Optimization of the N-body Problem using the Barnes-Hut Algorithm

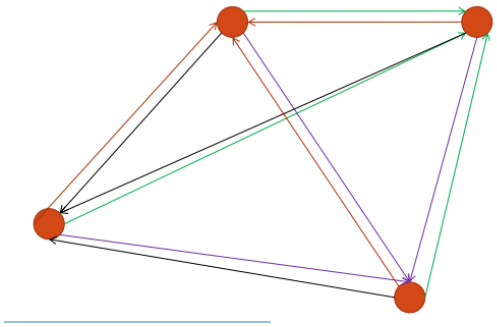
1 Project Description

The N-body Problem

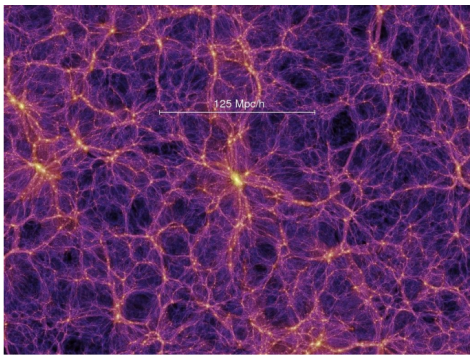
In physics and astronomy, an N-body simulation is a simulation of a dynamical system of particles, usually under the influence of physical forces, such as gravity. Direct N-body simulations are used to study the dynamical evolution of star clusters.

Beyond gravitational masses, a variety of physical systems can be modeled by the interaction of N particles, e.g., atoms or ions under electrostatics and van der Waals forces lead to molecular dynamics. Also, the integral formulation of problems modeled by elliptic partial differential equations leads to numerical integration having the same form, computationally, as an N-body interaction. In this way, Nbody algorithms are applicable to acoustics, electromagnetics, and fluid dynamics. Adding to this diversity of applications, radiosity algorithms for global illumination problems in computer graphics also benefit from N-body methods.

As shown in figure 1.0 a small system of four bodies on the left, has 12 force interactions between the bodies. This problem can increase drastically in terms of computations in the case of say a billion bodies as shown in the right part of the figure.



Simple n-body scenario with n=4 bodies



A not so simple N body simulation

Figure 1.0

N-body problems require a numerical approach. The direct simulation of the all-pairs interaction results in a computational complexity of O(n^2), which becomes too expensive to compute for large N. Nevertheless, the simplicity of direct integration admits ease of use of hardware accelerators leading to a prominent branch of research in this area. In this project we implemented the Barnes-Hut algorithm to calculate the final positions of the bodies. The complexity of this algorithm is O($n\ log\ n$), as explained in later sections. This reflects on their excellent performance on GPU hardware.

The Equation

The formula used to calculate the force on the bodies is based on Newtonian Gravitation as given below.

$$\vec{F_i} = -\sum_{j \neq i} \frac{G m_i m_j (r_i - r_j)}{(|r_i - r_j|^2 + \varepsilon^2)^{\frac{3}{2}}}$$

Where…..

$\mathbf{F_i}$ - Force on particle i
$\mathbf{m_i}$ – Mass of particle i
$\mathbf{m_j}$ – Mass of particle j
$\mathbf{r_i}$ – Direction vector for particle i
$\mathbf{r_j}$ – Direction vector for particle j
$\boldsymbol{\varepsilon}$ - Softening factor

*Assuming that other fundamental forces of interaction do not influence the system as much as gravity.

ε here is the softening error. This is necessary because it limits the maximum force between two bodies. When two bodies are near the distance between them can be small and will make the force very large. By placing the constant we cap the maximum force between the bodies so that we can get an acceptable value of the force.

2 SERIAL IMPLEMENTATION : CPU BASELINE VERSION

The serial implementation of the N-body simulation is straight-forward. As mentioned above the force on the body is calculated as sum of its interactions with all the other bodies.

An array of structures stores the data of mass and position and initial velocities for each of the n bodies. This data is initialized randomly but the masses of the bodies are kept constant for simplicity of understanding and the fact that, even though they were different, they would not change throughout the life of the program.

The program then iteratively takes each body and calculates the force and therefore the acceleration, velocity and position using a constant time step δt. The new velocity and positions obtained are updated back to the memory.

