Analysis and Optimization of Gyrokinetic Toroidal Simulations on Homogenous and Heterogenous Platforms

Khaled Z. Ibrahim[1], Kamesh Madduri[2], Samuel Williams[1], Bei Wang[3] , Stephane Ethier[4], Leonid Oliker[1]

[1]*CRD, Lawrence Berkeley National Laboratory, Berkeley, USA*

[2] *The Pennsylvania State University, University Park, PA, USA*

[3] *Princeton Institute of Computational Science and Engineering, Princeton University, Princeton, NJ, USA*

[4] *Princeton Plasma Physics Laboratory, Princeton, NJ, USA*

{ *KZIbrahim, SWWilliams, LOliker* }*@lbl.gov, Madduri@cse.psu.edu, beiwang@princeton.edu, ethier@pppl.gov*

Abstract—The Gyrokinetic Toroidal Code (GTC) uses the particle-in-cell method to efficiently simulate plasma microturbulence. This work presents novel analysis and optimization techniques to enhance the performance of GTC on large-scale machines. We introduce cell access analysis to better manage locality vs. synchronization tradeoffs on CPU and GPU-based architectures. Our optimized hybrid parallel implementation of GTC uses MPI, OpenMP, and nVidia CUDA, achieves up to a $2\times$ speedup over the reference Fortran version on multiple parallel systems, and scales efficiently to tens of thousands of cores.

1 Introduction

Continuing the scaling and increasing the efficiency of high performance machines are exacerbating the complexity and diversity of computing node architectures. Leveraging the power of these technologies requires tuning legacy codes to these emerging architectures. This tuning requires revisiting the constraints imposed by legacy designs and prior assumptions about computational costs. It also requires deep understanding of the algorithmic properties of the studied code and the behavior of the underlying architectures with different computational patterns.

Our work explores analysis and optimizations across a variety of architectural designs for a fusion simulation application called Gyrokinetic Toroidal Code (GTC). Fusion, the power source of the stars, has been the focus of active research since the early 1950s. While progress has been impressive — especially for magnetic confinement devices called tokamaks — the design of a practical power plant remains an outstanding challenge. A key topic of current interest is microturbulence, which is believed to be responsible for the experimentally-observed leakage of energy and particles out of the hot plasma core. GTC's objective is to simulate global influence of microturbulence on particle and energy confinement. GTC utilizes the gyrokinetic Particle-in-Cell (PIC) formalism [19] for modeling the plasma interactions, and is orders-of-magnitude faster than full force simulations. However, achieving high parallel and architectural efficiency is extremely challenging due (in part) to complex tradeoffs of managing runtime evolving locality and data hazards, and low computational intensity of the GTC subroutines.

In this study, we present novel analyses of cell access on both CPU and GPU-based architectures and correlate that with the observed performance. We also discuss how these analyses can guide the data layout and the parallelization strategies to better manage locality and to reduce data hazards. We quantify the architectural interactions with key GTC access patterns on multiple architectures through microbenchmarking.

Our tuning efforts involve the full GTC code for the multi- and manycore era, leveraging both homogeneous and heterogeneous computing resources. A key contribution is the exploration of novel optimizations across multiple programming models, including comparisons between flat MPI and hybrid MPI/OpenMP programming models. To validate the impact of our work, we explore six parallel platforms: IBM BlueGene/P, IBM BlueGene/Q, Cray's AMD Magny-Cours-based XE6, Cray's Intel Sandy Bridge-based XC30, as well as manycore clusters based on nVidia Fermi and Kepler GPUs. Our optimization schemes include a broad range of techniques including particle and grid decompositions designed to improve locality and multi-node scalability, particle binning for improved locality and load balance, GPU acceleration of key subroutines, and memory-centric optimizations to improve single-node performance and memory utilization. We additionally show how the choice of optimization strategy is greatly influenced not only by the target architecture, but also the generation of the architecture. For instance, particle sorting on GPUs is better

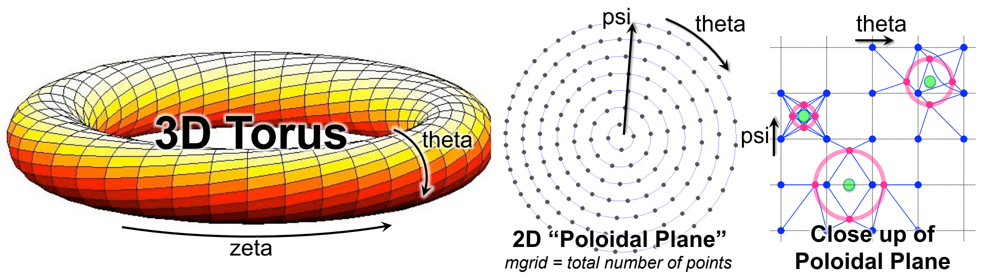

Figure 1: An illustration of GTC's 3D toroidal grid and the "four-point gyrokinetic averaging" scheme employed in the charge deposition and push steps.

not adopted on nVidia Fermi due to the high cost of atomic conflicts, but the same technique proves beneficial on the latest-generation Kepler architecture. Overall results demonstrate that our approach outperforms the highly-tuned reference Fortran implementation by up to 1.6× using 49,152 cores of the Magny-Cours "Hopper" system, while reducing memory requirements by 6.5 TB (a 10% saving). Additionally, our detailed analysis provides insight into the tradeoffs of parallel scalability, memory utilization, and programmability on emerging computing platforms.

The rest of this paper is organized as follows: Section 2 briefly overviews GTC and related work. In Section 3, we present experimental setup and the programming models used. We analyze the grid access patterns and their interactions with different architectures in Section 4. Our optimization strategies are discussed in details in Section 5. Section 6 covers performance results and analysis, and we conclude in Section 7.

2 GTC Overview and Related Work

The six-dimensional Vlasov-Maxwell system of equations [20] govern the behavior of particles in a plasma. The "gyrokinetic formulation" [19] removes high-frequency particle motion effects that are not important to turbulent transport, reducing the kinetic equation to a five-dimensional space. In this scheme, the helical motion of a charged particle in a magnetic field is approximated with a charged ring that is subject to electric and magnetic fields. The Particle-in-Cell (PIC) method is a popular approach to solve the resulting system of equations, and GTC [10, 11, 20] is an efficient and highly-parallel implementation of this method.

The system geometry simulated in GTC is a torus with an externally-imposed magnetic field (see Figure 1) characteristic of fusion tokamak devices. The self-consistent electrostatic field generated by the particles at each time step is calculated by using a grid discretization, depositing the charge of each particle on the grid, and solving Poisson's equation in Eulerian coordinates. The field value is then used to advance particles in time. GTC utilizes a highly specialized grid that follows the magnetic field lines as they twist around the torus. This permits use of a smaller number of toroidal planes in comparison to a regular grid that does not follow field lines, while retaining the same accuracy.

2.1 GTC Data Structures and Parallelization

GTC principally operates using two data representations: those necessary for performing grid-related work, and those that encapsulate particle coordinates in the toroidal space. Three coordinates (shown in Figure 1) describe particle 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 $mzetamax$, $mpsi$, and $mthetamax$. The average number of particles per grid cell is given by the parameter $micell$.

The charge density grid, representing the distribution of charge throughout the torus, and the electrostatic field grid, representing the variation in field strength (a Cartesian vector), are two key grid-related data structures in GTC. Note that the field grid, a vector quantity, requires three times the storage of the charge density grid. The highly-tuned and parallel Fortran implementation of GTC, which we will refer to as the reference version, uses multiple levels of parallel decomposition. A one-dimensional domain decomposition in the toroidal direction is employed to partition grid-related work among multiple processors in a distributed-memory system. GTC typically uses 64 to 128-way partitioning along the torus. In addition, particles within each of these toroidal domains can be distributed among multiple MPI processes for parallelizing particle-related work. A third level of parallelism is intra-node shared memory partitioning of both particle and grid-related work, and this is accomplished using OpenMP pragmas in GTC. The two-level particle and grid decomposition entails inter-node communication, as it is necessary to shift particles from one domain to another (due to the toroidal parallelization), and also obtain the globally-consistent states of grid data structures (due to the particle decomposition).

	Charge	Push	Shift
Flops	180 mi	450 mi	12 mi
Particle data (in bytes)			
read	40 mi	208 mi	$8\,mi + 100\,mi_s$
written	140 mi	72 mi	100 mi_s
Arithmetic intensity [a]			
(flops/byte)	**≤0.56**	**≤1.28**	**N/A**
Grid Data[b] (in bytes)			
read	≤320	≤768	0
written	≤256	0	0

Table 1: Characteristics of memory-intensive loops in particle-processing GTC kernels (charge deposition, push, and shift) for a single time-step in terms of total number of particles per MPI task (mi) or number of particles shifted (mi_s). [a]*Arithmetic intensity is computed as total flops divided by total data accessed, assuming a write allocate cache policy.* [b]*Grid data is per particle computation. Due to the high reuse, the total size of grid data is smaller than particle data, as shown in* Table 2.

Grid Size	B	C	D
$mzeta$	1	1	1
$mpsi$	192	384	768
$mthetamax$	1408	2816	5632
$mgrid$ (points per plane)	151161	602695	2406883
chargei (MB)[a]	2.31	9.20	36.72
evector (MB)[a]	6.92	27.59	110.18
Total Particles			
($micell = 20$)	3.02M	12.1M	48.1M
($micell = 96$)	14.5M	57.9M	231M

Table 2: The GTC experimental settings. $mzeta = 1$ implies each process operates on one poloidal plane. [a]*minimum per process.*

2.2 GTC Code Structure

The gyrokinetic PIC method involves five main operations for each time step: charge deposition from particles onto the grid using the four-point gyro-averaging scheme (***charge***), solving the gyrokinetic Poisson equation on the grid (***poisson***), computing the electric field on the grid (***field***), using the field vector and other derivatives to advance particles (***push***), and smoothing the charge density and potential vectors (***smooth***). In addition, distributed memory parallelization necessitates a sixth step, ***shift***, to move particles between processes or toroidal domains. In this paper, we assume a collisionless system and that the only charged particles are ions (adiabatic electrons).

