J. Parallel Distrib. Comput. 68 (2008) 1339–1349

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU☆

George Stantchev[a,*], William Dorland[a], Nail Gumerov[b]

[a] Center for Scientific Computing and Mathematical Modeling, University of Maryland, United States
[b] Institute for Advanced Computer Studies, University of Maryland, United States

ARTICLE INFO

Article history:
Received 12 March 2008
Received in revised form
10 May 2008
Accepted 10 May 2008
Available online 29 June 2008

Keywords:
GPU computing
Scientific computing
Parallel algorithms
Numerical simulations
Particle-In-cell methods
Plasma physics

ABSTRACT

Particle-In-Cell (PIC) methods have been widely used for plasma physics simulations in the past three decades. To ensure an acceptable level of statistical accuracy relatively large numbers of particles are needed. State-of-the-art Graphics Processing Units (GPUs), with their high memory bandwidth, hundreds of SPMD processors, and half-a-teraflop performance potential, offer a viable alternative to distributed memory parallel computers for running medium-scale PIC plasma simulations on inexpensive commodity hardware. In this paper, we present an overview of a typical plasma PIC code and discuss its GPU implementation. In particular we focus on fast algorithms for the performance bottleneck operation of Particle-To-Grid interpolation.

1. Introduction

Numerical simulations of magnetized plasma (ionized gas) comprise an active fundamental area of research in modern computational physics. Plasma consists of charged particles whose individual trajectories and collective behavior produces effects on a wide range of temporal and spatial scales. Although the fundamental principles governing plasma dynamics, namely Maxwell's equations and statistical mechanics, are well known, obtaining their solution under realistic assumptions and relevant physical regimes is a problem of outstanding computational complexity.

Particle-In-Cell (PIC) methods represent one of several kinetic approaches to plasma simulation. These methods, based on hybrid Lagrangian–Eulerian formulation of dynamics, involve following the trajectories of virtual (marker) particles in the system phase space. A major challenge for PIC methods is the sheer magnitude of the number of particles needed to faithfully represent the underlying physics, a constraint that has historically pushed PIC implementations into the realm of distributed memory supercomputing, [10]. Several reduction techniques, most notably gyro-kinetics, have helped relax this constraint and have allowed moderately sized, physically meaningful simulations to be performed on smaller scale platforms.

PIC methods operate on the assumption that interactions among particles are carried out through electromagnetic fields. A rectilinear grid is typically used to obtain a discretized representation of the fields over the simulation domain. Solving the evolution equations of the fields is one of main components of a PIC code; however its execution time accounts for only 2%–15% of the simulation's total time [4]. The remaining time is spent on evolving particles and calculating the particle-field interactions, expressed computationally as local interpolation operations. These operations are the major performance bottleneck of PIC codes. Strategies to optimize PIC performance have been studied and implemented extensively for codes running on large scale distributed memory architectures. These fall mainly in two categories: particle decomposition schemes for optimal load balancing and particle sorting for improving CPU cache coherence (see for instance [2,22,16]).

Inspired by the latest trends and developments in graphics processing technology we investigate the feasibility of implementing PIC methods on the GPU. In particular we focus on Particle-To-Grid interpolation as it is expected to dominate the execution pipeline and to be the most challenging component in terms of performance optimization. While we restrict our attention to PIC simulations, we emphasize that the same type of local interpolation is common to many other application areas (see for instance [18]) where conversion between scattered-point and structured-grid data takes place.

☆ This work was supported by the US DOE Center for Multiscale Plasma Dynamics.
* Corresponding address: Center for Scientific Computing and Mathematical Modeling, University of Maryland, 4145 CSIC Bldg., 20742-3289 College Park, MD, United States.
E-mail address: gogo@umd.edu (G. Stantchev).

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.05.009

General Purpose GPU (GPGPU) computing was pioneered six years ago with the advent of the first programmable graphics processors. It has experienced a significant boost within the past year, after the introduction of NVIDIA's Compute Unified Device Architecture (CUDA) technology, which offered not only a faster, massively parallel hardware platform, but also enhanced programming environments and tools for scientific computation. Since CUDA's public release successful implementations from various application areas have emerged and have demonstrated the efficiency and the tremendous performance gain potential that the new architecture has to offer.

At the same time it has become apparent that certain classes of algorithms are better suited to perform well on the GPU than others. The specific memory hierarchy, while beneficial to some types of operations, poses challenges to others; for instance, those that rely on random memory access suffer performance losses that have the potential to eliminate entirely the benefits of GPU's high theoretical throughput. Also, algorithms with low arithmetic intensity relative to the number of memory operations will be, in general, less efficient.

For instance plasma simulations based on nonlinear MHD models have shown excellent optimization potential on the GPU, [19,5] due to their natural formulation exclusively in terms of matrix-vector operations. Particle simulations based on computational strategies other than PIC, such as molecular dynamics [20], and N-body gravitational dynamics [12], have also been implemented efficiently. In these simulations, particles interact among themselves via long range forces which are either calculated exactly by summing over all possible particle pairs or approximated via a suitable series truncation. The *apriori* higher arithmetic intensity of these methods predicates the large performance gain achieved in their implementation

Particle-To-Grid interpolation, on the other hand, performs only a few arithmetic operations per data point and, without modification, exhibits memory access patterns that are inherently random. However, even with these *apriori* deficits, Particle-To-Grid interpolation can still be implemented efficiently on the GPU with proper data rearrangement and algorithm adjustment. For instance, Sorensen et al. [18] have recently demonstrated very good speedup factors for their GPU implementation of the gridding step of the Non-uniform FFT (NFFT) algorithm, which mathematically is identical to Particle-To-Grid interpolation. In the context of NFFT, however, the set of scattered data points is static and thus a preprocessing data rearrangement step needs to be executed only once. In contrast, in PIC simulations particles change positions from one time step to the next, thus requiring a dynamic data structure optimized *both* for efficient updates and fast memory access.

This paper is organized as follows: in Section 2 we give and overview of the current trends of GPGPU computing. Section 3 outlines the general PIC simulation framework. Section 4 discusses the Particle-To-Grid interpolation operation and possible implementation strategies. In Section 5 we derive a fast parallel Particle-To-Grid interpolation algorithm tailored specifically for the GPU. In Section 6 we present some of the CUDA implementation details and discuss performance results.

2. Overview of GPU computing

Bolstered by significant increase in performance capabilities over the past six years and recent improvements in programmability, graphics processing units (GPU) are entering into the mainstream of modern computing. The current high-end GPU is a powerful parallel processor whose functionality is no longer confined to the traditional graphics pipeline. GPU's have sustained super-Moore's law rate of performance increase for almost a decade now. The primary driver of this phenomenal growth has been the highly parallel nature of graphics computing, as well as order-independence, pipelining, and streaming, all part of the graphics processing paradigm.

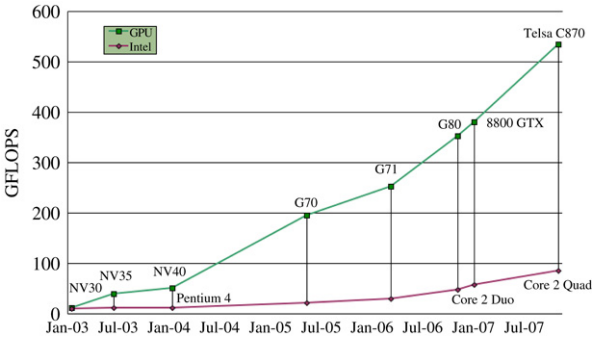

Fig. 1. Performance comparison between GPUs and CPUs measured in GFLOPS as peak throughput. Data points derived from graphs in [11,21,9].

