Efficient GPU implementation for Particle in Cell algorithm

Rejith George Joseph, Girish Ravunnikutty, and Sanjay Ranka
UFL
rjoseph, girishr, sranka@ufl.edu

Eduardo D'Azevedo and Scott Klasky
ORNL
dazevedoef, klasky@ornl.gov

***Abstract*—Particle in cell (PIC) algorithm is a widely used method in plasma physics to study the trajectories of charged particles under electromagnetic fields. The PIC algorithm is computationally intensive and its time requirements are proportional to the number of charged particles involved in the simulation. The focus of the paper is to parallelize the PIC algorithm on Graphics Processing Unit (GPU). We present several performance trade-offs related to small shared memory and atomic operations on the GPU to achieve high performance.**

***Keywords*-Particle in cell; GPU; CUDA;**

I. Introduction

The particle-in-cell algorithm is a method widely used to simulate plasmas and hydrodynamics. The movement of particles is computed by the interaction between each pair of particles in a self-consistent system. Instead of directly calculating the interaction of particles, the PIC algorithm employs a regular computational mesh on the particle domain and assigns the particle attributes to nearby grid points of the mesh. The field equations are then solved on the mesh and the force on each particle is obtained by gathering the attributes of nearby grid points from the resultant fields on the mesh. This force is then used to move the particle for the next step of the simulation.

The PIC algorithm has two different data structures, particle array and mesh grid array. In each iteration both arrays are updated based on values of the other. The interaction between two arrays can be modeled as a bipartite graph as depicted in Figure 1. The elements of the each array form the nodes of the graph and edges represent the communication between both arrays. The mesh grid array is typically spatially homogeneous. The particle array can be non homogeneous. The edges of the graph are dynamic and change with each simulation iteration.

An efficient implementation of the PIC Algorithm on distributed memory MIMD computers requires distributing these two data structures over the processors such that off-processor accesses are minimized. A good load balance in parallelization should also ensure that each data structure is nearly equally distributed among the processors. Further, the data access patterns between two arrays may change dynamically (albeit incrementally) and therefore the particles may need to be redistributed frequently to reduce communication cost.

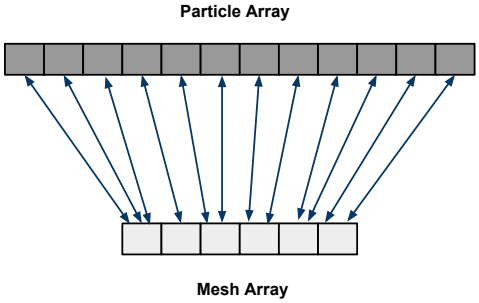

Figure 1: Interaction between particle and mesh grid array

There are many implementations of PIC algorithm on a CPU. The first fully toroidal edge PIC algorithm, XGC1 describes an openMP implementation of the particle in cell simulation using a triangular mesh [1]. Kraeva, Malyshkin uses the Assembly Technology [3] to develop a dynamic load balancing strategy for parallel PIC algorithm [2]. Liao, Ou et al. describes a scalable domain partitioning strategy for particle and mesh arrays that reduces the communication overhead in a distributed computing framework [4]. Madduri, Williams et al. describes 13 different approaches to parallelize the PIC problem on Multi-core processors [5]. This implementation limits the amount of particle movement in an iteration and also expects an initial ordering of particles.

Stantcheva, Dorlanda et al. describes a GPU implementation of the PIC algorithm using a rectangular mesh [6]. The algorithm uses binning in which the cell clusters in a mesh are grouped as bins. During execution, each bin is mapped to a GPU block. The particles are kept in bin-sorted order with an exchange mechanism using shared memory and registers. This is based on the assumption that particles can move only to neighboring bins. Viktor K. Decyk, Tajendra V. Singhet et al. developed a GPU global memory based approach for PIC algorithm that is not efficient because the fast local memory is not used at all [9]. Our approach differs from existing

1530-2075 2011
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/IPDPS.2011.46

implementations in the following ways:

1) The XGC1 [1] code requires a triangular mesh for simulation on GPU. Unlike a rectangular mesh, which is symmetric and can easily be partitioned and mapped to GPU parallel hardware, a triangular mesh is asymmetric. This makes the partitioning complex. A typical triangular mesh contains several hundred thousand triangles and millions of particles. We introduce hierarchical mesh partitioning strategies for effective mapping to the GPU hardware. The hierarchical partitioning can be symmetric or asymmetric.
2) The use of a triangular mesh requires explicit search for an enclosing triangle, for every particle to update the mesh. We describe several novel approaches for accelerating this search.
3) Our algorithms work for random ordering of particles without imposing any limit on the particle movement. To order particles and triangles, we describe a novel *parallel hierarchical bucket sort* algorithm. In this approach, the triangles are ordered only once whereas the particles are incrementally ordered after each iteration.

Our implementation addresses the limited memory bandwidth between GPU and the host as well as the limited amount of shared memory available in each GPU vector core. We explore various mechanisms to reduce the performance penalty due to memory conflicts while using atomic operations. We also study the performance of various levels of memory hierarchy in the GPU (Global, Shared and Registers) and the performance effects of atomic operations on the global and shared memories. Our experiments are based on the CUDA [8] framework provided by NVIDIA, though our algorithms can be ported to any GPU computing framework like openCL. Our GPU kernel algorithms leverage maximum utilization of the GPU shared memory. The GPU kernels are optimized to reduce the communication with the host machine. In this paper, we study and parallelize the computationally intensive portions of the PIC algorithm. The rest of the algorithm can easily be incorporated into our implementations.

The rest of the paper is organized as follows. Section II provides a high level overview of the PIC algorithm and GPU architecture. Section III details the parallelizing strategies and their implementations. Section IV describes experimental results followed by conclusion.