The parallel runtime and efficiency of GTC depends on several factors, including the simulation settings (the number of particles per grid cell $micell$, the discretized grid dimensions, etc.), the levels of toroidal ($ntoroidal$) and particle decomposition ($npartdom$) employed, as well as the architectural features of the parallel system, described in previous GTC performance studies on parallel systems [1, 11, 28].

Our work builds on our well-optimized C implementation of GTC, and further improves upon it to address the challenges of HPC systems built from multi- and manycore processors. Thus, understanding and optimizing the following key routines on multi- and manycore architectures is imperative for achieving high performance.

Charge deposition: The charge deposition phase of GTC's PIC primarily operates on particle data, but involves the complex particle-grid interpolation step. Particles, represented by a four-point approximation for a charged ring, deposit charge onto the grid. This requires streaming through a large array of particles and updating locations in memory corresponding to the bounding boxes (eight grid points each) of the four points on the particle charge ring (see Figure 1). Each particle update may thus scatter increments to as many as 32 unique grid memory locations. In a shared memory environment, these increments must either be guarded with a synchronization mechanism to avoid read-after-write data hazards, or redirected to private copies of the charge grid. We further discuss in detail the locality vs. synchronization challenges of accessing the grid in Section 4.

As seen in Table 1, for each particle, the reference implementation of this step reads 40 bytes of particle data, performs 180 floating point operations (flops), and updates 140 bytes of particle data — resulting in a low arithmetic intensity of 0.56 flops per byte (assuming perfect grid locality and no cache bypass). In a distributed-memory parallelization, particles within a toroidal segment may be partitioned among multiple processes by setting $npartdom$ greater than 1. The effect is that each process maintains a private copy of the charge density grid onto which it deposits charge from the particle it owns. GTC merges these copies into a single copy using MPI "Allreduce" primitive.

Poisson/Field/Smooth: These three routines constitute purely grid-related work, where the floating-point operations and memory references scale with the number of poloidal grid points ($mgrid$). Simulations employ high particle densities ($micell \geq 100$), and so the time spent in grid-related work is typically substantially lower than the particle processing time (charge, push, and shift). Note that multi-node parallelism in these steps is limited to the extent of 1D domain decomposition (the $ntoroidal$ setting). When $npartdom$ is set to a value greater than 1, each process begins with the global copy of the charge/potential/field grid and executes the Poisson/field/smooth routine independently. The reference GTC code (the version we optimize) uses a custom iterative method [21] for solving the gyrokinetic Poisson equation, and is efficient for the case of adiabatic electrons. Other versions of GTC use solvers [27] available through the PETSc library.

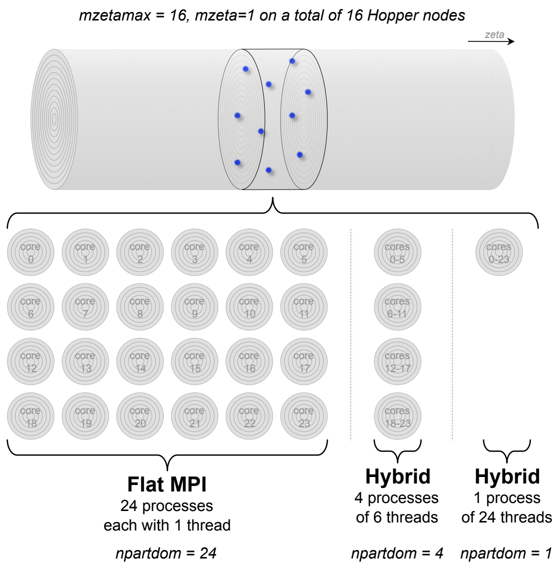

Figure 2: An illustration of the approaches used to exploit parallelism on Hopper, and the node-centric view of the replication of charge and field grids. Particles are partitioned, but never replicated.

Push: In this phase, the electric field values at the location of the particles are "gathered" and used for advancing them. This step also requires streaming through the particle arrays and reading irregular grid locations in memory (the electric field values) corresponding to the four bounding boxes of the four points on the ring. This can involve reading data from up to 16 unique memory locations. Additionally, as seen in Table 1, this kernel reads at least 208 bytes of particle data, performs 450 flops, and updates 72 bytes for every iteration of the particle loop. As this routine only performs a gather operation in a shared memory environment, it is devoid of data hazards. Moreover, since the arithmetic intensity of this kernel is higher than charge deposition's, it is somewhat less complex to optimize.

Shift: GTC's shift phase scans through the particle array looking for particles that have moved from the local domain (in the $\pm zeta$ directions). The selected particles are buffered, marking the resultant "holes" in the particle array where a particle has effectively been removed, and sending these buffers to the neighboring domains. Particles are moved only one step in $zeta$ at once (thus performing nearest-neighbor communication among processes along the toroidal direction), and so the shift phase iterates up to $mzetamax/2$ times. In the reference implementation, holes are filled sequentially as particles are received. However, such a sequential implementation can be inefficient in highly threaded environments (particularly GPUs). We discuss remediation strategies in Sections 5.1 and 5.2. Even when $npartdom$ is greater than one, processes still only communicate with their two spatially neighboring processes.

2.3 GTC Simulation Configurations

The most important parallel performance-related parameters describing a GTC simulation are the dimensions of the discretized toroidal grid and the average particle density. In all our initial experiments, we set $mzetamax$ to 16 and $mzeta$ to 1 (i.e., 16-way domain decomposition, with each process owning one poloidal plane). All scaling experiments in this paper are *strong scaling*.

In order to demonstrate the viability of our optimizations across a wide variety of potential simulations, we explore three different grid problem sizes, labeled *B, C, D*, and explore two particle densities: 20 and 96 particles per grid point. Therefore, a "B20" problem — often used in throughout this paper as it fits into all memory configurations — uses the class B grid size with an average particles per grid point count of 20 (assuming one process per node). Grid size C corresponds to the JET tokamak, the largest device currently in operation [16], and D to the forthcoming International Thermonuclear Experimental Reactor (ITER): a multi-billion dollar large-scale device intended to prove the viability of fusion as an energy source [15]. Table 2 lists these settings, which are similar to ones used in production runs [11]. Note, the grid memory requirements will scale with the number of processes per node. For the three examined GTC problem configurations, the maximum Larmor radius (a function of several GTC parameters) corresponds to roughly $mpsi/16$. The initial Larmor radii follow a uniform random distribution.

Figure 2 visualizes the confluence of programming model and simulation configurations on Hopper. As $mzetamax$

and $mzeta$ are 16 and 1 respectively, the torus is partitioned into 16 segments of 1 poloidal plane. To assign one of these segments to each 24-core Hopper node, we explore three approaches to parallelism within a node: one process per core, one process per chip, and one process per node. This is achieved by varying $npartdom$. Particles within this toroidal segment are simply partitioned among processes on a node. For instance, in a D96 simulation with $npartdom = 24$, the 231M particles on a node would be partitioned among the processes. However, this particle partitioning setting has two negative effects. First, the per-process particle density is only four particles per grid point, and second, the poloidal charge and field grids are replicated 24-way. Compared to the 24-thread hybrid implementation, the flat MPI implementation incurs a 24-way replication of charge grid data. All processes reduce their copies of the charge grid into one via the MPI Allreduce collective and solve Poisson's equation redundantly, so that each process has a copy of the resultant field grid. Moving to a hybrid implementation reduces the scale of the reduction, as well as the degree of redundancy in the solve. Conversely, the particle shift phase is done on a process-by-process basis. Shift is simplest in the flat MPI implementation and becomes progressively more complex and less concurrent in the hybrid version.

2.4 Related Work

PIC is a representative method from the larger class of *particle-mesh* methods. In addition to plasma physics (e.g., GTC's gyrokinetic PIC), particle-mesh methods find applications in astrophysics [3, 13], computational chemistry, fluid mechanics, and biology. There are also several popular frameworks from diverse areas that express PIC computations such as VPIC [5], OSIRIS [12], UPIC [8], VORPAL [26], and QuickPIC [14].

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 [4] and Marin et al. [24] 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 [7], and OhHelp [25] is a library for dynamic rebalancing of particles in large parallel PIC simulations. Koniges et al. [18] report performance improvements by overlapping computation with inter-processor communication for gyrokinetic PIC codes. The performance of the reference GTC MPI implementation has been previously well-studied on several large-scale parallel systems [1,11,28]. Prior research also examines expressing PIC computations via different programming models [2,6].

There has also been recent work on new multicore algorithms and optimizations for different PIC steps. Stanchev et al. [30] investigate GPU-centric optimization of particle-to-grid interpolation in PIC simulations with rectilinear meshes. Decyk et al. [9] present new parameterized GPU-specific data structures for a 2D electrostatic PIC code extracted from the UPIC framework. In our prior work on multicore optimizations for GTC, we introduced various grid decomposition and synchronization strategies [22, 23] that lead to a significant reduction in the overall memory footprint in comparison to the prior MPI-based GTC implementation, for the charge and push routines. This paper extends our prior work by developing an integrated CPU- and GPU-optimized version that accelerates a fully-parallel and optimized GTC simulation at scale.

