Computer Physics Communications 185 (2014) 708–719

Contents lists available at ScienceDirect

Computer Physics Communications

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

Particle-in-Cell algorithms for emerging computer architectures☆

Viktor K. Decyk[a,*], Tajendra V. Singh[b]

[a] Department of Physics and Astronomy, University of California, Los Angeles, CA 90095-1547, USA
[b] Institute for Digital Research and Education, University of California, Los Angeles, CA 90095-1547, USA

ARTICLE INFO

Article history:
Received 22 June 2013
Received in revised form
4 October 2013
Accepted 14 October 2013
Available online 22 October 2013

Keywords:
Parallel algorithms
Particle-in-Cell
GPU
CUDA
Plasma simulation

ABSTRACT

We have designed Particle-in-Cell algorithms for emerging architectures. These algorithms share a common approach, using fine-grained tiles, but different implementations depending on the architecture. On the GPU, there were two different implementations, one with atomic operations and one with no data collisions, using CUDA C and Fortran. Speedups up to about 50 compared to a single core of the Intel i7 processor have been achieved. There was also an implementation for traditional multi-core processors using OpenMP which achieved high parallel efficiency. We believe that this approach should work for other emerging designs such as Intel Phi coprocessor from the Intel MIC architecture.

1. Introduction

High Performance Computing (HPC) is going through a revolutionary phase with greatly increased parallelism on shared memory processors, but with different approaches. A number of these approaches make use of Single Instruction Multiple Data (SIMD) vector processors, which introduces another level of parallelism. Graphical Processing Units (GPUs) were an early pioneer in this kind of computing, especially after NVIDIA introduced CUDA, a parallel computing architecture with a new parallel programming model and instruction set architecture which substantially simplified programming their devices [1]. Intel has also introduced a similar architecture with the Phi coprocessor [2], but with a different programming model. The Intel Phi coprocessor is based on the Intel Many Integrated Core (MIC) Architecture.

This paper focuses on how to implement Particle-in-Cell (PIC) codes on such emerging architectures. PIC codes model plasmas by integrating self-consistently the trajectories of many charged particles responding to the electromagnetic fields that they themselves generate [3,4]. They are widely used in plasma physics, and with domain-decomposition techniques [5] they have run in parallel on very large distributed memory supercomputers [6]. PIC codes have three major steps: depositing some quantity such as charge or current from particles onto a grid, solving a field equation to obtain the electromagnetic fields, and interpolating electric and magnetic fields to particles from a grid. They have two data structures, particles and fields, that need to efficiently communicate with one another.

A variety of PIC codes have been implemented on GPU platforms in recent years [7–15]. In an earlier paper [8], we described how we implemented a two dimensional (2D) electrostatic spectral PIC code on the NVIDIA GT200 class of GPUs. The fundamental approach we used was to divide space into small tiles (typically containing 2×3 grid points) and to keep particles in each tile stored together in memory. Each tile could then be processed independently in parallel and only a small amount of fast local memory was needed.

More tiles were used than actual processors. The deposit step is very critical because data collisions are possible when two threads attempt to write to the same memory location simultaneously. Because the GT200 series GPUs (such as the Tesla C1060) are very slow in resolving data collisions for floating point data, the algorithm we designed was collision-free. This was accomplished by assigning a different thread to each tile. Because tiles were very small, the cost of reordering particles moving between tiles was substantial.

* Corresponding author. Tel.: +1 310 206 0371.
E-mail address: decyk@physics.ucla.edu (V.K. Decyk).

0010-4655/$ – see front matter © 2013 The Authors. Published by Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.cpc.2013.10.013

