Parallel implementation of Particle-Mesh mapping using CUDA enabled GPUs

Brijesh Kumar

Dhirubhai Ambani Institute of Information and Communication Technology, Gandhinagar – 382007, Gujarat

201101102@daiict.ac.in

Supervisor

Prof. Bhaskar Chaudhury

***Abstract* — This paper describe about how to use the GPU for mapping scattered distribution of particle to discrete grid. Mapping particle to grid reduces the compute complexity, without changing the effect particle contribution in a scattered distribution. For calculating this mapping an algorithm is designed and tested.**

Algorithm is implemented on CPU and a CUDA enabled GPU based on Serial and parallel implementation of algorithm. Paper also describe about how the throughput can further be increased using dynamic implementation.

***Keywords* — CUDA, HPC, heterogeneous computing.**

I. Introduction

Particle to grid/mesh interpolation is an important technique which is used in several computational problems, such as the kinetic simulation of plasmas using Particle in cell algorithms. This interpolation procedure is computationally very expensive when the number of particle is more than few millions.

If we use CPU to compute the contribution of individual particle in a grid then, computational overhead increases exponentially with increases in number of particle. Which is majorly needed of concern in research area.

Simulating randomly distributed nodes (of variable magnitude) contains the fractional value for particle position. Thus calculation related to doubly precision values further increases the computational overhead. If we map the contribution of nodes to corresponding coordinates then computation related to coordinated will be better, as the coordinates of map will minimise the number of double precision registers need for operation.

For example in electrostatic according to gauss law electric field passing through a closed loop is dependent on charge particle in enclosed surface.

$$\nabla \cdot \mathbf{E} = \rho/\varepsilon_0$$

Where $\nabla \cdot E$ is divergence of electric field, ρ is electric charged density, and ε_0 is electric constant. Therefore to calculate E we need charge density or contribution of individual particles.

Now according to Lorentz equation, electric force experiences by a positive charged particle is directly proportional to Electric field applied on it.

$$F = q\mathbf{E}$$

Where F is the force experience by particle with charge 'q' in the presence of electric field 'E'.

Once force applied in a body with mass, body will gain momentum, hence laws of electrodynamics will start acting. Mathematical calculation of electrodynamics, based on charged particle will be computationally expensive as there is huge number of charge particle acting simultaneously. But mapping from scattered charged particle to discrete grid, will reduce the complexity without changing the overall resultant effect of charged particles.

This mapping also have application in real life scenario like government policy implementation. For country with dense population it is difficult to point each individual on a geographical diagram. But if we divide area of country into a grid. Then we can map the population to the corresponding grid. Representation in grid form will be easier to interpolate as number of coordinate will be lesser then the overall population of country.

Comprehending information like rate of gradient change in population density will be easier to interpolate form this new information. Thus for government taking decision (such as budget allocation and medicine distribution) would be easier and more beneficial for the development of country.

Problem of type SIMD (Single instruction multiple data) are those where fixed set of computational intensive algorithm is needed to repeat on every data. If set of data are independent from other data, then there execution can be parallelised.

Serial implementation for processing the data in my algorithm takes 10 flop instruction per particle. As mentioned in Fig 1 with increase in number of particles, total flop increase linearly. Resulting in a higher CPU overhead. If we shift this repetitive computation from CPU to GPU then overhead can be eliminated, hence this hybrid form of computing will result in higher throughput of system.

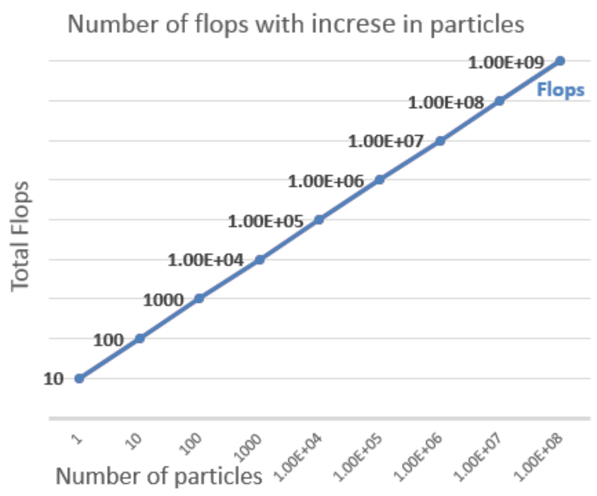


Fig. 1 Number of flop with increase in number of particles

II. Hardware Description

I have used Intel i5 for serial implementation and for parallel implementation. For parallel implementation I have used two machine, first half of project I worked on NVidia GT540 M based on Fermi architecture. Later during second half of project I got remote access to latest NVidia K40 which was based on Kepler architecture.