3 Experimental Setup

We list the evaluated systems and their raw performance in Table 3. "Intrepid," a BlueGene/P system that is optimized for power-efficient supercomputing, uses the PowerPC 450d processors with dual-issue, in-order, embedded cores (four). "Vesta," a BlueGene/Q machine, uses the latest-generation 64-bit PowerPC A2 processor. The A2 has larger number of cores (16 cores each with 4 threads), higher frequency, larger memory capacity, and an integrated NIC on-chip. The third system, "Hopper," is a Cray XE6 massively parallel processing (MPP) system, built from Magny-Cours Opteron with two dual hex-core chips, with strong NUMA properties. Hopper and Intrepid have a 3D torus interconnect, while Vesta has a 5D torus. Cray XC30, "Edison," is based on dual 8-core Intel Sandy Bridge architecture, with two NUMA domains per node, and has a dragonfly interconnect. The Dirac cluster at NERSC is a small GPU testbed cluster, and is built from 50 dual-socket, quad-core 2.4 GHz Xeon X5530 compute nodes. Dirac is a testbed for GPU technology. Currently, one Tesla C2050 (Fermi) GPU is installed on each Dirac node. Each C2050 includes 448 scalar "CUDA cores" running at 1.15 GHz and grouped into fourteen SIMT-based streaming multiprocessors (SM). For our experiments, we use the term "Fermi Cluster" to refer to heterogeneous CPU/GPU simulations. Finally, we evaluate the performance of Titan, the top machine in the top 500 list [31] as of Nov. 2012. Each node has 16-core AMD Opteron 6274 processor, accelerated by an nVidia Tesla K20x GPU. Tesla K20x (Kepler) has lower frequency compared with Fermi but larger L2 cache and more cores per multiprocessor (32 and 192 cores for Fermi and Kepler, respectively).

Core Arch	**IBM PPC450d**	**IBM A2**	**AMD Opteron**	**Intel SNBe**	**nVidia Fermi**	**nVidia Kepler**
Type	dual-issue in-order SIMD	dual-issue in-order SIMD	superscalar out-of-order SIMD	superscalar out-of-order SIMD	dual warp in-order SIMT	quad warp in-order SIMT
Clock (GHz)	0.85	1.6	2.1	2.6	1.15	0.732
DP GFlop/s	3.4	12.8	8.4	20.8	73.6[a]	171
$/core (KB)	32	16	64+512	32+256	48	48
Memory-Parallelism	HW Prefetch	HW Prefetch, SMT	HW Prefetch	HW Prefetch, SMT	Multi-threading	Multi-threading
Node Arch	**Blue-Gene/P**	**Blue-Gene/Q**	**Opteron 6172**	**Xeon E5 2670**	**Tesla C2050**	**Tesla K20x**
Cores×Chips	4×1	16×1	6×4	8×2	14[a]×1	14[a]×1
Last $/chip	8 MB	32 MB	6 MB	20 MB	768 KB	1.5 MB
STREAM[f]	8.3 GB/s	28 GB/s	49.4 GB/s	38 GB/s	78.2 GB/s	171
DP GFlop/s	13.6	204.8	201.6	166.4	515	1310
Memory	2 GB	16 GB	32 GB	32 GB	3 GB	6 GB
Power	31W[c]	80W[c]	455W[c]	338W[c]	390W[d]	439W[d]
System Arch	**Blue-Gene/P "Intrepid"**	**Blue-Gene/Q "Vesta"**	**Cray XE6 "Hopper"**	**Cray XC30 "Edison"**	**Dirac Testbed "Fermi Cluster"**	**Cray XK7 "Titan"**
Affinity	N/A	N/A	`aprun`	`KMP_AFFINITY`[e]	`KMP_AFFINITY`[e]	`aprun`
Compiler	XL/C	XL/C	GNU C	Intel C	Intel C + nvcc	GNU C + nvcc
Interconnect	custom 3D Torus	custom 5D torus	Gemini 3D Torus	Aries Dragonfly	InfiniBand Fat Tree	Gemini 3D Torus

Table 3: **Overview of Evaluated Supercomputing Platforms**. [a]*Each shared multiprocessor (SM for Fermi or SMX for Kepler) is one "core."* [b]*Two threads per core. Power based on Top500 [31] data*[c] *and empirical measurements*[d]*. The GPUs alone are 214W and 235W for Fermi and Kepler, respectively.* [e]*Affinity only used for full threaded experiments.* [f]*STREAM copy.*

3.1 Programming Models

Our study explores three popular parallel programming paradigms: Flat MPI, MPI/OpenMP, and MPI/OpenMP/-CUDA. MPI has been the de facto programming model for distributed memory applications, while the hybrid MPI/-OpenMP models are emerging as a productive means of exploiting shared memory via threads. The hybrid nature of the GPU-accelerated nodes necessitates the use of a MPI/OpenMP/CUDA hybrid programming model. For all conducted experiments, we utilize every CPU core on a given compute node. Therefore, the number of threads per MPI process is inversely related to the number of MPI processes per node, so as to ensure full utilization of cores. Furthermore, we restrict our hybrid experiments to one MPI process per core (flat MPI), one process per chip (NUMA node), and one process per compute node. In the case of the Hopper XE6, these configurations represent 24 processes/node, 4 processes of 6 threads/node, and 1 process of 24 threads/node. GPU accelerated code requires one controlling process on the host side per GPU for earlier generations. This constraint is relaxed in newer generations, but we did not measure performance advantage for this relaxation for our workloads. The Dirac cluster, as well as Titan, contains a single GPU per node, and we choose to have a single process invoking the GPU accelerated code while OpenMP threads are used for host-side computations.

4 Grid Access Analysis

This section discusses cell (or grid) access characteristics and their interaction with the underlying hardware for both the CPU- and GPU-based architectures. As discussed earlier, the particle-in-cell method relies on aggregating the effects of many particles onto few cells, solving the field equations on these cells, and thus attaining orders of magnitude performance improvements. We focus on the access pattern for cells when they are concurrently accessed with particle data for two reasons. First, data access is shaped such that particles (the larger dataset) are streamed, thus creating a difficult-to-optimize irregular access to the grid data (smaller in size). Second, most parallelization strategies involve distributing particles between execution units (processes or threads) with no concurrent access to particle data. In contrast, grid points are inherently shared between execution units, even if we create replicas to temporarily alleviate the need for concurrent accesses.

Concurrent access to grid points can be beneficial during data gather phase (particle push) because it can lead to improved locality. Concurrent access is also a challenge in the scatter phase (charge deposition) and can lead to conflicting updates. Most of the optimizations in accessing the grid are geared towards either improving locality in accessing the grid, during the gather phase, or reducing conflicts during the scatter phase. Both objectives impose conflicting requirements in the data layout and the access ordering.

We can create particle access in any arbitrary order, thus creating many possibilities for the associated access pattern to the grid. While sorting to group particles that access the same or neighboring cells is reportedly an effective technique to improve performance [22, 29], we present experimental results showing that the benefits are greatly

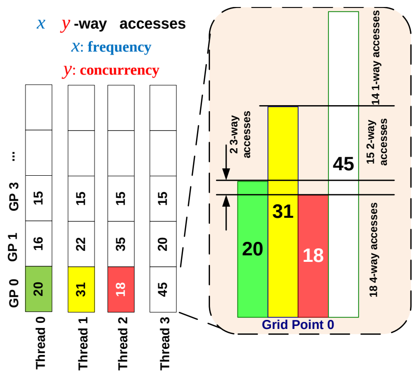

Figure 3: Profiling concurrent accesses for grid points due to thread concurrency.

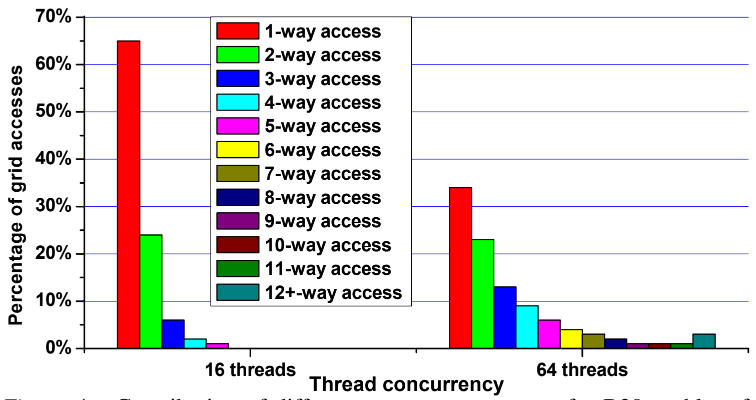

Figure 4: Contribution of different concurrent accesses for B20 problem for two thread concurrency levels 16 & 64.

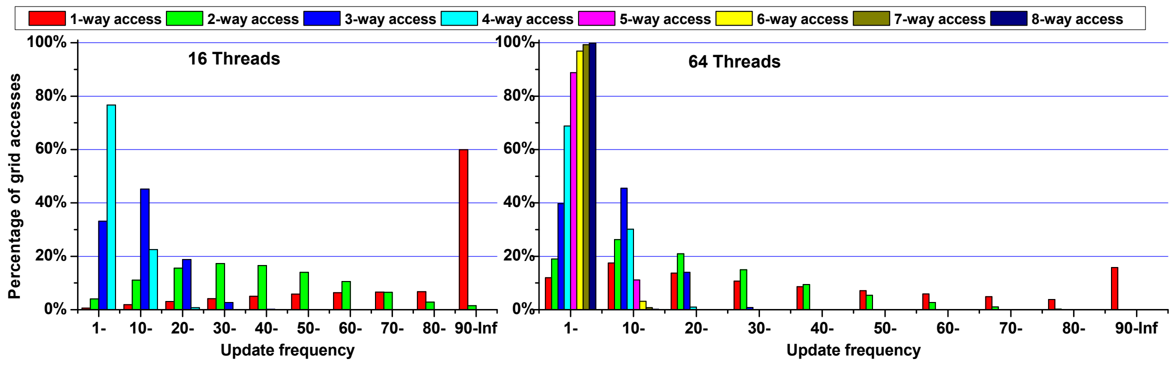

Figure 5: Percentage of memory accesses decomposed by access frequency and concurrency levels for B20 problem using two thread concurrency levels 16 & 64.

influenced by the underlying architecture.