In addition, high demand from the video gaming industry has helped maintain relatively constant GPU retail prices which has brought the performance cost of computing down to unprecedented levels. As a reference, the performance of the top NVIDIA G80 (GeForce 8800 Ultra) GPU is approximately 470 GFLOPS, which translates into about $1.40/GFLOPS; in contrast the performance of a 3.0 GHz Intel Core2 Duo CPU is close to 50 GFLOPS, or $4/GFLOPS [11]. Note that this improved performance is achievable only if the problem maps well to the underlying processor architecture.

The diagram in Fig. 1 shows the rapid rise in the floating-point performance of GPUs as compared to the multi-core Intel CPUs. GPU memory size is also growing, albeit more slowly, with current generation GPUs offering up to 1.5 GB of RAM. At present, GPUs support IEEE single-precision floating point arithmetic, but double precision support is expected on NVIDIA chips by the second half of 2008.

While all aforementioned features are undoubtedly attractive, one particular aspect of modern GPUs has proved crucial in enhancing their impact on scientific computing. The introduction of NVIDIA's Compute Unified Device Architecture (CUDA) has turned GPUs from esoteric special-purpose co-processors into highly accessible, programmer friendly, parallel supercomputers for the consumer-level desktop. As such, CUDA has the potential to "democratize" parallel computing and bring high-performance paradigms into the mainstream [9].

CUDA supports the single-program, multiple-data (SPMD) programming model, which is currently one of the dominant parallel processing paradigms. CUDA allows GPU programs to be written in ANSI C (with a few extensions), rather than graphics-oriented language, like Cg or GLSL, that were designed for shading algorithms. The current CUDA distribution provides BLAS and FFT libraries designed to take advantage of the hardware's new capabilities. In addition, various algorithm-oriented libraries are under development, most notably the CUDA Data Parallel Primitives (CUDPP) library [7].

Since CUDA's official introduction in early 2007, numerous scientific applications from areas as diverse as physics, numerical algebra, computational biology, and mathematical finance, have been implemented and tested on the new architecture. Speedup factors of up to several hundred have been noted in comparison with the respective serial CPU implementations [9]. Two main factors seem to have most effect on the acceleration potential for a given application: arithmetic intensity per memory operation ratio and the prevalence of random memory access patterns. In general, applications which perform many arithmetic operations between memory read/writes, and which minimize the number

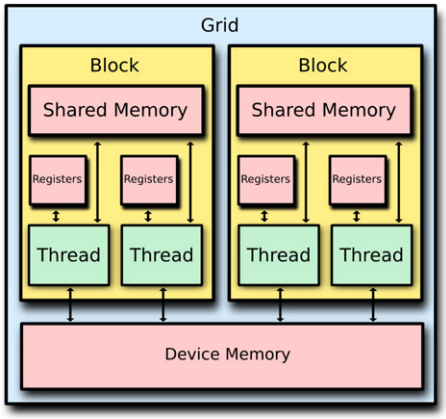

Fig. 2a. The G80 memory model.

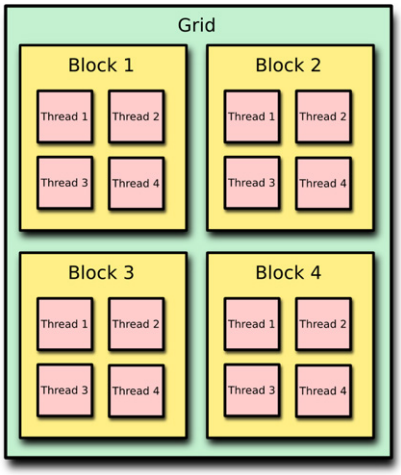

Fig. 2b. The G80 thread-block hierarchy.

of out-of-order memory accesses, tend to perform best. These two factors are not necessarily unique to CUDA but their impact seems to get disproportionately amplified by the architecture's specific hardware features.

We will not attempt to go any further into the details of CUDA's hardware and software models. Excellent introductions, reviews, and tutorials have been presented elsewhere in printed as well as in multimedia form (cf. [9,11,8,20,13,14]). However, for the sake of completeness we include a simplified version of the two basic diagrams that illustrate CUDA's execution patterns (Fig. 2b) and memory hierarchy (Fig. 2a) as presented in NVIDIA's CUDA Programming Guide, [11]. We also point out that while other GPU platforms, such as IBM's Cell and more recently AMD's FireStream, can be taken into consideration, our discussion is based specifically on CUDA. We believe this is not a major limitation as the main issues involved in parallel PIC simulations will persist among all current high-end programmable GPUs regardless of their underlying chip technology.

3. PIC method overview

Particle-In-Cell methods are a class of numerical simulation methods used to model physical systems whose behavior varies on a large range of spatial scales. On a macroscopic level the dynamics are typically described by a continuum model (a system of PDE's) whereas microscopically they are modeled by a collection of discrete particles. Dealing with each of these descriptions separately can be problematic; for instance the large scale nonlinear behavior of plasma is determined in part by

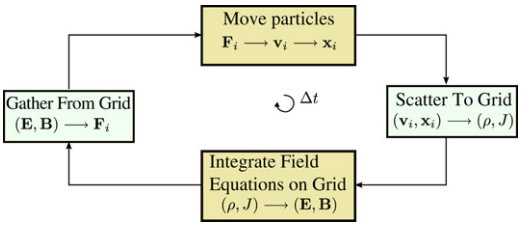

Fig. 3. Typical time step cycle of a PIC code. Particle related quantities are indexed by *i*. Field quantities are typically discretized on a uniform rectilinear grid; their indexing is not shown here.

Coulomb interactions between electrons and ions and thus cannot be fully described by a continuum model alone. On the other hand physically realistic simulations on a microscopic level require numbers of particles (on the order of 10^{23}) that are prohibitively large by any current computational standards. Particle-In-Cell (PIC) methods attempt to circumvent both of these issues and bridge the gap between the macroscopic dynamics of each particle and the macroscopic behavior of the system.

3.1. Main components of PIC codes

A PIC simulation is concerned with the time evolution of two types of objects: particles and fields. Particle trajectories are computed by integrating Newton's equations, usually written as a system of first-order ordinary differential equations. Fields are considered continuous quantities and are discretized on a (usually uniform, rectilinear) grid over the problem domain. In contrast with Molecular Dynamics or Gravitational N-Body systems, the effect of binary particle interactions is approximated via self-consistent particle-field interaction. Specifically, the motion of a single particle with position $\mathbf{x}$, velocity $\mathbf{v}$, and mass m is given by:

$$\begin{aligned} m\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} &= \mathbf{F} \\ \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} &= \mathbf{v} \end{aligned} \tag{1}$$

where $\mathbf{F}$ is the force acting on the particle. In a plasma the force in general has two components, corresponding to the electric and magnetic fields, respectively:

$$\mathbf{F} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}).$$

In this equation the fields are to be calculated at each particle position. Computationally, this requires interpolation from the fixed spatial grid of the fields to the scattered set of points at which particles are considered to reside within the given time step. We refer to this operation as *Grid-To-Particle Interpolation*.

