GPU-Accelerated Charge Mapping

Ahmed Sanaullah Saiful A. Mojumder Kathleen M. Lewis Martin C. Herbordt
Department of Electrical and Computer Engineering
Boston University, Boston, MA

***Abstract*—Charge Mapping is critical to electrostatics computations such as performed in Molecular Dynamics simulations. It reduces the complexity of evaluating long-range Coulombic forces by diffusing discrete particle charges onto a regular grid. Acceleration of this stage using GPUs is non-trivial, with the compute and memory intensive nature of the algorithm limiting performance benefits in naive implementations. In this paper, we explore methods for performing efficient charge mapping on GPUs. By utilizing available resources effectively and reducing compute and memory transactions, high throughput can be achieved. Our best case implementation shows $> 14\times$ speed-up over existing GPU codes and $> 25\times$ speed-up over production CPU codes such as NAMD.**

I. Introduction

Molecular Dynamics Simulations account for a large fraction of all High Performance Compute cycles. Their efficient computation has therefore received much attention with various types of computer systems being applied [1]–[5]. Reducing the complexity of the brute-force (all-to-all pairwise) method from $O(N^2)$ to at most $O(N \log(N))$ requires several steps. First the Coulombic forces are partitioned into range-limited and long-range components. For the former, bounding the range through use of structures such as cell and neighbor lists [6], reduces the pairwise complexity from $O(N^2)$ to $O(N)$. For the latter, transform-based methods such as Particle Mesh Ewald (PME) [7] reduce the complexity to $O(N \log(N))$.

Figure 1 illustrates the steps involved in computing long-range electrostatics. First, particles are diffused to a discrete grid using a charge mapping function. Next, the Poisson equation is solved using 3D FFTs or Multigrid to obtain a potential grid. These potential values are then back-interpolated to obtain the forces. The two major computations are the 3D FFT and Charge/Force Mapping. The FFT is well understood across multiple plaforms, with libraries/implementations available for doing transforms on CPUs (e.g., FFTW [8]), GPUs (e.g., cuFFT [9]) and FPGAs [10]–[13].

However, the grid-particle and particle-grid mappings are also crucial as they are often substantially more compute-intensive than the FFTs. Each is typically done using tricubic interpolation and requires on the order of 500 floating point operations ($500N$) per particle; for reference, the FFT requires around $100N$ for most MD simulations. Moreover, an N particle system with cubic interpolation can generate $65N$ memory transactions to read charge data and store computed grid values.

This work was supported in part by the National Science Foundation through Awards #CNS-1405695 and #CCF-1618303/7960; and by a grant from Microsoft. Email: (sanaulah|herbordt|kmlewis)@bu.edu

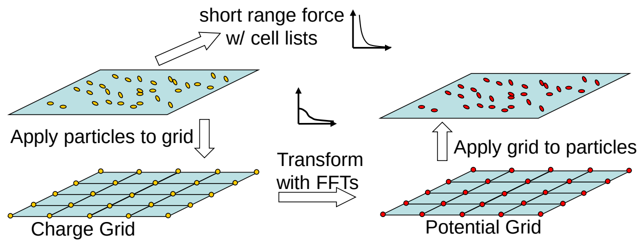

Fig. 1. Overview of the long-range electrostatics computation (from [19]).

Several methods of charge mapping have been proposed in the literature, with implementations taking advantage of the underlying compute platform hardware. CPU based designs such as NAMD [1] and Protomol [14] have access to fast memory hierarchies to address memory bottlenecks, but are unable to exploit all of the opportunities given by the large degree of data parallelism.

Harvey et al. [15] and Ganesan et al. [16] present implementations of charge mapping on GPUs. The former assigns particles to grid locations, with at most one charge per grid point, and maps them without the use of atomic operations. However, the remaining charges are unsorted and require atomic integer compare-and-set loop constructs to be mapped, which is slower. For a grid size of m per dimension, only m^3 particles can be mapped efficiently in the best case. In a 90K particle system mapped to a 32^3 grid, this is less than 50% of the total particles. On the other hand, Ganesan et al. maintain neighbor lists for each grid point to track all particles that will affect it. Spline values are computed using texture memory lookups. Apart from the overhead of creating and updating neighbor lists, the memory intensive nature of this design will likely limit potential performance improvements.