4.1 CPU Grid Access Analysis

On CPU-based architectures, we decompose the grid concurrent access by *access frequency*, *i.e.* updates per iteration, and the level of *concurrent accesses*, *i.e.* sharing among threads. We present how the pattern of access changes with *thread concurrency* assigned to a poloidal plan.

To analyze grid accesses for access concurrency due the thread concurrency, we annotate the code to count the number of accesses to each grid point. We analyze with radial binning of particles in each iteration to reduce concurrent accesses, and accumulate the accesses for each thread in a separate data structure. As shown in Figure 3, accesses for each grid point is decomposed based on the access concurrency and the access frequency associated with this concurrency level. The larger the number of threads accessing a grid point, the higher the conflict in the scatter phase in accessing the charge grid, *charge*, and the wider the sharing of electric field in the gather phase, *push*. We show, later, the impact of access frequency and concurrency on the observed performance using microbenchmarking.

As shown in Figure 4, the percentage of grid accesses without concurrency (1-way access) decrease from 65% to 34% when we increase the thread concurrency level from 16 to 64. We also observe higher percentage of access concurrency of grid points with the increase of thread concurrency, even though we do radial binning of particles after each iteration. For instance, 4-way concurrent access increases from 2% to about 9% when increasing the thread concurrency from 16 to 64. These results show that particle binning is less effective in reducing concurrent access in strong scaling experiments, where the same dataset is distributed between an increasing number of threads.

In Figure 5, we show the distribution of accesses for different concurrent accesses and access frequencies. We observe that highly concurrent accesses are typically associated with small access frequencies. We also observe that increasing thread concurrency leads to increasing accesses with large concurrency and frequency.

Of critical importance to performance is when access concurrency leads to conflicts. Therefore, we created a microbenchmark, listed in Appendix A, to assess the performance for atomics under multiple access concurrencies and frequencies. As shown in Figure 6, the impact of conflicts due to concurrent accesses on performance is more

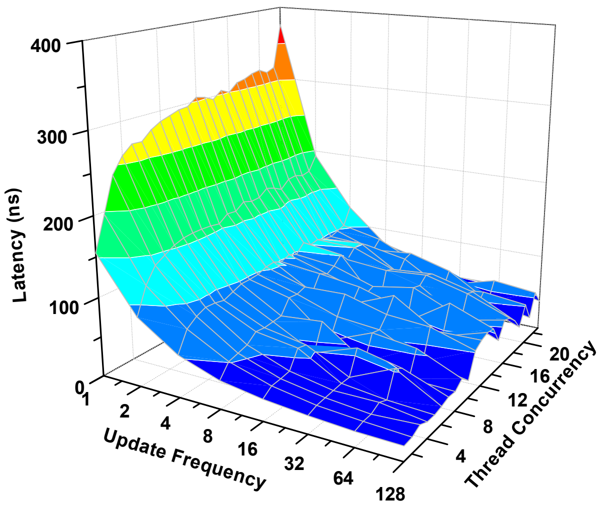

Figure 6: Atomic microbenchmark performance for different thread concurrencies and access frequencies on a Hopper node.

severe when the access frequency is small. Noting that the microbenchmark measures the lower bound of latency of atomics, large frequency can behave similar to small cluster depending on the access interleaving at runtime.

Generally, 24-way concurrency on Hopper cluster can increase the latency of atomic by at most 2.4× compared with no concurrent access. This low impact is due to having multiple cores sharing the same socket, thus making migration less expensive between cores. Having large access clusters, even under high concurrency provide a good chance of committing multiple updates locally before data migration, thus reducing the lower bound on latency by 7×.

Figure 6 shows the high cost of conflicts if we have a single grid, while the decay of concurrent accesses to the grid point, Figure 4, suggests that having fully replicated grid per thread is too wasteful for the memory. Another observation is that grid points with a large update frequency and a large concurrency can lead to a large variability in the access cost (the difference between the high cost for frequent migration and the low cost if updates are clustered). This can lead to unintentional load imbalance. Balancing the above conflicting requirements lead to choosing grids with ghost zones to balance between locality and conflict avoidance while accessing the charge grid, as detailed in Section 5.1. Electric field access always benefits from particle binning because accesses does not involve dependency hazards (no updates).

4.2 GPU Grid Access Analysis

On GPUs, access concurrency to the grid between threads in a thread block is far more critical to performance than the concurrent accesses between multiprocessors, which cannot be easily controlled. As such, the grid analysis introduced in Section 4.1 does not greatly influence the performance on GPU architectures. GPUs have limited hardware structures to capture temporal locality (L1 $\leq$ 48K, L2 $\leq$(768KB for Fermi, 1.5MB for Kepler) shared by all multiprocessors), and in nVidia Fermi architecture conflicting atomics are executed in the shared L2 cache, making the data access equally distanced from all multiprocessors. On Fermi architectures, committing atomic updates to the L2 cache voids the improvement with the increase in update frequency that we observe for the CPUs, shown in Figure 6, because frequent updates do not bring the data closer to the atomic issuer. Additionally, inter-multiprocessor concurrent accesses to a grid point do not cause data migration, but can rather cause contention. Capturing the contention effect with the small L2 cache—with respect to the large dataset accessed by all the multiprocessors—is difficult for GTC.

On GPUs, we have an additional parallelization level, in which consecutive particles assigned to a multiprocessor (or a thread block) are split between multiprocessor threads. As discussed earlier, in Section 4, we can control the level of access conflict to the grid through particle sorting. To assess the GPU response to different sorting strategies, we created a microbenchmark that controls the level of conflict between threads in a thread block.

Using a *dispersion factor* parameter, the microbenchmark in Appendix B controls the amount of conflicts between threads in a block. We create patches of random accesses in a radius of 512×*dispersion factor*. As shown in Figure 7(left), for a dispersion factor less than 4, the performance of atomics for Fermi architecture improves with reducing dispersion by about 7.4×. Surprisingly, as we reduce the dispersion further, below 4, the performance de-

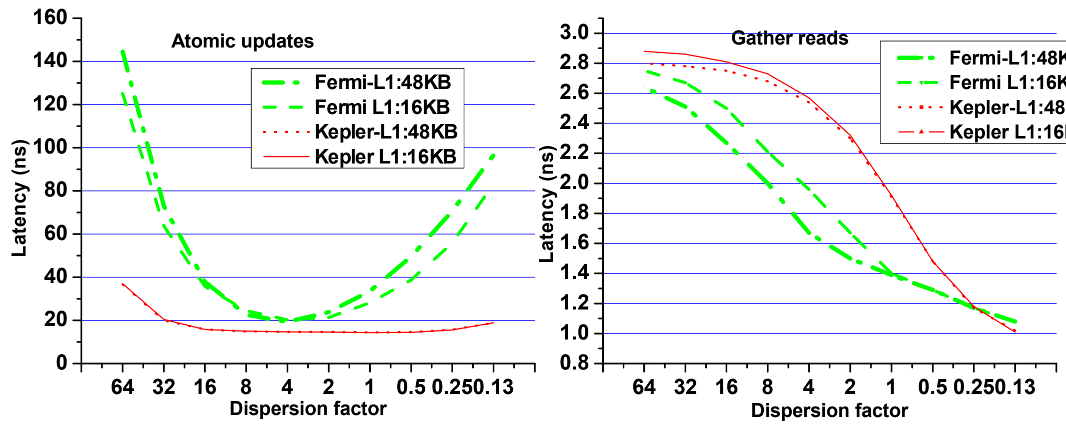

Figure 7: Left: atomic latency per multiprocessor for nVidia Fermi and Kepler GPUs for multiple conflict levels. Right: gather latency per multiprocessor for access patterns similar to atomic updates.

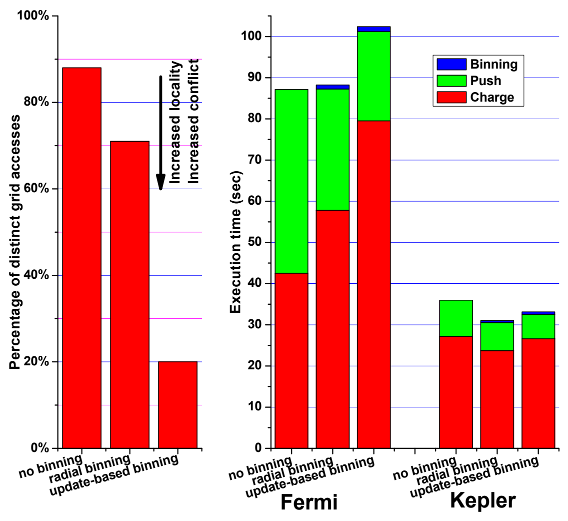

Figure 8: Impact of particle binning on the performance of GTC on Fermi GPUs.

teriorates by up to 5×. The Fermi performance for atomics with high locality is surprising but is also confirmed by nVidia [17]. Fortunately, this performance issue has been fixed on the latest generation nVidia Kepler Architecture. Not only does the performance improve for atomics, but the impact of conflicting updates is significantly reduced, as shown in Figure 7.

We additionally used the same microbenchmark access pattern to generate a gather operation, composed of reads based on the indirection array generated for the atomics. As shown in Figure 7(right), the latency per operation is reduced by 2.5× as we decrease dispersion, attesting for the improved locality and following what we intuitively expect in such experiments. For Kepler architecture, the read latency is higher due to the lower frequency of the multiprocessors. In both cases, scatter (through atomics) and gather, the choice of the L1 cache size does significantly affect the performance.