II. Background

A. PIC Problem

The PIC algorithm using a triangular mesh follows the evolution of the particles in a series of time steps, each of which consists of the following five phases:

1) Search phase: Each particle identifies the triangle (cell) in which it belongs.
2) Scatter phase: Using a linear interpolation scheme, each particle scatters its contributions to the current mesh grid points at the vertices of the cell in which it lies. A particle makes no contributions to other grid points. The total particle contributions at each grid point are summed to form the current local density.
3) Field solve phase: Solve Maxwell's equations on the mesh to determine the electric (E) and magnetic fields (B). Each grid point on the mesh needs data from its neighboring grid points to calculate new values for E and B.
4) Gather phase: The electric and magnetic fields at particles are obtained from their vertex grid points by a linear interpolation of particles' relative positions in the cell. Each particle sums these contributions from the enclosing triangle vertices to generate the force on the particle.
5) Push phase: The force obtained from the gather phase moves particles to their new positions.

B. GPU Architecture

GPU acts as a co-processor device to CPU (host). GPU is a dedicated computing device to address problems that have high arithmetic intensity i.e. high ratio of arithmetic operations to memory operations. Each of the GPU cores has a SIMT (Single Instruction Multiple Thread) architecture. The core of a GPU has a small cache and limited flow of control - both of these take up most of the transistors in a CPU whereas in a GPU, more transistors are used up for computing cores.

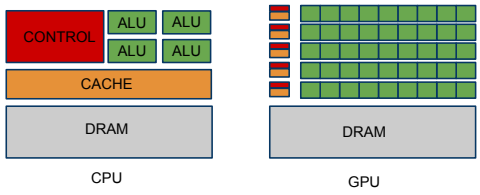

Figure 2: CPU vs GPU

Parallel portions of an application are expressed as device kernels which run on many threads. GPU threads are extremely lightweight and the creation overhead is very little. The scheduling is done by the hardware unlike the operating system in a CPU. A typical GPU needs hundreds of threads for full utilization of hardware whereas CPU can be saturated with only a few threads.

NVIDIA introduced the Compute Unified Device Architecture (CUDA) Framework [11] with a new parallel programming model and Instruction Set Architecture to leverage the capabilities of the massive parallel computing hardware in NVIDIA GPUs. CUDA framework consists of

- An extended C application programming interface
- A run time library which enables the access and control of devices from the host

A CUDA kernel is executed by an array of threads executing the same code in the SIMD (Single Instruction Multiple Data) fashion [13]. The thread array can be represented as 1D (one dimensional), 2D or 3D array. Each thread has a unique id that can be used to access memory for making control decisions. Threads are grouped into blocks. To simplify accessing and addressing of multi dimensional data, the threads are divided into monolithic blocks. Like threads, each block has a unique id. Blocks can be 1D or 2D. Threads within a block co-operate via shared memory, atomic operations and barrier synchronization. Blocks, shared memory and barrier synchronization forms the three core abstractions which provide fine grained data and thread parallelism. These abstractions enable the programmer to partition the problem into coarse sub-problems that can be solved independently by a thread block and each sub-problem into finer parts that can be solved co-operatively in parallel using the threads in a block. The hardware automatically does the scheduling and load balancing. These abstractions also enable automatic scalability of the CUDA program. Since there is no synchronization between blocks, hardware is free to assign blocks to any execution unit.

In a GPU, the processing cores are grouped into streaming multi processors (SMs). Each streaming multi processor has multiple scalar processors (SPs). SM maintains thread ids and block ids and schedules thread execution. Warp is the basic scheduling unit in an SM. A warp is a group of 32 threads in a block. Warps are not part of CUDA programming model and are invisible to a programmer. At any given time SM schedules only one of the warps for execution. All threads in a warp operate on the same instruction during the kernel execution. Figure 3 shows a high level view of GPU architecture.

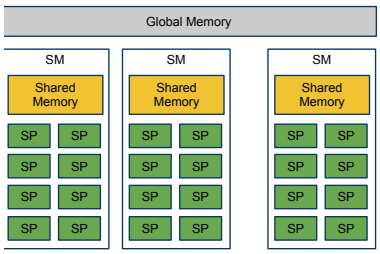

Figure 3: GPU Architecture

Each SM has a shared memory that is visible to all threads in a block executing in the SM. Shared Memory provides extremely fast read and write access. Shared memory is implemented as multiple memory banks and it is as fast as registers when there are no bank conflicts. Bank conflicts arise when multiple threads in a half warp access the same bank, which results in the serialization of access. Figure 4 depicts conflicts on banks 0, 1 and 2. Our algorithms use the access pattern that minimizes these bank conflicts.

Each SM has a register file that stores all the local variables of a kernel. The register file is dynamically partitioned across all the blocks executing in an SM. If a block uses more registers, it limits the registers available to other blocks. This reduces the number of blocks that can be concurrently executed in an SM and reduces performance. In our implementation, register usage is within limits so that an SM can execute maximum number of blocks as permitted by the hardware.

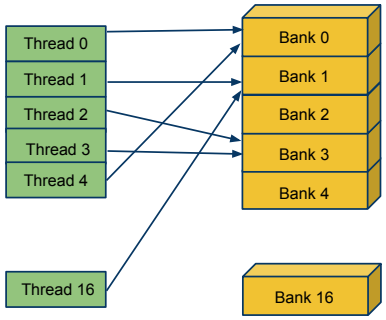

Figure 4: Shared Memory Bank Conflicts

GPU has its own DRAM called as global memory. Global memory is the main means of communication between host and the device. The contents in a global memory are visible to all threads. Since it's implemented as DRAM, access latency is also high. When accessing global memory, peak performance occurs when all the threads in a half warp access continuous memory locations. This type of access is called as coalesced memory access. Figure 5, shows coalesced and non coalesced global memory access. To improve performance, we use coalesced global memory accesses in our algorithms.

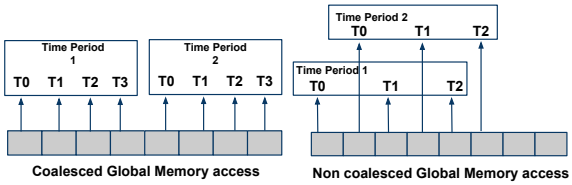

Figure 5: Global Memory Access

C. PIC Semantics

In the rest of the paper we use the semantics as shown in Figure 6.

- The term mesh entity refer particles, triangles, and vertices in a triangular mesh.
- We refer GPU thread block (CUDA thread block) as a GPU block or a block.