This process is repeated for the given number of iterations or time-steps. The time for each iteration is calculated.

Pseudo Code

```
Declare structure of Array
Define Nbodies
      Initialize/Randomize Values
      Foreach (iteration i)
            Start time
            Call to function bodyforce
                  Foreach (body)
                        For (every other body)
                              Calculate δx, δy, δz from position value of
                              other body
                              Calculate acceleration using the force
                              formula
                              Calculate and update velocity value of
                              current body
                  Foreach (body)
                              Update position
            End fuction bodyforce
            End time
            Calculate iteration time Start time – End time
            Display time of each iteration
      End iteration
End
```

3 RELATED WORK

The N-body problem being inherently computationally expensive, optimizing it has been the subject of extensive research. A few algorithmic techniques have been proposed to reduce the computational complexity of the calculations. Some of them are outlined below. A comparison of these has been done based on the floating point operations done by each. This is because measuring floating point operations is fairly representative of performance amidst several implementations. The performance analysis of these algorithms has been elaborated in [2]. The authors have used NESL as their programming language. Here we give a brief description of the Algorithms.

1. ***BARNES-HUT ALGORITHM:***

   The Barnes-Hut algorithm is based on a hierarchical octree representation of space in three dimensions. The algorithm has two phases. The first phase consists of constructing the octree by recursively subdividing the root cell containing all the particles into eight cubical subcells of equal size, until each subcell has at most one particle. Each cell contains the total mass and the position of the center of mass of all the particles in the subtree under it.

   In the second phase, the tree is traversed once per particle to compute the net force acting on it. We start at the root, and at each step, if the cell is well separated from the particle, we use the center of mass approximation to compute the force on the particle due to the entire subtree under that cell. Otherwise, each of its subcells is visited. A cell is considered well separated from a particle if its size, divided by the distance of its center of mass from the particle, is smaller than a parameter δ, which controls accuracy. This algorithm has a complexity of O(*n log n*) compared to the all pair approach which has a complexity of O(n^2).

2. ***FAST MULTI-POLE METHOD:***

   The Fast Multipole Method (FMM) uses an octree similar to that of the Barnes-Hut algorithm. The uniform version builds a balanced octree. It distributes the particles into leaf cells, and computes their multipole expansions, followed by a bottom-up phase in which it constructs the multipole expansions of the parent cells by shifting and adding the expansions of its children.
   After the tree is built, it has a top-down phase in which the local expansion of the parent cell (which describes the potential field due to distant particles) is shifted to the center of each child, and added to the multipole expansions of the cells in the child's interaction list to form its local expansion.

   Finally, the local expansions at the leaf cells, along with direct interactions with particles in neighboring cells gives us the total force on each particle. The number of terms in the multipole expansions, p, controls the accuracy of the algorithm.

   The primary difference between the FMM and the Barnes-Hut lies in the fact that the Barnes-Hut algorithm computes particle-cell interactions, whereas the FMM computes cell-cell interactions, thereby reducing its complexity.

3. *PARALLEL MULTI-POLE TREE ALGORITHM:*

The PMTA is a hybrid of the Barnes-Hut and the FMM algorithms. It uses a rule similar to that of Barnes-Hut to determine the well-separated-ness of two cells. Two cells are said to be well-separated from each other if the size of the bigger cell divided by the distance between the two cells is less than the parameter α.

The tree is built as in the Barnes-Hut method, but a cell is recursively subdivided until it contains no more than m particles (instead of one particle as in the case of the Barnes-Hut algorithm). Then the tree is traversed top down for each leaf cell, and when a cell is found to be well-separated from the leaf cell, its multipole expansion is translated into a local expansion about the center of the leaf cell, and the rest of the subtree below that cell is not visited.

All these translated local expansions are added and the gradient is found to get the force due to the far field on every particle in the leaf. The particles in the leaf cell interact directly with the particles in all the leaf cells that are not well separated from it. The number of terms in the multipole expansion, p, and the separation parameter a can both be varied to control accuracy. A theoretical error bound for this algorithm is not known.

The aim of our project is to implement the Barnes-Hut algorithm using CUDA and to take advantage of the obvious parallelism seen in the N body problem. Furthermore a comparison of performance of the serial all-pair N body implementation, it's naïve parallel implementation using shared memory and the Barnes-Hut implementation has been presented in the following sections.

