Gyrokinetic Particle-in-Cell Optimization on Emerging Multi- and Manycore Platforms

Kamesh Madduri[a], Eun-Jin Im[b], Khaled Z. Ibrahim[a], Samuel Williams[a],
Stéphane Ethier[c], Leonid Oliker[a]

[a] *Computational Research Division, Lawrence Berkeley National Laboratory, CA 94720*
[b] *School of Computer Science, Kookmin University, Seoul 136-702, Korea*
[c] *Princeton Plasma Physics Laboratory, Princeton, NJ 08543*

Abstract

The next decade of high-performance computing (HPC) systems will see a rapid evolution and divergence of multi- and manycore architectures as power and cooling constraints limit increases in microprocessor clock speeds. Understanding efficient optimization methodologies on diverse multicore designs in the context of demanding numerical methods is one of the greatest challenges faced today by the HPC community. In this work, we examine the efficient multicore optimization of GTC, a petascale gyrokinetic toroidal fusion code for studying plasma microturbulence in tokamak devices. For GTC's key computational components (charge deposition and particle push), we explore efficient parallelization strategies across a broad range of emerging multicore designs, including the recently-released Intel Nehalem-EX, the AMD Opteron Istanbul, and the highly multithreaded Sun UltraSparc T2+. We also present the first study on tuning gyrokinetic particle-in-cell (PIC) algorithms for graphics processors, using the NVIDIA C2050 (Fermi). Our work discusses several novel optimization approaches for gyrokinetic PIC, including mixed-precision computation, particle binning and decomposition strategies, grid replication, SIMDized atomic floating-point operations, and effective GPU texture memory utilization. Overall, we achieve significant performance improvements of 1.3–4.7× on these complex PIC kernels, despite the inherent challenges of data dependency and locality. Our work also points to several architectural and programming features that could significantly enhance PIC performance and productivity on next-generation architectures.

Keywords: Particle-in-Cell, multicore, manycore, code optimization, graphic processing units, Fermi

1. Introduction

The Particle-in-Cell (PIC) method [1, 2, 3] is a widely-used approach in plasma physics simulations, where charged particles (ions and electrons) interact with each other via a self-consistent grid-based field instead of direct binary interactions. Computationally, this has the important benefit of making the operation count proportional to N instead of N^2 (where N is the number of charged particles). The Gyrokinetic Toroidal Code (GTC) [4, 5, 6] is a 3D PIC application originally developed at the Princeton Plasma Physics Laboratory to study plasma micro-turbulence in magnetic confinement fusion. Turbulence is believed to be the main mechanism by which energy and particles are transported away from the hot plasma core in toroidal fusion devices called tokamaks. An in-depth understanding of this process is of utmost importance for the design of future experiments, since their performance and operation costs are directly linked to energy losses. Unlike many other applications, GTC's reliance on scatter/gather operations makes it notoriously difficult to optimize even in the sequential realm. In the shared memory environment, the resolution of fine-grained (word-level) concurrent access to shared data places GTC at the forefront of the programming challenges for the next decade.

Although GTC has been run at extremely high concurrencies, it has been done so mostly via a weakly-scaling message passing approach. As the technological scaling trend that enabled such parallelism has shifted, we are now forced to examine the challenges and solutions of GTC's principal kernels in a shared memory multicore environment to continue scaling performance. In this article, we build on our prior GTC multicore study for charge deposition [7] by significantly expanding the breadth of our research. First, we extend our study to include GTC's particle push phase. Push reads the electrostatic field and accelerates particles accordingly. Second, we extend algorithms, optimizations, and analysis to NVIDIA's Fermi GPU architecture. In order to gauge the limitations of slow atomic operations and large cache working sets, we create new mixed-precision (double+single or double+fixed) kernel implementations. We also broaden our multicore optimization space to include per-socket replicated grids, SIMDized floating-point atomic increment, and quantification of the impact of particle binning. We quantify our contributions on AMD and Intel's latest quad-socket shared-memory multiprocessors (SMPs), including a 24-core Opteron (Istanbul) and 64-thread Xeon (Nehalem-EX), as well as a Sun Niagara2 and a Tesla C2050 (Fermi) GPU. Moreover, we analyze the relative advantage of this set of machines to their predecessors (Barcelona, Nehalem-EP, and GT200).

Overall our work presents numerous node-level methodologies to significantly improve performance of gyrokinetic simulations, while providing in-depth analysis into the trade-offs of emerging multi- and manycore designs for challenging classes of numerical computations.

The rest of the paper is organized as follows. Section 2 provides a brief introduction to the PIC kernels studied in this paper. Section 3 details the machines and problem configurations used throughout the paper. Sections 4 and 5 present our novel optimizations for the charge and push phases respectively. We provide a detailed exploration and analysis of optimizations, architecture, and problems in Section 6. Finally, we provide several key insights in Section 7.

2. Overview of GTC kernels

PIC methods consist of the following steps in the main time loop: charge deposition on the grid (particle-grid interpolation or the "scatter" phase), field solve (converting density to field), gathering of field elements to calculate the force on each particle ("gather" phase), using that force to "push" each particle, and (in the case of distributed memory) shifting particles to other processors ("shift" phase). Overall, the charge deposition and gather/push phases account for a substantial amount of the total computation time (up to 80-90%) of the computational time. The parallel execution time depends on several factors, including the number of particles per grid cell on each process core, the number of parallel processors, and the architectural features of the parallel system [8, 9, 6]. Thus, understanding and optimizing these key routines on multi- and manycore architectures is imperative in achieving high performance on today's petascale machines.

2.1. Grids and Particles

Before proceeding with a discussion of the computational kernels, it is important to understand the characteristics of GTC's toroidal grid and charged particles. We model the spatial density of charge and the spatial variation in the resultant electric field via a 3D toroidal grid. Three coordinates (shown in Figure 1) describe position in the torus: ζ (*zeta*, the position in the toroidal direction), ψ (*psi*, the radial position within a poloidal plane), and θ (*theta*, the position in the poloidal direction within a toroidal slice). The corresponding grid dimensions are *mzeta*, *mpsi*, and *mthetamax*. The typical MPI decomposition results in each processor owning one poloidal plane and maintaining a copy of the next processor's plane. We maintain this approach in our experiments.

Unlike many other PIC codes, particles in GTC are not point objects. As the ions gyrate around magnetic field lines, GTC approximates their average charge distribution via four points on a ring with a gyroradius or the Larmor radius (ρ) [10]; this radius varies with the local value of the externally-imposed magnetic field B. Figure 1 shows three example particles (green) and their charge rings (red) in the bottom right sub-figure. The resultant memory references (blue) are discussed below.

2.2. Charge Deposition (scatter)

In the scatter phase, particles, represented by a four-point approximation to a charged ring, interpolate charge onto the charge grid (*chargei*). This process requires streaming through a large array of particles and updating locations in memory corresponding to the bounding boxes of the four points on the ring. As there are four points on each ring and eight points in each 3D bounding box, each particle update may access 32 unique grid memory locations (luckily, the 32 points are actually 8 stanzas of 4). Additionally, as seen in Table 1, for each particle, this kernel must read 40 bytes, perform 180 floating point operations (flops), and update 128 bytes. This works out to a low arithmetic intensity (assuming perfect grid locality and no cache bypass) of 0.61 flops per byte.

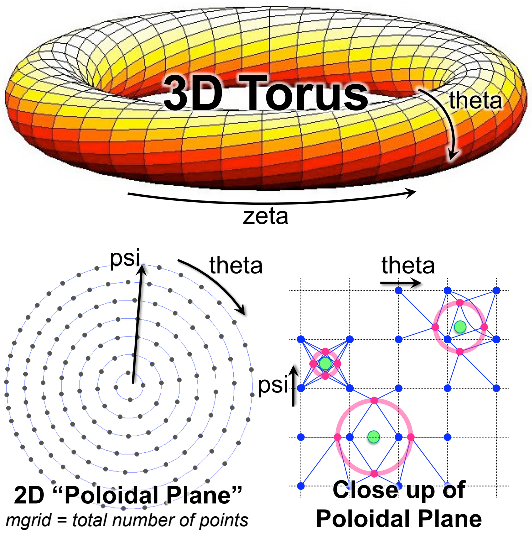


Figure 1: An illustration of the GTC's 3D toroidal grid and the four-point gyrokinetic averaging scheme employed in the charge deposition and push steps.

Note that the locality characteristics of this kernel are dependent on the order in which particles are accessed and the location of particles in the local domain. This in turn determines the locality in writes to the grid array. Enhancing locality of grid updates is one of the key targets of our new optimizations. The inability to maintain high grid locality will result in superfluous data brought by cache misses due to large cache line granularity, and will drive arithmetic intensity even lower.

In shared memory parallel programming models, the charge deposition phase is further complicated by multiple threads concurrently accessing the same grid points. Given the load-store nature of modern microprocessors, the corresponding interpolation operation is not inherently atomic. As such, a data hazard emerges and we must provide a synchronization mechanism to resolve it. Much of the charge phase optimization work presented in this paper is focused on efficiently resolving these data hazards.

2.3. Particle Push (gather)

In our experiments, we combine the nominally distinct "gather" and "push" phases into a single phase that performs both operations. In this new "push" phase, particles, still represented by a four-point approximation to a charged ring, interpolate an electric force field (*evector*) and push themselves. Again, this process requires streaming through a large array of particles, reading random locations in memory corresponding to the bounding boxes of the four points on the ring. As there are four points on each charge ring, eight points in each bounding box, and a cartesian vector electric field, each update may access 96

Table 1: The computational characteristics of the charge deposition and push kernels (double precision, $mzeta = 1$).

Single-iteration, per-particle counts	**Charge Deposition**	**Push**
Floating-point operations	180	450
Particle-array bytes read	40	184+24[a]
Particle-array bytes written	128	48+24[a]
Arithmetic intensity Upper Bound (flops/byte)[b]	0.61	1.61
Grid-array bytes read	$\leq$256[c] + 64[d]	$\leq$768[c]
Grid-array bytes written	$\leq$256[c]	–

[a] Memory references avoided with loop fusion optimization in the push phase (Section 5).

[b] Total flops per total particle data accessed. The impact of cache misses on grid data and superfluous data accesses are not considered.

[c] Exact value is dependent on each particle's Larmor radius.

[d] Memory references avoided with on-the-fly auxiliary array computation (Section 4).

unique grid memory locations. As seen in Table 1, this kernel (per point) reads at least 184 bytes of particle data, performs 450 flops, and updates 48 bytes of particle data. The arithmetic intensity of this kernel is higher than charge deposition's, but potentially irregular accesses to the electric field grid may hinder performance.

Unlike the charge phase, the push phase exhibits no data hazards. As such, it is somewhat simpler to optimize.