D. Parallelization Issues

As depicted in Algorithm 1, the simulation works in iterations. The $compute\_force()$ function encapsulates the

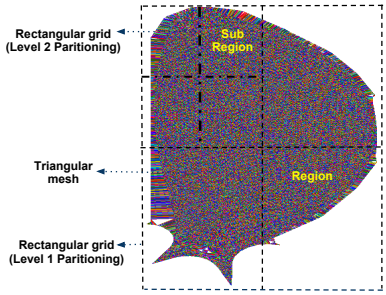

Figure 6: PIC Semantics used

Algorithm 1 Sequential Algorithm

```
1: for i = 0 to number of time steps do
2:    compute_force()
3:    move_particles()
4: end for
```

scatter, field solve, and gather phases. $move\_particles()$ function takes care of the particle push phase. Each iteration is dependent on the results (new particle position) from the previous iteration. Hence the successive iterations cannot be parallelized. However we can parallelize the computations done within an iteration.

In a parallel PIC algorithm, the distribution of particles and mesh grid points over processors should be made such that the overhead generated from the parallelization is carefully controlled to obtain a reasonable speedup. The PIC problem's parallelization strategies can be divided into two broad categories:

1) Direct Eulerian method: Particle domain is divided spatially into several non-overlapping regions (or sub-domains) and each processor is assigned one of these sub-domains. Particles are then assigned to processors according to their positions and may migrate between processors' sub-domains as the system evolves.
2) Direct Lagrangian method: After particles are assigned to processors, this assignment remains fixed throughout the simulation i.e. particles will not move from one processor to another.

Instead of replicating the mesh grids in each processor, the other option is to partition the mesh array into rectangular sub meshes and assign a different sub-mesh to each processor. In this case, the contributions of particles to their grid points are sent directly to the owner processors. The main issue is to decompose two arrays to reduce the amount of off-processor data access. Efficient parallelization of each iteration of the PIC algorithm during the system evolution requires that:

1) The number of particles assigned to each processor be nearly equal,
2) The number of grid points assigned to each processor be nearly equal, and
3) The communication between the two data structures be minimized.

The first two conditions are required to maintain good processor utilization for the computations in each of the five phases mentioned in Section II-A, while the third condition is required to reduce the communication overhead. Achieving all three conflicting goals together is difficult. We describe and analyze three potential ways to achieve the above goals:

1) Mesh Partitioning: Partition the mesh triangles such that each processor gets an equal number of triangles. The particles are then assigned based on the grid cells to which they belong.
2) Particle Partitioning: Partition the particle domain such that each sub-domain has an approximately equal number of particles. The grid cells are then assigned based on the corresponding sub-domain to which they belong.
3) Independent Partitioning: Partition the particle domain such that each sub-domain has an approximately equal number of particles. Also, partition the grid cells such that each processor gets an equal number of grid cells.

III. GPU Parallelization

The direct Eulerian method results in local interprocessor communication, but places no constraints on the computational load balance. The direct Lagrangian method imposes strict load balance, but makes no attempt to reduce communication. As mesh grid data distribution is spatially homogeneous, while the particle data can be non homogeneous, the decision to choose a proper strategy for particle movement among processors represents a trade-off between communication cost and computational load balance. Scalable implementation of the PIC problem for a large number of processors requires load balancing in all phases. The computational load in each phase is so large that a load imbalance in any phase could become a bottleneck. For this reason, we believe that scalable implementation of the PIC algorithm on a GPU would require using the direct Lagrangian method and an independent partitioning strategy that keeps the computation load, particles, and grid points strictly balanced. We devised methodologies to reduce the communication cost. Triangular mesh, requires an additional step to determine the enclosing triangle for each particle. To find this triangle efficiently, we need data structures that require additional memory. The type of partitioning used can have an indirect effect on the storage requirements and computational overhead for this computation.

The PIC problem's mesh decomposition is static. The particles move because of the forces acting on them making the decomposition dynamic. Hence we do the decomposition and optimizations in the mesh once. However, for particles the decomposition may change from iteration to iteration. We partition the mesh into rectangular regions. Vertices and

particles associated with triangles in a region are mapped to data structures on a GPU to facilitate parallel computation. In this way we can effectively use the smaller but faster shared memory effectively.

For a triangular mesh, the entities are particles and triangles. The vertices of a triangle store the forces from the contained particles. These force values are updated by the particles which in turn make the particle move. Particles, triangles, and vertices are separately stored as a linear array for faster access. Some triangles in the unstructured mesh can cross region boundaries. We refer such triangles as shadow triangles and the corresponding vertices as shadow vertices. Shadow triangles and vertices are depicted in Figure 7.

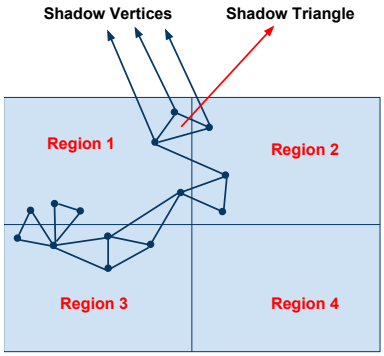

Figure 7: Shadow Triangles and Shadow Vertices

Shadow triangles and vertices will be replicated in all the regions they belong. Since the regions are executing in parallel, after each iteration, we aggregate the forces on shadow vertices from each region.

A. Partitioning Data Structures

As the mesh is partitioned into regions for mapping to GPU blocks, we need to sort the mesh entities (particles, triangles and vertices) in region order. Sorting enables the use of indexing to access the entities. We use linear arrays to store the entities. Each GPU block accesses the mapped region entities by index. i.e i^{th} index in the array corresponds to the entities of region i. The detailed description of the data structures can be found below.

Vertices: Each vertex is a tuple of X, Y co-ordinates and the force. Vertices are stored in a one dimensional array based on the order of regions. The shadow vertices in each region are stored next to the vertices in the region. Figure 8 shows the data structure for vertices .

Figure 8: Vertices array

Triangles: Each triangle is a tuple of 3 vertices $(V1, V2, V3)$ where $V1, V2, V3$ are indexes to the vertices array. The triangles are stored in the order of regions. Like shadow vertices, Shadow triangles in a region are stored next to the triangles in the region. The data structure is similar to Figure 8.

Particles: Each particle is a tuple of X, Y co-ordinates and the triangle index. Triangle index is an index to triangle array. This index stores the triangle in which the particle resides. Particles are stored in the order of regions.