Eq. (1) is solved using a numerical integration scheme, for instance the time-centered leapfrog method, [1]. At the end of each time iteration, the positions and velocities of all particles are updated, which induces an update to the particle density ρ and current $\mathbf{J}$, both evaluated at the grid vertices. Calculating ρ and $\mathbf{J}$ from the set of scattered particle positions onto the fixed spatial grid is carried out via *Particle-To-Grid* interpolation. From the values of ρ and $\mathbf{J}$ on the grid, the electric and magnetic fields are computed and evolved through the current time iteration step by numerically solving Maxwell's equations. The details of this step can vary significantly depending on the underlying physics, and we omit them as they lie outside the scope of this discussion. A schematic diagram of the fundamental steps involved in the PIC time iteration cycle is shown on Fig. 3.

As mentioned in Section 1, during a PIC simulation the most time is spent on the Grid-To-Particle and Particle-To-Grid interpolation steps. We focus on the latter since it proves to be

the more computationally challenging of the two; moreover the same strategy employed to interpolate quantities from particles to grid works almost verbatim in the opposite direction as these two operations are, in a sense, dual to each other.

4. The Particle-To-Grid interpolation operation

The Particle-To-Grid interpolation operation generalizes the familiar histogram approximation of the phase space probability density function onto a uniform rectilinear grid. Specifically, in dimension d, given a box shaped domain $D = [a_1, b_1] \times \cdots \times [a_d, b_d] \subset \mathbb{R}^d$, a set of points $\{p_1, p_2, \ldots, p_n\} \subset D$, and a uniform grid

$$G \stackrel{\text{def}}{=} \left\{ v_{\mathbf{s}} = \sum_{\lambda=1}^{\mathrm{d}} s_\lambda \mathbf{e}_\lambda \mid s_\lambda \in \mathbb{Z} \right\} \subset D$$

the value of the probability density function f at each vertex $v_{\mathbf{s}}$ is:

$$f(v_{\mathbf{s}}) = \sum_{i=1}^{n} w_i K(v_{\mathbf{s}}, p_i) \tag{2}$$

where $\mathbf{s} = \{s_1, s_2, \ldots, s_d\}$ is a multi-index, $\{\mathbf{e}_\lambda\}$ are orthonormal basis vectors in $\mathbb{R}^d$, $\{w_i\}$ are weight coefficients, and K is an interpolation kernel.

In most practical applications K is a local linear interpolation kernel of tensor product type, which in dimension one is exemplified by the familiar "hat" function; for instance a kernel with support $\operatorname{supp} K = (-1, 1)$ is defined as:

$$K(0, x) = \begin{cases} 1 - |x|, & \text{if } |x| < 1 \\ 0, & \text{otherwise.} \end{cases}$$

In dimension two, if K has support on the unit square,

$$K((0,0),(x,y)) = \begin{cases} (1-|x|)(1-|y|), & \text{if } |x| < 1, \text{ and } |y| < 1 \\ 0, & \text{otherwise} \end{cases}$$

is a quadratic function, whose graph is a piecewise hyperbolic sheet. This construction readily generalizes to higher dimensions.

The use of a local linear interpolation kernel of this type implies that, for each vertex $v_{\mathbf{s}}$, only particles contained in cells incident with $v_{\mathbf{s}}$ will have nontrivial contributions to the sum in Eq. (2):

$$f(v_{\mathbf{s}}) = \sum_{p_i \in \mathcal{P}(v_{\mathbf{s}})} w_i K(v_{\mathbf{s}}, p_i) \tag{3}$$

where $\mathcal{P}(v_{\mathbf{s}})$ is the collection of particles in the cells incident with vertex $v_{\mathbf{s}}$.

For a particle p_i let $\mathcal{V}(p_i) = \{v_{\mathbf{s}_1}, \ldots, v_{\mathbf{s}_q}\}$ denote the set of $q = 2^d$ vertices of the grid cell to which p_i belongs. The contribution of p_i to the value of f at each vertex $v_s \in \mathcal{V}(p_i)$ is precisely $w_i K(v_s, p_i)$. This expression has geometric interpretation as the Euclidean measure for the volume "opposite" to the vertex v_s with respect to p_i. For instance, in dimension $d = 2$, the contribution of a particle can be computed via the "opposite area" formula (see Fig. 4).

4.1. Computational strategies

The two maps defined above, particle-to-vertex: $p_i \mapsto \mathcal{V}(p_i)$, and vertex-to-particles: $v_{\mathbf{s}} \mapsto \mathcal{P}(v_{\mathbf{s}})$ are in some sense dual to each other; they motivate two complementary strategies for computing the particle density function f over the entire grid G.

4.1.1. Particle pull

In this case particles are "pulled" by the grid vertices. Pseudo code is given in Algorithm Listing (1)

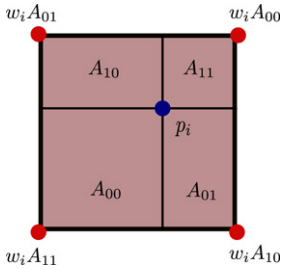

Fig. 4. Illustration of the "opposite area" formula. Each area A_{kl} multiplied by the weight coefficient w_i is the contribution of p_i to the particle density function at the vertex on the opposite diagonal.

```
// Loop over vertices first
foreach vertex v_s ∈ G do
    find P(v_s);
    f(v_s) ← 0 ;
    foreach p_i ∈ P(v_s) do
        f(v_s) ← f(v_s) + w_i K(v_s, p_i)
    end
end
```

Algorithm 1: Particle pull

```
// Initialize f(v_s)
foreach vertex v_s ∈ G do
    f(v_s) ← 0 ;
end
// Loop over particles first
foreach particle p_i ∈ D do
    find V(p_i);
    foreach v_s ∈ V(p_i) do
        f(v_s) ← f(v_s) + w_i K(v_s, p_i)
    end
end
```

Algorithm 2: Particle push

4.1.2. Particle push

In this case particles are "pushed" onto the grid vertices. Pseudo code is given in Algorithm Listing (2)

4.2. Method comparison

Each method has its own advantages and disadvantages. Let N denote the total number of particles and k the total number of vertices. The Particle Push method

- is *less efficient*: requires $O((2^d + 1)N)$ read/write operations;
- however, $\mathcal{V}(p_i)$ is of fixed length and is readily computed dynamically from the particle's coordinates.

At the same time, the Particle Pull method

- is *more efficient*: requires $O(2^d N + k)$ read/write operations; in practice $k \ll N$ and thus complexity is dominated by the magnitude of N;
- however $\mathcal{P}(v_{\mathbf{s}})$ would be prohibitively expensive to retrieve dynamically unless particles are organized in some efficient spatial data structure.

Furthermore, a direct parallel implementation of the Particle Push method is prone to memory collisions since $\mathcal{V}(p_i)$ and $\mathcal{V}(p_j)$ will have a non-empty intersection for each pair of particles $\{p_i, p_j\}$ that is processed simultaneously by concurrent threads. The availability of atomic memory operations could resolve this issue, however at the expense of reduced performance.