2.4. Crosscutting Overview

Table 1 presents the core characteristics of the charge and push phases. Although both phases access all particles, they read and update different sets of particle arrays. Arithmetic intensity is a useful metric in bounding performance for bandwidth-constrained kernels. Here we calculate arithmetic intensity as the ratio of flops to particle data accessed assuming a write-allocate cache. This is clearly an upper bound, as it assumes accesses to the grid do not generate a large number of capacity misses. Such an assumption is just for small grid sizes, but is inappropriate when the grid size is large.

Push is more compute-intensive overall, but also accesses a larger volume of particle data. Since the arithmetic intensity of push is higher than charge deposition, we expect push to achieve a commensurately higher performance rate. We also expect the performance of machines with a flop:byte ratio close to a kernel's arithmetic intensity to saturate after a certain parallel concurrency.

2.5. Related Work

PIC is a representative method from the larger class of *particle-mesh* methods. In addition to plasma physics (e.g., gyrokinetic PIC), particle-mesh methods find applications in astrophysics [2, 11], computational chemistry, fluid mechanics, and biology. VPIC [12], OSIRIS [13], UPIC [14], VORPAL [15], and

QuickPIC [16] are popular frameworks from diverse areas to express PIC computations.

Prior work on performance tuning of PIC computations has mostly focused on application-specific domain (mesh) decomposition and MPI-based parallelization. The ordering of particles impacts the performance of several PIC steps, including charge deposition and particle push. Bowers [17] and Marin et al. [18] look at efficient particle sorting, as well as the performance impact of sorting on execution time. A closely-related macro-scale parallelization issue is particle load-balancing [19], and OhHelp [20] is a library for dynamic rebalancing of particles in large parallel PIC simulations. Koniges et al. [21] report performance improvements by overlapping computation with inter-processor communication for gyrokinetic PIC codes. The performance of the GTC MPI implementation has been previously well-studied on several large-scale parallel systems [8, 9, 6]. Prior research also examines expressing PIC computations via different programming models [22, 23].

There has also been recent work on new multicore algorithms and optimizations for different PIC steps. Stanchev et al. [24] investigate GPU-centric optimization of the particle-to-grid interpolation step in PIC simulations with rectilinear meshes. Decyk et al. [25] discuss porting a 2D electrostatic code extracted from the UPIC framework to GPUs. In our prior work on multicore optimizations for GTC's charge deposition kernel, we introduce various grid decomposition and synchronization strategies [7] that lead to a significant reduction in the overall memory footprint in comparison to the prior MPI-based GTC implementation.

3. Experimental Setup

This section describes the extracted benchmarks, problems sizes, and experimental platforms. Additionally, we discuss the dual-socket machines used in our previous studies that are reused here as a baseline.

3.1. GTC Standalone Benchmarks and Problem Instances

Our study analyzes multicore performance of the charge deposition and push kernels by first extracting them from the GTC Fortran/MPI version and creating stand-alone benchmark routines. The setup, data representation, and computation performed are identical to the reference Fortran code.

There are several input parameters in GTC to describe a simulation. The ones most relevant to the examined kernels are the size of the discretized toroidal grid, the total number of particles, and the Larmor radius distribution of the particles for four-point gyrokinetic averaging. Often one replaces the number of particles with the average particle density as measured in the ratio of particles to grid points (labeled as *micell*). As our standalone benchmarks are designed to be representative of MPI simulations using a 1D decomposition, we mandate $mzeta = 1$ (*i.e.,* each process owns one poloidal plane and replicates the next). In order to demonstrate the viability of our optimizations across a wide variety

Table 2: The GTC experimental settings. *mzeta* = 1 implies each process operates on one poloidal plane.

Grid Size	**A**	**B**	**C**	**D**
mzeta	1	1	1	1
mpsi	90	192	384	768
mthetamax	640	1408	2816	5632
mgrid (grid points per plane)	32449	151161	602695	2406883
chargei grid (MB)	0.50	2.31	9.20	36.72
evector grid (MB)	1.49	6.92	27.59	110.18
Total Particles (micell=5)	0.16M	0.76M	3M	12M
Total Particles (micell=100)	3M	15M	60M	241M

of potential simulations, we explore four different grid problem sizes, labeled *A, B, C, D*, and vary the particle density from 5 to 100. Therefore, a “C20” problem — often used in experiments throughout this paper — uses the class C grid size with on average 20 particles per grid point. This simulation size corresponds to the JET tokamak, the largest device currently in operation [26]. Table 2 lists these settings, and these are similar to ones used in our prior experimental study [7], as well as GTC production runs [6]. All experiments run in this paper use the more challenging *strong scaling* regime. For all four GTC problem sizes used in this study, the maximum Larmor radius (a function of several other GTC parameters) corresponds to roughly *mpsi*/16. The radii are chosen from a uniform random distribution.

3.2. Architectures

In this section, we describe the seven platforms used to conduct our study; pertinent architectural details are shown in Table 3.

AMD Opterons: In this study we utilized both a four-socket, six-core Opteron 8431 (Istanbul) and a dual-socket, quad-core Opteron 2356 (Barcelona) [27]. In both cases, the Opteron architecture uses a semi-exclusive L3 cache resulting in somewhat larger attainable cache working sets than would be attainable via inclusive caches. To mitigate snoop effects and maximize the effective memory bandwidth, Istanbul uses 1 MB of each 6 MB cache for HT Assist (a snoop filter). The snoop filter enables higher bandwidth on large multi-socket SMPs. In many ways, this four-socket, 6-core Istanbul systems is an excellent proxy for the recently released dual-socket, 12-core Magny-Cours (Opteron 6100 series) which integrates two nearly identical chips on a multi-chip module.

As shown in Table 4, we used the GNU C compiler v4.4.1 to build our benchmarks on both the Opteron systems. Both the charge deposition and push kernels were initially implemented using the POSIX threads API (Pthreads). We further implemented the push kernel in OpenMP as well to take advantage of OpenMP’s various loop-scheduling schemes (static, dynamic, guided, etc.). All the results presented in this paper for the push kernel are with the OpenMP implementation. Further, we found that the performance of the Pthreads and

Table 3: Architectural details by platform. Note: with respect to multithreading on a GPU, we equate "threads" to the number of concurrent CUDA thread blocks per SM (core). Most experiments in this paper were run on the systems in the top half of this table. *Pin Bandwidth, †STREAM TRIAD Bandwidth, using cache bypass where possible, "MT" is multithreaded, "SS" is superscalar.

Core Architecture	**AMD** **Opteron**	**Intel** **Nehalem**	**Sun** **Niagara2**	**NVIDIA** **GF100**
		MT(2)	MT(8)	MT(48)
Type	SS	SS	dual-issue	dual-issue
	out-of-order	out-of-order	in-order	in-order
Clock (GHz)	2.4	2.27	1.16	1.15
DP GFlop/s	9.6	9.1	1.16	36.8
LS/L1D$/L2D$ (KB)	–/64/512	–/32/256	–/8/–	16/48/–
	Opteron	**Xeon**	**UltraSparc**	**Tesla**
	8431	**X7560**	**T2+**	**C2050**
System Architecture	**(Istanbul)**	**(Nehalem-EX)**	**(Niagara2)**	**(Fermi)**
sockets×cores×threads	4×6×1	4×8×2	2×8×8	1×14×48
Primary memory parallelism paradigm	HW prefetch	HW prefetch	Multithreading	Multithreading
Last-level cache	5 MB/chip	24 MB/chip	4 MB/chip	768 KB
(aggregate SMP$)	(33.5 MB)	(96 MB)	(8 MB)	
DRAM Capacity	64 GB	64 GB	32 GB	3 GB
DP GFlop/s	230.4	290.1	18.7	515
DRAM Bandwidth*	51.2 GB/s	136.5 GB/s	64 GB/s (2r:1w)	128 GB/s (ECC)
STREAM Bandwidth†	42 GB/s	34 GB/s	24 GB/s	79 GB/s
Flop:Byte ratio†	5.5	8.5	0.78	6.51

Core Architecture	**AMD** **Opteron**	**Intel** **Nehalem**		**NVIDIA** **GT200**
		MT(2)		MT(8)
Type	SS	SS		dual-issue
	out-of-order	out-of-order		in-order
Clock (GHz)	2.30	2.66		1.30
DP GFlop/s	9.2	10.7		2.6
LS/L1D$/L2D$ (KB)	–/64/512	–/32/256		16/24(T$)/–
	Opteron	**Xeon**		**Quadro**
	2356	**X5550**		**FX 5800**
System Architecture	**(Barcelona)**	**(Nehalem-EP)**		**(GT200)**
sockets×cores×threads	2×4×1	2×4×2		1×30×8
Primary memory parallelism paradigm	HW prefetch	HW prefetch		Multithreading
Last-level cache	2 MB/chip	8 MB/chip		256 KB (T$)
(aggregate SMP$)	(4.25 MB)	(16 MB)		
DRAM Capacity	16 GB	12 GB		4 GB
DP GFlop/s	73.6	85.3		78
DRAM Bandwidth*	21.3 GB/s	51.2 GB/s		102 GB/s
STREAM Bandwidth†	14.6 GB/s	32 GB/s		73 GB/s
Flop:Byte ratio†	5.0	2.66		1.07

OpenMP push kernel implementations with static loop scheduling (assigning equal-sized partitions of the particle array to each thread) were comparable. We chose the GNU compiler for the Opterons, since its OpenMP implementation provides the GOMP_AFFINITY environment variable to control thread pinning in OpenMP routines. To gauge the impact of alternate compilers and

Table 4: Programming model and software setup by platform.

Software Setup	AMD	Intel	Sun	NVIDIA
Threading Model	Pthreads (charge) OpenMP (push)	Pthreads (charge) OpenMP (push)	Pthreads (charge) OpenMP (push)	CUDA (charge & push)
Compiler, SDK, Drivers	GNU C v4.4.1	Intel C v10.1	Sun Studio C v12	CUDA C SDK

optimizations, we built the charge deposition kernel with the Intel C compiler and a selected hand-picked set of optimization flags. We observed that the parallel performance numbers (in GFlop/s) achieved for the C20 problem size using the GNU and Intel compilers were comparable on both the systems. In future work, we will attempt to automate the process of picking the best-performing compiler and appropriate optimization flags for a given problem instance.

Intel Nehalems: The recently released Nehalem-EX [28] is the latest enhancement to the Intel "Core" architecture, and represents a dramatic departure from Intel's previous large multisocket designs. Paralleling our analysis of AMD's designs, we include both a quad-socket, octal-core Xeon 7560 (Nehalem-EX), and a dual-socket, quad-core Xeon X5550 (Nehalem-EP). Unlike the dual-socket Nehalem-EP which integrated three DDR3 memory controllers directly on-chip, Nehalem-EX has four high-speed serial links to off-chip scalable memory buffers (SMB) which provide the scalability benefits of FBDIMM at the cost of commodity DDR. In our experiments, the Beckton system was equipped with only two SMB's per chip. However, concurrent experiments showed that increasing the number of SMB's per chip to four would not improve per-socket performance. Nehalem-EX is also unique in its last-level cache architecture. Cores are connected to cache banks via a ring. This allows for parallel cache references, but may also result in non-uniform cache access times. We believe the increased cache performance can strongly affect application performance.