The triangle array and vertices array are initialized only once in the pre-processing step and will not be changed during the simulation step except while updating force in the vertex array. Particle movement changes the particle array during each simulation iteration. Hence the particle array needs to be re-sorted after each simulation iteration. We map each step in the iteration of Algorithm 1 to a kernel that executes in GPU. As mentioned before, a Kernel is a group of GPU blocks that executes in parallel. The parallel algorithm is depicted in Algorithm 2.

Algorithm 2 Parallel Algorithm

```
1: sort_triangles()
2: sort_particles()
3: for i = 0 to number of time steps do
4:   for all GPU blocks in parallel do
5:     for all threads in a block in parallel do
6:       compute_force()
7:       aggregate_force(shadowVertices)
8:       move_particles()
9:       incremental_sort(particles)
10:    end for
11:  end for
12: end for
```

The first two steps of the Algorithm 2 sort triangles and particles respectively in the region order. Triangle sorting addresses vertices' sorting as well. For sorting, we create GPU blocks based on the entities (particles or triangles) to be sorted and number of threads within a GPU block. The number of GPU blocks will be equal to the number of entities divided by the number of threads in a GPU block. NVIDIA Tesla hardware limits the number of blocks in a dimension to 65535. If the number of blocks exceeds this value, we reduce it by increasing the workload of each thread i.e., instead of working on a single entity, each thread in a block will be responsible for multiple entities. The number of blocks in this case will be the number of entities divided by the product of number of threads in a block and number of entities per thread. GPU blocks used for our kernels are shown in Table I.

Initially the mesh entities are randomly distributed. Hence concurrent threads update the data structures. This necessitates the need for locks and we use atomic locks provided by the GPU hardware. The GPU hardware provides atomic op-

Table I: Determining GPU blocks

GPU kernel	Number of blocks	Load on GPU
sort_particles	N_PARTICLES / (THREADS_PER_BLOCK * LOAD_PER_THREAD)	Uniform
compute_force	N_SUB_REGIONS	Non Uniform

erations for global and shared memories. Atomic operations in global memory have two major considerations:

1) High access latency for accessing global memory
2) The number of atomic locks can increase due to requests from all the blocks, resulting in more serialized execution of the code

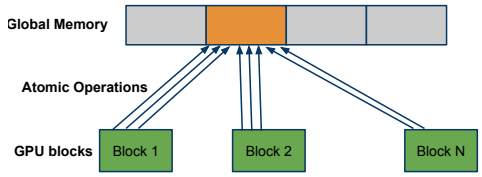

Figure 9: Atomic Updates in global memory

Atomic updates in global memory are depicted in Figure 9. We solve the drawbacks of global atomic operations by concentrating the atomic operations on shared memory. Threads in each block perform all atomic operations in the shared memory. After all the processing is completed for a particular block, we perform a single atomic update in the global memory as shown in Figure 10.

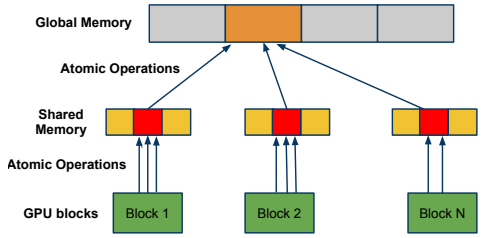

Figure 10: Atomic Updates in shared memory

Another approach to reduce the number of atomic updates is to color the triangles. The triangles in the mesh are considered as nodes in a graph and are colored such that the triangles with at-least one common vertex are of different colors. A GPU kernel works on only one particular color at a time. As seen from the Figure 11, kernel 1 processes the triangles and associated particles with color 1, kernel 2 processes those with color 2 etc.

Coloring eliminates the need for atomic updates as no two adjacent triangles are updated concurrently. This approach involves additional overhead for multiple kernel invocations for each color. For efficient indexing in a GPU, we also need to maintain a color sorted order of triangles and containing particles which involves additional computation. Due to

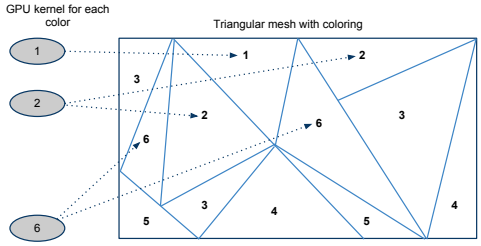

Figure 11: Mesh coloring and mapping to GPU blocks

coloring, some kernels can get less workload which results in under utilization of the GPU hardware.

As a result, for better performance, instead of coloring, we use the shadow triangle replication approach. We store the regions in shared memory for updating the entity count. Each region will be represented by an integer value in the shared memory which stores the number of entities in that region. Every thread in a block will compute the region for the assigned entities and updates (atomic) the integer counter in the shared memory. NVIDIA Tesla provides $16k$ of shared memory per SM. Shared memory also stores the arguments to kernel calls. Hence this approach is constrained by the amount of shared memory available for storing region counters.

For the rest of the discussion, we assume that the shared memory can hold integer counters for 1024 (using $32\ X\ 32$ rectangular grid) regions. Clearly this number will be larger, if the next generation machines have larger shared memory, and is smaller if they have smaller shared memory. We use 256 threads per GPU block and is abbreviated as TPB in the algorithms. We denote the atomic updates in shared memory and global memory using the abbreviations $s\_atom()$ and $g\_atom()$ respectively.

The mesh is partitioned into regions using Level 1 rectangular grid of fixed dimensions. There are two broad scenarios:

1) If the number of regions after partitioning of triangular mesh is within 1024, we proceed with the sorting, keeping all regions in the shared memory.
2) If the number of regions after partitioning the triangular mesh (because of its size), is larger than 1024, we use a two level grid for regions. We chose the Level 1 grid such that the partitioning creates only 1024 regions and proceed with the sorting. Then we use a Level 2 grid to partition each of the region into sub regions and perform a bucket sort so that entities will be in sub region sorted order. The number of GPU blocks that operate on each region is determined by the entity density of the region.

