 operations per clock cycle per multiprocessor for Kepler GPUs [18].

Figure 4 illustrates the intra-CU reduction implemented using warp shuffle instructions. The figure shows a warp of 32 threads, organized such that 8 threads are assigned to a row (or mesh point), i.e., CU=8. To perform a reduction operation within a row using 8 threads, 3 iterations of warp shuffle operations are needed, following the binomial tree algorithm.

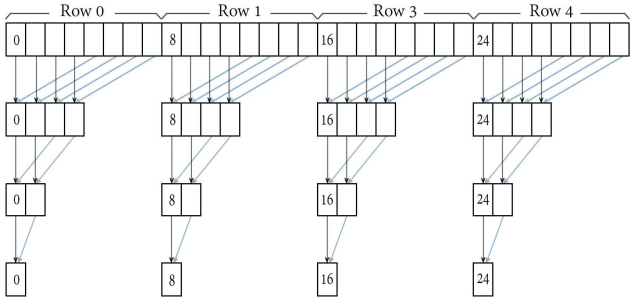


Figure 4: Illustration of intra-CU reduction using warp shuffle operations. The reduction across 4 sets of 8 threads is performed in 3 iterations following the binomial tree algorithm (see text).

The performance of the spreading step using the single mesh method is dependent on the choice of the size of CU. Here, we describe a heuristic of choosing the size of CU, which can be expressed as

$$CU_{optimal} = \begin{cases} 1 & \text{if } Np^3/K^3 < 1 \\ 2^t & \text{if } 2^t \le Np^3/K^3 < 2^{t+1} \text{ and } 0 \le t < 4 \\ 16 & \text{if } Np^3/K^3 \ge 16 \end{cases}$$

where Np^3/K^3 represents the average number of spreading contributions per mesh point (*ASM*). In sparse matrix terms, *ASM* is the average number of nonzeros per row.

To explain the heuristic, we use CU sizes that are powers of 2 for efficiency of the warp shuffle reduction. The CU size should also be at least larger than *ASM*, otherwise some threads will be idle. When *ASM* is larger than 16, the heuristic selects the optimal CU size as 16. Increasing the CU size from 16 to 32 does not significantly improve the load balance as 16 appears to be fine enough parallelism. Also, increasing the CU size from 16 to 32 increases the number of warp shuffle iterations from 4 to 5. We have run some experiments to verify the heuristic. Figure 5 shows the results.

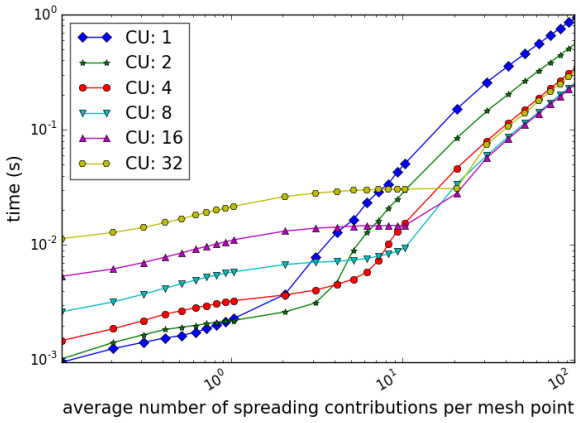


Figure 5: The performance of the spreading step using the single mesh method with various sizes of CU. $p = 6$ is used in the test.

The performance of the group mesh algorithm depends on the selection of the group size (*gsize*), i. e., the number

of mesh points that are grouped together. On the one hand, there is more write contention when *gsize* is small. On the other hand, the total number of warps that can be used for spreading is smaller when *gsize* is larger. We experimentally determined that an optimal value of *gsize* is 64 for any average number of spreading contributions per mesh point. The number may vary on different GPUs. On the GPU hardware used in our test, using $gsize = 64$ appears to be an appropriate compromise between parallelism and memory access conflicts. Figure 6 shows this result.