FPGA implementations, such as that by Gu et al. [17], take advantage of memory interleaving [18] using the fast on-chip block RAMs. They are able to couple flexible memory organizations with deep specialized pipelines to obtain large degrees of data and task parallelism.

In the present work, we design optimized charge mapping algorithms for GPUs. The contributions are as follows.

- We identify and address computational bottlenecks in CPU implementations. This enables GPU kernels to perform serial calculations in an efficient manner.
- We show that efficient memory structure is critical to charge mapping performance on GPUs. Our memory-aware implementations perform $14\times$ better than existing GPU codes and $25\times$ better than common CPU codes such as NAMD.

- We implement two different GPU algorithms: Particle Centric and Grid Centric. We start with naive baseline implementations and discuss the optimizations required to improve performance. Particle Centric methods have significantly fewer calculations and memory operations and benefit from low resource contention in atomic operations to give better performance than Grid Centric methods.
- We compare our GPU algorithms on different GPU architectures and use various benchmarks to determine the scalability of Particle Centric and Grid Centric methods.

II. Interpolation

Particle-grid interactions for charge mapping are typically computed using spline interpolation polynomials. Particles spread their charge to neighboring grid points based on their separation using a basis function (ϕ) as shown in Eq 1. Indices p and g denote particle and gridpoint respectively while ρ_g is the discretized charge distribution resulting from the mapping operation.

$$\rho_g = \sum_p Q_p \phi(|x_g - x_p|)\phi(|y_g - y_p|)\phi(|z_g - z_p|) \quad (1)$$

Selection of a basis function is important since its order governs the accuracy of results, the number of FLOPs per polynomial evaluated, and the number of interactions computed per particle. Moreover, the basis function is constrained to be C^1 continuous since its gradient is required for force computation at a later stage in electrostatic computations. Skeel et al. [20] proposed one such third order basis function shown in Eq. 2. Here ξ is the distance between the particle and any grid point.

$$\phi(\xi) = \begin{cases} (1-|\xi|)(1+|\xi|-\frac{3}{2}\xi^2) & |\xi| \le 1 \\ -\frac{1}{2}(|\xi|-1)(2-|\xi|)^2 & 1 \le |\xi| \le 2 \\ 0 & 2 \le |\xi| \end{cases} \quad (2)$$

Taking advantage of unit spacing between grid points, Gu et al. [17] modify the above basis function by first expressing all distances from grid points with non-zero contributions as a function of the distance of a particle from its nearest grid point (oi). An example of this is illustrated in Figure 2. By substituting ξ in Eq. 2 with $oi+1$, oi, $1-oi$ and $2-oi$, four spline polynomials are obtained. All polynomials take oi as an input and the polynomial selected depends on the particle-grid point distance.

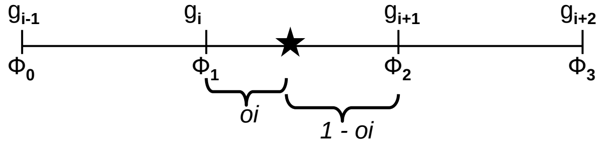

Fig. 2. Extract grid index and offset.

$$\begin{cases} \phi_0(oi) = -\frac{1}{2}oi^3 + oi^2 - \frac{1}{2}oi \\ \phi_1(oi) = \frac{3}{2}oi^3 - \frac{5}{2}oi^2 + 1 \\ \phi_2(oi) = -\frac{3}{2}oi^3 + 2oi^2 + \frac{1}{2}oi \\ \phi_3(oi) = \frac{1}{2}(oi^3 - oi^2) \end{cases} \quad (3)$$

III. Serial Charge Mapping

A. Computation

Designing an efficient GPU implementation requires optimizing the serial code that is executed by each thread. Consequently, we begin by identifying the potential performance bottlenecks in naive CPU implementations and how these can be addressed. Algorithm 1 shows the baseline CPU code. An outer loop iterates through all particles in the data-set. For each particle, a set of nested loops iterates over all 64 grid points for which interactions must be computed. Due to periodic boundaries, a wrapping function is required to determine valid grid coordinates for particles at system edges. The innermost loop evaluates the basis function for each dimension, multiplies with particle charge, and accumulates the result at the appropriate grid point. Here, PX and nGP refer to the particle coordinate and nearest grid point vectors respectively.