We use parallel bucket sort for ordering particles. The buckets in this case are regions created by the rectangular grid during partitioning of triangular mesh. As depicted in Algorithm 3 and 4, we use two GPU kernels to accomplish

Algorithm 3 Parallel Bucket Sort Kernel 1

```
1: regions array in Shared memory
2: particles array in Global memory
3: particleCount array in Global Memory
4: for all GPU blocks in parallel do
5:   thread = blockID * TPB + threadID
6:   for all threads in a block in parallel do
7:     particle = particles[thread]
8:     rIndex = particle.region
9:     s_atom(regions[rIndex], 1)
10:    barrier_synchronize()
11:    for i = 0 to total number of regions do
12:      gIndex = threadID + i * TPB
13:      newStart = g_atom (particle-
         Count[gIndex], regions[gIndex])
14:      regions[gIndex] = newStart
15:    end for
16:    barrier_synchronize()
17:    particle.position = s_atom
       (regions[rIndex], 1)
18:    particles[thread]=particle
19:  end for
20: end for
21: prefix_sum_gpu()
```

Algorithm 4 Parallel Bucket Sort Kernel 2

```
1: for all GPU blocks in parallel do
2:   for all threads in a block in parallel do
3:     thread = blockID * TPB + threadID
4:     particle = particles[thread]
5:     regionStart = get_region_start(
       particle.region)
6:     sortedParticles[particle.position
       + index] = particle
7:   end for
8: end for
```

Algorithm 5 compute_force

```
1: Load vertices and triangles in
   theregion to shared memory
2: barrier_synchronize()
3: for all threads in this block in parallel do
4:   for all particles in the region assigned to this thread
     do
5:     triangle = find_triangle(particle)
6:     s_atom (vertices, force)
7:   end for
8:   barrier_synchronize()
9:   g_atom (vertices, force)
10: end for
```

parallel bucket sort of particles:

1) Algorithm 3 outlines the kernel that finds the unique position of all the particles in the buckets (steps 1 - 20) and a GPU kernel for doing parallel prefix sum (step 21). Steps 7-9 compute the region contained by a particle and increments (atomic) the counter corresponding to that region in shared memory. Steps 11-14 write back the particle count to global memory. In step 13, we utilize an important property of atomic add in GPU. The return value of an atomic add is the value of the memory location before the atomic add. In step 14, we store that value in shared memory location corresponding to that region. Steps 17-18 use this information to identify the unique bucket position. Step 21 uses the standard parallel prefix sum algorithm for GPU to aggregate the particle counters of each region and compute the exact location to put the particle in the sorted array.
2) The kernel in Algorithm 4 puts the particle in the exact location computed from the Algorithm 3. After this kernel the particles will be sorted order according to the regions

Algorithm 6 move_particles

```
1: Load vertices and triangles in that
   region to shared memory
2: barrier_synchronize()
3: for all threads in this block in parallel do
4:   for all particles in the region assigned to this thread
     do
5:     triangle = get_triangle()
6:     get_force (triangle)
7:     move_particle()
8:   end for
9: end for
```

Algorithm 5 and Algorithm 6 depicts the GPU kernels for $compute\_force()$ and $move\_particles()$ respectively, which forms the most important steps in a simulation iteration. The $aggregate\_force()$ function performs a parallel atomic summation of the forces in shadow vertices. The function $incremental\_sort()$ for particles is explained in Section III-B2. As shown in Figure 12, there are two approaches to map the mesh regions to GPU blocks

1) Each region is mapped to multiple GPU blocks. The number of GPU blocks is determined by the particle density of the region. This ensures that all GPU blocks will be equally loaded. The disadvantage is that all GPU blocks in a region will have to load the triangles and vertices in that region to shared memory and do an atomic write back to the global memory.
2) Each region is mapped to a single GPU block. In this approach, each block loads the region entities to

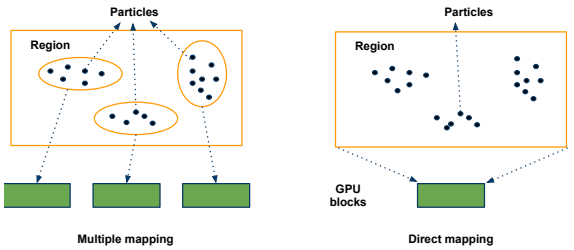

Figure 12: Region Mapped to multiple GPU blocks

the shared memory and writes back only once. The disadvantage is that the particle dense blocks will have more workload. This approach will be better when there is a uniform distribution of particles and triangles across the regions. Otherwise the runtime always depends on the densest mesh.

Choosing the number of blocks can depend on the type of memory access and the amount of the contention between the accesses. There can be two access patterns based on the kernel algorithm.

1) Independent Access: Memory access pattern is such that each block accesses independent global memory locations. A GPU kernel can be configured in two ways:
   a) Kernel with more workload per block and fewer number of blocks.
   b) Kernel with more blocks and less workload per block.

   The total number of global memory access is the same in both approaches. The second approach (as shown in Figure 13) is faster compared to the first as long as the number of blocks is within limits specified by the hardware. The reason is when the memory accesses remains constant, increase in the number of blocks will keep all the execution units in the hardware busy. The access pattern is demonstrated in Algorithm 7.
2) Conflicting Access: Memory access pattern in which many blocks access the same global memory location. This access can lead to more contention. Our experiment (as shown in Figure 13) shows that increasing workload per block gives better performance in this case. The reason is when the workload per block increases, the number of blocks decreases which in turn reduces contention from concurrent global accesses. The access pattern is demonstrated in Algorithm 8.

Another advantage with the replication of shadow triangles and vertices is the reduction of atomic updates in the global memory. As mentioned earlier, atomic updates in shared memory are very fast compared to atomic updates in global memory. Replication ensures each region can operate independently of the other. Each GPU block has its own copy of the triangles and vertices of the mesh that it needs

Algorithm 7 Scenario 1

```
1: for all GPU blocks in parallel do
2:   for all threads in a block in parallel do
3:     start= (blockID * TPB *
       DATA_PER_THREAD)
4:     for i = 0 to DATA_PER_THREAD do
5:       index = start + i * TPB +
         threadID;
6:     end for
7:   end for
8: end for
```

Algorithm 8 Scenario 2

```
1: for all GPU blocks in parallel do
2:   for all threads in a block in parallel do
3:     start = get_start()
4:     data_per_thread = n_elements_block
       / TPB
5:     for i = 0 to data_per_thread do
6:       index = start + i * TPB +
         threadID;
7:     end for
8:   end for
9: end for
```

for completing its computations. If replication was not there, $atomic\_force\_update()$ depicted in Algorithm 5 would be partly on shared memory and partly on global memory. This in turn may degrade the performance.