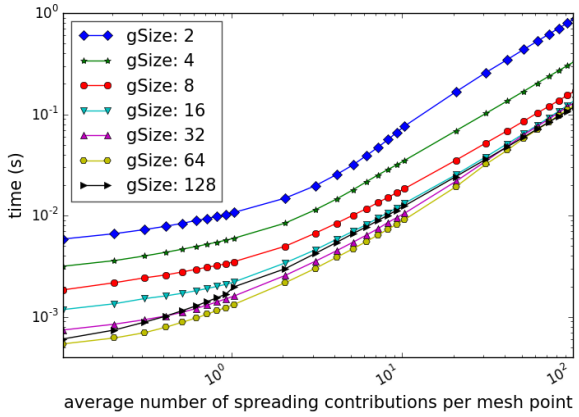


Figure 6: Performance of the spreading step using the group mesh method with various *gsize*. The tests used $K = 128$ and $p = 6$.

D. Hybrid Mesh Approach

In general, the spreading operator can be stored in any sparse matrix format. For completeness, we consider the cuSPARSE ELL-COO hybrid format, which is perhaps considered the most efficient matrix format for SpMV on GPUs [16].

A two-step preprocessing phase is needed to construct the spreading operator in this format. First, the spreading operator is constructed in compressed sparse column (CSC) format. This is natural because the interpolation coefficients can be efficiently computed on a per particle basis, and the coefficients for each particle corresponds to a column in the spreading operator matrix. The CSC format is then transformed to ELL-COO format using the cuSPARSE `csc2hyb` function. In this function, the maximum number of spreading contributions that can be stored in the ELL format, which is denoted by N_{hybrid}, was chosen automatically.

The spreading operation for this format is simply the hybrid SpMV operation provided by cuSPARSE. This operation does not need atomics, and due to extensive work by NVIDIA in optimizing SpMV in cuSPARSE, we expect this spreading operation to be very efficient.

IV. Experimental Results

A. Test Platforms

Table I lists the GPUs used in our tests. Most experiments were conducted on a NVIDIA K40c with the Kepler GK110 microarchitecture. For evaluating the effect of using atomic operations, we also used a GTX 480, based on the earlier Fermi microarchitecture. CUDA version 6.5 toolkit was used in all the experiments.

Table I: NVIDIA test platforms

GPU	K40c	GTX 480
Architecture	Kepler	Fermi
Compute capability	3.5	2.0
CUDA cores	2,880	448
GPU clock rate	876MHz	1,401MHz
Memory clock rate	3,004 MHz	1,848MHz
L1 cache size	16KB	16KB
L2 cache size	1,536KB	768KB
Global memory size	12GB	1.5GB
No. of registers per block	64K	32K
Shared memory per block	48KB	48KB

B. Test Problems

The performance of particle mesh spreading will be problem dependent, and therefore no single test problem is sufficient, and we expect that different algorithms will be best for different particle configurations. We propose a class of test problems for particle-mesh problems. The key parameter is the average number of spreading contributions for each mesh point, abbreviated ASM. To construct problems with different values of ASM, we use different numbers of particles ranging from 1000 to 10,000,000, and different mesh dimensions $K \times K \times K$, with K chosen as 32, 64, 128 and 256. We also use values of the interpolation parameter p of 4 and 6. In this paper, we generate random positions for the particles using a uniform distribution over the mesh. Nonuniform distributions will create load balance issues which we do not address in this initial study.

C. Atomic Operation Overhead for Different Platforms

In previous work [10], [11], particle based approaches were considered less efficient than mesh based approaches method due to the use of atomic operations. While this may be true on earlier GPU microarchitectures, the Kepler GK110 microarchitecture has significantly improved performance of atomic operations [19]. We are thus interested in the improvement of the particle based approaches compared to mesh based approaches on the contemporary GPU hardware.