4 PARALLEL IMPLEMENTATION

4.1 NAIVE IMPLEMENTATION

In this implementation we have used block-size shared memory. We parallelize the body-force function used in the previous serial implementation. Each block reads block-size of position data from global memory and stores it in the shared memory. Threads of each block are then synchronized using __syncthreads().

Each thread then calculates the change in the force/acceleration due to the interaction with all other bodies and updates this change in its unique registers. The threads are again synchronized before calculating and updating the global velocity array using the accumulated force/acceleration from the registers. This is then used to update the position array based on the dt which is the constant time-step. The positions are then updated accordingly. This implementation though parallelizes the force calculation, it does so with an O(n^2) complexity.

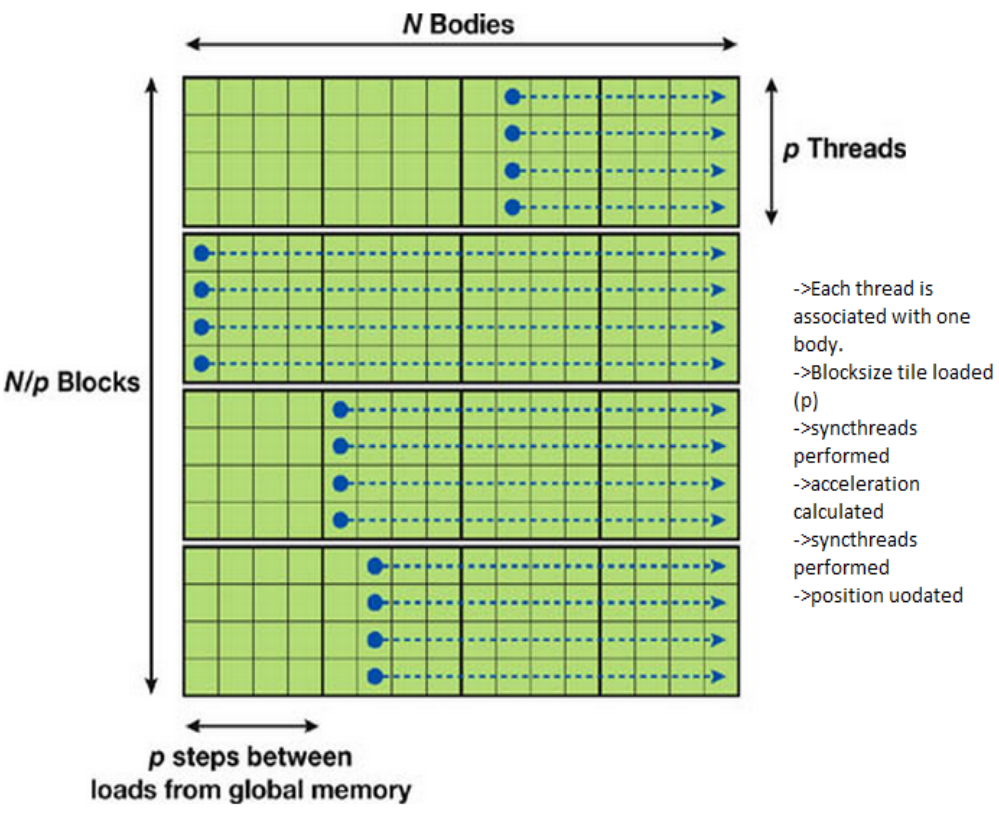


Figure 4.1

Pseudo Code

```
Kernel body-force function:
      i = thread index
            initialize force register Fx,Fy,Fz to 0;
            foreach (tile)
            load tile(BLOCK_SIZE) data in shared memory
            __syncthreads()
            Calculate δx, δy, δz from position value of other body in shared
            memory.
            Calculate acceleration/force and store in register for
            force/acceleration
            __syncthreads()
      Calculate and update velocity value of current body
End kernel
Update position array in main, calculate iteration time and repeat next
iteration like in the serial version.
```

4.2 BARNES HUT ALGORITHM

The Barnes Hut force-calculation algorithm is widely used in N Body simulations such as modeling the motion of galaxies. It hierarchically decomposes the space around the bodies into successively smaller boxes, called cells and computes summary information for the bodies contained in each cell, allowing the algorithm to quickly approximate the forces that n bodies induce upon each other. The hierarchical decomposition is recorded in an octree. This algorithm reduces the complexity of the problem from $O(N^2)$ to O(NlogN).