In addition, it becomes clear that, if implemented without modification, both methods would suffer from a common drawback, namely *random memory access*. This would occur since for each i, and each $\mathbf{s}$, not all elements of $\mathcal{V}(p_i)$, or $\mathcal{P}(v_\mathbf{s})$ will, in principle, reside in contiguous aligned memory locations. While a large cache CPU may alleviate the problem in the case of a serial implementation, on the GPU, random-memory access can be a big performance bottleneck. In particular, for CUDA a read/write operation from/to global memory takes 400–600 clock cycles (the same operation to shared memory takes only 4). This latency can be hidden if threads perform their read/write operations in a coalesced fashion, i.e. a block of threads reads from (writes to) a block of adjacent type-aligned memory locations.

We attempt to overcome the deficits of each method by following a hybrid type approach and organizing data in a way that avoids the aforementioned problems and improves the time complexity of the memory access operations.

5. Toward a fast Particle-To-Grid interpolation algorithm

We now focus on describing an efficient interpolation algorithm specifically tailored for the GPU parallel environment. One major constraint of the GPU that we recognize is the amount of on-board memory, currently 768 MB for G80 chips and 1.5 GB for Tesla. Thus, in order to to fit as many particles as possible, we must minimize the memory footprint of the data structures we use and whenever possible rely on in-place data transformations. The following discussion assumes that these objectives are taken into consideration at all times.

From the analysis in the previous section it becomes apparent that a fast algorithm's data processing strategy should be based on the Particle Pull method. One way to deal with the issue of the inherently inefficient $\mathcal{P}(v_\mathbf{s})$ retrieval is to organize particles in a linear array such that for each $v_\mathbf{s}$ the elements of $\mathcal{P}(v_\mathbf{s})$ occupy a contiguous section of that array.

5.1. The bookmarked particle array

Suppose that pointers marking the endpoints of each section are given as a satellite "bookmark" array. Access to $\mathcal{P}(v_\mathbf{s})$ is efficient as it only requires sequential traversal of the corresponding section of the particle array. To ensure non-overlapping access from vertices sharing common subsets of particles in their associated $\mathcal{P}(v_\mathbf{s})$ sets, this type of data structure would have to be made redundant in the sense that each particle would have to appear in multiple sections of the array. This would result in memory complexity of $O(2^dN)$ which, given the memory limitations of the current generation GPUs, would make it unfeasible for any practically meaningful values of N.

5.2. The cell based particle pull method

A viable alternative is to "transfer" the memory redundancy over to the grid data structure. To this end, we work with the grid $\overline{G}$ dual to G; i.e. the elements of $\overline{G}$ are the 2^d-dimensional cells of G. Let $c_\mathbf{s}$ denote the cell with multi-index $\mathbf{s}$ and let $\mathcal{P}$be the map from the set C of grid cells to the set P of particles:

$$\mathcal{P}\colon C \longrightarrow P$$
$$c_\mathbf{s} \longmapsto \{p \in P \mid p \in c_\mathbf{s}\}$$

i.e. $\mathcal{P}(c_\mathbf{s})$ is the collection of particles contained in $c_\mathbf{s}$. We modify the Particle Pull method described in Section 4.1.1 to accommodate for this change in the underlying data structure. In particular, we keep a copy of the accumulated value of f at each of the 2^d vertices of $c_\mathbf{s}$. This means that the memory complexity of this method will be $O(2^dk)$ which for $k \ll N$ is a considerable improvement over the original version. We call this modified method the *Cell Based Particle Pull*, or CBPP for short, since particles are being "pulled" out of their data container on a per cell basis.

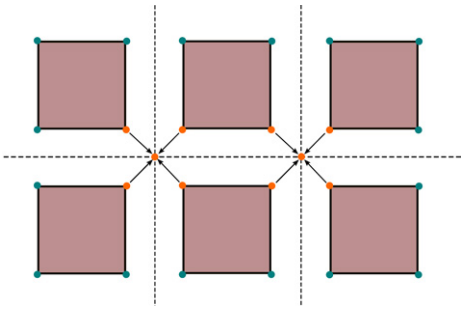

Fig. 5. Illustration of the vertex accumulation process: corresponding values from all cells sharing a vertex are accumulated and the result is stored in the output array.

For the CBPP method the bookmarked particle array data structure requires some modification accordingly. If, for instance, particles are arranged so that $\mathcal{P}(c_\mathbf{s})$ occupies a contiguous section of the bookmarked array for each grid cell $c_\mathbf{s}$. Then access to each subset $\mathcal{P}(c_\mathbf{s})$ is still efficient and, since distinct cells do not share particles, duplication of data is eliminated. As a result, the time complexity of the CBPP method is $O(N + 2^dk)$, which for $k \ll N$ is superior compared to $O(2^dN + k)$ for the "Vertex Based" Particle Pull method discussed in Section 4.1.1.

5.3. The Cell-To-Vertex density accumulation pass

The output from the CBPP method is an array of 2^d-tuples of values; each 2^d-tuple represents the contribution of the particles in the associated cell to the values of f at the vertices of that cell. To obtain the final per-vertex value of f an additional pass through the grid data structure is required. Specifically, each vertex $v_\mathbf{s}$ of G picks up exactly one value from each of the 2^d cells incident with it (see Fig. 5 for a 2D example) and since the grid traversal is uniform, the execution time is fixed for given k and d; it is at least an order of magnitude smaller compared to that of the Cell Based Particle Pull step (see discussion below).

5.4. The preprocessing step

The issue that remains to be addressed is how to arrange the particles in the bookmarked particle array requisite for the efficient execution of the CBPP algorithm. One way to accomplish this is for each particle to compute the global array index of the cell to which it belongs; then use that index as an integer key to sort all particles. Bookmarks can then be computed readily on a second pass by comparing adjacent key values. However, performing a *full sort* at the end of each iteration of the PIC code can be prohibitively expensive; for desirably large values of N the execution time of a full sort could be several orders of magnitude larger compared to that of the CBPP step itself.

The requirement for a full particle sort can be relaxed if we consider the dynamics of particle motion modeled by the PIC code. In many applications, for instance, particles do not traverse more than a few grid cells per time iteration step. This means that most particles will not leave their respective cells and a few will drift at most n_c cells away. Let us assume, without loss of generality, that $n_c = 1$; the case when $n_c > 1$ but still small, is treated similarly. Assume that an initial global sort has been performed and that particles have been arranged in a bookmarked particle array. We show that after a PIC iteration step particles can be rearranged and bookmarked in place in $O(N)$ time.

To achieve this we change the "granularity" of the bookmarked particle array. In particular, we increase the size of each

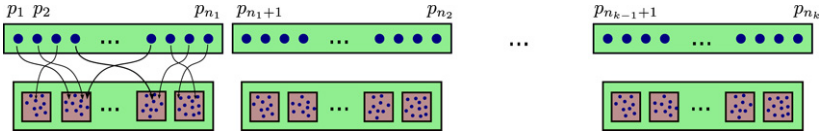

Fig. 6. Particles (black (blue in web version) circles) are partitioned into bins; cells (white (brown in web version) squares) are grouped in clusters. All particles in a bin belong to a single cluster of cells. The order of particles within each cluster is irrelevant.