On paper, the Nehalem-EX machine has 6× the cache, 4× the core count, and 2.7× the bandwidth of the Nehalem-EP system. In practice, the STREAM bandwidth was virtually identical. Through the two AMD and two Intel architectures, we gain insight into the relative strengths and weaknesses of the respective designs, as well as the future scalability of multicore SMPs.

We use the Intel C compiler v10.1 to build both the kernels on the Intel systems. The push phase uses Intel's OpenMP and the KMP_AFFINITY environment variable for controlling thread placement. We also found that, for the C20 problem size, the Nehalem-EX Intel compiler build for a tuned charge deposition implementation was around 1.2× faster than the executable built with the GNU C compiler (v4.1.2). Hence we use the Intel compiler on both the systems.

Sun UltraSparc T2+ (Niagara2): The Sun "UltraSparc T2 Plus" [29], a dual-socket × 8-core Niagara2 SMP, presents an interesting departure from mainstream x86 multicore processor design as it relies heavily on 8-way per core multithreading to hide latency. Moreover, both the raw per-core performance and available cache size per core is much lower than their x86 competitors. We view this machine's reliance on massive thread parallelism as a vanguard for future architectural exploration as it tests the limits of our algorithms.

Niagara2 has no hardware prefetching, and software prefetching only places data in the L2 cache. Multithreading may hide instruction and cache latency, but may not fully hide DRAM latency. Our machine is clocked at 1.16 GHz, does not implement SIMD, but has an aggregate 64 GB/s of DRAM bandwidth in the usual 2:1 read:write ratio associated with FBDIMM. As such, it has significantly more memory bandwidth than either Barcelona or Nehalem, but has less than a quarter the peak flop rate. With 128 hardware thread contexts, this Niagara2 system poses parallelization challenges that we do not encounter in the Gainestown and Barcelona systems.

NVIDIA GPUs: Paralleling our x86 exploration, in this paper we optimize code for NVIDIA's latest GPU (a Fermi-based Tesla C2050) [30] and compare the results to their previous generation (a GT200-based Quadro FX 5800) [31]. Both GPUs are designed primarily for high-performance 3D graphics (and similar applications) and are available only as discrete PCIe graphics cards. While the Quadro FX 5800 used a 240-"core" (30 streaming multiprocessors, or SMs) design, Fermi increases the core count to 448 and reorganizes them into SM's of 32 cores. Additionally, Fermi includes three other major improvements. First, Fermi's theoretical double-precision performance is half its single-precision (instead of the 1:8 ratio associated with the GT200). Second, Fermi includes non-coherent L1 and L2 caches (instead of the GT200's texture cache). Finally, Fermi is the first NVIDIA GPU to support ECC DRAM. All experiments in this paper were conducted with ECC-enabled. We observe that Fermi's ECC-enabled streaming bandwidth is not substantially higher than the GT200's. Like the GT200, Fermi preserves CUDA's *shared* memory concept, but augments it by allowing the programmer to partition each SM's 64 KB of memory among shared and cache. This facilitates programming, as it mitigates the onus on the programmer to find and exploit spatial and temporal locality. However, it also increases the tuning and performance optimization space.

GPUs are also of interest in that they implement atomic operations at the memory controllers rather than in the L1 caches. As such, they provide a testbed for evaluating the benefits of supporting one-sided atomic operations (*e.g.*increment), rather than emulating them with a compare-and-swap (CAS) operation. One expects the one-sided approach to be more efficient, as it allows injection of more parallelism and is not impaired by high on-chip latencies.

There is only 3 GB of fast (device) memory on the C2050. Unfortunately this is insufficient to encapsulate every problem instance discussed in the previous section. Although the impact of the resultant host-GPU transfer time is not examined in this study, our previous work [32] has examined this potentially significant source of performance overhead.

4. Charge Deposition Phase Optimization

Optimizing the charge deposition kernel for multicore involves balancing three contending forces: improving locality (spatial and temporal) of grid accesses, load balancing work (*i.e.,* charge updates) among threads, and efficiently resolving fine-grained (word-sized) data dependencies. Additionally, we explore mixed-precision implementations (64b computations + 32b charge grid – or – 64b computations + fixed-point charge grid) in order to examine the impacts of a reduced cache working set and faster integer atomic operations.

Our prior work [7] began with the MPI implementation of this kernel where the grid was fully replicated, and presented several different memory-efficient shared-memory threaded variants for dual-socket multicore systems. These approaches were all based on radially partitioning and only partially replicating the charge grid to ensure that the memory usage remains constant as thread-level parallelism increases. This study also quantified the benefits of using tuned atomics versus Pthreads locks, and the effect of low-level optimizations such as NUMA-aware initialization and thread pinning.

This paper builds on the previous study and explores additional multicore optimizations, as well as new approaches for the Fermi GPU. We observe that at a high-level, there is substantial overlap between CPU and GPU optimizations; however, their optimal parameterizations may be different. Additionally, some optimizations are prohibitively costly given Fermi's massive parallelism and limited memory capacity, and others (like NUMA) are unnecessary. Conversely, the GPU's single instruction multiple thread model (SIMT) necessitates vectorization. Note that although most optimizations can be implemented and benchmarked independently, our work examines the impact of incrementally laying optimizations techniques.

4.1. Parallelization

The charge deposition phase is parallelized across all architectures by partitioning the particle array (and the associated loop iterations) among threads. The remainder of this section primarily discusses how to efficiently resolve the resultant deluge of data hazards. Our CPU Pthreads implementation performs a static partitioning of the particle array. Further, threads are pinned to cores and arrays are initialized in a NUMA-aware fashion.

On the GPU, parallelization is more complex due to the explicit hierarchical nature of CUDA programming. The number of particles per CUDA thread and the number of CUDA threads per CUDA thread block must be selected (the total number of thread blocks is based on the number of particles). We tune the implementation by varying these parameters and results indicate that the optimal parameter settings are dependent on the problem size. For example, in case of C20 on Fermi, the best performance was achieved with the number of particles per thread set to 99, the number of threads per thread block set to 256, and 476 thread blocks. Future work will examine modeling techniques to determine optimal parameters for a given problem configuration.

A second challenge on the GPU is to ensure that the parallelization strategy provides (or facilitates) memory coalescing. By choosing a linear mapping of particles to threads (modulo 32), we guarantee memory coalescing for particle reads and writes. Unfortunately, the random gather/scatter nature of this kernel makes coalescing for grid updates challenging. We explore a number of techniques, including *GPU cooperative threading* (see Section 4.6), to resolve this.

4.2. Locality-Improving Optimizations

Particle Binning. Particle locality is paramount to the performance of charge deposition, as it ensures that each thread's cache working set of grid points (a thin annulus) is minimized. Furthermore, the radial grid replication strategies discussed later in this section will be of maximum benefit when particles are processed roughly in the order of their radial grid position (*i.e.,* radially sorted). Options range from ordering particles via a full sort (sorting by radial coordinate, then *theta*, and then *zeta* coordinates) to simple binning in the radial direction. It is also reasonable to assume that binning is not required for every simulation time step, as the particle position variation in the radial direction is considerably less than the variation in the θ and ζ directions.

In this paper, we gauge the impact of particle locality by implementing a fast, shared-memory parallel, out-of-place binning routine. Our approach reuses auxiliary particle arrays that are unused in the charge deposition phase, and thus does not incur an additional memory overhead. Additionally, our work also explores binning in the θ direction, which would further improve locality by constraining particles to small sector of an annulus. Note, however, that particle binning just mitigates the locality problem in charge deposition, as the four-point gyro-averaging scheme still leads to irregular and unconstrained memory accesses. By constraining the centers of the charge rings to an annulus (with radial binning) or sector of an annulus (with radial and theta binning), we hope to similarly constrain the four points on the perimeter of the circle.

Local/on-the-fly computation of auxiliary grid arrays. The charge deposition phase utilizes about 120 flops to determine the bounding box for each particle. In addition to memory accesses that stream through the particle arrays, this step involves irregular lookups to two different auxiliary grid arrays used for four-point gyro-averaging. These arrays are the size of the charge grid, and thus entail similar cache working set sizes and random access penalties. To mitigate this, we create a version of charge deposition that eliminates accesses to these auxiliary arrays, but instead redundantly recomputes these values for every particle. In effect, increased computation is traded for a reduced cache working set (see Table 1). Such strategies are generally viable for a range of large grid sizes, low particle densities, and machine flop:byte balances when the savings in cache misses exceed the cost of about 20 flops (which include two instances of transcendental functions).

4.3. Synchronization Optimizations

In a shared-memory implementation of the charge deposition phase, the particle array is partitioned but the charge grid is shared. Although this dramatically reduces the memory requirements compared to the MPI version, it necessitates fine-grained (word-level) data dependency resolution, as any thread

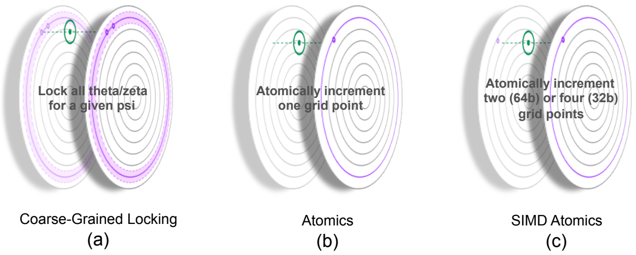

Lock all theta/zeta for a given psi
Atomically increment one grid point
Atomically increment two (64b) or four (32b) grid points
Coarse-Grained Locking
(a)
Atomics
(b)
SIMD Atomics
(c)

Figure 2: Synchronization Strategies.

can increment any location in the shared grid at any time. As a result, in our multicore processors, up to 128 threads will contend for access, and on the GPUs thousands of threads can contend for access. In this section, we discuss the three strategies used in this paper to resolve data dependencies and illustrate them in Figure 2.

Coarse-Grained Locks. The simplest approach is to use a lock variable for each ring (all θ and ζ for a given ψ) in the charge grid. To perform an update, a thread must acquire the lock, update the four points on the ring, and then release the lock. Unfortunately, the Pthreads lock mechanism requires a substantial overhead, and in this approach the attainable parallel concurrency is limited by the number of locks (ψ, or 90-way parallelism on the smallest problem). As the GPU threading model forbids locks, they were not implemented on Fermi.