A. *Intel i5 Serial Board*

TABLE I
Serial hardware configuration

No	Intel i5	
	Property	**Details**
1	Cache	3.0MB
2	Clock Speed	2.90 GHz
3	Number of cores	2
4	Number of threads	4
5	Thermal design power	28 W
6	Max memory bandwidth	25.6GB/S

B. *Fermi and Kepler Parallel boards*

TABLE III
Parallel Hardware configuration

	CUDA machine		
No	**Property**	**Fermi**	**Kepler**
1	Card	GT540	K40
2	Compute capability	2.1	3.5
3	Thread / Warp	32	32
4	Max warp / Multiprocessor	48	64
5	Max Thread / Multiprocessor	1536	2048
6	Max Thread Block / Multiprocessor	8	16
7	32-bit Registers / Multiprocessor	32768	65536
8	Maximum registers / Thread	63	255
9	Max Thread / Thread Block	1024	1024
10	Shared Memory	48KB	48KB
11	Multiprocessor Count	2	15

NVidia CUDA enabled Fermi architecture card have compute capacity of 2.1. Kepler architecture is the next generation of Fermi with compute capability of 3.5. Kepler board contain more number of cores and have larger global memory. As the number of cores increases the cost of power consumption to compute the process decreases, making it 3 time more efficient than previous Fermi board.

III. CUDA Properties

A. *Architecture*

CUDA enabled machine works similar to Von Neumann architecture. It contains memory, I/O, program counter, registers, arithmetic logical unit, processing unit and bus. The only difference is it has multiple number of processing unit with independent arithmetic logical unit, registers and program counter.

B. *Functionality*

GPU can only be launched by CPU. CPU act as host and GPU act as device. In this form of heterogeneous computing CPU initiate the kernel and launch it into GPU. CPU core have faster clock thus it can compute large flop instruction per second, as compare to single GPU core. This make CPU more efficient for doing computation for smaller number. But as in case of GPU has more number of core, it have higher throughput as compare to CPU. Which enables GPU to complete bunch of instruction in a single instruction cycle.

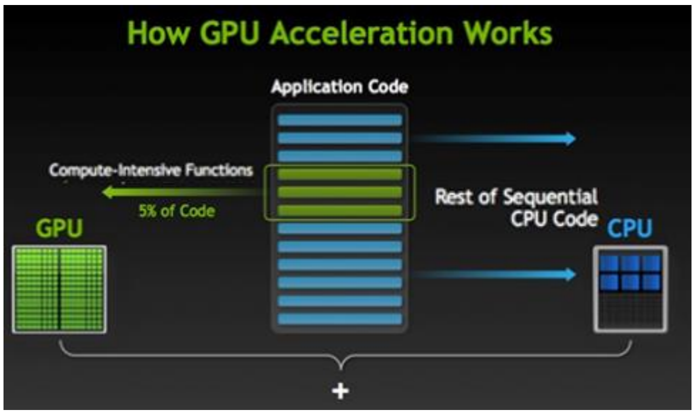

How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
GPU
Rest of Sequential CPU Code
CPU
+

Fig. 2 CUDA functionality.

C. Memory hierarchy

Similar to L1, L2 and L3 cache memory architecture. GPU has different kind of memory. Similar to CPU memory access time and memory size within GPU increases as we go from Registers – Shared memory – Global memory. Registers are accessible within individual processors. Shared memory is visible within a group of processors called block. Whereas global memory is visible and accessible by every process.