Algorithm 1 Baseline CPU Algorithm

```
1: for $Particles = 1 : N$ do
2:     $nearestGridPoint = floor(ParticleCoordinates)$
3:     for $z = nGP.z - 1$ to $nGP.z + 2$ do
4:         for $y = nGP.y - 1$ to $nGP.y + 2$ do
5:             for $x = nGP.x - 1$ to $nGP.x + 2$ do
6:                 wrap(z)
7:                 wrap(y)
8:                 wrap(x)
9:                 Basis Function(z,PX.z - nGP.z)
10:                Basis Function(y,PX.y - nGP.y)
11:                Basis Function(x,PX.x - nGP.x)
12:                Multiply and Accumulate
13:            end
14:        end
15:    end
16: end
```

Despite its low complexity, there are several drawbacks in this implementation. Addressing the wrap-around and basis functions is critical to compute performance since each function is called 192 times per particle. These functions perform multiple cases of fixed function form evaluation. For example, basis function calls compute one of four possible polynomials depending on the distance of a grid point from the particle. The large number of conditional branches results in performance degradation due to branch miss-prediction penalties. By introducing conditional assignments for wrap-arounds and by precomputing all dimension-independent basis function values and affected grid-coordinates, the conditional branches can be remove completely. This reduces the overall number of function calls by 93.75%.

Algorithm 2 shows the improved serial charge mapping code. Here, the arrays gx, gy and gz contain all possible dimensional coordinates of the nearest 64 points while the arrays $phix$, $phiy$ and $phiz$ contain all possible basis function polynomials.

Algorithm 2 Optimized CPU Algorithm

```
1:  for Particles = 1 : N do
2:      Compute nearest four grid points in each dimension
3:      gz[4] = boundary ? wrap around : gz[4]
4:      gy[4] = boundary ? wrap around : gy[4]
5:      gx[4] = boundary ? wrap around : gx[4]
6:      phiz[4] = Basis Function(PX.z - gz[2])
7:      phiy[4] = Basis Function(PX.y - gy[2])
8:      phix[4] = Basis Function(PX.x - gx[2])
9:      for $z = 1$ to $4$ do
10:         for $y = 1$ to $4$ do
11:             for $x = 1$ to $4$ do
12:                 ChargeGrid(gz[z],gy[y],gx[x]) =
13:                 phiz[z]*phiy[y]*phix[x]*Q
14:             end
15:         end
16:     end
17: end
```

Apart from optimizing function evaluation, parallelism in the outermost loop can also be exploited through vectorization. Since the same functions are evaluated for every particle, multiple particles can be processed simultaneously which can improve performance despite persisting memory bottlenecks.

B. Binning

To improve memory performance, data-sets can be pre-sorted into bins. All particles within a bin have a common nearest grid point and contribute charges to the same set of grid points. Particles in consecutive bins overlap for 75% of the grid locations accessed. Consequently, in-order traversal of such bins results in better locality and high cache hit rates.

There are two primary constraints on functions that perform binning. First, they should bin particles in a single pass with $O(N)$ time, which is bound by the charge mapping complexity. Second, dynamic arrays cannot be used due to their negative impact on performance. The latter is of importance since the number of particles per bin is not fixed and can vary up to a maximum bounded by the typical system density constraint of 0.1 particle per A^3. Performing two passes of the data set, one to determine bin sizes and allocate memory and second to fill bins, is inefficient as it increases timeframes significantly despite still being $O(N)$. As a result, we declare static bins of fixed sizes in memory to address the worst case. Empty bin spaces are filled with "ghost" particles which have no charge and hence do no contribute to grid-charges. A counter variable in each bin structure is used to track the number of filled bin slots.