Atomics. In our prior paper [7], we exploited the x86 `cmpxchg` instruction coupled with custom inline assembly code to perform atomic floating-point increments. This work extends this methodology to both single-precision via a 32b compare-and-swap (CAS), and to SIMD (via the `cmpxchg16B` instruction). The SIMD operation atomically increments two (or four) contiguous addresses by two (or four) unique floating-point numbers. As GTC always updates four contiguous memory locations, these SIMD atomic increments minimize the atomic overhead per floating-point increment. We can perform atomic SIMD floating-point increment on 2×32b, 4×32b (8B or 16B aligned), or 2×64b (16B aligned) SIMD vectors. Unfortunately, although the current version of GPUs implement integer and single-precision atomic increment, they do not support double-precision floating-point atomic increment. We may however leverage intrinsics within CUDA to implement a CAS-based emulation of double-precision floating-point increment. Unlike CPUs, the CAS operation is performed at the memory controllers rather than in the cache — incurring much higher latency and on-chip bandwidth due to the round trip nature of CAS.

Sorting. In addition to CAS-based atomics, there is an alternate pure double-precision GPU approach: We can implement a data-parallel solution in which we produce a list of charge updates from particle data (embarrassingly parallel), sort this list (we use an optimized radix sort [33] using the routine

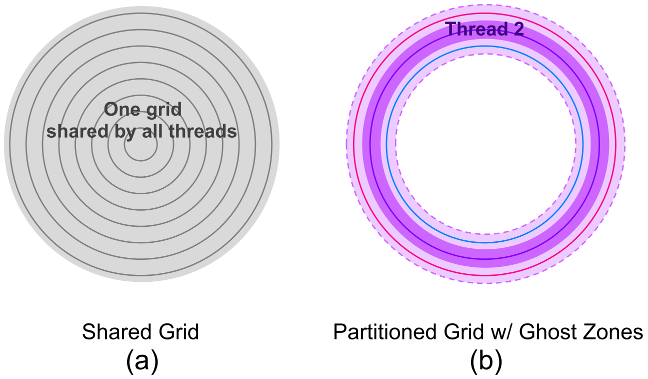


Figure 3: Grid Replication Strategies.

`thrust::sort_by_key()` provided by Thrust template library [34], and finally update the grid after a segmented scan. Although sorting is efficiently parallelized and does not require locks or atomics, it makes the overall charge deposition step work-inefficient. It also requires far more kernel invocations and increased memory requirements (an additional 256 bytes per particle) — potentially resulting in lower performance. In fact, for even the modest C20 problem, the sheer number of updates is so large that the entire list of charge updates cannot be generated on the GPU. We stream batches of particle data, generate partial lists, sort the lists, and then apply the charge updates.

4.4. Partial Grid Replication Optimizations

Although a shared grid certainly reduces memory capacity requirements, it requires costly locks or atomic updates. To mitigate these performance impediments as well as locality and inter-socket cache line thrashing, we explore three (partial) grid replication strategies shown in Figure 3.

None (Shared Grid). The simplest approach is to perform no replication, utilizing one large shared grid. Binning of particles limits each socket's updates to a quarter (or half for Niagara2) of the grid. Unfortunately this approach requires every thread to potentially contend with all other threads for updates. Moreover, if contending threads are located on different sockets it may result in cache line thrashing.

Partitioned Grid with Ghost Surfaces (PG). To mitigate the contention challenges, we create an auxiliary grid that is partitioned in ψ (creating overlapping annuli) among threads. The degree of overlap is dictated by the Larmor radius. Each thread may either safely update its own partition free of synchronization and locality challenges or update the shared grid and contend with locality, synchronization, and thrashing. At the end of the charge phase, all partitions are reduced to a single shared grid. This was consistently the best-performing grid replication strategy in our prior work [7].

Replicated Partitioned Grid (RPG). Unfortunately, high parallelization (greater than 16-way) reduces the size of each thread's partition and results in many more updates to the shared grid. We see this pushed to the extreme on Nehalem-EX (64 threads) and Niagara2 (128 threads). Our new work thus extends the partitioned grid approach by creating one (or more) auxiliary grids per socket. By varying the number of replicas, we may limit the parallelism within a grid to 16-way (or lower). In effect, each group of 16 threads shares a replica of the full grid and each thread in the group creates a partitioned grid replica of a subset of that grid. One can visualize this approach by viewing Figure 3(a) as private to each socket, and Figure 3(b) as a replica of each thread within that socket. As such, the (grid) working set per thread is larger than what might have been possible if the pure partitioned grid approach were parallelized among 128 threads.

The reduction for this strategy is more complicated, as $P/16$ replicas of the full grid must be merged with P larger private partitioned grid replicas. As a result, this approach improves load balancing, synchronization, locality, and thrashing at the expense of increased cache capacity and reduction time. Importantly, we still classify this approach as memory-efficient, as the number of replicas will not scale proportional to the thread count.

Table 5 estimates the cache working set size as a function of grid partitioning and thread-level parallelism within a socket for a generic 4-socket SMP, assuming radially-binned initial particle positions. Note that as we stream through the particle arrays without reuse, the particle arrays do not necessitate significant cache usage. As we move from a shared grid, to partitioned grid, and finally to a replica strategy (to mitigate inter-socket cache line thrashing), the cache working set size grows considerably. Our empirical tuning allows us to weigh the benefits of locality and load balancing against the impact from contention and thrashing.

Table 5: Cache working set sizes per socket (in MB) for 4-way SMPs.

Grid Size	**A**	**B**	**C**	**D**
Shared Grid (4 sockets)	0.22	1.03	4.11	16.4
Partitioned Grid (>16 threads)	0.60	2.77	11.0	43.9
Replica + Partitioned (4×6)	0.92	4.28	17.1	68.2
Replica + Partitioned (4×16)	1.75	8.14	32.4	129.6

4.5. Mixed-precision

The presence of single-precision and integer atomic increments in GPU memory controllers coupled with the desire to gauge the impact of improved locality motivated us to take the novel approach of utilizing mixed-precision. Mixed-precision implementations promise improved GPU performance by leveraging GPUs support for one-sided 32b atomic increment operations instead of iterative CAS-based solutions. In our mixed-precision implementation, all data structures except for a temporary copy of the charge grid are 64b double precision; a temporary charge grid in a 32b (fixed or floating-point) representation is created. The charge deposition computation proceeds normally in double precision, except that charge grid updates are first down-sampled to 32 bits and the temporary 32b (fixed or floating-point) grid is atomically updated. After all charge updates are complete, the 32b temporary charge grid is converted to its full 64b double precision representation. Such a method presumes the cost of conversion is offset by the increased performance when incrementing the grid.

In this work, we explore both a 32b floating-point and a 32b (8.24) fixed point representation. The latter transforms floating-point increments into integer increments (in millionths of a unit charge). One should remember that the number of particles (and thus the number of updates) per grid point is relatively small (less than 100), there is no need for a large dynamic range as each update may not deposit more than 1.0 units of charge.

To determine the impact of mixed precision in the charge accumulation phase, we carried out several simulations with the full GTC application, with and without mixed precision and for the exact same set of parameters, including identical initial particle positions and velocities. We used grid size A and 3 different numbers of particles per cell, specifically 10, 40, and 80. We observed that both the linear stage, during which the turbulence develops, and the final steady-state level are nearly identical for both the mixed precision and full double precision simulations, and for all 3 numbers of particles per cell. Only the saturation phase and the full non-linear phase preceding the steady-state show a small variation in the dynamics, and this variation gets smaller as the number of particles increases. Initializing the particles with different random positions and velocities can produce the same variation in the dynamics, and so we conclude that it is not statistically relevant. Comparing the results closely, we notice that using a single-precision array for the charge accumulation has the same effect as adding a small amount of smoothing to the grid density. The PIC method always includes some smoothing of the grid-based fields in order to minimize the fluctuations at the scale of the grid spacing [3]. These fluctuations are mainly due to the finite number of particles used for sampling the phase space, which is usually referred to as discrete particle noise. The smoothing affects only the shortest wavelengths in the system and allows for the use of a smaller number of particles, since the level of short-scale fluctuations decrease as the number of particles increases ($\propto 1/\sqrt{N}$).

The use of mixed precision is thus physically justified by the fact that it does not affect the longer wavelengths being generated in the system by instabilities resulting from the collective motion of a much larger number of particles. The production GTC code used to be run entirely in single precision when only the ions were treated explicitly and the required number of time steps was relatively small. We thus employ this mixed-precision strategy for both GPU and CPU platforms.

4.6. Low-level Optimizations

Structure of Arrays. As only a subset of each particle's data is used in each phase, we represent the particle arrays in a structure-of-arrays (SOA) layout. Doing so maximizes spatial locality on streaming accesses. This optimization is employed on both CPUs and GPUs.

CPU threaded Implementations. Our new Pthreads codes include data reorganization and loop fusion optimizations discussed in our earlier work [7]. We perform a "NUMA-aware" initialization of the particle and grid arrays by explicitly pinning threads to cores, and relying on the first-touch page allocation policy for exploiting thread-memory affinity. Affinity is exploited in the charge deposition computation by pinning threads to cores, and employing a static thread scheduling scheme. Further, we use SSE2 intrinsics on the x86 systems for charge density increments to the thread-local grids in PG approaches.

GPU cooperative threading. GPUs operate best when memory transactions can be coalesced. Nominally, that happens when thread_i accesses element_i (modulo 32). Unfortunately, particle-to-grid interpolation dramatically complicates this. Although it is quite easy to assign successive threads to successive particles and thereby attain memory coalescing on reads from a structure of arrays representation of particles, the resultant updates to the grid will not be coalesced as there is no reason to expect particle_i to update a grid location adjacent to that updated by particle_{i+1}. To rectify this, we introduce a cooperative model in which threads, after reading in particle data and performing interpolation locally, exchange data through shared memory and then update grid points. Although perfect memory coalescing is not possible on the grid accesses, we can ensure that threads $i ... i + 3$ access grid elements $j ... j + 3$ and thereby attain some measure of spatial locality. The limits on coalescing are a result of the $2 \times 2 \times 2$ bounding box interpolation method used by GTC.

An overall summary of optimizations by type and architecture is presented in Table 6.

5. Particle Push Phase Optimization

Optimizing the push deposition kernel for multicore involves balancing two main forces: improving locality (spatial and temporal) of grid accesses, and load balancing work among threads. Unlike the charge deposition phase, there are no fine-grained data dependencies. We build on our knowledge gained in charge and explore additional multicore and GPU optimizations. At a high-level, there remain a number of similarities between CPU and GPU optimizations. Intriguingly, some graphics hardware features like texture caches synergize well with this interpolation.

Table 6: Charge Deposition Optimizations.

	x86	Niagara2	GPU's
Particle Binning	✓	✓	✓
On-the-fly computation of auxiliaries	✓	✓	✓
Coarse Locks	✓	✓	
Scalar Atomics	✓	✓	✓
SIMD Atomics	✓		
Shared Grid	✓	✓	✓
Partitioned Grid	✓	✓	
Replicated Partitioned Grid	✓	✓	
Mixed Precision (+32b Single-Precision)	✓	✓	✓
Mixed Precision (+32b Fixed-Point)			✓
Mixed Precision (+64b Fixed-Point)			✓

5.1. Parallelization

Once again, we simply divide the loop over the particle array among threads. On the CPUs, we report performance for implementations with OpenMP-based threading. To ensure optimal performance, we pin threads and initialize arrays in a NUMA-aware fashion.