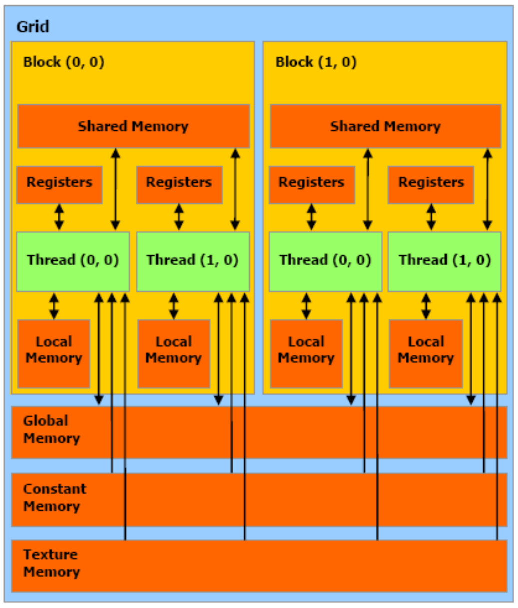


Fig. 3 CUDA memory architecture and accessibility.

D. Compute capability

NVidia has launched various architecture for parallel processing till now. Every successive architecture contains include the new improves functionality. Compute capability of a machine define its property and types of function it can handle.

IV. Serial Implementation

A. Algorithm

First I initialised the particle with random position within a defined grid. Each particle is assumed to carry value equal to unit magnitude. Once the particle are initialised they contain information of their magnitude and position in 2D domain.

Once the particle are initialised I have defined a grid of unit length in 2D plane covering the whole canvas of particle. Every coordinate of grid is initialised with value zero.

For every particle I took its coordinate and stored its x any y position in a float variable. From this information I located its position in grid and choose the four coordinate in grid surrounding the particle (named left, right, top and bottom). Later I defined four integer values containing information of two x position in particle left side and right side. I have also defined two y position which tells top and bottom line of particle in grid.

```
//initialise array of particle
   foreach particle p(i) in list
            p(i).x = random position in grid;
            p(i).y = random position is grid;
   end
//Initialising grid
   foreach position in grid
            value of coordinate = 0;
   end
```

Algorithm 1: Initialising particle with random position in grid.

Once I know the corresponding x and y positions in grid. I can create the coordinate pair which will make the corner of grid made by point A, B, C and D in Fig. 4. This can be done by using (left, top), (right, top), (right, bottom) and (left, bottom). I subtracted the float value of particle position with the integer type value of grid coordinate. Once we repeat the process for all coordinate. I get four float type value named fL, fR, fT and fB which tells the fractional distance from left, right, top and bottom. These set of fractional value gives exact information about the location of particle within grid block.

Now once I know the position, I calculated the contribution of particle on a point based on the ratio of area covered by diagonally opposite side of corner, to the total area of grid. Multiplying this fraction with the magnitude of particle will give me the exact contribution of particle in a corner. Once we get the value we can update the main grid. Then I repeated this process for all particle.

```
//Mapping particle to grid position
  foreach particle p(i) in list
    left = integer part of p(i).x ;        right = left+1;
    bottom = integer part of p(i).y ;      top = bottom+1;

    fL = p(i).x – left ;                   fR = 1-fL;
    fB = p(i).y-bottom ;                   fT = 1-fB ;

    grid( left, bottom)  = (own value) + ( fT x fR );
    grid( right, bottom) = (own value) + ( fT x fL );
    grid( left, top)     = (own value) + ( fB x fR );
    grid( left, bottom)  = (own value) + ( fB x fL);

end
```

Algorithm 2: Mapping particle to grid

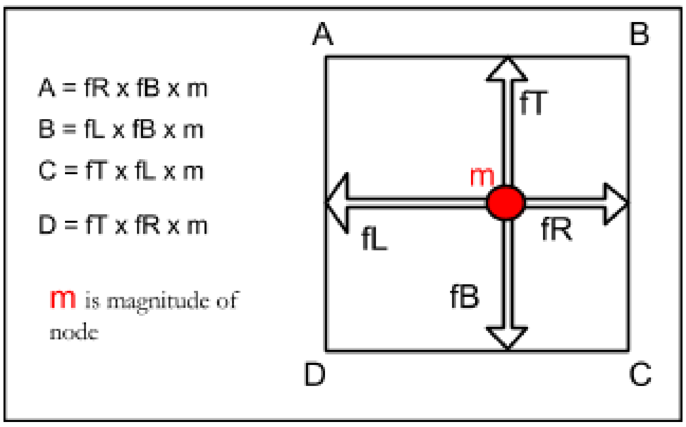


Fig. 4 Algorithm design for serial implementation.