IV. GPU Accelerated Charge Mapping

Developing an efficient GPU algorithm for charge mapping requires identifying opportunities for parallelism that exist despite data-dependence between particles. In our design, we consider two methods, Particle Centric (PC) and Grid Centric (GC) (illustrated in Figure 3). PC methods assign workloads from the particle perspective. Each thread represents one or more particles and computes the 64 particle-grid interactions for each. The dominant memory transaction is in the form of write-backs; each thread reads from one particle/bin but writes to multiple locations on the charge grid. On the other hand, GC methods assign workloads from the grid perspective. Each thread calculates and accumulates all possible contributions to a unique grid point. This is a read intensive operation, with multiple particles/bins read in order to generate a single grid-charge value. Harvey et al. and Ganesan et al. both use GC methods. In this section, we will develop algorithms for both methods starting with naive implementations, identifying bottlenecks, and discussing optimizations performed.

A. Particle Centric

1) Baseline: Algorithm 3 shows a baseline implementation for PC mapping. Each thread represents a unique particle, indexed from an unsorted data set in the global memory by using the thread ID. This particle is processed by first computing grid coordinates of interest in each dimension along with all possible basis function values. These values are used to evaluate particle-grid interactions in nested loops and subsequently accumulated in memory using atomic operations. Since floating point atomic operations are available in CUDA libraries, the native 'atomicAdd' function is used. Despite the large degree of parallelism (N threads for a N particle system), irregular memory access patterns on both intra-block and inter-block levels lead to sub-optimal performance. Particles close to each other in real space are potentially not processed by threads within the same block due to the unsorted database. In the worst case, there is no locality exploited for writes within a block resulting in a large number of compulsory and capacity cache misses.

2) Optimizations: Algorithm 4 shows the first level of optimization, where we sort particles into bins and assign each bin to a unique thread. This increases the number of read operations required, both on a per-thread and a global level. The former is due to more particles being processed per thread while the latter occurs because fetching an entire bin results in counters and "ghost" particles being read from global memory. Binning also increases the register usage per thread, reduces the overall number of threads scheduled (number of threads is now equal to the number of grid points), and increases code serialization since a thread loops over all particles in the bin. Despite these drawbacks, however, the overall system performance improves since major limitations outlined previously are addressed. Particles now have similar physical and logical layouts resulting fewer cache misses and greater locality. Another benefit is reduced resource contention for atomic operations since particles writing to the same 64

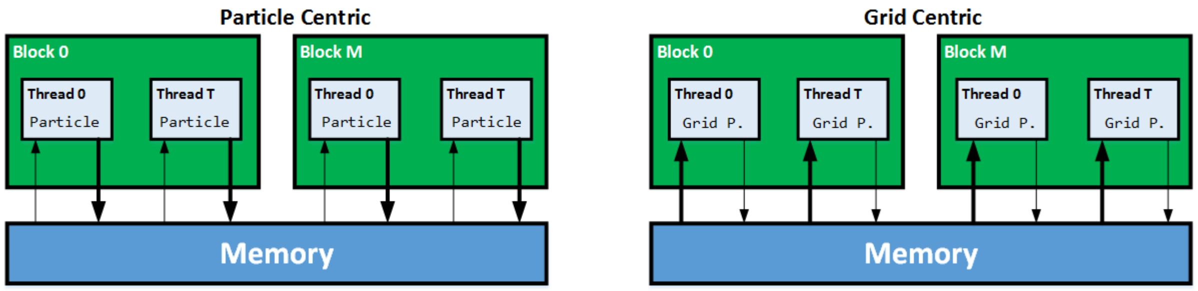

Fig. 3. Particle and Grid Centric methods for implementing charge mapping on GPUS. The width of arrows represents the amount of data transferred in the indicated direction.

Algorithm 3 Baseline Particle Centric Algorithm

```
 1: ID ← getThreadID()
 2: Particle = Mem[ID]
 3: Compute nearest four grid points in each dimension
 4: gz[4] = boundary ? wrap around : gz[4]
 5: gy[4] = boundary ? wrap around : gy[4]
 6: gx[4] = boundary ? wrap around : gx[4]
 7: phiz[4] = Basis Function(PX.z - gz[2])
 8: phiy[4] = Basis Function(PX.y - gy[2])
 9: phix[4] = Basis Function(PX.x - gx[2])
10: for $z = 1$ to 4 do
11:     for $y = 1$ to 4 do
12:         for $x = 1$ to 4 do
13:             AtomicAdd (ChargeGrid(gz[z],gy[y],gx[x]) ,
14:             phiz[z]*phiy[y]*phix[x]*Q )
15:         end
16:     end
17: end
```