Before updating the force, each particle needs to identify the triangle in which it belongs. The is accomplished by the $find\_triangle(particle)$ step in Algorithm 5. This step can be a major bottleneck in the PIC algorithm as it involves a linear search over the triangles for each particle in a region. We use two strategies to reduce the search time:

1) Each GPU block stores the triangles associated with its region in the shared memory for faster access. All the threads in the block perform a linear search over these triangles. We use the shared memory access pattern with minimal bank conflicts.
2) Partitioning the triangular mesh into regions using a rectangular grid. Finding the region contained by a particle is straight forward and takes only constant number of operations. This limits the triangle search to only triangles in that region and reduces the search time.

As mentioned earlier, we are mapping a region to GPU block and the block loads the vertices and triangles in the region to shared memory. We have to ensure that the region is small enough so that the shared memory can accommodate the vertices and triangles. This creates another challenge, as the region gets smaller, the number of regions increases. GPU

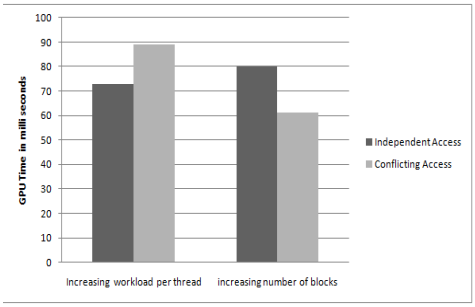

Figure 13: Comparison of blocking strategies

hardware imposes a limit on the number blocks per kernel invocation. In case of $find\_triangle(particle)$, we have to linearly scan through the list of the triangles in a given region. This function benefits if the region size is smaller as the number of triangles to scan reduces.

For solving this, we further partition the regions into sub regions. Sub regions are created based on the triangle density in a region. This will be explained in Section III-B. By doing so we can keep all the triangles to be linearly scanned in the shared memory. This ensures effective sharing of the mesh entities by all threads in a GPU block. As the sub regions increase, the number of GPU blocks will also increase. We have to find an optimal partitioning such that triangles can fit into the shared memory and the number of GPU blocks is within the limits imposed by the hardware. With these constraints in place, we explore different mesh decompostion strategies as described in Section III-B.

B. Mesh Decomposition

In this paper, we explore three different mesh partitioning strategies. The three approaches only differ in the mesh partitioning strategy used. The rest of the PIC algorithm remains the same.

1) Density based Partitioning: In this approach, we partition the mesh into regions such that each region has approximately the same triangle density (Figure 14). Uniform triangle density ensures effective load balancing across the regions. This approach requires slightly expensive pre-processing step to find the triangle density for creating regions. A KD tree [13] is created on all triangles in the mesh. The KD tree prunes the search space for triangles. Each particle traverses down the KD tree finds the triangle in which it belongs to. This approach requires an expensive pre-processing step for creating a KD tree. Moreover KD tree is an asymmetric data structure that cannot be mapped very well to the SIMT architecture of the GPU.

In next two approaches, we superimpose a rectangular grid over the triangular mesh as shown in the Figure 15. The rectangular grid partitions the triangular mesh into regions which can be solved independently. The pre-processing step is very fast compared to KD tree approach. One disadvantage

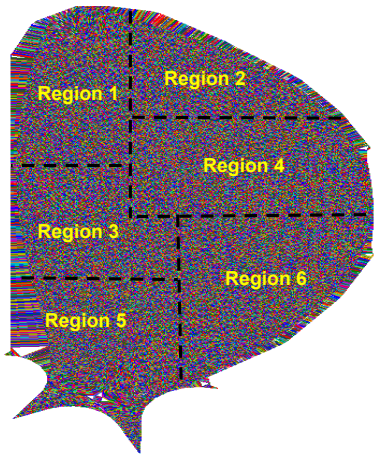

Figure 14: Partitioning based on triangle density

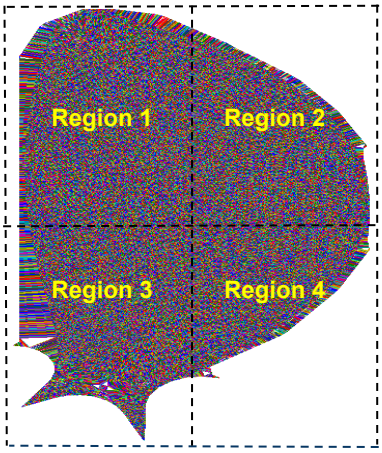

Figure 15: Partitioning using a rectangular mesh

with this approach is the triangle density varies between regions. For example in Figure 15, Region 1 is denser than Region 2. This results in some regions having more workload compared to other regions. We reduce the load imbalance by superimposing a second level grid over the rectangular grid which partitions the region into sub regions. We use similar data structures as described in Section III-A to store the mesh entities. Each of these data structures will be sorted according to the order of sub regions.

2) Uniform Partitioning: The level 2 grid partitions the level 1 regions into sub regions. The structure of the level 2 grid is uniform across the regions as shown in the Figure 16.

The additional grid granularity reduces the time for the linear search to identify which triangle a particle belong to. One disadvantage with this method is the upper limit on the number of blocks a kernel call can handle. Since each sub region maps to a block, we cannot increase the number of sub regions beyond a certain limit.

Due to symmetric nature of the partitioning grid, to

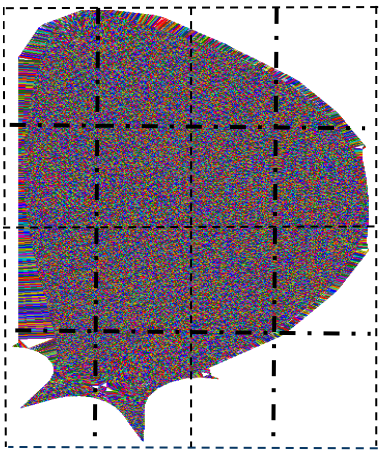

Figure 16: Uniform Partitioning using a rectangular grid

overcome the shared memory limitation, we use heuristics to incrementally sort the particles after each simulation iteration. In reality, most of the particles will move to adjacent regions. Hence we do not have to keep all the regions in the shared memory. The regions that are in a threshold neighborhood are kept in the shared memory and other regions are kept in the global memory. In this way, most of the updates will be in the shared memory and fewer updates in the global memory. As shown in Figure 17, while processing the particles in region X, we keep the neighboring regions marked as Y in the shared memory. All the other regions in the mesh will be in the global memory. The algorithm is depicted in Algorithm 9.