To show the impact of this GPU behavior on the GTC code, we present in Figure 8 two different binning strategies (affecting particle traversals) and we compare them with no binning. The first binning strategy is based on the radial position of the particles, similar to that used with the CPU analysis, while the second is based on the placement of the majority of updates to the grid. In GTC, a particle updates upto four different cells, based on the gyro radius. Because of the radial structure, these four cells are stored in 8 different memory segments (two per cell, one for the inner and the other for the outer ring). To create a single index for sorting based on the update locations, we divide the grid into 4KB regions. We then calculate which region of the grid receives most of the updates. The index of this region is used in sorting the particles. The second strategy is more superior in making the updates clustered to a small grid region.

As shown Figure 8 (left), the percentages of distinct memory locations in a frame of 2K updates (32KB) are 88%, 71%, 20% for no binning, radial binning, and update-based binning, respectively. These 2K updates correspond to what a 64-thread block commits in one iteration to all grid cells. The three binning points can roughly be associated with dispersion factors less than 4 in Figure 7 with sorting leading to a smaller dispersion and thus a reduced performance, for nVidia Fermi architecture. Binning improves the gather phase (*push*) of computation by up to 2.1x, while it degrades the scatter phase (*charge*) by 1.9x. Overall the performance with binning is reduced by up to 16%. For Kepler, sorting improves the overall performance due to the better support of Atomics. The best strategy for sorting is determined by the problem size. In the B20 problem shown in Figure 8, simple radial binning is the best for performance. For larger problems, the second sorting technique, based on update regions, delivers better performance.

Earlier for Fermi, we attempted to exploit clustering of updates based on binning by committing updates to the GPU shared memory before committing them to the GPU global memory. Unfortunately, this degrades the performance because of the overhead of managing the shared memory and the reduced occupancy of thread blocks sharing a multiprocessor. To sum up, binning, which is provably beneficial to CPUs, is associated with a large performance penalty for the charge phase on nVidia Fermi architecture. In contrast, Kepler carries an improved support for atomics making sorting a viable strategy.

5 Code Optimization

The reference optimized GTC code was developed in Fortran 90, and uses the MPI and OpenMP libraries for parallelism. We have rewritten the entire application in C for several reasons. First, our new implementation provides a common optimization substrate and simplifies GPU exploitation. We can also leverage optimized C/OpenMP multithreaded implementations from prior work, developed for the particle-mesh interpolation performed in charge and push [23]. The new C version permits exploration of several data layout alternatives, especially for the grid data. To maximize spatial locality on streaming accesses and thus improving sustained bandwidth, we chose the structure-of-arrays (SOA) layout for particle data, which necessitated moving away from the reference GTC representation (array-of-structures). This optimization is employed on both CPUs and GPUs, while particle binning strategy and grid replication is based on the underlying architecture, guided by the discussion in Section 4.

5.1 CPU Optimizations

We now discuss the details of the code restructuring as well as CPU specific optimizations for the different computation kernels. Common to all routines, we apply low-level restructuring such as flattening 2D and 3D grid arrays to 1D, zero-indexed arrays in C, pre-allocation of memory buffers that are utilized for temporary storage in every time step, aligned memory allocation of particle and grid arrays to facilitate SIMD intrinsics and aligned memory references, NUMA-aware memory allocation relying on the first-touch policy, and C implementations of Fortran 90 intrinsics such as modulo and cshift. Our work optimizes all six GTC phases on the CPU. On all kernels, we use OpenMP for thread-level parallelism. The key parallelization strategy and new contributions are discussed below.

Charge: Our previous single-node work [22, 23] explored optimizing GTC's charge deposition via multiple replication and synchronization strategies, exhaustively searching for an optimal performance. The analysis in Section 4 provides an empirical evidence that partial grid replication can provide minimal occurrence of expensive conflicts in accessing the charge grid, while not increasing the memory requirements. Our grid replication is based on creating a number of static replicas in addition to the original copy of the charge grid. Each replica can constitute either a full poloidal plane, or a small partition of it extended with a ghost zone in psi to account for the potentially-large radius of gyrating particle. Threads may quickly access their own replica without synchronization, or slowly access the globally shared version with a synchronization primitive. Thus, the size of the replica may be traded for increased performance.

Our optimized version leverages OpenMP to parallelize the particle loop, and using our previous results as guidance, selects the best replication and synchronization for each underlying platform. Note that as the BlueGene/P does not support 64-bit atomic operations, we employ the full poloidal grid replication strategy when running hybrid configurations (MPI/OpenMP). In case of the x86 systems, we have an additional cache-bypass optimization. As seen in Table 1 charge deposition involves substantial write traffic to particle data. Given that these unit stride writes are not accessed again until the push phase, we implement SSE intrinsics for streaming (cache-bypass) stores, boosting the computational intensity of the interpolation loop to 1.0.

Poisson/Field/Smooth: The grid-related routines are primarily comprised of regular accesses to grid data structures such as the charge density, potential, and the electric field. The main optimization for these routines is NUMA-aware allocation of both temporary and persistent grid data structures, in order to maximize read bandwidth for grid accesses. Additionally, we fuse OpenMP loops wherever possible to minimize parallel thread fork-join overhead.

Push: Similar to charge,we use a threaded implementation of push. The key optimizations performed in this study is loop fusion to improve the arithmetic intensity and NUMA-aware allocation. We also improve load balancing via OpenMP's guided loop scheduling scheme. This increases performance for larger cache working sets seen by particles near the outer edge, with a reduced number of loop iterations assigned to those threads. Load balancing is thus far more efficient than what is possible with a flat MPI reference approach.

Shift: Recall that shift is an iterative process in which particles move one step at a time around the torus. We optimize the shift phase by threading the MPI buffer-packing and unpacking, as well as via the use of additional buffer space at the end of the particle arrays. In practice each thread enumerates a list of particles that must be moved, buffer space then is allocated, threads fill in disjoint partitions of the buffer, and the send is initiated. When particles are received, they may be filled into the "holes" emptied by departing particles. However, performing hole-filling every step can unnecessarily increase overhead. Therefore, particle arrays are allocated with extra elements, and incoming particles are copied in bulk into this extra space. If the buffer is exhausted, holes are filled with particles at the end of the particle arrays. The hole filling frequency can thus be a runtime parameter, that we tune based on the particle movement rate.

Particle binning: For efficient parallel execution of charge and push, and for reducing the cache working set size of grid accesses, we make the important assumption that particles are approximately radially-binned. This is however not likely to be satisfied as the simulation progresses, as particles may move in the radial direction as well as across toroidal domains (causing holes in the particle array). Thus, we implement a new multithreaded radial binning routine, as an extension to the shift routine. Radial binning frequency is be a parameter that can be set dynamically, and is currently based on the number of toroidal domains and the expected hole creation frequency.

On-the-fly auxiliary array computations: GTC requires twelve double precision values per particle ($96mi$ bytes) to maintain the coordinates and state of each particle. This also represents the data volume per particle, exchanged in the shift routine. In addition, seven auxiliary arrays of size mi words each ($44mi$ bytes total) are maintained just to facilitate exchange of data from the disjoint charge and push phases of the simulation. The auxiliary information, accounting for particle data write traffic in charge and read traffic in push, can be avoided if we redundantly perform approximately 120 floating point operations per particle in the push phase. While we implemented this optimization, it did not lead to an overall improvement in performance for the problem configurations and parallel systems tested in this paper. This may however be an important optimization for future memory- and bandwidth-constrained systems.

5.2 GPU Acceleration

To investigate the potential of manycore acceleration in a cluster environment, we employ GPUs based on Fermi and Kepler architectures. A judiciously selected subset of the computational phases were ported to the GPU. As particle arrays can constitute the bulk of memory usage, our implementation strives to keep particles on the GPU to maximize performance. In practice, this constrains the size of possible problem configurations. We leverage our previous research into tuning the charge and push phases for GPUs [22]; however, significant additional effort was required to implement and optimize the GPU shift functionality.

Our implementation strategy requires that most auxiliary arrays are kept in the GPU memory to communicate data between GTC's computational phases. For instance, the indices of the electrostatic field that are computed during the charge phase are stored for use by the push phase. Additionally certain memory-intensive optimizations, such as keeping additional particle buffer space for shifted particles, were not suitable with the limited GPU memory capacity.

Charge: Similar to the OpenMP version, CUDA parallelizes the iterations of particle arrays among thread blocks with special attention paid to attain memory coalescing. The thread blocks then perform the requisite scatter-add operations to a charge grid resident in the GPU's device memory. However, unlike the CPU implementations of charge deposition, there is no grid replication on the GPU. Rather, the GPU version relies solely on fast double-precision atomic increment (implemented via a compare-and-swap operation). Unfortunately, the scatter nature of this kernel makes memory coalescing of grid data all but impossible. Nevertheless, we alleviate this performance impediment via a transpose within shared memory. Thus, computation is organized so that CUDA threads access successive particles when reading, but is reorganized so that a thread block works together on one charge deposition cluster when writing. Given the limited use of shared memory, we configure the GPU to favor the L1 cache.

We also highlight that the charge phase is also partly executed on the CPU, thus exploring architectural heterogeneity. Specifically, the communication involved in reduction of the charge density across domains and subsequent boundary condition updates are managed by the CPU. The solve routines utilize the charge density grid and produce the electorostatic field vector used in the push phase.