This implementation uses caching, throttles threads to drastically reduce the accesses to main memory. It also employs array-based techniques to enable memory coalescing. Because traversing irregular data structures often result in thread divergence, we group similar work together to minimize thread divergence.

4.2.1 Barnes Hut Algorithm : An Overview

Following is the pseudo code for the algorithm:

```
0. Read input data and transfer to GPU

For each timestep do {

1. Compute bounding box around all the bodies
2. Build hierarchical decomposition by inserting each body into octree.
3. Summarize body information in each internal octree node
4. Approximately sort the bodies by spatial distance
5. Compute forces acting on each body with help of octree.
6. Update body position and velocities.

}

7. Transfer result to CPU
```

Steps 1-6 are executed on the GPU. Each of these steps is implemented as a separate kernel, which is necessary because we need a global barrier between the steps. It is also beneficial because it allows us to individually tune the number of blocks and threads per block for each kernel.

Kernel 1 computes a bounding box around all bodies. This box becomes the root node of the tree. It is represented in Fig 4.1.

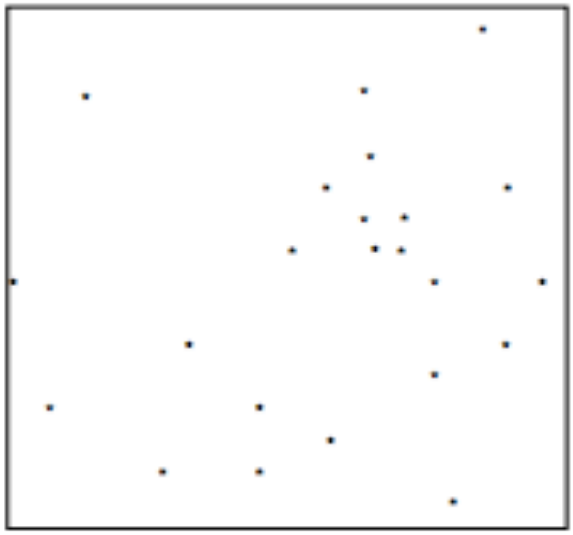


Fig 4.1 Bounding Box in space

Kernel 2 hierarchically subdivides this cell until there is at most one body per innermost cell. This is accomplished by inserting all bodies into the tree. It is represented in Fig 4.2

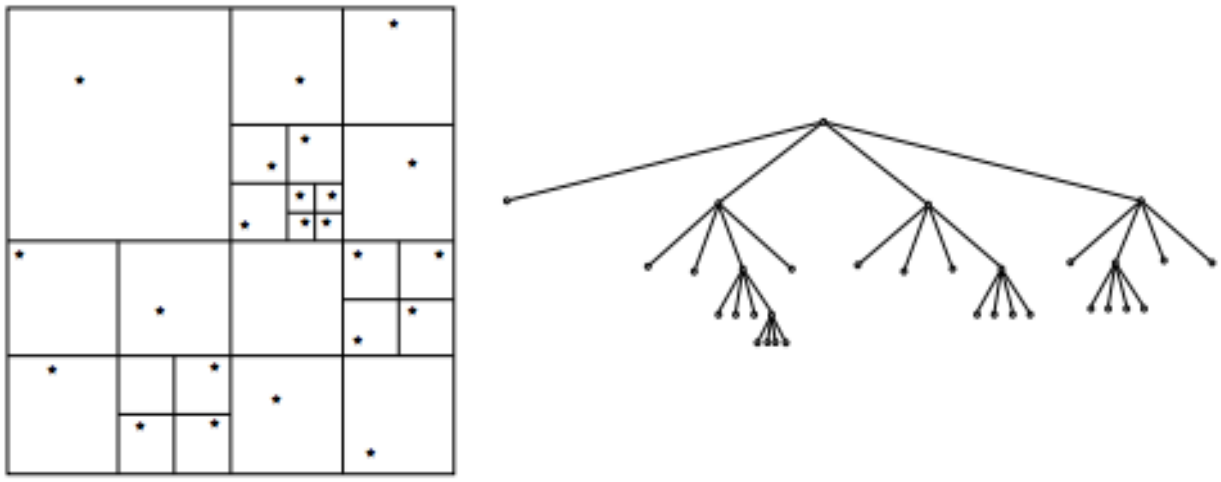


Fig 4.2 Hierarchical decomposition kernel

Kernel 3 computes, for each cell, the center of mass and the cumulative mass of all contained bodies. This is as indicated in Fig 4.3. “X” indicates the center of mass.