Once again, the GPU implementation was far more complex. By choosing a linear mapping (modulo 64) of particles to threads, we guarantee memory coalescing for particle reads and writes. Unfortunately, like charge, the random gather nature of this kernel makes explicit management of memory coalescing for field reads a non-profitable optimization. We explored moving blocks of the field grid to CUDA shared memory, but with only 12 memory locations to coalesce, under-utilized threads limit performance. Finally, we explored different configurations of particles per CUDA thread (parallelization granularity) and CUDA threads per thread block. We found the optimal parameters to be 1 particle per thread and 64 threads per thread block.

5.2. Locality-Improving Optimizations

Eliciting good spatial and temporal locality from the underlying memory architecture is the key push phase challenge. The push phase baseline version is comprised of two loops. The first gathers data from the field grid, storing into an array proportional to the number of particles. The second loops reads the gathered data and updates the corresponding particles. As the gathered field data is never used again, elimination these auxiliary arrays by fusing the two loops reduces the cache working set and therefore eliminates capacity misses. This optimization is employed on both CPUs and GPUs.

As particles are radially sorted (the gather randomness is constrained to a small range in ψ, but encompasses all θ), the resultant data working size per thread is relatively small (about 12% of grid size). For class A problems this is less than 180 KB/thread, but it grows quickly reaching 13 MB/thread for the class D problem. As threads are packed densely and there is no replication of the grid in the push phase, their cache working sets are actually overlapping. Thus, for the four-socket SMPs, the cache working set in the last level cache is roughly 25% of grid size, regardless of the thread parallelism. As such, we expect good locality on the CPUs up to a key problem size (depending on the machine's cache), beyond which performance will fall off due to cache misses.

On Fermi, reads from device memory are now cached. However, the cache is only 768 KB and thus incapable of capturing even the class A problem. As there is a 2D locality to interpolation, we use the texture language attribute to force reads to pass through the multilevel texture cache. The GT200 and C2050 (Fermi) do not support double-precision data handling by the texture memory. Luckily, unlike charge deposition where updates must be atomic, the high and low 32 bits can be read sequentially and then combined into a 64b double-precision number using `__hiloint2double`. Although the entire grid cannot fit into the texture cache, this nonetheless increases the benefits of temporal locality.

5.3. Load-Balancing Optimizations

On profiling the CPU code, a minor load imbalance issue was observed when using a static particle partitioning. Due to their slightly larger cache working sets, threads that are assigned particles at the outer torus periphery take longer to finish. We alleviate this imbalance by employing OpenMP's guided loop scheduling scheme, while retaining the initial NUMA-aware allocation.

Load balancing in GPUs is done implicitly by the thread block spawner. A typical GPU program is composed of many fine-grained threads that are grouped into blocks. In our implementation, each thread does the computation for one particle, and a thread block is chosen to be 64 threads. The number of blocks ranges from a few thousands (grid size A, micell =5) to millions (grid size D). As the number of GPU multiprocessor (14 in Tesla C2050) is much smaller than the number of blocks, load-balancing can easily be achieved by the spawner by incrementally assigning thread blocks to free multiprocessors.

5.4. Low-level Optimizations

Structure of Arrays. Similar to the charge deposition kernel, we represent the particle arrays in an SOA layout to maximize spatial locality.

CPU threaded Implementations. As mentioned previously, we perform a NUMA-aware memory initialization of particle and grid arrays and also pin threads to cores using OpenMP environment variables. In future work, we will utilize SSE intrinsics to vectorize amenable parts of the push kernel.

GPU. There were a number of GPU features exploited to further improve performance. First, we replaced conditional statements that often lead to divergent execution with code amenable to predicated execution. Next, in addition to the SOA data layout, the array alignment was modified and padded to facilitate coalesced accesses to the memory. Shared memory is now used to hold data with temporal locality, especially those carried across fused loops. We also moved certain runtime constants into the GPUs special constant memory. On GPUs, declaring data as texture (read only) allows caching them without coherency concerns. Experiments showed that texture declaration improved performance for small dataset (grid size A) by up to 53%, while the benefit shrank to 6% for grid size D.

Finally, the Fermi GPU architecture allows users to configure the fraction of each SM's memory dedicated for shared memory and L1 cache. Preferring larger L1 configuration yielded better performance only for large dataset. In general the interaction of these cache configurations and the dataset sizes is profound on performance.

6. Experimental Results and Analysis

In this section, we present the results of our performance optimization efforts and provide a detailed performance analysis of both GTC's charge deposition and particle push phases.

6.1. Charge Deposition Phase

Optimization of the charge deposition phase must resolve two fundamental challenges: resolution of data dependencies and management of data locality. In this section, we proceed through our set of optimizations quantifying and analyzing their benefits.

6.1.1. Impact of Locality Optimizations

In a naïve implementation, the particles are randomly distributed throughout the full grid. This necessitates a per-socket cache working set equal to the full grid (almost 10 MB for Class C) and results in substantial inter-socket cache line thrashing. To mitigate this, the particles can be periodically binned, so that subsets of the particle array are contained in a corresponding subset of the charge grid. Two versions of this approach are explored. In the first, particles are sorted based on their *psi* (radial) coordinate. While this ensures that all particles with the same radial coordinate are contiguous in memory, the charge updates are distributed in a annulus around this radial coordinate due to the variation in the particle Larmor radius. As a result, we reduce the cache working set size from the full grid to roughly $1/8^{th}$ of it. This can then be improved upon by sorting particles based on their *theta* coordinate.

With particles now ordered first their radial and then their angular coordinates, the working set size has been reduced to roughly $1/64^{th}$ of the full grid. Effectively, the Larmor radius defines a moving cuboid as one streams through particles.

Figure 4 quantifies this benefit and cost on the C20 problem configuration for each architecture by examining three cases: no binning, radially binning, radial and theta binning. Moreover, the asymptotic limit can be estimated if we perform one bin (total) rather than bin every time step. Results show that on

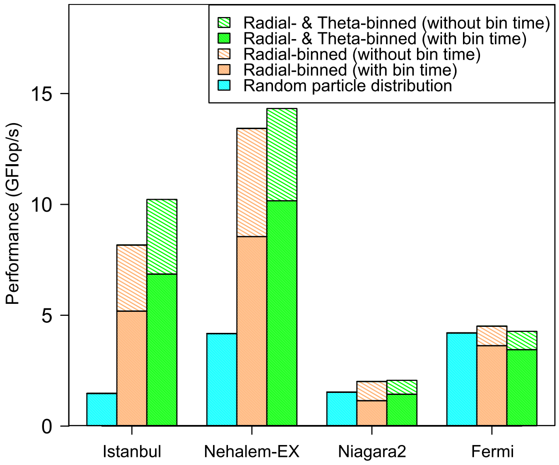


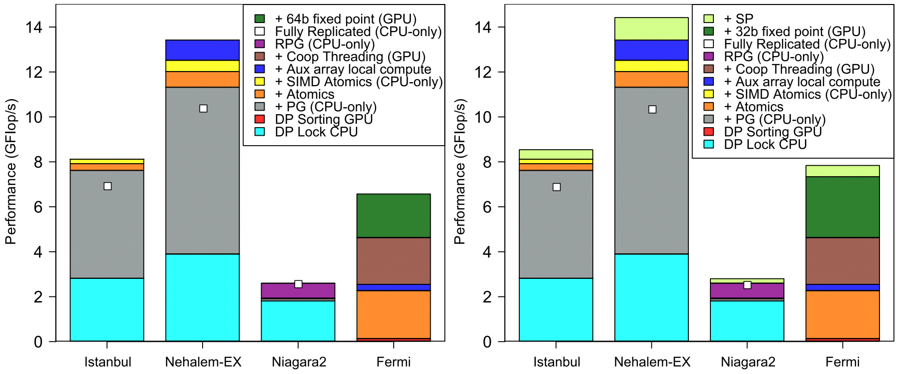
Figure 4: SMP/GPU Charge deposition performance as a function of particle locality optimization. Performance is shown for the *partitioned grid with ghost flux surfaces and atomic increments* version for the C20 (grid=C, density=20) problem.

Istanbul and Nehalem-EX, the performance benefit is substantial, as the inability to maintain a small cache working set dramatically impedes performance. Moreover, the time required for such a bin is not negligible, as evidenced by the amortized bin time performance. If particles move sufficiently slowly (or if the bin operation is included in preparation for inter-node MPI communication), this asymptotic limit can be attained.

Fermi presents a more nuanced effect. The time required to sort particles significantly exceeds any benefits gained from sorting. When this time is amortized (lighter bars), there is a moderate performance boost. One can surmise the small GPU caches cannot capture the requisite temporal locality in either scenario. As such, the performance benefit is negligible. All subsequent charts leverage this insight and assume radial binning has been performed on both CPUs and GPUs.

6.1.2. Benefit of Synchronization and Data Replication Optimization

Our grid replication strategies constrain the complexity of the data dependency problem, while we improve the efficiency of data dependency resolution by selecting the architecturally-optimal synchronization mechanism. By examining a mixed-precision implementation, we reduce cache working sets and, in the case of GPUs, gauge the potential benefit of hardware support for atomic double-precision floating-point increment. To that end, Figure 5(a) shows double-precision performance as a function of grid replication, optimization, and use of mixed-precision on each architecture for the C20 problem. The baseline CPU implementation utilizes a double-precision shared grid with coarse-grained radial locks for update synchronization (indicated as DP CPU in the figure) while the baseline GPU implementation utilizes the update sorting algorithm. Note that



(a) Double-Precision (64b charge grid)

(b) Mixed-Precision (32b charge grid)

Figure 5: SMP/GPU Charge deposition performance as a function of optimization for the C20 problem. Notes: The CPU baseline implementation uses a shared charge grid with locks, while the GPU baseline is pure double-precision (DP) with the cooperative threading optimization, and uses an Atomic CAS for synchronization. The performance of the shared memory full replication implementation (also the algorithm employed in the MPI implementation) is shown as a white square. "PG" is partitioned grid with ghost zones. Optimizations are cumulative. *i.e.,* "+ SP" changes the grid precision to single while keeping the partitioned grid and SIMD atomics optimizations.

at this scale, the performance of the sorting algorithm is virtually unnoticeable.