Poisson/Field/Smooth: The three solve-related phases are performed on the CPU as they mandate communicating

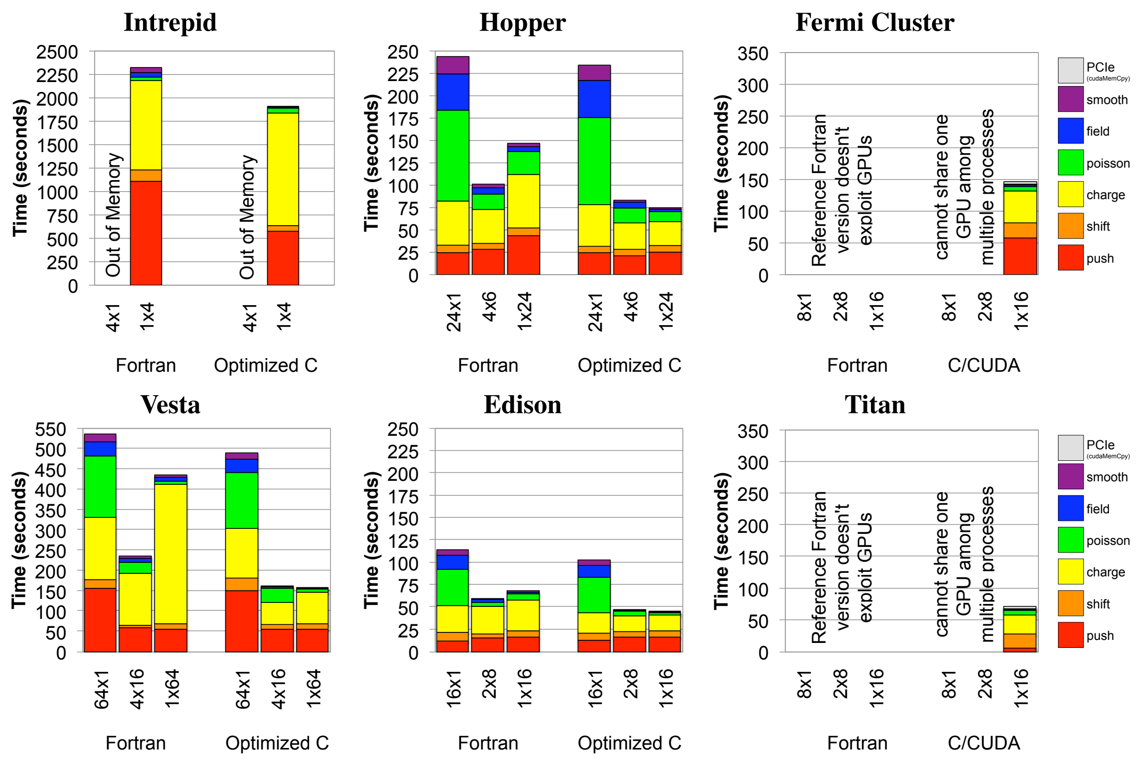

Figure 9: Breakdown of runtime spent in each phase for the B20 problem as a function of architecture, optimization, and threading. Note, minor axis is "processes×threads", or equivalently, "`npartdom`×`OMP_NUM_THREADS`". Additionally, observe each is plotted on a different timescale. PCIe time has been removed from each kernel's time and tabulated in a separate bar.

the computed values with other nodes via MPI. We predict that migrating these computations to the GPU will likely not substantially improve Fermi Cluster performance, as their fraction of the runtime is small (details in Section 6).

Push: For the push phase, we leverage our extensive GPU optimizations [22]. Unfortunately, the transpose operation that facilitated charge showed no benefit here. Our analysis indicates that a simple decomposition of one particle per CUDA thread and 64 threads per block performed well. Additionally, conditional statements were replaced with redundant computation to avoid divergent code, and some variables were judiciously placed in CUDA's "constant" memory. We also observe that the per-SM SRAM should be configured to prefer the L1 cache (i.e., 16 KB shared + 48 KB L1) for the larger grid size. Finally, prior to execution, push must transfer the field grid produced by the CPU solver from host memory to the GPU's device memory.

Shift: The shift phase is far more challenging to implement on the GPU. First, the GPU must enumerate a list of all particles currently residing in device memory whose toroidal coordinate is now outside of the GPU's domain and pack them into special buffers. Although a sequential implementation of such an operation is straightforward, a GPU implementation must express massive parallelism. To that end, each thread block maintains small shared buffers that are filled as it traverses its subset of the particle array. Particles are sorted into three buffers for left shift, right shift and keep buffer. Whenever the local buffer is exhausted, the thread block atomically reserve a space in a pre-allocated global buffer and copy data from the local buffer to the global one. Packed particles are also flushed over the original particle array, thus clustering particle holes towards the end of each thread block assignment. To efficiently implement this functionality we used low-level intrinsics such as ballot voting and bit-counting instructions.

The array of structure that benefited the charge and push routines proved problematic here. Reducing the number of memory transfers, from 24 to 2 in our case, required using the array of structures organization. Consequently data is transposed while flush to the global buffer. This organization also helps in facilitating the messaging mechanism. The global buffer is copied back to the host where the normal iterative shift algorithm is executed. Upon completion, the host then transfers a list of incoming particles to the GPU, where unpacking involves filing clustered holes in the particle arrays and transposing the data back to the GPU structure of arrays layout.

Particle binning: The decision of whether to use particle binning is based on the target GPU generation. For Fermi, we showed in Section 4.2 that the performance degradation due to atomic conflicts in the charge deposition outweighs the performance improvement in the push phase from the improved locality. For Kepler, particle binning improves the performance, and hence better be adopted. The best strategy of binning is based on the dataset size: for small datasets simple radial binning is enough, while a more sophisticated particle binning based on update regions, discussed in Section 4.2, is better used for larger datasets.

6 Results and Analysis

In this section, we present the performance of GTC using 16 nodes on all platforms, where the underlying problem (simulation) remains constant. Our optimizations explore the balance between threads and processes via the $npartdom$ (particle decomposition) configuration parameter. Additionally we present large-scale simulations to understand strong-scaling behavior at high concurrency.

6.1 Optimization and Threading

To highlight the benefits of our optimizations as well as the trade-offs of using multiple threads per process, we examine the B20 problem in detail. Figure 9 shows the time spent in each phase for 200 time steps of the simulation, as a function of machine, optimization (major axis), and threading (minor axis). The three threading configurations per platform include the flat MP (one MPI process per core), MPI/OpenMP (one MPI process per chip), and MPI/OpenMP (one MPI process per compute node) — for both the reference Fortran GTC as well as our optimized C version. As Intrepid and Vesta have only one chip per node, both hybrid implementations represent identical configurations. Moreover, as all simulations use every core on a node, changes in threads per process are realized by decreasing $npartdom$.

Results demonstrate a dramatic increase in performance when threading is used. This benefit primarily arises from retasking cores from performing redundant solves (Poisson/field/smooth) to collaborating on one solve per node. Interestingly, it appears that the Fortran version on Hopper and Edison does not effectively exploit the NUMA nature of the compute nodes. For instance, when using 24 threads per process on Hopper, there is a marked degradation in Fortran performance in push, charge, and Poisson. Conversely, our optimized implementation delivers moderate improvements in charge (obviates the intra-node reduction) and solve (trading redundancy for collaboration), but a slight decrease in push performance. Finally, the benefit of threading on Intrepid is more basic. This problem size per node cannot be run using the flat MPI programming model as it runs out of memory. This highlights the importance of moving to a threaded model, allowing us to exploit shared memory and avoid redundant poloidal replicas in the solver routines. BlueGene/Q (Vesta) nodes have more memory thus can accommodate larger problem configurations, thus the need for threading shifts to larger problem sizes. We also note that on all platforms, our optimized implementation attains best performance with one process per node.

Figure 9 also relates the relative contribution to runtime for each GTC computational component for a given architecture and threading configuration. Results show that the threading and optimization choice can have a profound impact on the relative time in each routine. Generally, as the number of threads increase, there is an improvement in the solver routines (due to reduced redundancy) as well as the shift routine (particles need not be shifted if they remain on the node). However, in the reference Fortran implementation, threading has a minimal impact on charge or push performance. As such, the fraction of time spent in charge and push can become dramatic, exceeding 71% of the best runtime on the x86 machines and constituting a combined 93% and 85% of the runtime on Intrepid and Vesta repsectively.

Results also show that, contrary to conventional wisdom, PCIe overhead does not substantially impede GPU performance. On Fermi, the challenges of data locality and fine-grained synchronization result in slower GPU execution times compared with running on the CPU alone. Unfortunately, the extra GPU memory bandwidth cannot make up for its inability to form a large working set in its relatively small L2 cache. Migrating the solver computations to the GPU would have a negligible impact on performance as it constitutes a small fraction of the runtime.

The Fermi performance is significantly impacted by the low atomic performance. Comparing Figure 6 & Figure 7, we see that the best throughput of a GPU multiprocessor atomics (with all its SMT threads) is equivalent to the best single-core performance. Additionally, we had to choose a slow implementation of the push phase, which is not impacted by the atomics, for a better overall performance, as discussed in details in Section 4.2. Kepler [17] shows a much better overall performance, about 2.06$\times$ the Fermi performance. The charge routine improves by 1.63$\times$, while the push routine received a 11.5$\times$ improvement due to running with sorted particles in addition to the increase of Kepler throughput over Fermi. In fact Kepler delivers the best performance for the push phase over all other architectures. The shift routine is slowed down by 22% because of its dependency on shared memory. In our implementation of the shift phase, a small thread block (of 64 threads) uses the whole shared memory resources, thus

Kernel	BGP Intrepid	BGQ Vesta	XE6 Hopper	XC30 Edison	Fermi Cluster	XK7 Titan
push	1.93×	1.05×	1.12×	0.92×	0.95×	9.75×
shift	2.03×	0.63×	0.89×	0.66×	0.40×	0.41×
charge	0.79×	1.63×	1.43×	1.72×	1.73×	2.00×
poisson	0.70×	2.99×	1.53×	1.97×	2.52×	3.00×
field	3.52×	4.42×	2.43×	3.09×	3.10×	3.19×
smooth	9.09×	6.51×	2.76×	3.75×	13.8×	3.91×
overall speedup	**1.22×**	**1.50×**	**1.35×**	**1.77×**	**1.34×**	**2.12×**

Figure 10: Speedup of B20 problem by kernel (best threaded C vs. best threaded Fortran). Note, baseline for the GPU's is the best of Fortran implementations running on the node's CPUs.