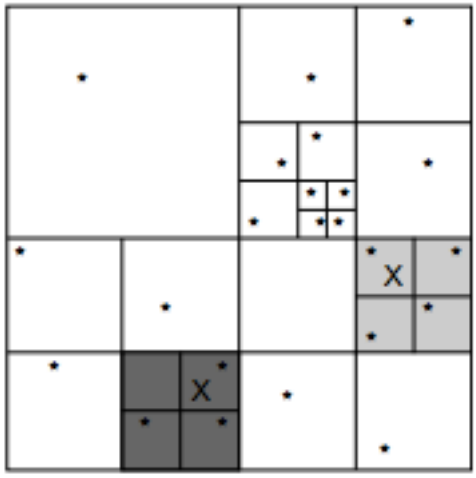


Fig 4.3 Center of Mass kernel

Kernel 4 sorts the bodies according to an in-order traversal of the octree, which places spatially close bodies close together while preserving the order of traversal.

Kernel 5 computes the force acting on each body. Starting from the root node, it checks whether the center of gravity is far enough away from the current body for each cell. If it is not (solid arrow), the sub-cells are visited to perform a more accurate force calculation. If it is (dashed arrow) only one force calculation with the cell’s center of gravity and mass is performed and individual sub-cells are not visited. This greatly reduces the total number of force calculations.

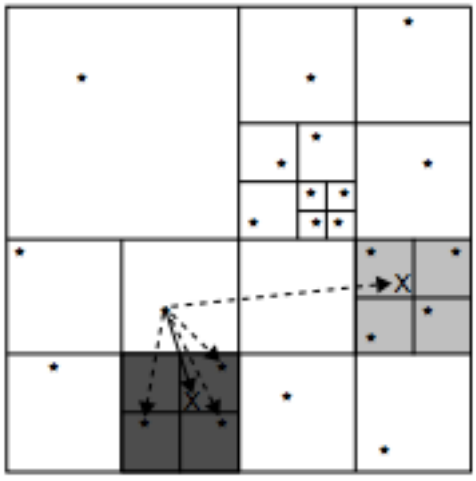


Fig 4.4 Force Calculation

Kernel 6 nudges all bodies into their new positions and updates their velocities.

4.3 IMPLEMENTATION

4.3.1 Global Optimizations

Dynamic data structures such as trees are typically built from heap objects where each heap object contains multiple fields and is allocated dynamically. Because dynamic allocation of and accesses to heap objects tend to be slow, we use an array-based data structure. Accesses to arrays cannot be coalesced if the array elements are objects with multiple fields, so we use several aligned scalar arrays, one per field, as shown in Fig 4.5.

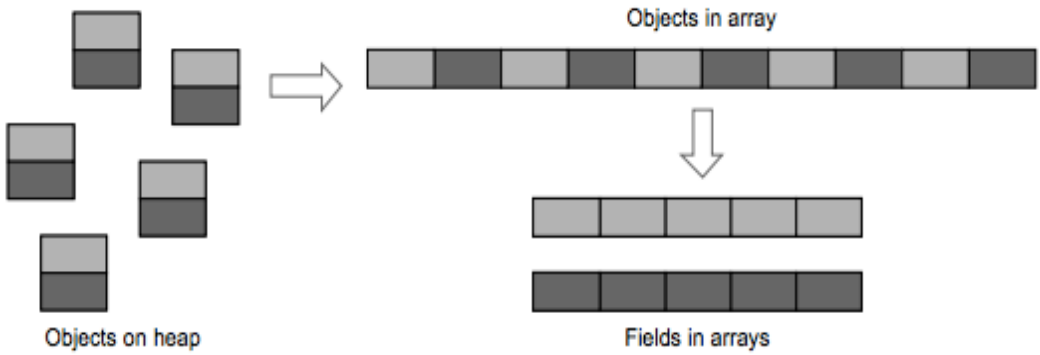


Fig 4.5 Using multiple arrays instead of an array of objects or separate heap objects

In the tree, the cells, which are internal tree nodes and the bodies which are the leaves of the tree have some common fields (mass and position). Other fields are needed only by either the cells or the bodies. However, to simplify and speed up the code, it is important to use the same arrays for both bodies and cells.