In this section, we test the particle based algorithm, and show the overhead of atomic operations by comparing the execution time of the algorithm itself and a modified version that replaces atomic operations with normal global memory store operations. We use both the Kepler platform and the older Fermi platform. Although the modified version does not generate correct results, it is useful for determining the performance impact of atomic operations.

As shown in Figure 7, on the Fermi microarchitecture, atomic operations add a very large overhead to the particle based algorithm. On the Kepler microarchitecture, the overhead is much smaller, and is only a small fraction of the overall execution time.

Figure 8 compares the performance of the particle based algorithm and the Gather algorithm on the Fermi and Kepler microarchitectures. On Fermi, the particle based algorithm requires more time than the Gather algorithm, but on Kepler, the Gather algorithm requires more time. This change is directly related to the improvement in performance of the atomic operations on Kepler.

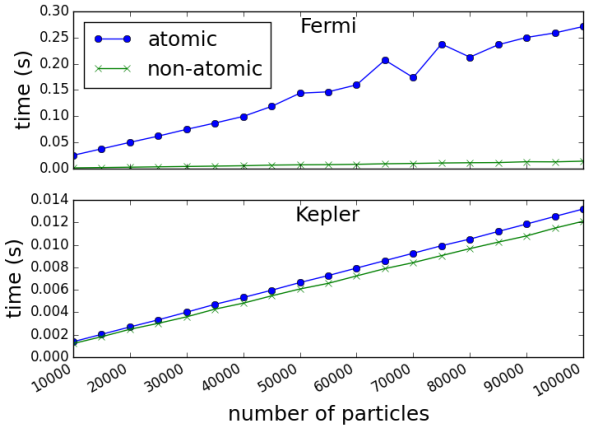


Figure 7: Performance impact of atomic operations in the particle based algorithm. Blue circles show performance of the particle algorithm. Green crosses show the performance if atomic operations are replaced by global memory writes. Top figure shows result using the Fermi microarchitecture; bottom figure shows result using the Kepler microarchitecture. The test problems used $K = 64$ and $p = 6$.

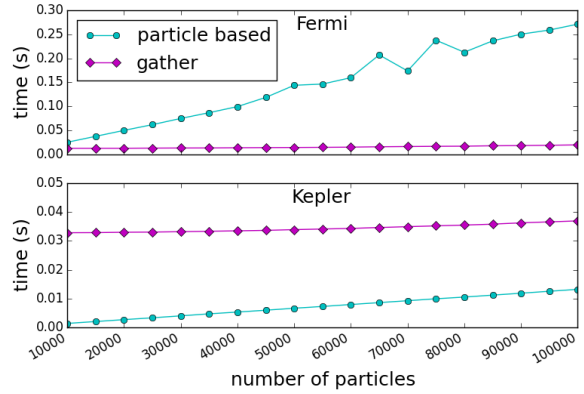


Figure 8: Performance comparison between the particle based algorithm and the Gather algorithm on Fermi and Kepler microarchitectures. The test problems used $K = 64$ and $p = 6$.

D. Comparison of Spreading Costs

In this section we compare the cost of the spreading operations. For the mesh based algorithms, we do not include the time for constructing the mesh based data structures. These will be considered separately later in this paper.

Figure 9 shows the timing comparison between particle based algorithm and mesh based algorithms. For the mesh based algorithms, the preprocessing time is not included here, but will be analyzed in a later section. We make the following observations from the figure.

1. For small numbers of particles, the particle based algorithm is best. Threads are less likely to experience contention on atomic writes when there are fewer particles, which gives this algorithm an advantage in this regime. It can be observed in the figures that the slope of the timing curve for this algorithm (red triangles) increases very slightly as the number of particles is increased. This effect may be due to greater contention due to more particles.

2. Except for small numbers of particles, the hybrid mesh algorithm, using the cuSPARSE SpMV operation for the hybrid format, is generally best.