Algorithm 4 Optimized Particle Centric Algorithm

```
 1: ID ← getThreadID()
 2: Bin = Mem[ID]
 3: Compute nearest four grid points in each dimension
 4: gz[4] = boundary ? wrap around : gz[4]
 5: gy[4] = boundary ? wrap around : gy[4]
 6: gx[4] = boundary ? wrap around : gx[4]
 7: for $itr = 1$ to $Bin.counter$ do
 8:     Particle = Bin[itr]
 9:     phiz[4] = Basis Function(PX.z - gz[2])
10:     phiy[4] = Basis Function(PX.y - gy[2])
11:     phix[4] = Basis Function(PX.x - gx[2])
12:     for $z = 1$ to 4 do
13:         for $y = 1$ to 4 do
14:             for $x = 1$ to 4 do
15:                 AtomicAdd (ChargeGrid(gz[z],gy[y],gx[x]) ,
16:                 phiz[z]*phiy[y]*phix[x]*Q )
17:             end
18:         end
19:     end
20: end
```

grid locations are processed sequentially. Moreover, thread divergence does not impact performance. Despite each thread processing bins of different sizes, threads in a warp either perform the same computation or remain idle.

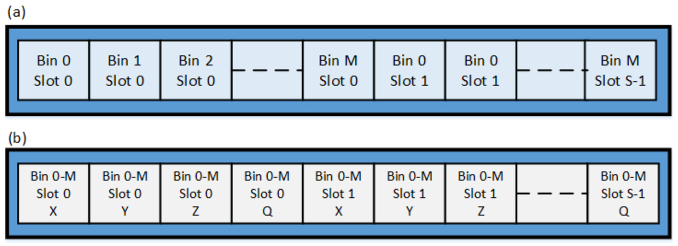

Fig. 4. Modified bin data-structures used for a) reducing invalid reads and b) coalescing. Each of M possible bins has a maximum of S slots.

For the next level of optimization, we reduce the number of reads from global memory. This is important since a thread cannot be scheduled unless the entire bin, including ghost particles, has been read. To achieve this, the bin data structure described previously is replaced by a single array containing particle data and a corresponding counter array (as shown in Figure 4a). A complex addressing scheme is used which maps each bin slot to a different location (equidistant) in the array. Data is organized such that similar slots of all bins are placed in consecutive memory locations. This allows threads to read individual particles in a bin instead of the entire data structure. Based on the value stored in the counter array, threads only traverse data up until the last valid particle. Thus the overhead of reading ghost particles from memory is removed.

Finally, to further improve read memory operations through coalescing, we reorder particle data such that the coordinate and charge values from similar bin slots are placed in consecutive memory locations. This is illustrated in Figure 4b. It is achieved by changing the memory indexing function only; no further changes are required to the data structure.

B. Grid Centric

1) Baseline: Algorithm 5 shows a simple GPU implementation for GC mapping. In contrast to PC mapping, GC methods are aimed at reducing the number of memory write operations and removing data dependencies that necessitate atomic functions. Each thread indexes a grid point based on its thread ID. Using this local grid ID, we can determine the bins which contain particles that will interact with the grid point. These bins are read into memory using nested loops. For each inner loop iteration, a new remote bin is traversed. By comparing local grid and remote bin IDs, the basis function polynomial for particles in the remote bin is determined and evaluated. Local variables can be used for charge accumulation.

Threads now process a larger number of particles but compute the same number of interactions on average. Consequently, threads have true data parallelism, and overhad of coherency misses and atomic additions is completely removed. There are several potential areas of optimization in this implementation which we identify.

Algorithm 5 Baseline Grid Centric Algorithm

```
1: Grid Point ← getThreadID()
2: Compute nearest four grid points in each dimension
3: gz[4] = boundary ? wrap around : gz[4]
4: gy[4] = boundary ? wrap around : gy[4]
5: gx[4] = boundary ? wrap around : gx[4]
6: for $z = 1$ to $4$ do
7:     for $y = 1$ to $4$ do
8:         for $x = 1$ to $4$ do
9:             RemoteBinID = f(gz[z],gy[y],gx[x])
10:            Bin = Mem[RemoteBinID]
11:            for $itr = 1$ to $Bin.counter$ do
12:                Particle = Bin[itr]
13:                phiz = Basis Function(z,PX.z - floor(PX.z))
14:                phiy = Basis Function(y,PX.y - floor(PX.y))
15:                phix = Basis Function(x,PX.x - floor(PX.x))
16:                Local Variable += phiz*phiy*phix*Q
17:            end
18:        end
19:    end
20: end
21: ChargeGrid(getThreadID) = Local Variable
```