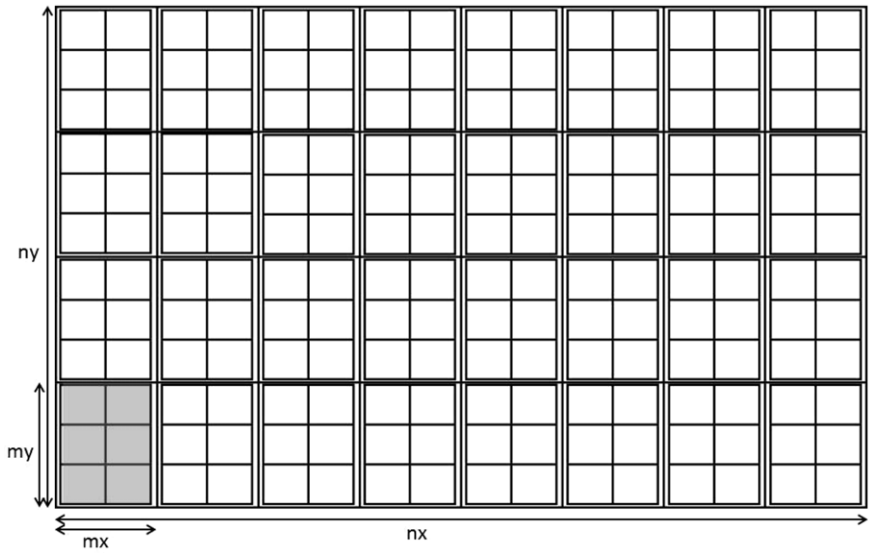

Fig. 1. Partition of grid space into tiles. In this figure, the variables defined in the text have values $\mathtt{nx} = 16$, $\mathtt{ny} = 12$, $\mathtt{mx} = 2$, $\mathtt{my} = 3$, and their sizes are shown. The total number of tiles is 32, with layout $\mathtt{mx1} = 8$, $\mathtt{my1} = 4$. Additional guard cells for the tiles are not shown.

With NVIDIA's introduction of the Fermi architecture, several new features were added that had a substantial impact on optimal PIC algorithms: increased fast local memory, automatic cache, and native support for atomic adds for floating point data. The use of tiles is still optimal, but now it is preferable to assign a block of threads to a tile rather than one thread. We call this algorithm collision-resolving. This paper describes the changes made to the earlier algorithm and the resulting performance of the new algorithm. Furthermore, we show that the general tiling approach can also work well on conventional shared-memory multi-processors and this can be used as the basis for a portable approach to PIC codes on emerging architectures. Both electrostatic and electromagnetic spectral algorithms are described.

2. Collision-free GPU PIC algorithm

An earlier version of this algorithm is described in [8]. Here we summarize the main points and describe some changes. The algorithm is described in 2D space, but can be extended to 3D.

Space is divided into tiles each of which contains mx, my uniformly spaced grid points in x and y respectively, plus additional guard cells. If there are nx, ny grid points in the entire computational region, then there is a total of $\mathtt{mx1} \times \mathtt{my1}$ tiles, where $\mathtt{mx1} = (\mathtt{nx} - 1)/\mathtt{mx} + 1$ and $\mathtt{my1} = (\mathtt{ny} - 1)/\mathtt{my} + 1$. Tiles at the edges can contain grid points which are not active. Fig. 1 describes the partitioning.

To determine which tile a particle belongs to, one first determines the grid point n, m, that the particle belongs to. The tile number is then given by: n/mx and m/my. The same tiling is used in both the collision-free and collision-resolving algorithms.

Both GPUs as well as other high performing processors make use of SIMD processors, which perform nvec vector operations simultaneously (in lockstep). The value of nvec varies from 4 on the Intel SSE to 32 on NVIDIA Fermi devices. These SIMD processors also read data in large blocks, such as 128 bytes on the Fermi device. It is thus advantageous that adjacent memory locations needed by a vector operation are stored in adjacent memory locations. On the GPU, each vector operation is performed by a group of threads (called a thread block in CUDA).

For the collision-free algorithm, each vector element (thread) is responsible for one tile, and the particle data is stored in an array where adjacent threads in a thread block read adjacent memory locations. Such data layout for the particles is described by the following Fortran declaration:

```
real dimension partc(lvect,idimp,nppmx,mbxy1)
```

where lvect is typically the blocksize on the GPU, idimp is the number of particle co-ordinates (2 positions plus 2 or 3 velocities), nppmx is the maximum number of particles expected in a tile, and mbxy1 the number of independent blocks of tiles given by: $(\mathtt{mx1} * \mathtt{my1} - 1)/\mathtt{lvect} + 1$. An auxiliary array kpic is used to store the number of actual particles in each tile in partc. It is declared as follows:

```
integer kpic(mx1*my1)
```

Normally, the code initializes the particle co-ordinates on the host computer in the usual way, then rearranges them by tile into the new data structure and copies them to the GPU.

Advancing particles is now easily and efficiently parallelized. The input to the push procedure is a global field array, which for linear interpolation is declared as follows:

```
real dimension fxye(idimp-2,nx+1,ny+1)
```

where idimp-2 corresponds to the number of velocity co-ordinates (2 for the electrostatic code, 3 for the electromagnetic code). Each thread block copies the fields it requires from global memory into a small local (shared) memory array which can be read much faster than