3. The cost of the Gather algorithm is composed of gathering contributions at each mesh point, and processing the overflow particles (these are steps 2 and 3 of the Gather algorithm, as explained in Section II.B.). When the number of particles is much less than K^3, there are few if any overflow particles, and thus the cost of the algorithm is independent of the number of particles. For K^3 particles or more, the overflow phase adds to the execution time. The cost of this phase increases linearly with the number of overflow particles. Thus there is an expected knee in the timing for the Gather algorithm, as observed.

E. Comparison of Preprocessing Costs

In this section, we compare the costs of constructing the mesh based data structures. From the sparse matrix point of view, transferring from a particle based data structure to a mesh based data structure is a matrix transpose operation. However, note that in particle-mesh applications, there is no sparse matrix corresponding to the particle based data structure.

Figure 10 shows the overhead of constructing the mesh based data structures. The group mesh data structure can be constructed the fastest, due to better memory access patterns. The hybrid mesh data structure (the ELL-COO format) is generally slowest to construct. Unfortunately, hybrid mesh spreading was the fastest among the mesh based approaches.

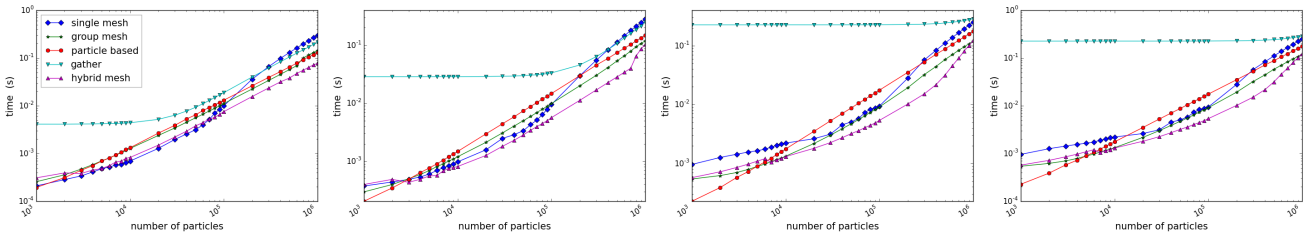
Figures 11 and 12 show the data structure construction cost for the single mesh method and group mesh method, respectively, for different grid size parameters, K. In both cases, when the number of particles is small, the $O(K^3)$ term of the cost dominates; when the number of particles is large, the Np^3 term of the cost dominates, where N is the number of particles.

F. Spreading Multiple Times

In applications where spreading is performed multiple times for the same particle configuration, the cost of constructing the mesh based data structure can be amortized. Here we report the total cost (preprocessing plus spreading) for 1 spreading step and 20 spreading steps with the same particle configuration.

Figure 13 shows the overall performance of different algorithms when the spreading is performed only once. As seen in the figure, the particle based algorithm has the best performance when the spreading is performed only once or for very small number of times.

Figure 14 shows the overall performance when the spreading is performed 20 times. Several conclusions can be drawn from these figures. When the number of particles is relatively small, the particle based algorithm still has the best performance for the particle-mesh configurations. When the number of particles is relatively large, the group mesh method is best. Although the group mesh method is slower than the hybrid mesh algorithm, the group mesh method has lower data structure construction cost. It is expected that the hybrid mesh algorithm is best if its data structure



(a) K = 32 (b) K = 64 (c) K = 128 (d) K = 256

Figure 9: Performance comparison between the particle based algorithm and the mesh based algorithm. The test problems used $p = 6$.