Once all particle are mapped. I divided the every coordinate of grid with the theoretical average value in case of even distribution of particle. This gives me the normalised result telling about the degree of deviation of particle in grid.

Once the result are generated on server. I stored the information into a text file, where grid value were separated by comma. Then to plot the file I imported the resulted text file into excel sheet and drew the corresponding graphs.

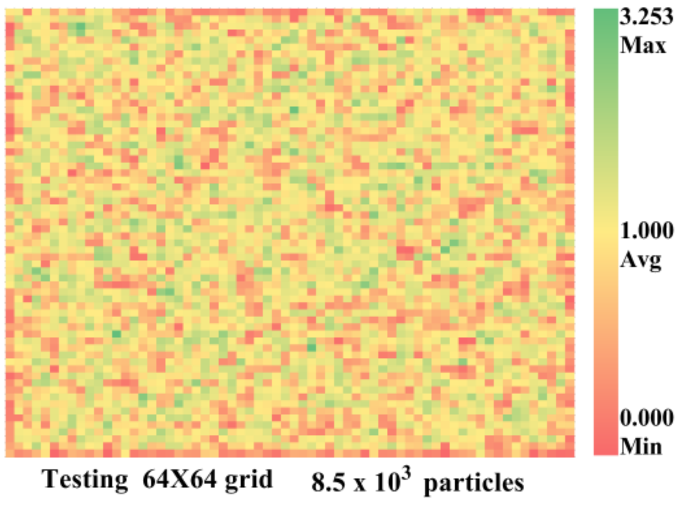
B. Testing algorithm

To test the algorithm I have taken a fixed grid of size 64 x64. Keeping the grid side fixed I have increased the number of particles from 10^3, 10^4 and 10^6. Once the particle are mapped to grid I normalised every node with theoretical average node value, in case of uniform distribution.

For smaller number of particle node value sharply diverge from average value in a large range (as depicted by column coloured scale along with figure). Thus in sparse particle distribution image is pixel value vary sharply. As the number of particle increases the degree of divergence from mean value reduces to smaller range as compare to space particle distribution. In case of large number of particle rage of divergence is smallest and node value of grid is close to the mean value. Thus in dense particle distribution the map is even and pixel colour are near to average value pixel.

```
//Calculating divergence of node in grid from average value
Variable average = ( total particle ) / (total grid coordinates);
   foreach position in grid
      node value   = (own value) / (average);
   end
// if ratio > 1 means node value is more than average;
// if ratio < 1 means node value is less than average;
```

Algorithm 3: Calculating divergence of coordinate in grid from average value



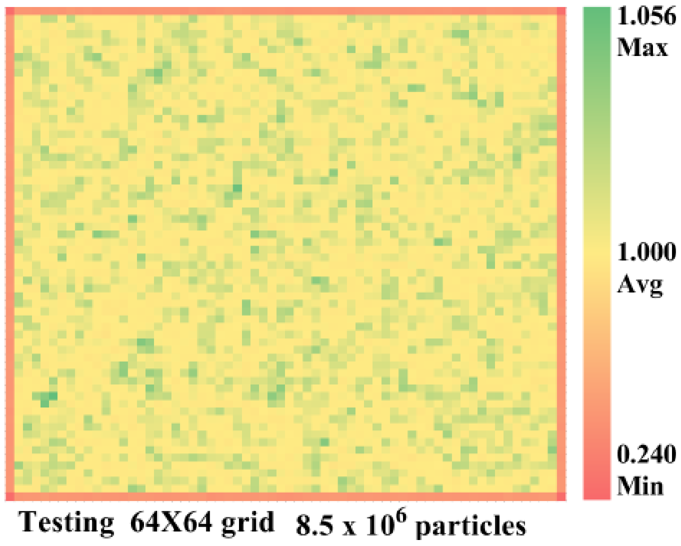


Fig. 5 Normalised visual representation of resultant mapping for various number of particles. Colour scale represent the divergence of node value from theoretical average value of node.