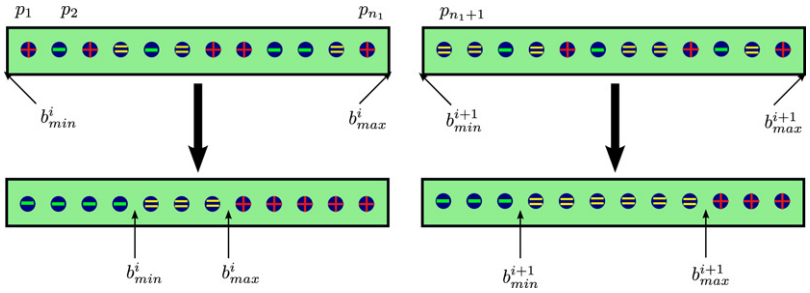

Fig. 7. During the Defragmentation step particles are labeled "+", "−", and "=" according to where they go: right neighbor, left neighbor or same bin respectively, and then shifted to the corresponding section of each particle bin. Bookmarks are adjusted to mark the endpoints of the middle section.

bookmarked section to include all particles that map not to single cell but rather to a *cluster* of cells. We refer to these enlarged bookmarked particle array sections as *bins*. More precisely, let $C_n = \{c_{\mathbf{s}_1}, \ldots, c_{\mathbf{s}_n}\}$ be a cluster of cells. Then the bin associated with C_n is $\mathcal{P}(C_n)$, where $\mathcal{P}$ is the cell-to-particles map defined in Section 5.2. We choose the geometry of the clusters in such a way that after one PIC iteration step particles will either stay within the same bin or will move at most to the adjacent one. A natural solution is to define a cell cluster to consist of all cells that lie on a hyper-slab of co-dimension 1 in D: for instance take the collection of cells with a single fixed index. Two adjacent bins in the bookmarked array map to adjacent clusters in the grid cell array (see Fig. 6). For specific architectures (e.g. CUDA), and problem sizes, however, such hyper-slab cluster partitioning may not be feasible due to the limited shared memory resources available per multiprocessor. Alternative partitioning schemes are discussed in Section 6.4.

The key observation that makes a Particle Binning algorithm of $O(N)$complexity possible is that the CBPP method does not depend on the order of particles within each bin and thus an "incomplete" particle sort would suffice. Our algorithm consists of two passes:

Particle defragmentation. After a PIC iteration step, each existing particle bin will contain three types of particles: those that remain within the bin; and those that need to be moved to the bin on the right or to the bin on the left, respectively. During the Particle Defragmentation pass particles are partitioned in three groups with respect to the associated cell index with pivot values determined by the cell cluster boundary index. More specifically, let $P = \{p_{i_1}, \ldots, p_{i_n}\}$ be a bin of particles that maps to a cluster of cells $C = \{c_{\mathbf{s}_1}, \ldots, c_{\mathbf{s}_m}\}$. Since a cell cluster, by definition, is laid out in contiguous memory, the set of multi-indices $\{\mathbf{s}_1, \ldots, \mathbf{s}_m\}$ corresponds to a set of consecutive integer grid array indices $\{j, j+1, \ldots, j+m\}$. For a particle p_i let $I(p_i)$ denote the associated integer grid array index. The goal of this pass is to partition the set P into three subsets:

$$L = \{p \in P \mid I(p) < j\}$$
$$R = \{p \in P \mid I(p) > j + m\}$$
$$S = \{p \in P \mid j <= I(p) <= j + m\}.$$

This is accomplished by element swapping in two consecutive traversals of P, one forward, with pivot value equal to j and one backward, with pivot value equal to $j + m$ (see Algorithm Listing (3)). In the end the elements of P are permuted in such a way that $P = \{L; S; R\}$. Bookmarks are adjusted to indicate the endpoints of S (see Fig. 7).

```
// Loop over particle bins
foreach particle bin P_l do
    t_min ← lowest cell index in the cluster associated with P_l ;
    t_max ← highest cell index in the cluster associated with
    P_l;
    α ← lowest particle index of P_l ;
    ω ← highest particle index of P_l ;
    // Forward swapping pass
    foreach p_i ∈ P_l in ascending order do
        if I(p_i) < t_min then
            swap (p_i, p_α) ;
            α ← α + 1 ;
        end
    end
    // Backward swapping pass
    foreach p_i ∈ P_l in descending order do
        if I(p_i) > t_max then
            swap (p_i, p_ω) ;
            ω ← ω − 1 ;
        end
    end
    // At this point α and ω are the new
       temporary bookmarks indicating the
       boundaries of the subset S_l of
       non-migrating particles
end
```

Algorithm 3: Particle Defragmentation

Particle rebracketing. After a Defragmentation step particles in the L and R subsets of each P need to be swapped with those in the adjacent bin. More precisely, if $P_l = \{L_l; S_l; R_l\}$ and $P_{l+1} = \{L_{l+1}S_{l+1}; R_{l+1}\}$ are two adjacent bins then the Particle Rebracketing step will swap the subset L_{l+1} with R_l resulting in new bins $P_l = \{L_l; S_l; L_{l+1}\}$ and $P_{l+1} = \{R_l; S_{l+1}; R_{l+1}\}$. The bookmarks of P_l and P_{l+1} will be readjusted to mark the new boundary between them (see Fig. 8).

This step is based on the same swapping strategy used in one of the two passes of the Defragmentation step; in the Rebracketing case there is a single pass that for each l traverses only the elements of R_l and L_{l+1} and swaps them with the pivot value equal to the associated cell cluster boundary. The complexity of this step depends on the number of particles that leave their bins, which is typically only a small fraction of N. It is important to note that in the case where the maximum cell traversal number $n_c > 1$, the

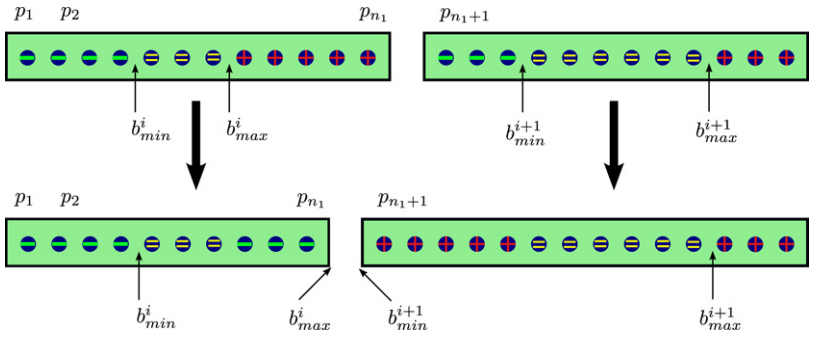

Fig. 8. During the Rebracketing step particles from adjacent bins are swapped according to their designation from the Defragmentation step. Bookmarks are adjusted to mark the endpoints of the beginning and the end of the reshaped bins.

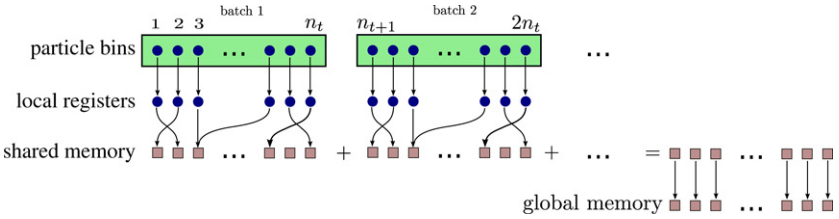

Fig. 9. A single particle bin is processed by a block of n_t threads. Since n_t is much smaller than the number of particles in the bin several batches of size n_t each are processed in succession. The intermediate results are accumulated in a shared memory cell data structure of size n_t which in the end gets written to global memory in a coalesced fashion.