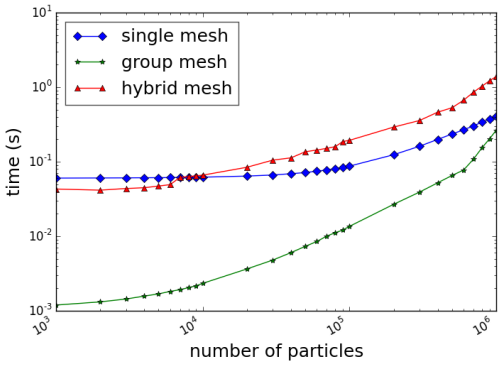


Figure 10: Performance comparison of constructing the mesh based data structure for different algorithms ($K = 128$ and $p = 6$ was used for the test problems).

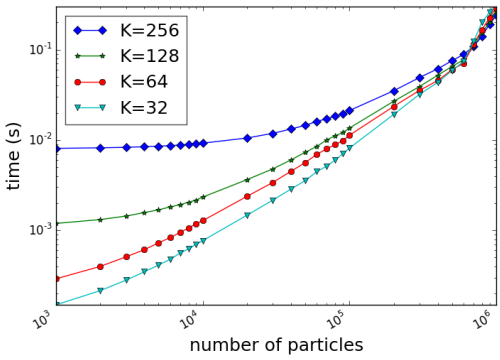


Figure 12: Performance of matrix construction of the group mesh method under various mesh dimensions. The spreading order $p = 6$.

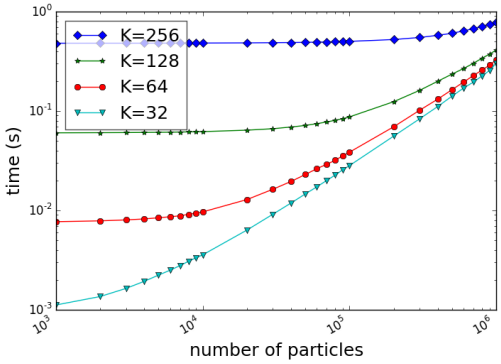


Figure 11: Performance of matrix construction of the single mesh method under various mesh dimensions. The spreading order $p = 6$.

construction time can be amortized over a very large number of spreading steps.

When the dimension of the mesh is very small, e. g., 32, and the number of particles is between 10,000 and 100,000, the single mesh method has the best performance.

G. Comparison of Reduction Performance Using Warp Shuffle and Shared Memory

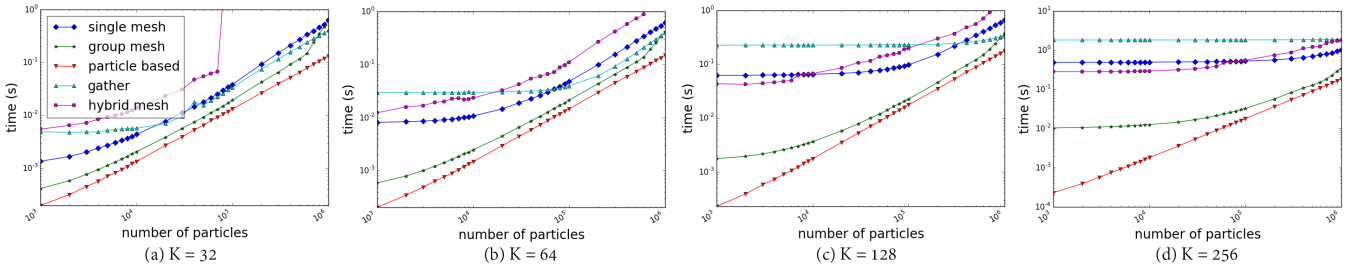
In Tesla and Fermi microarchitecture based GPUs, sharing data between parallel threads can only be done in shared memory. In the newer Kepler GPU microarchitecture, NVIDIA introduced a way to directly share data between threads that are part of the same warp, using so called *warp shuffle* instructions. By allowing threads of a warp read each other's registers, the warp shuffle instruction can be used to achieve throughput that is usually much higher than by using communication through shared memory.