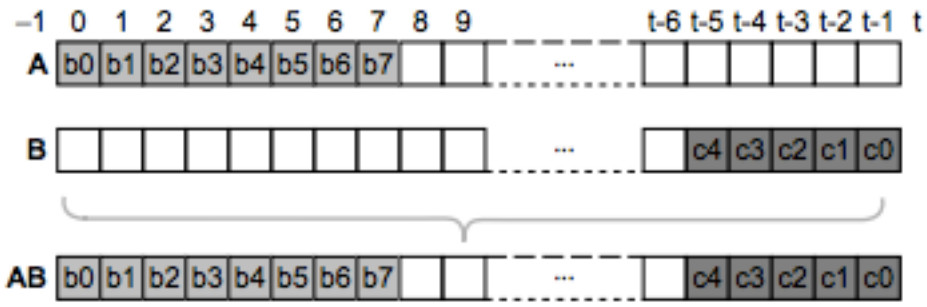


Fig 4.6 Array layout and element allocation order for body-only (A), cell only (B) and combined arrays (AB)

Bodies are allocated at the beginning and the cells at the end of the arrays, as illustrated in the above figure. A value of "-1" is used as a null pointer. This allocation order has several pros. First, a simple comparision of the array index with number of bodies determines whether the index points to a cell or

body. Second, we need to find out whether an index points to a body or null and because -1 is smaller than number of bodies, a single integer comparision suffices to test both conditions.

Also, because all parameters passed to the kernels, such as starting addresses of various arrays, stay the same throughout the timestep, we copy them once on the GPU's constant memory. This is much faster than passing them with every kernel call.

Moreover, data is transferred from CPU to the GPU only once. This approach avoids slow data transfers over the PCI bus.

4.3.2 KERNEL 1: BOUNDING BOX KERNEL

The first kernel computes a bounding box around all the bodies ie. It finds the root of the octree. In order to do that, it has to find the minimum and maximum co-ordinates in the 3D space. This kernel breaks up the data into equal sized chunks and assigns one chunk to each block. Each block then performs a reduction operation in shared memory to find the local minima and maxima in each block.

The data is loaded from global memory in a fully coalesced manner. It also minimizes the thread divergence. In built *fminf* and *fmaxf* functions are used to find min and max since these are faster than if statements. The final result from each block is written to main memory. The last block as determined by a global atomicInc operation, combines these results and generates the root node.

The index of root node is 0 and is put at the end of the array. Mass of the root node is -1.

Pseudo code

1. Insert a temp value into each thread as a local minima and maxima
2. For each thread, compare its local minima and maxima with other threads in a block.
3. Do a reduction in shared memory to find local minima and maxima in each block
4. Do a reduction in global memory to find global minimum and maximum amongst all threads in a grid.
5. The distance between minima and maxima is the bounding box radius.
6. Assign it as root node
7. Assign its mass to –1 and start index to 0

4.3.3 KERNEL 2: TREE BUILDING KERNEL

This kernel builds an iterative tree that uses lightweight locks, only locks child pointers of leaf nodes. The bodies are assigned to the blocks and threads within a block in round-robin fashion. Each thread inserts its bodies iteratively by traversing the tree from root to desired last level cell and then attempts to lock the child pointer by writing a "-2" value to use it in an atomic operation. If the locking succeeds, the thread inserts new body, thereby overwriting the lock value by its thread ID, which releases the lock.

If a body is already is stored at this location, the thread first creates a new cell by atomically requesting the next array index, inserts the new and the original body into a new cell. It then inserts a memory

```
if (-1 == atomicCAS((int *)&childd[locked], -1, i)) {  // if
null, just insert the new body
          ……….
        }
      } else {  // there already is a body in this position
        if (ch == atomicCAS((int *)&childd[locked], ch, -2))
{  // try to lock
          ……..……
```

fence so that the new subtree is visible to other threads as well.