On CPUs, we observe the partitioned grid (*+PG*) replica eliminates most of the writes to the shared grid. The use of atomics (*+Atomics*) results in lower contention and reduced per-update overhead compared to Pthreads locks. These optimizations already achieve a substantial boost in performance on Istanbul and Nehalem-EX. Further improvements are gained via our new SIMDized floating-point atomic increment operations (*+SIMD Atomics*). These optimizations achieve more than a 3× performance improvement over the baseline Pthreads implementation on Istanbul and Nehalem-EX. Moreover, it is important to note that the PG approach now outperforms the memory-inefficient replica per thread version examined in our previous work [7] and used in the MPI implementation. This was not the case on the dual-socket Nehalem-EP and Barcelona systems of that study. We believe that the increase in parallelism (3-4×) has allowed us to reach a crossover point in which the final reduction substantially impacts overall running time. Interestingly, the PG does not yield a performance improvement on the Niagara2 primarily due to the higher overhead for atomic increments, load imbalance, and compute-bound nature of the machine. Thus, we utilize the new RPG (multiple replicas per socket with partitioning) implementation (*Replica/Socket+PG*) to constrain the grid partitioning to 8- or 16-way. This trades atomics for increased cache pressure, but is ultimately beneficial on the Niagara2.

Examining the GPU implementation shows an order of magnitude performance boost via the shared grid strategy with atomic (CAS-based emulation of double-precision increment) updates. Moreover, results show that *cooperative threading* can provide a further doubling of performance. Unfortunately, performance is ultimately limited by the rather slow compare-and-swap based emulation of floating-point atomic increments. We can gauge the impact by implementing a 64b fixed-point charge grid and using a one-sided integer atomic increment. Despite the overhead of having to subsequently convert a fixed-point grid to floating-point, we see that performance increases by almost 50%. Clearly CAS-based solutions provide a substantial benefit over sorting, but one-sided atomics would dramatically boost performance.

Additionally, we examine the performance impediments of atomic operations and finite caches by mixing 64- and 32-bit operations. Figure 5(b) presents a mixed-precision implementation on CPUs and GPUs in which all computation is still 64b double-precision, but the charge grid is either 32b floating-point or 32b fixed-point. Note that for clarity, the lower bars are simply reproduced from Figure 5(a). Results show that the CPU-mixed precision implementation shows little benefit on Istanbul, while there is a moderate benefit on Nehalem-EX. The GPU shows improvement using the one-sided 32b fixed-point operations which is higher than the 64b fixed-point performance in Figure 5(a). This suggests that there is an advantage in reducing the cache working set, but ultimately, the small GPU cache is ineffective for this class problem. Interestingly, the one-sided 32b floating-point mixed-precision implementation outperforms the 32b fixed-point version on GPUs. Overall, we see the coupling of larger caches and atomics performed in their L1 caches give the CPUs an advantage.

6.1.3. Perturbations to Problem Configuration

Finally, in order to understand the resilience and broad applicability of our optimizations, we examine and analyze performance relative to the grid size and density. Varying grid size allows us to study the performance impacts on the working set size; changing the density impacts temporal locality and amortizes the overhead of capacity cache misses. Note that as some problem configurations are extremely large, the GPU lacks sufficient DRAM to run them.

Figure 6 shows the best attained performance as a function of problem configuration over our entire set of implementations including PG, PG2 (Partitioned grid at half the maximum thread concurrency), PGL (Partitioned Grid with Local/On-the-fly auxiliary array computation), and variants of RPG (with 1, 4, and 8 replicas per socket). In case of the GPU, the best-performing approach uses CAS atomics with cooperative threading and on-the-fly auxiliary array computation.

On Istanbul and Nehalem, performance drops considerably when grid size is increased beyond classes B and C respectively. This is directly correlated with the cache working set requirements and the available shared last-level cache size (see Tables 3 and 5). Additionally, we observe a transition in the optimal implementation from PG to PGL. Recall that PGL is the PG variant in which the gyro-averaging constants are recomputed on-the-fly (*i.e.,* locally) instead of being gathered from memory. As these constants are proportional to grid size, PGL reduces cache pressure dramatically at the cost of increased computation.

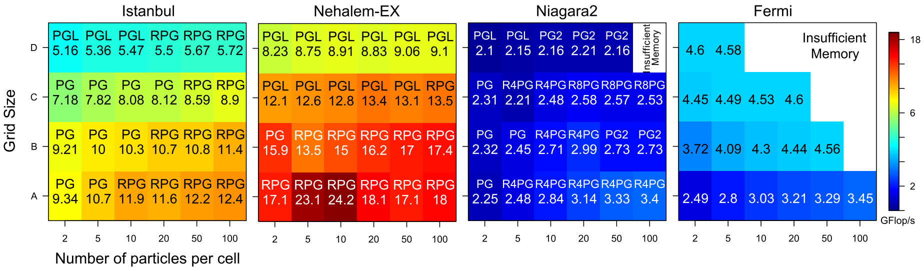


Figure 6: SMP/GPU Charge deposition performance in GFlop/s (180 flops per particle per iteration) achieved by the best double-precision implementation on each system as a function of grid size and particle density. Note: an execution time of 1 second corresponds to 2.12 GFlop/s for grid size C at 20 particles per cell. "PG" is partitioned grid, "RPG" is replicated partitioned grids, "PGL" is partitioned grid with local computation of auxiliary arrays, "R#PG" creates # (partitioned grid) replicas per socket.

Thus, as the grid's cache working set size becomes critical, the PGL approach becomes superior (C for Nehalem-EX, and D in the case of Istanbul and Niagara). At high particle densities, the cache working set is less critical. On Niagara2, the performance of 128-way grid partitioning is severely limited by the low throughput of atomics, and hence variants of RPG optimizations dominate in most cases. Four-way replication per socket sufficed for smaller grid sizes, but Niagara2 requires up to an 8-way replication per socket at higher particle densities. For grid size D, a 64-way partitioning (utilizing only 4 threads per core) was slightly faster than RPG. As RPG already lad to a 8-way or 16-way increase in cache working set requirements, the last-level cache size is insignificant compared to the increased load balancing and reduced inter-socket thrashing. Ultimately, Nagara2 is an extremely underpowered machine resulting in a flat performance map. On Fermi the performance trend is the opposite that of CPUs, where larger problems yield better performance. We believe this is an artifact of how increased parallelism results in decreased contention for atomic operations.

Overall, we expect RPG to be faster for the smaller grid sizes, as PG may over-parallelize at 128-, 64-, and possibly 24-way concurrency. As the grid size increases, the impact of over-parallelization is decreased, but the cache working set associated with RPG becomes and impediment as it no longer fits in cache. To that end we see a transition to the PG approach as the best solution. Ultimately, as PG begins to fall out of cache, replacing grid constants with on-the-fly-computation of them slightly improves grid locality.

As particle density is increased, so is temporal locality (assuming particles were radially sorted), and thus a performance improvement is expected. Results in Figure 6 confirm this trend to generally hold true. On the Nehalem-EX, performance increases quickly from A2 to A10, where it abruptly falls. The combined grid and particle arrays are likely small enough that the entire simulation might have fit in the SMP's extremely large 96 MB of cache.

Typically, Nehalem-EX outperforms Istanbul by more than 50% despite having only a 26% higher peak performance and comparable STREAM bandwidth; thus, the 3× larger aggregate cache is presumably the deciding factor. Both machines significantly outperform the floating-point weak Niagara2. Interestingly Istanbul and Nehalem-EX outperform the Fermi GPU by a factor of two or more. As previously analyzed, this is likely a result of the CPUs having larger caches and faster atomic operations.

6.2. Particle Push Phase

Unlike charge deposition, the push phase is straightforward to parallelize, as loop iterations are independent. Although a mixed-precision version is technically possible, the primary motivation for its examination (atomic increments being limited to 32 bits on GPUs) has been eliminated. Nevertheless, due to the random nature of the gather operation, push phase performance can suffer significantly from a lack of temporal and spatial locality.

6.2.1. Impact of Optimizations

We explored a number of optimizations within push to reduce memory traffic, improve cache behavior, and improve load balancing. Figure 7 shows C20 push performance as a function of architecture and optimization. Clearly, baseline performance varies dramatically, with Nehalem-EX nearly 3× faster than Istanbul and Niagara, and about 4× faster than the Fermi. Further, we chose the CPU baseline versions to be without NUMA-aware allocation and process pinning, in order to quantify the impact of these optimizations. Adding proper NUMA-aware allocation for the particle and grid arrays and process pinning (*+Pinning+NUMA*) can dramatically improve CPU performance (3× on Istanbul and Nehalem-EX). The benefit on Niagara is less pronounced due to the combination of a compute-intensive kernel on a architecture with superfluous bandwidth.

Fusing the nominal gather and push loops (sub-phases) together (*+Loop fusion*) significantly reduces CPU memory traffic and alters the read:write ratio from 2.9 to 3.8. This results in up to a 2× improvement on Istanbul (write-allocations are expensive), and a moderate improvement on Nehalem-EX. The lack of significant Nehalem-EX improvement is likely due to two factors. First, the FBDIMM-like interface has a natural 2:1 read:write ratio and is tolerant of write-allocate. Second, the cache working set for the class C problem is smaller than EX's aggregate last-level cache. On the GPU, loop fusion is only beneficial when used in conjunction with low-level optimizations (*+Low level opt*) such as constant and shared memories for storage of intermediate variables, and detailed code tweaking to improve memory coalescing. Clearly, the inclusion of a generic cache has now obviated much of the value of a texture cache on the GPU.

Figure 7 also shows that there is a load imbalance in the push loop with a static equi-sized partitioning of the particle array. Using OpenMP's guided scheduling (*+OpenMP*), we mitigated the potential 20% loss in performance by 10-15%.

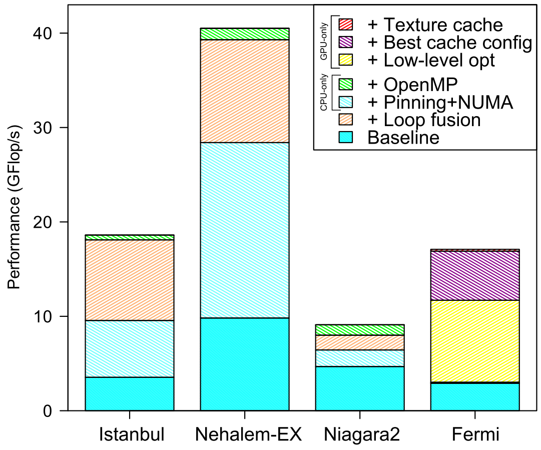


Figure 7: SMP/GPU Particle push performance for each platform and optimization on C20. Optimizations are cumulative. Note: On Fermi, the benefits of loop fusion (before Low-Level Optimizations) are not visible at this scale, but are critical for Low-Level Optimizations.