In the single mesh method, we perform an intra-warp reduction when applying spreading. The reduction is performed by using warp shuffle optimizations. In this section, we show the performance gain of this optimization, by comparing the performance of single mesh spreading using warp shuffle and shared memory reduction.

Figure 15 compares the execution time of the single mesh method using two different reduction methods. As can be seen, the single mesh method using warp shuffle reductions is never worse than the shared memory counterpart. When the average number of spreading contributions per mesh point (ASM) is larger than 20, these two versions achieve approximately the same performance. One explanation for this phenomenon is that, when ASM is sufficiently large, the shuffle or shared memory load is hidden by other costs such as warp divergence or poor cache usage (Figure 5 tells us one warp only uses half the cache line when ASM is larger than 20).

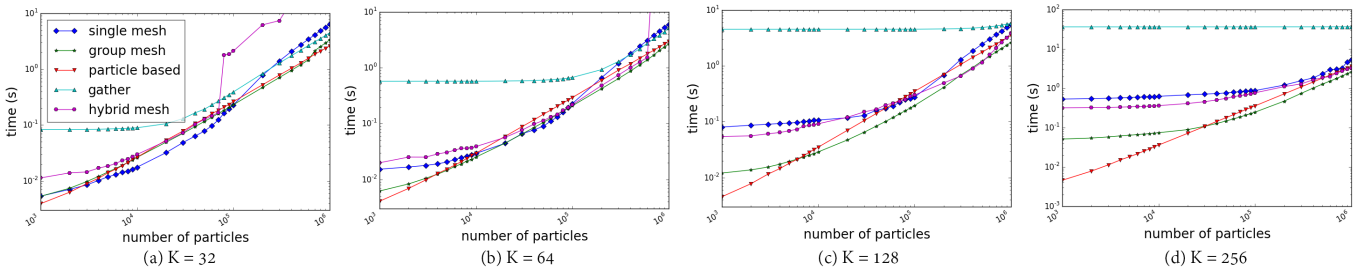
V. Conclusion

In this paper, we discussed the advantages and disadvantages of various algorithms for particle-mesh spreading. We categorized algorithms as being particle based or mesh based. Those that are particle based generally require atomic operations. Those that are mesh based require the construction of mesh based data structures. We introduced single



(a) K = 32 (b) K = 64 (c) K = 128 (d) K = 256

Figure 13: Comparison of the overall performance for 1 spreading operation.



(a) K = 32 (b) K = 64 (c) K = 128 (d) K = 256

Figure 14: Comparison of the overall performance for 20 spreading operations with the same particle configuration. The mesh based data structures are only computed once and are reused.

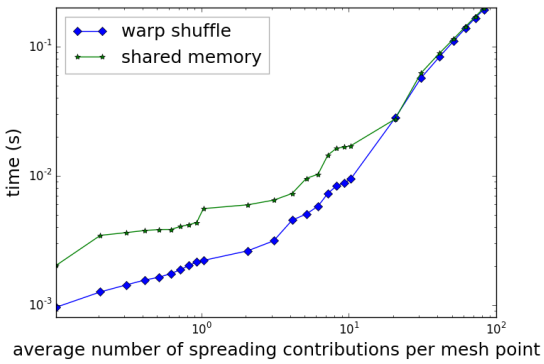


Figure 15: Performance comparison of the single mesh method using warp shuffle and shared memory reduction ($K = 128$ and $p = 6$).

mesh and group mesh data structures that are related to sparse matrix data structures.

Timing tests were used to determine which algorithms are best for a test set parameterized by the average number of particles per mesh point. When only a single spreading operation is performed for a given particle configuration, the simple particle based method is fastest. This is due to very fast atomic operations on current GPU architectures. When multiple spreading operations are performed and the preprocessing costs can be amortized, the single mesh and group mesh algorithms are marginally better, for moderate numbers of spreading operations (around 20). For very large numbers of spreading operations, the hybrid mesh approach using the hybrid sparse matrix data structure in cuSPARSE is fastest. This is due to very fast spreading but relatively high data structure construction times.