Particle Defragmentation and Rebracketing steps do not change; they just need to be run n_c times in a row. For small values of n_c the resulting complexity is still small compared to that of full particle sorting.

5.5. Summary

Combining the algorithm ingredients discussed so far we summarize the main components of a fast Particle-To-Grid Interpolation Algorithm

1. *Particle Binning* consisting of particle defragmentation followed by particle rebracketing;
2. *Particle-To-Cell Density Deposition* carried out by the Cell Based Particle Pull algorithm;
3. *Cell-To-Vertex Density Accumulation* which produces the final array of particle density values at each grid vertex.

6. CUDA implementation of Particle-To-Grid interpolation

We now discuss the implementation of the Particle-To-Grid interpolation operation on an NVIDIA CUDA graphics processor. We focus on the case of a three dimensional domain with periodic boundary conditions. One of the main benefits of the fast Particle-To-Grid Interpolation Algorithm is that the data structures and the data management strategies fit well into the CUDA programming/execution model. In particular, random global memory access has been eliminated and data has been organized in units that can be processed independently by concurrent threads. The key performance enhancing strategy here is to allow threads to cooperate in processing batches of particles staged in shared memory.

6.1. Data structures

There are four real valued quantities associated with a particle: three spatial coordinates and a weight. The spatial coordinates determine uniquely the index of the grid cell in which the particle is contained. Although technically redundant this integer index is stored separately and used as a sorting key during the Particle Binning step. Following one of the fundamental stream processing mantras we organize the particle data container as a structure of streams (vs. a stream of structures). Thus there are five separate arrays of length N, one for each of the five particle attributes. Similarly for the grid cell data container: the contribution of particles to each of the eight vertices in a cell is maintained in a separate array. One of these arrays is ultimately used for output during the stream reduction operation in the Cell-To-Vertex Density Accumulation Step (see below).

6.2. Thread organization

One CUDA thread block is assigned per particle bin. In practical applications the bin size is larger than the maximum allowable threads per block (512 at the time of this writing) and so each thread makes several strides over its assigned bin. Unless the bin size happens to be a multiple of the block size, some threads will be idle during the last stride.

While processing its assigned particle bin, a thread block stores the intermediate values of the particle contributions to shared memory (Fig. 9). Since particles within a bin are not sorted, write access to shared memory is random resulting in possible memory collisions. Resolving these collisions is made possible by a "thread tagging" trick that emulates atomic shared memory operations (see discussion in 6.3.1). The output from a kernel executed by a thread block is written in the associated cell cluster in the global cell data container. Threads are synchronized at the onset of the output stage and writing is coalesced ensuring optimal performance.

6.3. Efficient use of parallel cache

During the Particle Binning and the Particle-To-Cell Density Deposition steps shared memory is used as parallel cache in which threads stage data for processing. Each step has a different parallel cache management strategy that reflects the nature of the specific algorithms involved.

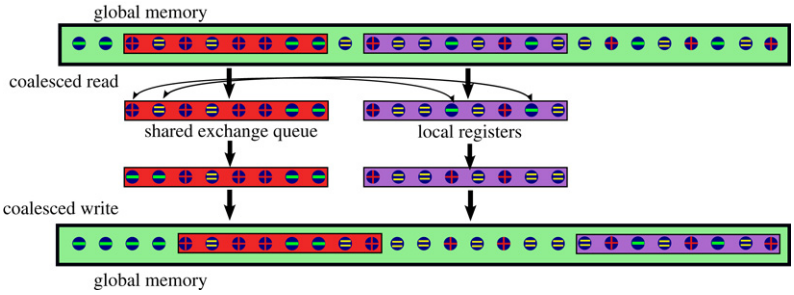

Fig. 10. An illustration of Particle Defragmentation in CUDA. A thread block reads the current exchange queue (dark grey (pink in web version) background) into shared memory, and the current particle batch (white (purple in web version) background) into local registers. The particle batch and the exchange queue are initialized identically: with the first n_t bin elements for the *forward swapping pass*; with the last n_t bin elements for the *backward swapping pass*. After all necessary swaps are completed, the two batches are written (coalesced) back into global memory. The head of the exchange queue is shifted according to the number of swaps in the previous pass; the next particle batch is offset by the number of CUDA threads per block.

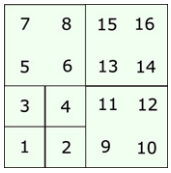

Fig. 11. Illustration of the spatial decomposition and the associated memory layout for hierarchical bins. Numbered boxes indicate cell clusters each containing a bin of particles. Particles within a bin can be in arbitrary order.

6.3.1. Particle Binning

During this step a pair of particle batches are read (in a coalesced fashion) from the global particle array, one into shared memory and one into local (per thread) registers. The batch size is equal to the number n_t of CUDA threads per block. The batch in shared memory serves as an "exchange queue": each thread compares the cell index of the particle in its local register with the current pivot value and swaps accordingly with the particle in the next available slot in the exchange queue. After all swaps are completed the threads perform a coalesced write of the two batches back into global memory. Fig. 10 illustrates the CUDA version of the Particle Defragmentation algorithm pictorially.

Since comparisons are carried out concurrently, the one outstanding problem is how to update the exchange queue's tail attribute in a synchronized fashion.

The exchange queue is implemented as an array of size n_t and its tail attribute is simply an integer index into the next available element in the array. Conflict arises if several concurrent threads attempt to update the index simultaneously. To deal with this issue we borrow the "thread tagging" trick used in the NVIDIA's histogram calculation example [15]. Essentially, the tail attribute is declared as a shared memory integer variable with the C++ *volatile* type qualifier and tagged by inserting the index of the last thread that updated it into its 5 most significant bits. A thread that needs to update the tail attribute attempts a sequence of read-write operations into the tail variable until it verifies that the last updated value was indeed its own one (see [15] for details). During each attempt the tail value is incremented by one; upon success, the thread has obtained a valid slot index which it uses to swap its particle with the corresponding element in the exchange queue. Since the thread tagging trick works only on a per-warp basis, separate queue indices are maintained for each warp. During a preliminary pass each thread determines whether it needs to swap its particle with one in the exchange queue; in the case when it does, it obtains a valid per-warp slot index. Threads are then synchronized and subsequently all those that need to swap do so in parallel by converting the per-warp index into a unique exchange queue index. The key point here is that after synchronization the number of swapped particles per warp is already known and is used as an offset for the set of indices in the higher numbered warps.

We point out that operations that rely on similar types of parallel shared counters, or shared memory atomic operations in general, can be essential to the efficient implementation of important parallel algorithms in CUDA. We believe that it would be very beneficial if GPU manufacturers consider implementing shared memory atomic operations in hardware. Atomic operations in global memory are already available for CUDA devices of capability 1.1 and higher.

6.3.2. Particle-To-Cell Density Deposition

During this step particles are read (in a coalesced fashion) in batches into local (per thread) registers from the corresponding bin in the global particle array. A single CUDA thread block is assigned to a cell cluster; each cell in the cluster is assigned to an element in a shared memory Density Deposition array. Elements of this array consist of two quadruples of real numbers corresponding to the value of the density function at each cell's eight vertices. After a thread has read its current particle it determines the particle's cell index and deposits its contribution to the cell vertices. As the process of deposition is carried out by concurrent threads it is prone to memory collisions and may require simultaneous updates to elements in the Density Deposition array.