C. Memory overflow

As number of particle increases more than 10^9 the memory becomes the limiting factor. The available fundamental data types has maximum limit. To resolve this issue I used the concept of dynamic programming and started Using address location to use as variables, rather than declaring a variable. This reduces overhead as no pointer to keep track of next memory location is need. Thus working on a consecutive set of memory location and directly updating value for every major iteration resolve my issue.

V. Parallel Implementation

A. *Kernel launch*

For parallel implementation in code, first separate variable for host and device are declared. Then I have initialized the host variable according to 'Algorithm 1'. Once value are initialized. Using 'cudaMalloc' I allocated memory space equivalent to size of data to be processed within GPU memory. After that I copied the information from host to device memory using 'cudaMemcpy' function. As the number of particle huge, I braked the number of particle in a group of 32 particle Block. Then I created a group of blocks called grid.

Since Fermi board are capable to handling 32 threads a time on each of its 5 SM (Streaming Multiprocessor). It can support 5 blocks at a time. On the other hand kepler board can support 192 threads on each of its 15 SM. Hence it can support 6 block on each SM. Thus in total 90 blocks can run at a time in kepler board.

While defining the kernel, I declared the kernel in global memory. Then launched it for each individual particle. Within kernel I take in parameters like, pointer to particle, pointer to grid and size of grid as parameter.
I configured kernel to run according to 'Algorithm 4'.

```
// Algorithm to launch the kernel from CPU
Int main()
{
       Variable hostGrid, hostParticle;
       Variable deviceGrid, deviceParticle;
// initialize host variable using Algorithm 1.
       cudaMalloc( space in GPU for device variable);
       cudaMemcpy(from host to device);
// define block dimension
      Dim3 bD;   //block dimension;
      Dim 3 gD;   //grid dimension;
// bD x gD = total number of particles;
      Launch kernel<< bD, gD>>(parameters);
      cudaMemcpy(from device to host);
      Store data into file;
}
```

Algorithm 4: Kernel launch

```
// Algorithm to describing the kernel
__global__ void parMap(float *pD, float *netD, int grid)
{
       unsigned int rID= blockDim.x*blockIdx.x + threadIdx.x;
       int left, right, top, bottom;
       float x,y, fL,fR,fB,fT;
   // Algorithm 2
}
```

Algorithm 5: Kernel definition

B. *Issue faces due to parallelisation*

As threads are running parallel for every particle. Their execution is independent of each other. Thus particle present in neighbor of common coordinate, tries to update the value at same time. It create issue of collision and result in overwrite the value of coordinate in a grid. This scenario is depicted in figure 6.

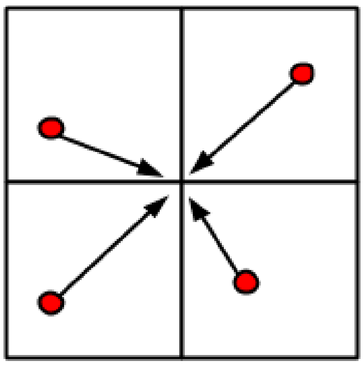


Fig. 6 Collision while writing the data into grid.

C. *Solution for collision avoidance*

To resolve the collision I have configured the kernel to write the value in anticlockwise direction order as shown in Fig 7. This allows each thread to access coordinate one at a time. Later when function run in synchronization then parallel threads avoided the over write condition.

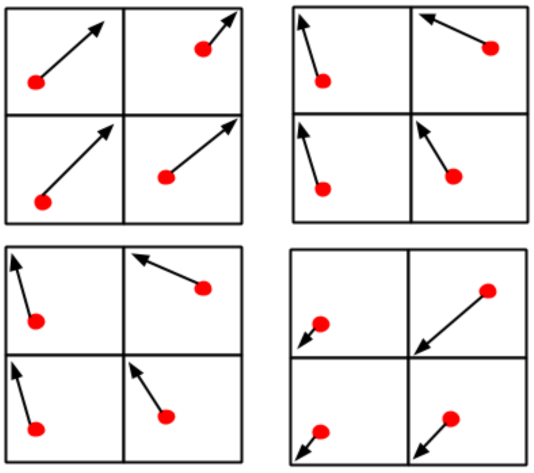


Fig. 7 Four state of cyclic write to avoid the memory overwrite.

VI. Result

Following result are for grid size of 1024 x 1024. I have varied the number of particle and compared the processing time.