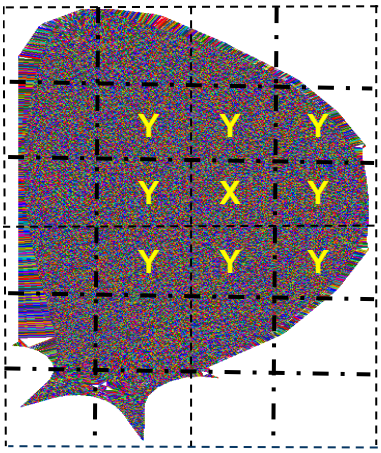

Figure 17: Partioning regions between shared and global memory for Incremental Sorting

3) Non-Uniform Partitioning: In this approach, the level 2 grid is not uniform across the regions. Figure 18 shows non uniform partitioning of the mesh. The level 2 grid is dense in the regions where triangle density is more and coarse in the regions where density is less. For non-uniform partitioning,

Algorithm 9 incremental_sort

```
1: for all blocks in parallel do
2:   threshold regions in shared memory.
3:   regions outside threshold in global
     memory.
4:   for all threads in this block in parallel do
5:     for all particles in the region assigned to this thread
       do
6:       newRegion = get_region(particle)
7:       if newRegion != region then
8:         if newRegion is within threshold then
9:           s_atom(newRegion)
10:        else
11:          g_atom(newRegion)
12:        end if
13:      end if
14:    end for
15:    barrier_synchronize()
16:    g_atom(threshold_regions)
17:  end for
18: end for
```

we define a threshold value for triangles. The threshold value is the maximum number of triangles that can be part of a sub region. Non-Uniform Partitioning reduces the effective number of sub regions and therefore GPU blocks. Like uniform partitioning the additional grid granularity reduces the time taken by $find\_triangle(particle)$ function. Non-uniformity creates asymmetry and requires more complex pre-processing and indexing. Due to asymmetric nature of mesh, applying the heuristics for incremental sort as in uniform case increases the complexity of code and can introduce conditionals, which we believe will reduce the performance.

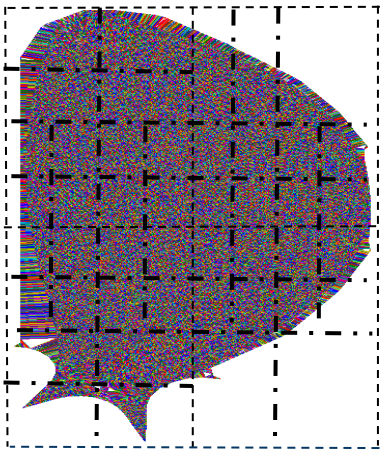

Figure 18: Non Uniform Partitioning using a rectangular grid

IV. Empirical evaluation

We used the triangular mesh from Oak Ridge National Laboratory, which is used for XGC1 [1] benchmarks.The mesh has 1.8 million triangles and represents a tokamak geometry with effects of the separatrix and x-point visible. We randomly distributed 18 Million particles on this mesh. For level 1 partitioning of the mesh, we used a $32\ X\ 32$ rectangular grid (1024 regions). Rest of the experimental section is organized as follows. In Section IV-A, we outline the hardware and software configuration of the benchmark machine. Section IV-B compares different particle sorting approaches. In Section IV-C, we study the best workload per thread for the particle sorting kernel. In Section IV-D, we compare the triangle search times in two mesh partitioning (uniform and non-uniform) approaches.

A. Benchmark Machine

The experiments are conducted with the following hardware and software configurations. We used an NVIDIA Tesla machine for benchmarking. The machine has a four core Intel Xeon Processor clocked at $2.8GHz$ and two NVIDIA Tesla T10 GPUs with a clock rate of $1.3GHz$. The machine runs on $64bit$ Linux Operating System. The GPU has $4GB$ global memory and 30 Streaming Multiprocessors (SM). Each SM has 8 Scalar Processors (SP) resulting in 240 computing cores. The NVIDA software stack uses version 3.0 of CUDA Driver and Runtime. The experiments used CUDA compute Capability mode 1.3. The compilation is done using NVIDIA nvcc compiler with optimization level 3 ($-O3$).

B. Comparison of different particle sorting schemes

As mentioned before, using bucket sort, we re-arrange particles based on sub-region sorted order. In Algorithm 3, we use an integer counter to count the particles in a region. In this experiment, we assume that the number of sub regions are so large that counters for all sub-regions cannot be kept in the shared memory. We use the non-uniform partitioning strategy (Section III-B3, Figure 18) for the level 2 grid. We compare two sorting strategies:

- Sorting without using shared memory: The integer counters corresponding to all the sub regions are kept in global memory. We do not use shared memory.
- Sorting using shared memory: In this approach sorting is done in two steps. The first step orders the particles based on level 1 grid regions. The number of level 1 regions are small and can be stored in shared memory. The second step works on each of these regions and sorts in sub region sorted order.

Experimental results are depicted in Table II. The $maximum\ number\ of\ triangles\ in\ a\ subregion$ is the threshold for level 2 partitioning i.e. level 2 grid is placed in such a way that each sub-region has at-most the threshold number of triangles. When shared memory is not used, the performance degrades as we increase the triangle threshold. As triangle threshold value increases, the number of level 2 regions decreases. The number of particles in a grid will increase resulting in more global memory contentions while updating the region counter. When we use fast shared memory, there is a $20X$ speedup (for threshold value of 5000) in the sorting time. The sorting time remains constant for the threshold values 10 to 5000. This reason being majority of atomic operations is on shared memory and is confined with in a block. Hence the execution of other blocks will not be affected because of atomic locks. We cannot increase the threshold beyond a certain limit because the time to search during $find\_triangle(particle)$ step will increase proportionally. Also, we will not be able to keep all triangles in shared memory for faster searching and force updation.