To deal with this issue in the absence of atomic shared memory operations one may attempt to apply the thread tagging trick from Section 6.3.1. However, the difference here is that the updated quantities are floating point numbers. One option to get around this is to assume that no two particles within a warp-sized batch will have exactly the same contribution to the cell vertices. If this is the case then the contribution values themselves can be used as unique particle identifiers and the same process of incremental updating can be applied. Another option is to use the 5 least significant bits in the floating point representation of the contribution value for thread tagging (as opposed to the 5 most significant bits for the integer case). This in effect reduces the floating point precision to 27-bits, which, depending on the application, may be computationally sufficient.

Neither of these approaches is ideal, although the assumption in the former is rather non-restrictive: in most cases the probability for a repeated pair in a sequence of 32 floating point numbers drawn from a close to uniform distribution is virtually zero. This is the approach we take in our test implementation. Clearly, the availability of hardware supported shared memory atomic operations would be a highly desirable feature in future CUDA devices.

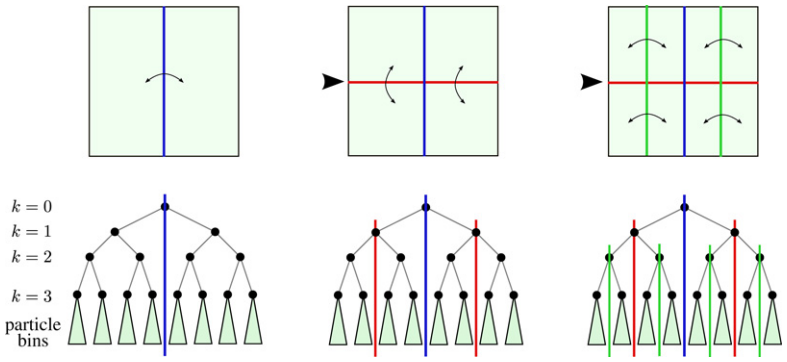

Fig. 12. Bins are laid out in memory so that defragmentation can be performed hierarchically. Traversing the BSP tree takes $O(N \log C)$, where C is the total number of bins. The leaves of the tree contain unsorted particle sequences.

Particles	Full Sorting	Binning	Ratio
1M	117.6	5.8	20.3
2M	173.7	9.7	17.9
4M	351.8	18.3	19.2
8M	709.8	37.8	18.8
16M	1407.7	73.5	19.4

Fig. 13. Performance comparison of our Particle Binning algorithm against full sorting. Particle numbers are given in powers of 2, i.e. 1M = 1048 576. Full sorting is carried out using the CUDA Data Parallel Primitives library (CUDPP) radix sort function. Execution times are measured in milliseconds.

6.4. Hierarchical Particle Binning

The restrictions imposed by the limited amount of shared memory per multiprocessor (presently 16 KB) have an impact on the CUDA implementation of the Particle Binning step. Increasing the number of threads per block benefits performance but at the same time diminishes the amount of available shared memory per thread block. Optimal performance for the Particle-To-Cell Density Deposition step is achieved at a cell cluster size of 64; in the 3-dimensional case this means that the product of two of the grid's dimensions has to be 64, too small for practical purposes. To address this issue we implement a hierarchical version of the Particle Binning algorithm and adjust the underlying particle and cell data structures accordingly.

The main idea is to treat the particle bins as leaves of a pointer-less binary space partition (BSP) tree. Associated with it is a spatial domain decomposition whereby cell clusters are arranged along a space filling curve [17], such as a Z-curve (Fig. 11). Hierarchical binning is carried out by performing particle defragmentation and exchange among the two branches of the tree at each node starting from the root. This process is similar in spirit to the recursive partitioning performed in the classical Quicksort algorithm, [3]; the main differences are that here tree traversal is partial and breadth first; also pivot values are fixed in advance and need not be part of the sorted sequences. An illustration of the hierarchical binning algorithm is given in Fig. 12

This algorithm has two main benefits:

- it doesn't assume anything about the particle dynamics and specifically about how far particles drift in each time step;
- it parallelizes well (just like Quicksort).

An issue, however, is that the number of allowable thread blocks is low at levels close to the root of the BSP particle bin tree: at level k there can be at most 2^{k+1} thread blocks launched simultaneously. With parallel execution at level k, the number of elementary operations per thread block is $N/2^{k+1}$ and thus the expected parallel execution time is proportional to

$$N\left(\frac{1}{2}+\frac{1}{2^2}+\cdots+\frac{1}{2^{k+1}}\right)=N\left(1-\frac{1}{2^{k+1}}\right) \tag{4}$$

which $\approx N$ for large enough k. If hierarchical binning is performed over a 3-dimensional domain, this bound can be improved by first running the Defragmentation algorithm in parallel with respect to some slab partitioning (as outlined in Section 5.4). Then within each slab run a 2-dimensional hierarchical binning in parallel for all slabs. This gives parallel execution time proportional to:

$$\frac{N}{s}+\frac{N}{s}\left(\frac{1}{2}+\frac{1}{2^2}+\cdots+\frac{1}{2^{k+1}}\right)\approx 2\frac{N}{s} \tag{5}$$

where s is the number of slabs.

For instance, on a domain of size $n_x \times n_y \times n_z$ we identify slabs with two dimensional slices along a coordinate axis, say z, and thus $s = n_z$. We set cell clusters to have size 64 (due to the shared memory limitations outlined above) and consequently there are $n_x \times n_y/64 = 2^k$ cell clusters per slab. There are $k+1$ Defragmentation passes: the first one with n_z thread blocks to repartition between adjacent slabs; the next k with $n_z 2^i$ thread blocks each, $1 < i < k$, to perform hierarchical binning at level i, within each slab.

We emphasize that the estimates given in Eqs. (5) and (4) are theoretical best case scenarios. In practice, if k is too large, the terms of the geometric sequence in Eq. (5) are expected to saturate for some $k_c < k+1$. This is due to the fact that at some level of the BSP particle bin tree there will not be enough available thread blocks for concurrent execution and the kernel would run longer than the ideal $O(N/2^{k_c+1})$ time.

7. Performance results

We discuss the performance of our algorithms on various test case scenarios. We consider the Preprocessing Step and the Particle Density Deposition Step separately so that the effect of sorting can be isolated. We leave out the Cell-To-Vertex step whose execution time depends only on the grid size and is relatively small; for example, it is 19 and 6 milliseconds for grids of size 64^3 and 32^3 respectively. The tests have been performed on a 2.67 GHz 64-bit quad core Intel processor using Intel's Fortran 10.1 compiler for the CPU version, and NVIDIA's CUDA 1.1 `nvcc` for the GPU version of the code. The standard `O2` compiler optimization option was used to compile both versions.

Preprocessing step. We compare the execution times of Particle Binning versus full sorting, both performed on the GPU. The results are shown on Fig. 13 for a grid size of 64^3 and cell cluster size of 64.