2) Optimizations: The first optimization is removing conditional branches in the kernel code. In previous cases, all spline values of a particle were needed and hence they were precomputed and reused. For GC mapping, a read particle only computes one spline value per dimension. Using branches allows this required polynomial to be evaluated without the overhead of unnecessary calculations. However, there are a large number of nested conditional branches corresponding to possible relative orientations of the local grid point and remote bin. The performance benefits of avoiding extra floating point operations may potentially be outweighed by branch miss penalties. Therefore, conditional assignments are used which evaluate all four polynomial before selecting the appropriate one.

A second optimization is performed by using shared memory. Shared memory is a low latency unit accessible by all threads in a block. By cooperatively populating the shared memory, threads can reduce the number of global memory accesses through storage of common variables locally. This is potentially useful for GC methods since threads in a block have significant overlap in remote bins they access due to locality. Moreover, data read into shared memory remains valid since particle values are not modified during mapping.

Finally, we use the reduced-read data structure developed for PC mapping to read only valid particles. Since GC mappings read $64\times$ more bins, we expect that the performance improvement obtained will also be substantially greater. This is illustrated in Algorithm 6.

Algorithm 6 Optimized Grid Centric Algorithm

```
1: Grid Point ← getThreadID()
2: Compute nearest four grid points in each dimension
3: gz[4] = boundary ? wrap around : gz[4]
4: gy[4] = boundary ? wrap around : gy[4]
5: gx[4] = boundary ? wrap around : gx[4]
6: for $z = 1$ to $4$ do
7:     for $y = 1$ to $4$ do
8:         for $x = 1$ to $4$ do
9:             RemoteBinID = f(gz[z],gy[y],gx[x])
10:            for $itr = 1$ to counter[RemoteBinID] do
11:                BinSlot = h(itr,RemoteBinID)
12:                Particle = Mem[BinSlot]
13:                phiz[4]=BasisFunction(PX.z - floor(PX.z))
14:                phiy[4]=BasisFunction(PX.y - floor(PX.y))
15:                phix[4]=BasisFunction(PX.x - floor(PX.x))
16:                LocalVariable+=phiz[z]*phiy[y]*phix[x]*Q
17:            end
18:        end
19:    end
20: end
21: ChargeGrid(getThreadID) = Local Variable
```

V. Results

A. System Specifications

We have tested our algorithms using the ApoA1 (92K particles) and DMPC (68K particles) benchmarks. CPU codes are compiled and run on a single core of an Intel Xeon i5 3.3GHz processor using g++ v5.3.0. GPU codes are run on a TESLA M2070 (Fermi, 440 cores) and a TESLA k40m (Kepler, 2880 cores) GPU using the CUDA NVCC 7.5 compiler. In order to get a complete performance comparison across common technologies, we also compare our results with an FPGA implementation using an Altera Stratix V board.

B. Performance

1) Algorithm: Table 1 and 2 show the performance results for both PC and GC implementations with GPU baseline codes. Results are expressed in the number of particles processed per microsecond (PPUS). PC implementations show greater throughput than GC codes. In both cases, the best performance is achieved by reducing the number of reads from global memory through efficient binning.

Table 3 lists the results of profiling the best case PC and GC implementations for the ApoA1 benchmark.