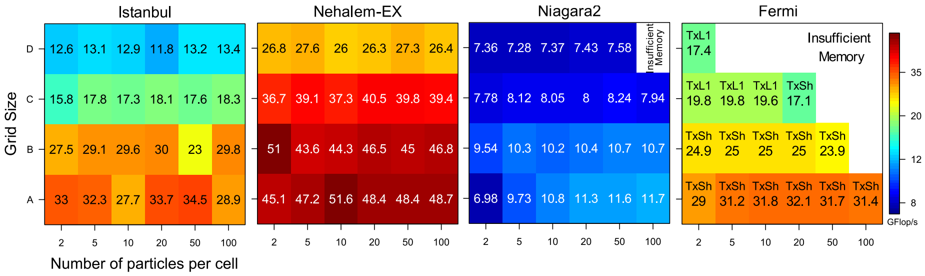


Figure 8: SMP/GPU Push performance in GFlop/s (450 flop/s per particle per iteration) achieved by the best implementation on each system for varying grid sizes and particle densities. An execution time of 1 second corresponds to 5.4 GFlop/s for the C20 problem.

6.2.2. Perturbations to Problem Configuration

Like the charge deposition phase, we examine and analyze performance relative to grid size and density in order to understand the resilience and broad applicability of our optimizations. Figure 8 shows the best attained performance over our entire set of implementations as a function of problem configurations. Once again, the vertical axis is a measure of cache working set size for the last level cache, and the horizontal axis is a measure of temporal locality. Again, some problem configurations are so large that the GPU lacks sufficient DRAM to run them.

Istanbul and Nehalem-EX can see sizable variations in performance if the grid falls out of cache (B to C on Istanbul, and C to D on EX). Unlike the previous generation of GPUs, we now see the cache-based GPU performance trends are now in line with CPU behavior. Increased cache working set sizes result in more cache misses leading to decreased performance. Moreover, the relatively small caches (14×48KB L1 + 768KB L2) can capture some temporal and spatial locality, but performance only improves slightly with increased problem temporal locality. This indicates that the GPUs cannot fully exploit the particle grid's temporal locality, as the L2 cache is likely polluted by the large data volume incurred due to streaming particle data. Loads that bypass the cache may show promise on all architectures in the future. It is important to note that on GPUs, the performance trends for push are opposite of what we saw on charge. We believe this is attributable to the throughput of atomic operations and their total elimination in the charge phase.

Observe that all CPU architectures (with their large caches) are relatively insensitive to changes in temporal locality. Recall that a particle density of two can result in each grid point being referenced 64 times. As expected, there is some benefit from increased locality, but ultimately, the streaming characteristics of this problem dominate. Niagara is likely compute-bound (push performs 450 flops per particle) for all problem configurations and thus shows little performance variation whatsoever.

Results demonstrate that the Nehalem-EX is often 50% faster than Istanbul on small problems and up to twice as fast on the large problems, due to its larger caches and higher compute rate. Typically, GPU performance is on par with Istanbul, but Nehalem-EX is easily 50% faster than Fermi. All machines deliver at least twice the performance of Niagara2.

6.3. Cross-Cutting Study: A Comparison to the Previous Generation

In this paper, we examine GTC performance on two of the newest x86 processors: AMD's quad-socket, hexa-core Istanbul (a proxy for Magny-Cours) and Intel's quad-socket, octal-core Nehalem-EX. Additionally, we explore performance optimization on NVIDIA's latest GPU (Fermi). A key question is the performance advantage achieved with these systems compared to the respective previous generations, as this allows extrapolation of these architectural paradigms into the future. To that end, Figure 9 shows the speedup of the newer system by kernel (top row is charge, bottom row is push) and by respective vendor (AMD on the left, Intel in the middle, NVIDIA on the right). Note that the "insufficient memory" cases arise from limited memory capacity on the Barcelona, Nehalem-EP, GT200, and/or Fermi. When those machines lacked sufficient memory capacity to run the particular problem configuration, a quantitative comparison was not possible.

The Istanbul SMP has three times the cores, roughly 2.5× the DRAM bandwidth, but nearly eight times the aggregate cache as its predecessor Barcelona. Observe that for charge deposition, despite its complex data-locality and data-dependency challenges, Istanbul lived up to its expectations and delivered approximately 3× the performance for virtually every problem configuration. We note a boost in performance for the larger problem sizes, but less than ideal scaling for the smaller problems (low density × small grids). The former is attributed to the vastly larger caches (working set fits in Istanbul's cache but not Barcelona's), but the latter is likely a more complex phenomenon resulting

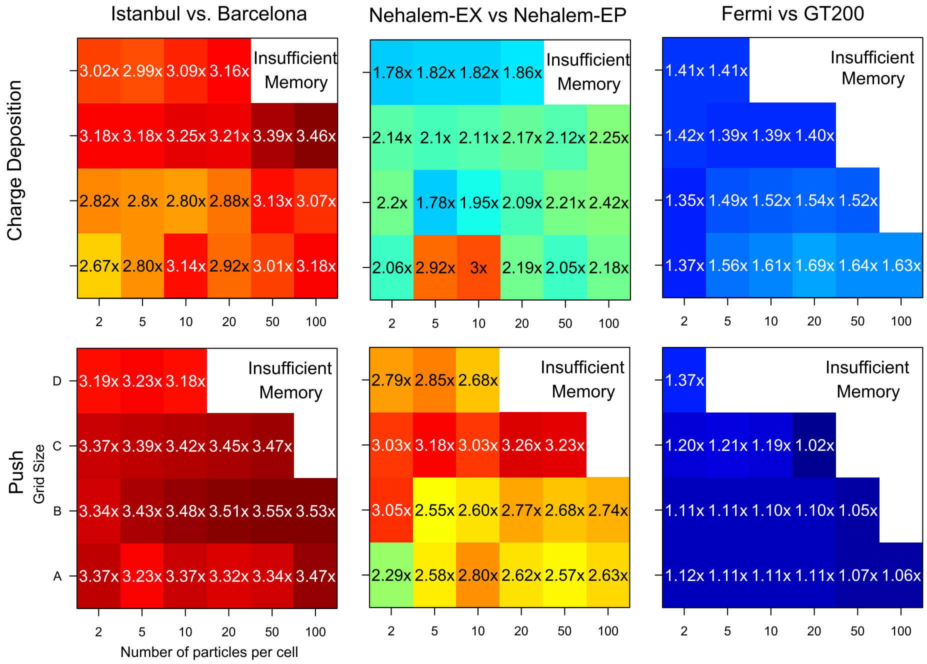


Figure 9: Performance ratio of Istanbul vs. Barcelona SMPs (left column), Nehalem-EX vs. Nehalem-EP SMPs (middle column), and Fermi vs GT200 GPUs (right column) for the best implementations of charge deposition (top row) and push (bottom row).

from limited parallelism and cache line thrashing across a larger (four NUMA node) SMP. As the particle push phase does not perform writes to the grid, the cache coherency protocol allows shared lines to reside in multiple caches without thrashing. Moreover, the higher arithmetic intensity tends to make the push phase more compute-bound. Thus, Istanbul's performance advantage on push is extremely smooth across all problem sizes.

Nehalem-EX machine has four times the cores (2× per chip, 2× the chips), 2.66× the raw DRAM pin bandwidth, and 6× the cache as the previous quad-core Nehalem-EP server. Nevertheless, on the charge deposition phase, we observe only a 2× typical speedup. Moreover, we see no discernible performance boost from the larger cache on larger problems. We believe the two outlier cases (A5, A10) arise from the entire simulation fitting in cache. For push, observe a pronounced performance spike for the class C grids; this is likely an artifact that smaller problems fit in both EP and EX's caches, larger grids fit in neither, but at this size, the working set fits in EX's cache, but not EP's. Aside from these cases, EX performance is less than 3× EP's. We believe that EX's paltry 6% improvement over EP in observed STREAM bandwidth (see Table 3), and possible performance inefficiencies due to a 64-way radial grid replication, likely inhibit any higher expected performance gains.

As advertised, Fermi has 6.6× the peak flops, and 40% higher pin bandwidth than the previous GT200 based GPUs. Nominally, this would suggest a huge speedup for the computationally intense push, and a moderate (40%) speedup on the charge deposition phase. Paradoxically, we see the opposite. Charge performance increases by about 1.5× while push performance increases by less than 20%. This can be attributed to a number of factors. First, unlike the GT200, Fermi may now cache the values gathered and scattered in charge. This can dramatically reduce the memory traffic and boost performance. Moreover, Fermi's atomic operations, although slow, are still faster than the GT200's. In the gather-heavy push phase, Fermi's use of a generic cache provides little value over GT200's use of a texture cache (although its increased size likely provides a boost similar to what we saw on the x86 machines). Additionally, measurements show that the ECC-enabled streaming bandwidth on Fermi is only approximately 8% higher than the GT200. Although push is more computationally-intense than charge, it is still bandwidth-bound. Thus Fermi's performance gains are simply due to its bandwidth advantage.

6.4. Cross-Cutting Study: Combined Charge/Push Performance

GTC simulations often spend over 80% of their total execution time in the charge and push phases. However, the relative time spent in each kernel will vary from one architecture to the next. Figure 10 shows the time spent in the baseline implementations of push and charge deposition as a function of architecture (lower is better). Note that in many cases, the time spent in push is greater than the time spent in charge. Before optimization, on Nehalem-EX, charge consumes the majority of the time, where on Fermi, the push dominates.

As described in detail, the benefits of optimization clearly vary between kernels and architecture — the overall speedup (charge+push) is shown for each platform in Figure 10. On Niagara, observe a moderate benefits to charge, but negligible improvements to push. Clearly, efficient exploitation of 128-way parallelism comes easily in push, but requires substantial optimization in charge. Observe that all architectures see a substantial benefits of optimization of charge (efficient management of data dependencies is not something that comes easily in a reference implementation). Conversely, we see that the embarrassingly parallel push phase saw varied speedups among architectures. Nehalem-EX and Niagara achieves little benefit, while Istanbul attains a moderate speedup and the GPUs see dramatic improvements. On the GPUs, much of the benefit came from low-level optimizations, and tuning for the appropriate balance between shared memory and L1 cache. Overall results show that optimization led to more than a 2× combined performance improvement on the x86 systems, while almost quadrupling GPU combined performance.