TABLE IV

Number of particles	Timing		
	CPU	**GPU**	**Ratio**
256 K	13.1	0.7	18.7
512 K	21.9	1.3	16.8
1 M	42.1	2.7	15.6
2 M	77.2	5.3	14.5
4 M	151.1	11.2	13.4
8 M	297.2	24.1	12.3
16 M	589.6	51.2	11.5

TABLE IV performance comparison of particle to mesh mapping for a fixed grid. Particle are given in power of 2, i.e. 1K = 1024; 1M = 1048576.

VII. Future work on dynamic parallelism

To further increase the performance of mapping algorithm we can use the dynamic parallelism. To implement that I have created the local variables within each grid and stored it into the shared memory. In this new approach memory access time of global memory reduces. Thus reducing the overheads.

In this approach ever coordinate of grid is divided into four different sections. Ever section is updated individually by the running threads. Once the processing is completed value of all four sub section is added and stored into the coordinate they represent.

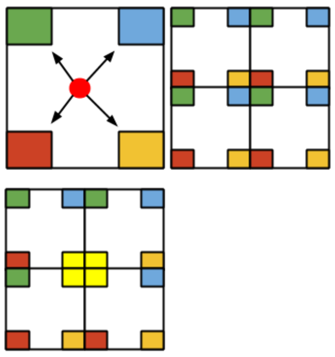


Fig. 8 creating 4 variables for each side of individual grid block.

Recent release of cuda 3.5 kepler architecture. It provide the functionality of dynamically launching the core from the GPU. According to this approach CPU can launch single kernel and remaining kernel can be launched by the first kernel. This reduce the waiting time of GPU and improves the performance.

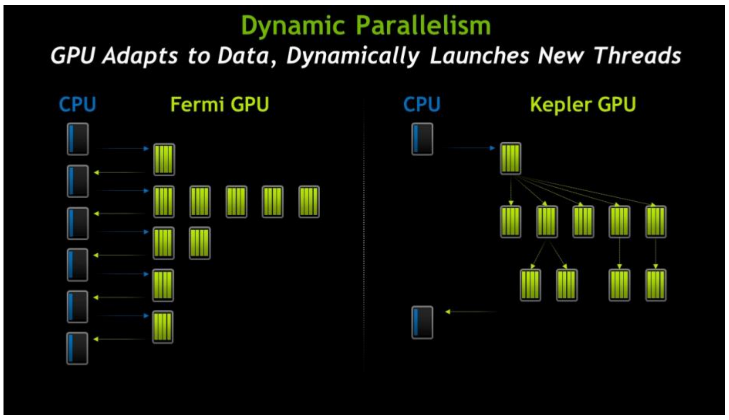


Fig. 9 dynamically launching kernel from GPU.

VIII. Challenges faced

While working on project I learned how to work with MobaXtrem. Debugging code was the toughest part. As compare to GDB debugger in CPU, there is no debugger for GPU. Thus finding point of error was difficult. I learned how we can design our own header files as per requirement. Another challenge was to drawing the visual presentation of result. CUDA is still in development phase hence most of the basic function available in CPU are not be available in GPU. Working with huge number of particle in a limited memory space was difficult.

IX. Conclusions

Parallel GPU based implementation of particle to mesh interpolation increases the performance of the method by an order of magnitude compared to serial implantation. However several precautions (such as data collision) must be taken for accurate result. Further improvement in the speedup is possible with the newer version of GPU which allow the functionality to process and transfer data within memory simultaneously.

Acknowledgment

I would like to thanks my guide Prof. Bhaskar Chaudhury for his clear guidance. Without his moral and intellectual support this work was impossible for me. I am grateful to him for providing me access to Georgia state university and DAIICT server.

References

[1] Shane Cook Book 'CUDA programming , a developer guide'.
[2] David B. Kirk, book 'Programming massively with parallel processor'
[3] F. Buyukkececi, O. Awile, I.F. Sbalzarini, A portable openCL implementation of generic particle-mesh and mesh-particle interpolation in 2D and 3D, sciencedirect 2013
[4] G.Stantchev, W. Dorland,N.Gumerov, Fast parallel particle-to-grid interpolation for plasma PIC simulation on the GPU, J.Parallel Distrib,comput (2008)