The PC code performance is observed to be better due to orders of magnitude fewer computations and memory

TABLE I
PARTICLE CENTRIC CHARGE MAPPING PERFORMANCE (PPUS)

Version	A-M2070	D-M2070	A-K40m	D-K40m
Baseline	5.6	5.1	129.7	124.0
Binning	46.2	39.3	144.5	125.0
Reduced Reads	50.9	45.1	201.8	193.0
Coalescing	50.9	45.9	198.4	194.0

TABLE II
GRID CENTRIC CHARGE MAPPING PERFORMANCE (PPUS)

Version	A-M2070	D-M2070	A-K40m	D-K40m
Baseline	2.4	2.0	3.5	3.3
Compute Efficient	2.2	1.8	3.7	3.7
Shared Memory	2.2	1.7	3.0	2.3
Reduced Reads	15.2	14.1	26.6	28.2

operations. The former is due to the difference in data reuse. For each bin read, a thread computes 4 basis function values in each dimension. PC kernels use all these values to compute artial charge contributions for 64 grid points. GC kernels only use one value per dimension to compute the contributions to a single grid point. The latter is due to the large number of bins read by each thread in the GC algorithm. Despite having a higher cache hit rate, the aggregate time spent on these operations is greater than for PC. Moreover, higher multiprocessor and warp efficiency for PC kernels indicates that, even though more than 50% of stalls were caused by the approximately 400K atomic operations, waiting time for contesting threads is low owing to the low probability of a large number of threads attempting to write to the same location simultaneously.

2) GPU Technologies: Using GPUs with different compute capabilities allows us to determine how our proposed mapping algorithms are likely to scale with available resources. First, the higher memory bandwidth available on the Kepler GPU makes the design more tolerant to irregular memory accesses leading to a $> 24\times$ increase in throughput for the baseline PC mapping. On the other hand, latencies of reads are largely independent of the GPU technology. This is indicated by the insignificant performance improvement in the first three GC versions and the comparable performance of PC binning (more reads due to ghost particles) to its baseline. Overall, PC mapping shows better scalability since the best case version shows an $\approx 4\times$ throughput increase as compared to an $\approx 2\times$ increase for GC.

TABLE III
PROFILE STATISTICS FOR OPTIMAL GPU IMPLEMENTATIONS

Metric	Particle Centric	Grid Centric
Global Load	85,372	3,834,694
Global Store	0	1022
Local Load	578,396	1,650,781
Local Store	175,951	1,650,781
L2 Read	3,439,981	17,178,508
L1 Hit Rate (Local)	1.19%	17.01%
L1 Hit Rate (Global)	47.89%	57.19%
L2 Hit Rate	7.98%	93.66%
Single Precision FLOPs	17,333,048	56,983,823
Double Precision FLOPs	3,873,408	72,334,599
Instructions Executed	2,892,563	42,690,098
Multiprocessor Efficiency	98.43%	92.8%
Warp Execution Efficiency	57.06%	46.83%
Data Request Stalls	77.3%	24.56%

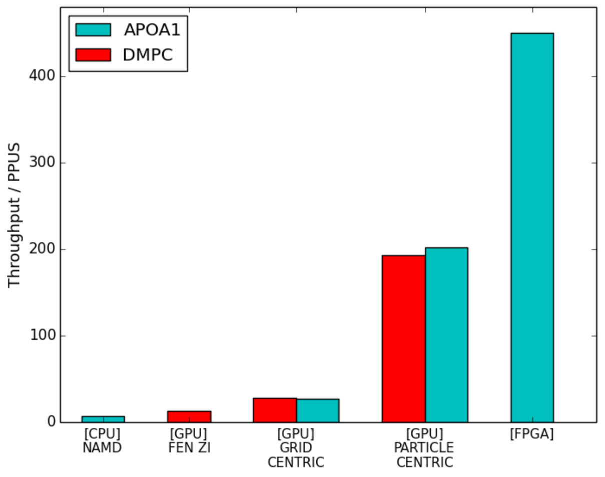

Fig. 5. Throughput comparison with existing implementations.

3) Benchmark: Performance results of the benchmarks show that better throughput is achieved when larger number of particles are processed. Since the difference is only a small fraction, it is potentially due to overhead of kernel execution being amortized over a greater data set.

C. Impact

To evaluate the impact of our work, we compare the best case throughputs of Grid Centric and Particle Centric methods with existing CPU (NAMD), GPU (Fen-ZI on TESLA K40m) and FPGA implementations. From the results shown in Figure 5, we can see that our memory aware methods achieve $> 25\times$ speedup over NAMD (single core) and $> 14\times$ speedup over Fen-ZI. Moreover, our Grid Centric implementation achieves $> 2\times$ speedup over Fen-ZI, which is also a grid centric implementation. The throughput of the FPGA remains higher than that of the GPUs on account of the fast on-chip memories and the ability to compute and store multiple iterations simultaneously.