This paper also introduced the use of warp shuffle operations for performing reductions for summing contributions to a mesh point with multiple threads. This idea can be extended to optimize the SpMV operation on GPUs for row-based data structures.

Acknowledgements

This work was supported by the U.S. National Science Foundation under grant ACI-1306573, the National Natural Science Foundation of China (No. 61272087, 61363019 and 61073008), the Beijing Natural Science Foundation (No. 4082016 and 4122039), and the Sci-Tech Interdisciplinary Innovation and Cooperation Team Program of the Chinese Academy of Sciences.

References

[1] A. Dutt and V. Rokhlin, "Fast Fourier transforms for nonequispaced data," *SIAM Journal on Scientific Computing*, vol. 14, no. 6, pp. 1368–1393, 1993.

[2] A. F. Ware, "Fast approximate Fourier transforms for irregularly spaced data," *SIAM Review*, vol. 40, no. 4, pp. 838–856, 1998.

[3] H. Schomberg and J. Timmer, "The gridding method for image reconstruction by Fourier transformation," *IEEE Transactions on Medical Imaging*, vol. 14, no. 3, pp. 596–607, 1995.

[4] D. B. Twieg, "The k-trajectory formulation of the NMR imaging process with applications in analysis and synthesis of imaging methods," *Medical Physics*, vol. 10, no. 5, pp. 610–621, 1983.

[5] M. K. M. Soumekh, "Computer-assisted diffraction tomography," *Image Recovery: Theory and Application*, p. 369, 1987.

[6] T. Darden, D. York, and L. Pedersen, "Particle mesh Ewald – an $N\log(N)$ method for Ewald sums in large systems," *Journal of Chemical Physics*, vol. 98, no. 12, pp. 10 089–10 092, 1993.

[7] A. W. Götz, M. J. Williamson, D. Xu, D. Poole, S. Le Grand, and R. C. Walker, "Routine microsecond molecular dynamics simulations with AMBER on GPU. 1. Generalized Born," *Journal of Chemical Theory and Computation*, vol. 8, no. 5, pp. 1542–1555, 2012.

[8] D. Saintillan, E. Darve, and E. S. G. Shaqfeh, "A smooth particle-mesh Ewald algorithm for Stokes suspension simulations: The sedimentation of fibers," *Physics of Fluids*, vol. 17, no. 3, p. 033301, 2005.

[9] X. Liu and E. Chow, "Large-scale hydrodynamic Brownian simulations on multicore and manycore architectures," in *28th IEEE International Parallel & Distributed Processing Symposium*, 2014.

[10] M. Harvey and G. De Fabritiis, "An implementation of the smooth particle mesh Ewald method on GPU hardware," *Journal of Chemical Theory and Computation*, vol. 5, no. 9, pp. 2371–2377, 2009.

[11] W. M. Brown, A. Kohlmeyer, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers–particle–particle particle-mesh," *Computer Physics Communications*, vol. 183, no. 3, pp. 449–459, 2012.

[12] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, and L. G. Pedersen, "A smooth particle mesh Ewald method," *The Journal of Chemical Physics*, vol. 103, no. 19, pp. 8577–8593, 1995.

[13] T. Oberhuber, A. Suzuki, and J. Vacata, "New row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA," *arXiv preprint arXiv:1012.2270*, 2010.

[14] Z. Koza, M. Matyka, S. Szkoda, and L. Miroslaw, "Compressed multirow storage format for sparse matrices on graphics processing units," *SIAM Journal on Scientific Computing*, vol. 36, no. 2, pp. C219–C239, 2014.

[15] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[16] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 18:1–18:11.

[17] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPU," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 115–126.

[18] NVIDIA, "CUDA C Programming Guide," 2014.

[19] ——, "NVIDIA Kepler GK110 Architecture Whitepaper," 2012.