Number of Particles		Timings		
Total	Per Cell	CPU	GPU	Ratio
256K	8	12.9	0.7	18.4
512K	16	22.1	1.3	17.1
1M	32	41.4	2.6	15.9
2M	64	77.3	5.3	14.6
4M	128	149.9	10.9	13.6
8M	256	296.3	24.1	12.3
16M	512	589.4	50.9	11.6

Fig. 14. Performance comparison of the Particle-To-Cell step for grid size $32 \times 32 \times 32$ and varying number of particles. Particle numbers are given in powers of 2, i.e. 1K = 1024; 1M = 1048 576. The number of particles per cell is the average. Execution times are measured in milliseconds.

Number of Particles		Timings		
Total	Per Cell	CPU	GPU	Ratio
1M	4	67.8	3.2	21.2
2M	8	104.2	5.5	18.9
4M	16	178.9	10.3	17.4
8M	32	324.3	20.3	15.9
16M	64	617.2	42.3	14.6

Fig. 15. Performance comparison of the Particle-To-Cell step for grid sizes $64 \times 64 \times 64$.

Particle-To-Cell step The Particle-To-Cell step has been implemented on the CPU using the Particle Push method but assuming a sorted input particle array. This improves significantly CPU cache coherency and results in 3-5 times speedup over the unoptimized CPU version. The CPU times shown on Figs. 14 and 15 correspond to the "sorted" version, ensuring that both the GPU and the CPU implementations are compared as fairly as possible. Two observations are immediately apparent:

- that execution times scale with the number of particles: linearly for CPU and almost linearly for GPU
- that the CPU/GPU time ratio depends only on the average number of particles per cell.

Furthermore, these results provide an encouraging initial benchmark that unequivocally justifies the feasibility of developing high performance PIC codes on the GPU.

8. Conclusion

We have given an overview of Particle-To-Grid interpolation in the context of plasma PIC simulations. We have discussed the potential issues involved in implementing Particle-To-Grid interpolation on the GPU and we have presented a fast parallel algorithm that maps efficiently on CUDA platforms. We have demonstrated considerable speedup over an equivalently optimized CPU implementation.

As future work we envision incorporating the CUDA accelerated Particle-To-Grid component into a fully functional GPU based PIC code. Further, we plan to carry out careful investigation and benchmarking of the impact of cell cluster geometry and size on the performance of the Hierarchical Particle Binning step. Comparison with other computational methods whose GPU implementations exhibit similar memory access patterns issues, such as the NFFT [18], and the FMM [6], is also in order. An issue of paramount importance is how PIC codes would scale in a hybrid distributed GPU environment, e.g. a cluster of GPU enabled multi-core compute nodes where network communication will become the dominant bottleneck. Such a migration would be ultimately necessary if realistically sized PIC simulations should find their way into the mainstream of GPU computing.

Acknowledgments

We would like to thank Ramani Duraiswami, Amitabh Varshney, Marc Swisdak, Ingmar Broemstrup, and Derek Juba for helpful discussions and suggestions during the development phase of our work. We are grateful to the paper referees for their detailed comments and suggestions.

References

[1] C.K. Birdsall, A.B. Langdon, Plasma Physics Via Computer Simulations, McGraw-Hill, Inc., New York, NY, USA, 1985.
[2] K. Bowers, Accelerating a Particle-In-Cell simulation using a hybrid counting sort, Journal of Computational Physics 173 (19) (2001) 393–411.
[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, The MIT Press, 2001.
[4] V. Decyk, Skeleton PIC codes for parallel computers, Computer Physics Communications 87 (1–2) (1995) 87–94.
[5] N. Gumerov, R. Duraiswami, W. Dorland, Middleware for programming NVIDIA GPUs from Fortran 9x, Supercomputing '07, Reno, NV (2007).
[6] N.A. Gumerov, R. Duraiswami, Fast multipole methods on graphics processors, Journal of Computational Physics (2008), doi:10.1016/j.jcp.2008.05.023.
[7] M. Harris, J. Owens, S. Sengupta, Y. Zhang, A. Davidson, CUDA data parallel primitives library (2007). URL http://www.gpgpu.org/developer/cudpp/.
[8] W. Hwu, D. Kirk, Programming massively parallel processors, University of Illinois at Urbana-Champaign (2007). URL http://courses.ece.uiuc.edu/ece498/al1/index.html.
[9] D. Luebke, The democratization of parallel computing, AstroGPU Conference, Princeton, NJ (November 2007).
[10] W.M. Nevins, G.W. Hammett, A.M. Dimits, W. Dorland, D.E. Shumaker, Discrete particle noise in particle-in-cell simulations of plasma microturbulence, Physics of Plasmas 12 (2005) 122305.
[11] NVIDIA Corporation, NVIDIA CUDA Programming Guide (2007).
[12] L. Nyland, N-body on the GPU, AstroGPU Conference, Princeton, NJ (November 2007).
[13] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips, GPU computing, Proceedings of the IEEE 96 (5).
[14] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, T.J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (1) (2007) 80–113.
[15] V. Podlozhnyuk, Histogram calculation in CUDA (2007). URL http://www.nvidia.com/object/cuda_sample_data-parallel.html.
[16] V. Przebinda, J. Cary, Some improvements in PIC performance through sorting, caching, and dynamic load balancing, University of Colorado (2005).
[17] S. Seal, S. Aluru, Handbook of Parallel Computing: Models, Algorithms and Applications, Chapman & Hall/CRC, 2008, Ch. 44, pp. 44.1–44.24.
[18] T. Sorensen, T. Schaeffter, K. Noe, M. Hansen, Accelerating the nonequispaced fast fourier transform on commodity graphics hardware, IEEE Transactions on Meidcal Imaging 27 (4) (2008) 538–547.
[19] G. Stantchev, D. Juba, W. Dorland, A. Varshney, High-performance computation and visualization of plasma turbulence on graphics processors, Computing in Science and Engineering (in press).
[20] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, Journal of Computational Chemistry 28 (16) (2007) 2618–2640.
[21] S. Stone, H. Yi, W. mei Hwu, J. Haldar, B. Sutton, Z.-P. Liang, How GPUs can improve the quality of magnetic resonance imaging (October 2007). URL http://www.gigascale.org/pubs/1175.html.
[22] D. Tskhakaya, R. Schneider, Optimization of pic codes by improved memory management, Journal of Computational Physics 225 (1) (2007) 829–839.

George Stantchev is a Research Associate at the Center for Scientific Computing and Mathematical Modeling and the Center for Multiscale Plasma Dynamcis at the University of Maryland, College Park. He received his Ph.D. in Applied Mathematics and Scientific Computation from the University of Maryland in 2003. His current research focuses on high-performance computing, scientific visualization, and plasma turbulence simulations.

William Dorland is an Associate Professor at the Department of Physics with joint appointments at the Center for Scientific Computing and Mathematical Modeling and the Institute for Research in Electronics and Applied Physics at the University of Maryland, College Park. He holds a Ph.D. in astro-physics from Princeton University (1993). His research interests include plasma turbulence, astrophysical gyrokinetics, and high-performance computing.

Nail Gumerov is an Associate Research Scientist at the University of Maryland Institute for Advanced Computer Studies. He holds a Sc.D. degree in Physics and Mathematics from the Tyumen State University, Tymen, Russia (1992). His research interests comprise a number of disciplines ranging from acoustics and hydrodynamics to computer vision and high-performance computing, among others.