Before optimization, Nehalem-EX was clearly the fastest machine, followed closely behind by Nehalem-EP and Istanbul, and then the slower GPU systems. After optimization, Nehalem-EX maintains its expected performance lead (more flops, more bandwidth, more cache), with Istanbul now a distant second, while the Fermi GPU delivers optimized performance no better than the older Nehalem-EP. Although GPUs promise superior bandwidth and floating-point

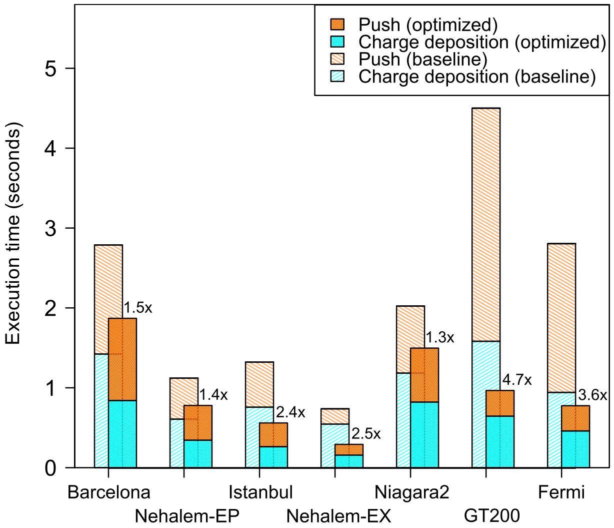


Figure 10: Execution times (for one time-step) of the baseline and best-performing double precision parallel versions of charge deposition and push kernels for C20 (numbers indicate overall optimization speedup) on a variety of multicore SMPs and GPUs. The CPU charge deposition baseline implementation uses a shared grid with locks, while the GPU baseline uses a shared grid with AtomicCAS for synchronization. The CPU baseline implementations include NUMA-aware memory allocation.

potential, their small caches and slow atomics limit performance on challenging scientific applications like GTC.

7. Conclusions

In this work, we examined the optimization of GTC's two principal phases, charge deposition (a scatter-increment operation devoid of locality with inherent data hazards), and push (a numerically intensive gather operation challenged by data locality), across emerging multicore SMPs and the newest manycore GPUs. Result show that fine-grained synchronization via atomic operations can greatly enhance performance without wasting increasingly precious memory capacity. The commonly-proposed CUDA strategy built on geometric decomposition and parallel primitives like sorting and scanning are unfortunately inadequate for a codes of this nature. Moreover, the relatively small caches found on GPUs are ill-equiped to full capture the extremely large cache working sets. As such, the GPU's memory bandwidth is wasted satisfying capacity misses. On the large SMPs we see that some replication is necessary to ensure that synchronization is limited to fast intra-socket operations. In terms of data locality, results show that caches are an essential ingredient. User management of random and unpredictable data locality in the presence of writes makes charge deposition optimization extremely challenging.

Overall, we observe the Nehalem-EX delivered the best performance; however, in relation to the Nehalem-EP (and considering its raw floating-point and DRAM capabilites), Nehalem-EX underperformed. Istanbul (our proxy for a dual-socket Magny-Cours), despite having only half of EX's computational peak, delivered much better performance relative to its raw potential and its Barcelona predecessor. Even after extensive optimization (including using fixed-point), the GPU typically underperformed Istanbul, Nehalem-EX, and Nehalem-EP — a testament to the complexities of this kernel on an architecture optimized for algorithms with minimal data dependencies and trivially expressed temporal locality.

As we moved to larger multicore architectures, load balancing becomes an increasingly large impediment to performance. In this respect, CUDA's task model coupled with a massive expression of parallelism has a significant advantage over the CPU SPMD model. In the future, we plan to investigate replacing the CPU SPMD model with a task model in order to better load balance these extremely parallel machines. Moreover, we believe that even within one multicore socket, GTC must migrate towards a 2D decomposition of particles and grid to mitigate the complexities of locality, data dependencies, and load-balancing.

Interestingly, results show that continually recomputing some of the constants on the fly could actually improve performance. Although this may run contrary to conventional wisdom, for bandwidth-bound applications with immense cache pressure and superfluous flop/s wasted as idle cycles, computing temporaries on the fly reduces cache pressure by exploiting the untapped floating-point capability of these machines.

We also observe that despite the growing popularity of accelerators and heterogeneity, results show that — even when all the data is kept local on a GPU — the overall performance (and even each phase) is slower than an Istanbul SMP. Thus, scientists must carefully weigh the potential advantages and costs before embarking on a heterogenous solution.

Finally our work points to the limitations of the manycore philosophy of simply instantiating as many cores as technology allows. Unlike many other domains, hardware/software co-design efforts for PIC codes must carefully balance bandwidth, compute, synchronization, and cache capacity. Failure to do so may result in implementations stalled for inter-thread synchronization operations or squandering their increasingly limited memory bandwidth with capacity misses.

Acknowledgments

All authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. Dr. Im was supported by Mid-career Researcher Program and by Basic Science Research Program through National Research Foundation of Korea (NRF) grant funded by the Ministry of Education, Science and Technology under contract numbers 2009-0083600 and 2010-0003044, and by research program 2010 of Kookmin University. Dr. Ethier was supported by the DOE Office of Fusion Energy Sciences under contract number DE-AC02-09CH11466. Additional support comes from Microsoft (Award #024263) and Intel (Award #024894) funding, and by matching funding by U.C. Discovery (Award #DIG07-10227). Further support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems. We would like to express our gratitude to Intel and Sun for their hardware donations. Access to the Istanbul and GPU resources were made possible through the DOE/ASCR Computer Science Research Testbeds program and NERSC.

References

[1] C. Birdsall, A. Langdon, Plasma Physics Via Computer Simulation, McGraw Hill Higher Education, 1984.

[2] R. Hockney, J. Eastwood, Computer simulation using particles, Taylor & Francis, Inc., Bristol, PA, USA, 1988.

[3] C. Birdsall, Particle-in-cell charged-particle simulations, plus Monte Carlo collisions with neutral atoms, PIC-MCC, IEEE Transactions on Plasma Science 19 (2) (1991) 65–85.

[4] Z. Lin, T. Hahm, W. Lee, W. Tang, R. White, Turbulent transport reduction by zonal flows: Massively parallel simulations, Science 281 (5384) (1998) 1835–1837.

[5] S. Ethier, W. Tang, Z. Lin, Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms, Journal of Physics: Conference Series 16 (2005) 1–15.

[6] S. Ethier, W. Tang, R. Walkup, L. Oliker, Large-scale gyrokinetic particle simulation of microturbulence in magnetically confined fusion plasmas, IBM Journal of Research and Development 52 (1-2) (2008) 105–115.

[7] K. Madduri, S. Williams, S. Ethier, L. Oliker, J. Shalf, E. Strohmaier, K. Yelick, Memory-efficient optimization of gyrokinetic particle-to-grid interpolation for multicore processors, in: Proc. ACM/IEEE Conf. on Supercomputing (SC 2009), 2009, pp. 48:1–48:12.

[8] L. Oliker, A. Canning, J. Carter, J. Shalf, S. Ethier, Scientific computations on modern parallel vector systems, in: Proc. 2004 ACM/IEEE Conf. on Supercomputing, IEEE Computer Society, Pittsburgh, PA, 2004, p. 10.

[9] M. Adams, S. Ethier, N. Wichmann, Performance of particle in cell methods on highly concurrent computational architectures, Journal of Physics: Conference Series 78 (2007) 012001 (10pp).

[10] W. Lee, Gyrokinetic particle simulation model, Journal of Computational Physics 72 (1) (1987) 243–269.

[11] E. Bertschinger, J. Gelb, Cosmological N-body simulations, Computers in Physics 5 (1991) 164 – 175.

[12] K. Bowers, B. Albright, B. Bergen, L. Yin, K. Barker, D. Kerbyson, 0.374 Pflop/s trillion-particle kinetic modeling of laser plasma interaction on Roadrunner, in: Proc. 2008 ACM/IEEE Conf. on Supercomputing, IEEE Press, Austin, TX, 2008, pp. 1–11.

[13] R. Fonseca et al., OSIRIS: A three-dimensional, fully relativistic particle in cell code for modeling plasma based accelerators, in: Proc. Int'l. Conference on Computational Science (ICCS '02), 2002, pp. 342–351.

[14] V. K. Decyk, UPIC: A framework for massively parallel particle-in-cell codes, Computer Physics Communications 177 (1-2) (2007) 95–97.

[15] C. Nieter, J. Cary, VORPAL: a versatile plasma simulation code, Journal of Computational Physics 196 (2) (2004) 448–473.

[16] C. Huang et al., QUICKPIC: A highly efficient particle-in-cell code for modeling wakefield acceleration in plasmas, Journal of Computational Physics 217 (2) (2006) 658–679.

[17] K. Bowers, Accelerating a particle-in-cell simulation using a hybrid counting sort, Journal of Computational Physics 173 (2) (2001) 393–411.

[18] G. Marin, G. Jin, J. Mellor-Crummey, Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code, Journal of Physics: Conference Series 125 (2008) 012087 (6pp).

[19] E. Carmona, L. Chandler, On parallel PIC versatility and the structure of parallel PIC approaches, Concurrency: Practice and Experience 9 (12) (1998) 1377–1405.

[20] H. Nakashima, Y. Miyake, H. Usui, Y. Omura, OhHelp: a scalable domain-decomposing dynamic load balancing for particle-in-cell simulations, in: Proc. 23rd International Conference on Supercomputing (ICS '09), 2009, pp. 90–99.

[21] A. Koniges et al., Application acceleration on current and future Cray platforms, in: Proc. Cray User Group Meeting, 2009.

[22] E. Akarsu, K. Dincer, T. Haupt, G. Fox, Particle-in-cell simulation codes in High Performance Fortran, in: Proc. ACM/IEEE Conference on Supercomputing (SC'96), 1996, p. 38.

[23] S. Briguglio, B. M. G. Fogaccia, G. Vlad, Hierarchical MPI+OpenMP implementation of parallel PIC applications on clusters of Symmetric MultiProcessors, in: Proc. Recent Advances in Parallel Virtual Machine and Message Passing Interface (Euro PVM/MPI), 1996, pp. 180–187.

[24] G. Stantchev, W. Dorland, N. Gumerov, Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU, Journal of Parallel and Distributed Computing 68 (10) (2008) 1339–1349.

[25] V. Decyk, T. Singh, S. Friedman, Graphical processing unit-based particle-in-cell simulations, in: Proc. 10th International Computational Accelerator Physics Conference, 2009.

[26] JET, the Joint European Torus, http://www.jet.efda.org/jet/, last accessed Nov 2010.

[27] AMD server platforms, http://www.amd.com/us/products/server/processors/Pages/server-processors.aspx, last accessed Nov 2010.

[28] Intel microarchitecture, http://www.intel.com/technology/architecture-silicon/microarchitecture.htm, last accessed Nov 2010.

[29] Sun SPARC Enterprise T-Series, http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/index.html, last accessed Nov 2010.

[30] NVIDIA Fermi Architecture, http://www.nvidia.com/object/fermi_architecture.html, last accessed Nov 2010.

[31] NVIDIA Quadro FX 5800, http://www.nvidia.com/object/product_quadro_fx_5800_us.html, last accessed Nov 2010.

[32] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: Proc. 2008 ACM/IEEE Conf. on Supercomputing (SC 2008), 2008, pp. 1–12.

[33] D. Merrill, A. Grimshaw, Revisiting sorting for GPGPU stream architectures, Tech. Rep. CS2010-03, University of Virginia (2010).

[34] Thrust: A template library for CUDA applications, http://code.google.com/p/thrust/, last accessed Nov 2010.