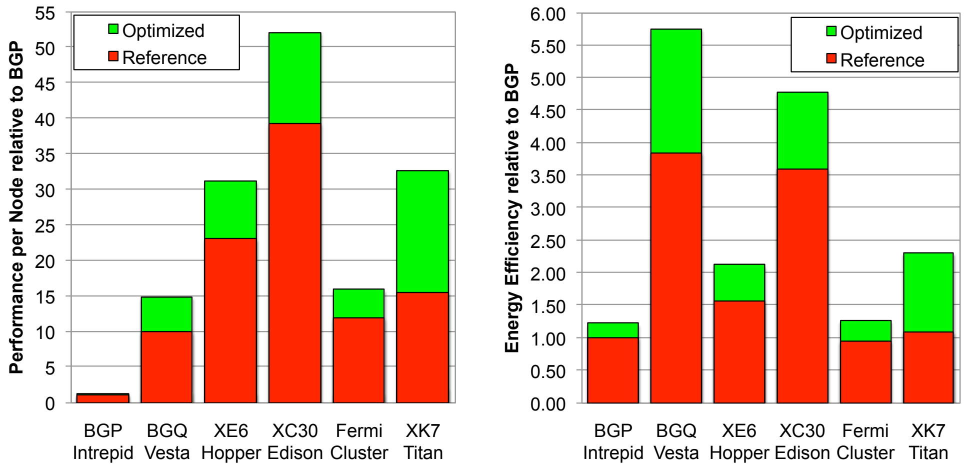

Figure 11: Performance and energy efficiency (normalized BGP = 1.0) before and after optimization for B20. Baseline for the GPU is fastest Fortran version on the CPU. Speedups through optimization are labeled. Power based on Top500 [31] and empirical measurements (see Table 3).

allowing reduced occupancy of the multiprocessor. Running one thread block per multiprocessor, with a small thread count, makes Kepler in a disadvantageous position compared with Fermi because Kepler has a lower frequency. The shift routine will be a subject of further tuning for Kepler architecture in our future work.

Table 10 details the B20 performance benefits of our optimized approach compared with the fastest reference Fortran implementation on a kernel-by-kernel basis. For our x86 optimizations, the complex charge routine delivers more than a 2× speedup due to our locality-aware, low-synchronization particle-grid interpolation approach. Unfortunately, the lack of 64-bit atomic operations prevents us from leveraging those techniques to improve Intrepid charge performance. The benefits observed in the three grid-related routines is even more dramatic, partially due to the attained peak performance with one process per node (thus all the on-node parallelism is tasked for one solve). Interestingly, BlueGene/P speedups vary wildly — a testament to the differences between XLF and XLC compilers, as well as the lack of ISA-specific BlueGene optimizations in our current implementation. Our shift routine also includes time for radial binning, which is performed every four time steps. Hence we do not achieve a substantial speedup on the x86 platforms. Fortunately, the BlueGene/Q (Vesta) A2 core supports 64-bit atomics that are executed in the L2-cache, in addition to supporting transactional memory. This new support allows our optimizations to improve the performance on the BlueGene line of architectures. We notice that the performance boost of Vesta compared with Intrepid is about 12.2×, the largest across all node architectures.

Finally, Table 10 shows that the GPU implementation benefited from speedups on the solver phases (as particles are not flushing the grid points from the caches. The significant improvement of the GPU performance with Kepler for GTC does not significantly reduce performance the gap with CPU architectures because of their continued performance improvement.

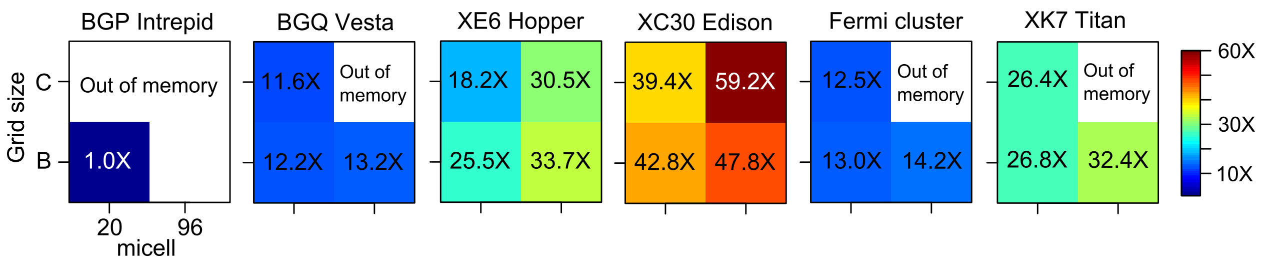

Figure 12: Heatmaps showing per-node performance (particles pushed per second per time step) as a function of system and problem configuration. All numbers are normalized to B20 Intrepid performance (for $mzeta = 16$).

6.2 Performance and Energy Comparison

Figure 11(left) summarizes our optimization impact (compared with the fastest reference Fortran implementation), as well as the relative per-node performance across all architectures (normalized to the BlueGene/P reference version). Results show the significant speedup of an Edison node compared with GPU-accelerated Titan, and low-power BlueGene/Q node, attaining a 1.6×, and 3.51× performance advantage (respectively). Overall, through our optimizations, we attain significant application-level speedups of 1.22×, 1.5×, 1.35×, 1.77×, 1.34×, and 2.12×on Intrepid, Vesta, Hopper, Edison, the Fermi Cluster, and Titan respectively, compared to the fastest Fortran version.

We now turn to Figure 11(right) that shows the energy efficiency trade-offs (derived from empirical measurements and the Top500 [31] data, see Table 3) between our evaluated architectures — an increasingly critical metric in supercomputing design. The energy efficiency of reference (or optimized) is calculated as the reference (or optimized) performance divided by the total node estimated power. The improvement of energy efficiency with optimization compared with the Fortran reference increased on newer generation platforms, Vesta, Edison, and Titan. BlueGene/Q improved node architecture delivers the the highest efficiency across all architectures, 5.7× the BlueGene/P efficiency. Titan, accelerated by nVidia Kepler, delivers 1.9× the performance efficiency of BlueGene/P. The Kepler improvement in efficiency is surpassed by other architectures because of the memory access attributes of GTC. Nevertheless, we believe that Kepler is a major step for accelerator-based architectures for applications with irregular access patterns, especially those relying on atomics.

6.3 Memory Savings

Computer architecture design is seeing memory capacity per node growing slower than peak performance. Thus, reducing memory usage is an important success. Using the threading paradigm (in either C or Fortran), reduces the processes per node, and thus the total number of field grid copies per node. These savings in memory scale with the number of cores and problem size. For example, the class B problem on Intrepid memory requirements are only reduced by 21 MB, while the D size configuration on Hopper results in significant saving of 2.5 GB. Additionally, in our optimized version, threads share one copy of the grid and partitions a second (unlike the reference Fortran, where each thread creates a redundant charge grid copy), resulting in an additional 800 MB of savings for the class D Hopper problem. Overall, our approach saves 330 MB, and about 3.3 GB for Intrepid, and Hopper respectively, corresponding to 16%, 4%, and 1% of each node's DRAM. These savings will have increasingly important impact on forthcoming platforms, which are expected to have higher core count and less memory per core.

6.4 Perturbations to Problem Configurations

Having examined B20 in detail, we now explore a range of configurations to understand our optimization impact across a wide range of problems. Figure 12 provides insight into how performance varies as a function of problem configuration and architecture. The vertical axis corresponds to the grid size. Whereas the class B grid is relatively small (2.3 MB per component) and could likely fit in cache, the class C grid is larger (9.2 MB per component) and will challenge the cache subsystems used on most CPUs. The horizontal axis of Figure 12, $micell$, denotes the average number of particles per grid point (particle density). Increasing $micell$ has two effects that are expected to increase performance. First, because the computational complexity of charge, push, and shift phases vary linearly with $micell$, the time spent in those routines increases relative to the time spent in the solver routines (whose computational complexity depends on $mzetamax$). Additionally, a larger $micell$ increases the temporal locality of accessing the grid.

Given that the total memory required for a simulation grows linearly with grid size and particle density, it is not surprising that at 16 nodes all but the B20 problem was too large for Intrepid and the C96 was too large for the Fermi,

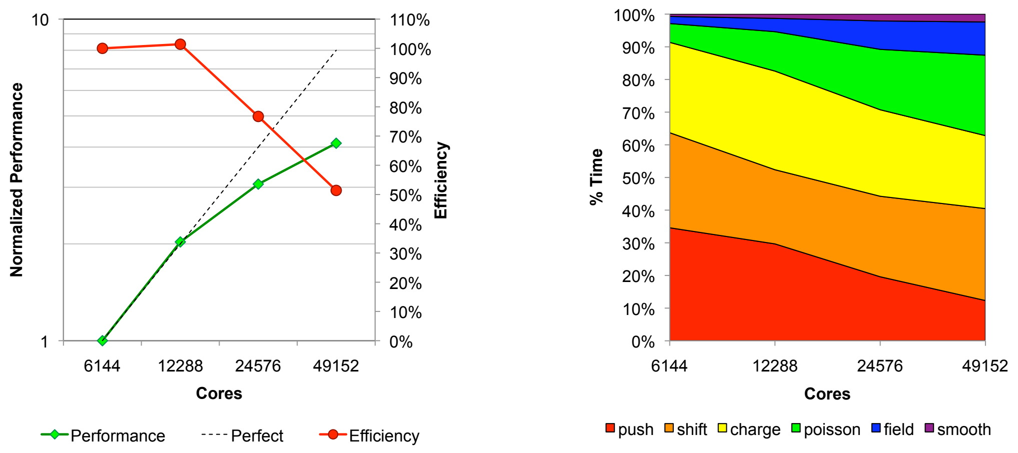

Figure 13: Strong-scaling results on Hopper using the D96 problem. (left) performance and efficiency as a function of concurrency, normalized to 6144 cores. (right) the breakdown of runtime among phases.