Table II: Comparison of Particle Sorting Schemes

Maximum number of triangles in a sub region	**Sorting using shared memory (Time in ms)**	**Sorting without using shared memory (Time in ms)**
10	185.68	1492.55
200	183.88	1881.56
1000	183.55	2441.08
5000	183.99	3773.92

C. Choosing the best workload per thread

In this experiment, we study the effect of increasing the workload of threads in a block. We use the parallel bucket sort algorithm (Algorithm 3) for this study and increase the number of particles assigned to a thread. As shown in table III, when the workload of threads increases, the number of GPU blocks will reduce. When GPU blocks are less, we do not have enough blocks to keep the computing units in GPU busy and this leads to relatively poor performance. It is also worth noting, that when the number of particles per thread reaches 256, there is a huge deterioration in performance. The reason is an access width of 256 between threads can result in resulting in large number of cache misses which results in the performance penalty. For parallel bucket sort, the best performance is achieved when each thread processes 4 particles.

D. Comparison of triangle search times

Table IV and Table V shows the triangle search times for uniform and non-uniform partitioning respectively. Both the tables show search time for ten iterations. Non-uniform partitioning with fewer number of blocks scheme has comparable performance with uniform partitioning with more number of blocks. The reason is, even though there can be some blocks having less workload in uniform partitioning, the automatic load balancing provided by the GPU hardware

Table III: Comparison of sorting time with variable workload per thread

Number of GPU blocks	Particles Per Thread	Time (ms)
275	256	95
276	255	65.07
314	224	63.89
550	128	65.79
1099	64	61.13
4395	16	60.73
17579	4	60.49
35157	2	61.49

takes care of the load imbalance. Hence the simpler uniform partitioning would be a better choice as it can also support the heuristics for incremental sorting. In general, uniform partitioning will be a better methodology for automatic code generators even when the underlying computational grid is highly asymmetric.

GPU blocks	Time (ms)
4096	3111.11
9216	1366.21
16384	877.23
25600	609
36864	500.92
50176	427

Table IV: Uniform Partitioning

GPU blocks	Time (ms)
1024	12561.06
2779	7235.16
22471	989.88
33464	428.51

Table V: Non-Uniform Partitioning

V. Conclusion

The parallelization of PIC problems requires careful partitioning of particle and mesh domains to balance the computational load and minimize communication cost. We explored the parallelization of PIC algorithm on an asymmetric structure like triangular mesh. We discussed various methodologies to explore the capabilities of the GPU hardware for PIC algorithm. The paper also provides an insight into the decisions to be made before parallelizing an application on the GPU. Our experimental results show that the use of shared memory is critical for optimal performance of any algorithm on GPU. We introduced the concept of replication which is a simpler solution than graph coloring that involves the overhead of multiple GPU kernel invocations for each color and ordering based on color. The use of replication for shadow triangles and vertices does not deteriorate performance during force aggregation step. In our experiment, with the optimal configuration of parameters, each simulation iteration takes $82ms$. The aggregation of force from the vertices and shadow vertices takes $3ms$ which contributes only 3.5% to the total simulation time. The algorithms we discussed are scalable with size of mesh and number of particles and can easily be ported to execute on multiple GPUs.

References

[1] Adams MF, Ku S, Worley P, et al., *Scaling to 150K cores: Recent algorithm and performance engineering developments enabling XGC1 to run at scale*, Journal of Physics: Conference Series, 2009.

[2] Kraeva MA, Malyshkin VE, *Algorithms of Parallel Realisation of the PIC Method with Assembly Technology*, Lecture Notes in Computer Science, 1999, Volume 1593/1999, 329-338.

[3] Kraeva MA, Malyshkin VE, *Implementation of PIC Method on MIMD Multicomputers with Assembly Technology*, in: Proceedings of the (HPCN) High Performance Computing and Networking International Conference, Europe, Lecture Notes in Computer Science, Vol. 1255, Springer, Berlin, 1997, pp. 541-549.

[4] Liao W, Ou C, Ranka S, *Dynamic Alignment and Distribution of Irregularly Coupled Data Arrays for Scalable Parallelization of Particle-in-Cell Problems*, ipps, pp.57, 10th International Parallel Processing Symposium (IPPS '96), 1996.

[5] Madduri K, Williams S, Ethier S et al., *Memory-efficient optimization of Gyrokinetic particle-to-grid interpolation for multicore processors*, SIAM Conference on Parallel Processing for Scientific Computing (SIAM PP10), 2010.

[6] Stantchev G, Dorland W, Gumerov, *Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU*, Journal of Parallel and Distributed Computing, Volume 68, Issue 10, October 2008, Pages 1339-1349.

[7] Ryoo S, Rodrigues CI, Baghsorkhi SS et al., *Optimization principles and application performance evaluation of a multithreaded GPU using CUDA.*, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, PPoPP'08.

[8] Ji Qiang, Robert D. Ryne, Salman Habib and Viktor Decyk, *Object-Oriented Parallel Particle-in-Cell Code for Beam Dynamics Simulation in Linear Accelerators*, sc, pp.55, Proceedings of the 1999 ACM/IEEE conference on Supercomputing, 1999.

[9] Viktor K. Decyk, Tajendra V. Singh and Scott A. Friedman, *GRAPHICAL PROCESSING UNIT-BASED PARTICLE-IN-CELL SIMULATIONS*, Proc. Intl. Computational Accelerator Physics Conf. (ICAP2009), San Francisco, CA, Sept, 2009.

[10] V. K. Decyk, *UPIC: A framework for massively parallel particle-in-cell codes*, Computer Physics Communications, Volume 177, Issues 1-2, July 2007, Pages 95-97 Proceedings of the Conference on Computational Physics 2006 - CCP 2006, Conference on Computational Physics 2006.

[11] NVIDIA, *CUDA C Programming Guide*, Version 3.1.

[12] NVIDIA, *CUDA C Best Practices Guide*, Version 3.1.

[13] KD tree, www.en.wikipedia.org/wiki/Kd_tree

[14] CUDA, www.nvidia.com/object/what_is_cuda_new.html