VI. CONCLUSION

In this paper, we present optimizations for charge mapping on GPUs which address both computation and memory bottlenecks. Calculation timeframes were minimized primarily through data reuse, reducing memory transactions, and forcing regular data access patterns. Particle Centric implementations show the highest performance across all devices and benchmarks despite using atomic operations. Overall, both our Particle Centric and Grid Centric algorithms achieved significantly better performance than previous CPU (NAMD) and GPU (Fen-ZI) codes. Achieving performance on par with FPGAs, however, potentially requires more aggressive optimizations to both the algorithms and to the underlying GPU memory architecture.

REFERENCES

[1] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[2] J. Phillips, J. Stone, and K. Schulten, "Adapting a message-driven parallel application to GPU-accelerated clusters," in *Proc. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis – Supercomputing*, 2008.

[3] Shaw, D.E., et al., "Anton, A Special-Purpose Machine for Molecular Dynamics Simulation," in *Proc. Int. Symp. on Computer Architecture*, 2007, pp. 1–12.

[4] M. Chiu and M. Herbordt, "Efficient filtering for molecular dynamics simulations," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2009.

[5] ——, "Molecular dynamics simulations on high performance reconfigurable computing systems," *ACM Trans. Reconfigurable Tech. and Sys.*, vol. 3, no. 4, pp. 1–37, 2010.

[6] M. Chiu, M. Khan, and M. Herbordt, "Efficient calculation of pairwise nonbonded forces," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2011.

[7] T. Darden, D. York, and L. Pedersen, "Particle mesh Ewald: An N log (N) method for Ewald sums in large systems," *The Journal of chemical physics*, vol. 98, no. 12, pp. 10 089–10 092, 1993.

[8] M. Frigo and S. Johnson, "FFTW, C subroutine library," *URL http://www. fftw. org*, 2005.

[9] NVidia, "CUFFT :: CUDA Toolkit Documentation," *URL http://docs.nvidia.com/cuda/cufft/*, accessed May 2016.

[10] B. Humphries, H. Zhang, J. Sheng, R. Landaverde, and M. C. Herbordt, "3D FFTs on a Single FPGA," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 2014, pp. 68–71.

[11] A. Lawande, A. George, and H. Lam, "Simulative Analysis of a Multidimensional Torus-based Reconfigurable Cluster for Molecular Dynamics," in *Proc. Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications*, 2014.

[12] J. Sheng, B. Humphries, H. Zhang, and M. Herbordt, "Design of 3D FFTs with FPGA Clusters," in *IEEE High Perf. Extreme Computing Conf.*, 2014.

[13] J. Sheng, C. Yang, and M. Herbordt, "Towards Low-Latency Communication on FPGA Clusters with 3D FFT Case Study," in *Proc. Highly Efficient and Reconfigurable Technologies*, 2015.

[14] T. Matthey, T. Cickovski, S. Hampton, A. Ko, Q. Ma, M. Nyerges, T. Raeder, T. Slabach, and J. A. Izaguirre, "ProtoMol, an object-oriented framework for prototyping novel algorithms for molecular dynamics," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 3, pp. 237–265, 2004.

[15] 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.

[16] N. Ganesan, M. Taufer, B. Bauer, and S. Patel, "FENZI: GPU-enabled Molecular Dynamics Simulations of Large Membrane Regions based on the CHARMM force field and PME," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 472–480.

[17] Y. Gu and M. Herbordt, "FPGA-based multigrid computations for molecular dynamics simulations," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2007, pp. 117–126.

[18] T. VanCourt and M. Herbordt, "Application-dependent memory interleaving enables high performance in FPGA-based grid computations," in *Proceedings of the IEEE Conference on Field Programmable Logic and Applications*, 2006, pp. 395–401.

[19] A. Sanaullah, A. Khoshparvar, and M. Herbordt, "FPGA-Accelerated Particle-Grid Mapping," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2016.

[20] R. D. Skeel, I. Tezcan, and D. J. Hardy, "Multiple grid methods for classical molecular dynamics," *Journal of Computational Chemistry*, vol. 23, no. 6, pp. 673–684, 2002.