Kepler (Titan), and Vesta. The only way to run larger problems is to use multiple nodes per plane. Broadly speaking, on the x86 machines, increasing the grid size and the corresponding working set results in a performance degradation. Conversely, an increase in particle density (increased locality, decreased solve time), causes a performance improvement.

6.5 Large Scale runs

Ultimately, ultra-scale simulations must focus on large grid sizes with high particle densities. To understand the potential scaling impediments, we examine the ITER-scale D96 problem with $ntoroidal = 64$ on the Cray XE6 Hopper. Compared to the B20 problem in the previous subsection, the size of the grid per node is increased by a factor of 16×, while simultaneously the particle density is increased by almost a factor of 5×. Our experiments scale the number of nodes from 256 to 2048 (6144 to 49,152 cores) in a *strong scaling* regime by increasing $npartdom$ from 4 to 32. Thus, at 256 nodes, each node uses a field grid of 110 MB to push 58 million particles. All experiments leverage our fastest optimized hybrid implementation on Hopper.

Figure 13(left) shows that performance increases by a factor 4.1× (relative to the 6144-core reference point), attaining a 1.6× overall improvement compared with the fastest reference implementation on 49,152 cores. Note, however, that parallel and energy efficiency (red line) achieves only 51%, mostly due to the increasing fraction of time spent in the solver routines, clearly visible in Figure 13(right). In reality, the actual solve times remain almost constant, but the time spent in push, shift, and charge decrease by up to 11.5×. Mitigating the solver scaling impediment requires either parallelization of the 2D solvers across $npartdom$, or increasing the particle density proportionally with $npartdom$ via weak scaling simulations. Nonetheless, the unique characteristics of particle distributions within a poloidal plane allowed push to attain super-linear scaling, while shift and charge improved by 4.3 and 5.1× respectively.

7 Conclusions

In this work, we analyze a gyrokinetic fusion code and explore the impact of various multicore-specific optimizations. The global capability of the GTC code is unique in that it allows researchers to study the scaling of turbulence with the size of the tokamak and to systematically analyze important global dynamics. Our work explores novel multi- and manycore centric optimization schemes for this complex PIC-based code, including multi-level particle and grid decomposition to alleviate the effects of irregular data accesses and fine-grained hazards, increasing computational intensity through optimizations such as loop fusion, and optimizing multiple diverse numerical kernels in a large-scale application. Results across a diverse set of parallel systems demonstrate that our threading strategy is essential for reducing memory requirements, while simultaneously improving performance by 1.22×, 1.50×,1.35×, 1.77×, 1.34×, and 2.12× on BlueGene/P, BlueGene/Q, the Cray XE6, the Cray XC30, Fermi Cluster, and Titan, respectively.

The B20 problem provided insights into the scalability and performance optimization challenges (CPU and GPU) facing GTC simulations. For example, our memory-efficient optimizations on CPUs improved performance by up to 1.77× while reducing memory usage substantially. Proper use of threading placed simulations in the ideal regime where they are limited by the performance of on-node charge and push calculations.

Our highly-optimized GPU charge and push routines, we observe that GTC's data locality and fine-grained data synchronization challenges are at odds with the underlying Fermi architecture's synchronization granularity and small cache sizes—thus mitigating the GPU's potential. Kepler improved atomic performance significantly puts GPUs in a better position with other architectures for the GTC irregular access pattern, which is typically challenging on GPU streaming-based programming model. We also observe that Vesta and Edison deliver the highest overall energy efficiency compared with other studied architectures.

To evaluate our methodology at scale, we explored strong-scaling D96 experiments on Hopper using up to 49,152 cores, and showed that our optimization scheme achieves a 1.6$\times$ speedup over the fastest reference Fortran version, while reducing memory requirements by 6.5 TB (a 10% savings). Results show that in the strong scaling regime, the time spent in solver remains roughly constant, while charge/push/shift exhibit parallel scalability. Thus, the redundancy in the solver (performing the same calculation ($npartdom$ times) impedes scalability and efficiency, motivating exploration of solver parallelization schemes that will be the subject of future work.

Additionally, future work will address the limited DRAM capacity of next-generation system via radial partitioning techniques that lower memory requirements.

8 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. Ethier is supported by the DOE contract DE-AC02-09CH11466. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This research also used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Finally, this research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

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

[2] E. Akarsu, K. Dincer, T. Haupt, and G. Fox. Particle-in-cell simulation codes in High Performance Fortran. In *Proc. ACM/IEEE Conference on Supercomputing (SC'96)*, page 38, Nov. 1996.

[3] E. Bertschinger and J. Gelb. Cosmological N-body simulations. *Computers in Physics*, 5:164–175, 1991.

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

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

[6] S. Briguglio, B. M. G. Fogaccia, and 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)*, pages 180–187, Sep–Oct 1996.

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

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

[9] V. K. Decyk and T. V. Singh. Adaptable particle-in-cell algorithms for graphical processing units. *Computer Physics Communications*, 182(3):641–648, 2011.

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

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

[12] 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)*, pages 342–351, Apr. 2002.

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

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

[15] The ITER project. http://www.iter.org/.

[16] JET, the Joint European Torus. http://www.jet.efda.org/jet/, last accessed Apr 2011.

[17] NVIDIA's next generation CUDA compute architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, last accessed Apr 2013.

[18] A. Koniges et al. Application acceleration on current and future Cray platforms. In *Proc. Cray User Group Meeting*, May 2009.

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

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

[21] Z. Lin and W. Lee. Method for solving the gyrokinetic Poisson equation in general geometry. *Physical Review E*, 52(5):5646–5652, 1995.

[22] K. Madduri, E. J. Im, K. Ibrahim, S. Williams, S. Ethier, and L. Oliker. Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms. *Parallel Computing*, 37:501–520, Sept 2011.

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

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

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

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

[27] Y. Nishimura, Z. Lin, J. Lewandowski, and S. Ethier. A finite element Poisson solver for gyrokinetic particle simulations in a global field aligned mesh. *Journal of Computational Physics*, 214(2):657–671, 2006.

[28] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier. Scientific computations on modern parallel vector systems. In *Proc. 2004 ACM/IEEE Conf. on Supercomputing*, page 10, Pittsburgh, PA, Nov. 2004. IEEE Computer Society.

[29] C. Shon, H. Lee, and J. Lee. Method to increase the simulation speed of particle-in-cell (pic) code. *Computer Physics Communications*, 141:322329, December 2001.

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

[31] Top500 Supercomputer Sites. http://www.top500.org.

A Pseudo-code for benchmarking atomic performance on multicore systems

Performance of concurrent atomic accesses to a grid poloidal plane. This microbenchmark creates multiple access frequencies and thread concurrency levels for randomly generated memory addresses.

```
procedure CPU_ATOMIC_CONCURRENCY_TEST
    for $i = 0$ to $update\_count - 1$ do
        Compute a random $index[i]$ within the grid space
    end for
    for $u = 1$ to $max\_update\_frequency - 1$ do
        for $t = 1$ to $max\_threads$ do
            Start the timer
                                                        ▷ Each thread do the same
            for all $i = 1$ to $update\_count - 1$ do
                for $j = 0$ to $u$-1 do
                                                        ▷ Control of concurrency level
                    if $my\_thread < t$ then
                        AtomicUpdate($grid[index[i]]$, $update\_val$)
                    end if
                end for
            end for
            Record the elapsed time to $time[u][t]$
        end for
    end for
end procedure
```

B Pseudo-code for benchmarking atomics on GPU Architectures

This microbenchmark creates multiple levels of access conflict by threads sharing a thread block. A dispersion factor controls the radius of update around a focal point in the grid. A high dispersion factor leads to accesses that are less likely to be captured by the cache hierarchy. A small dispersion leads to increased conflicts and a better locality.

```
procedure CPU_DRIVER
    pick an update radius ($dispersion\_factor$ * 512 double)
    for $block$ = 0 to $multiprocessor\_count - 1$ do
        for $i = 0$ to $update\_count - 1$ do
            Pick a random grid point
                                                        ▷ Assume a Thread Block of 128 threads
            for $thr$=0 to 127 do
                for $cell$=0 to 3 do
                    Pick a random cell
                                                        ▷ 4 points per cell × 2 doubles per point
                    for $c = 0$ to 7 do
                        Calculate $index[block][i][cell$*8+$c][thr]$ randomly within the update radius
                    end for
                end for
            end for
        end for
    end for
    Allocate the $grid$ array on the GPU.
    Transfer the update array $index$ to the GPU
    Time the execution time of running GPU_KERNEL on the GPU
end procedure
procedure GPU_KERNEL
    Calculate the update index, $my\_index$, based on my block
    for $i = 0$ to $update\_count - 1$ do
        for $c = 0$ to 23 do
            AtomicUpdate($grid[my\_index[i][c][my\_thr]]$, $update\_val$ )
        end for
    end for
end procedure
```