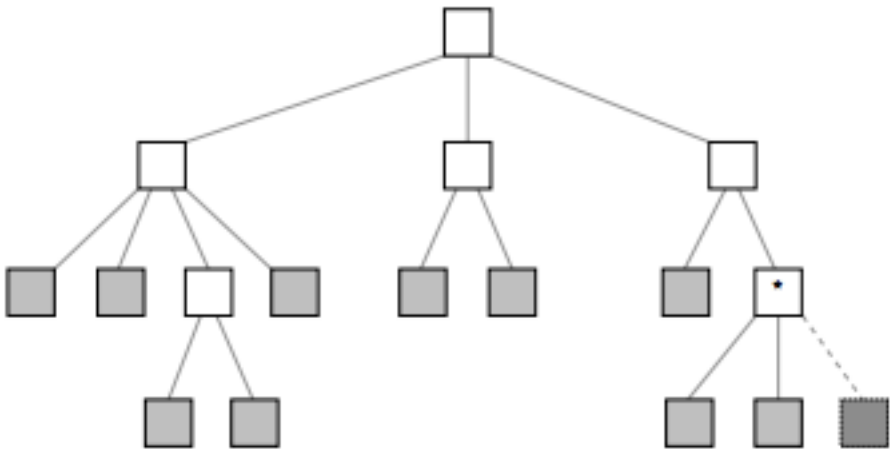


Fig 4.7 Locking the child pointer of the "*" cell to insert body

Pseudo Code:

```
1. cell = find_insertion_point(body) //nothing is locked, it retries
   by caching the cell
2. child = get_insertion_index(cell,body)
3. if(child!=locked)
   {
        if(child == atomicCAS(cell[child],child,lock)) {
             if(child=null) {
```

```
                cell[child] = body       //insert the body and
release lock
            }
      else {
      new cell = …   //atomically get the next unused cell
      //insert the existing and the new body into new cell

      __threadfence();
      cell[child] = new cell;       //inset new cell and release
lock
}

success= true   //flag indicating the insertion succeeded
}
}
__syncthreads;   //wait for other warps
```

4.3.4 KERNEL 3: CENTER OF MASS CALCULATION

This kernel traverses the octree from bottom up to compute the center of mass and the sum of the masses of each cell's children. Fig 4.8 shows the octree nodes in global arrays and the corresponding tree representation.

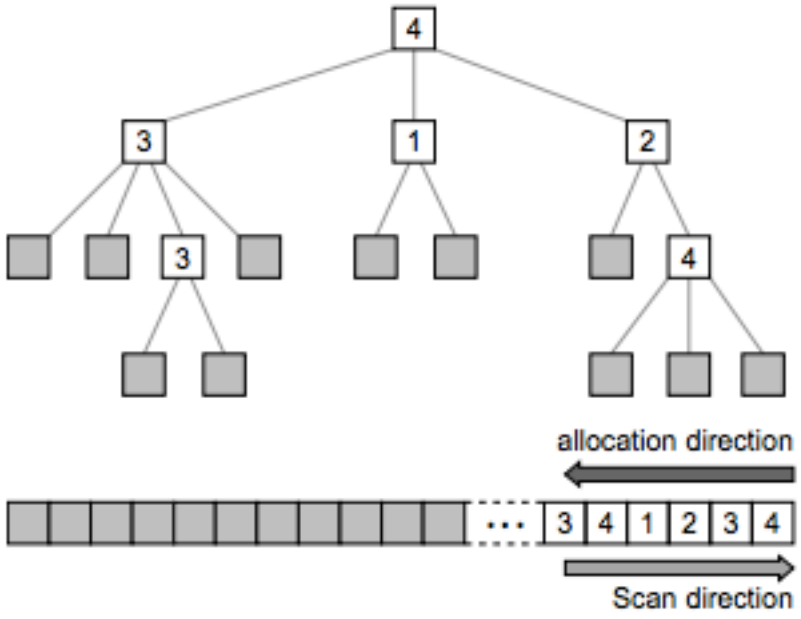


Fig 4.8 Allocation of bodies (shaded) and cells in global arrays and corresponding tree representation

The cells are assigned to blocks and to the thread within a block in round robin fashion. The threads are aligned to WARPSIZE to ensure memory coalescing. Initially all the cells have negative masses, indicating their true masses will be computed. Because the majority of cells in the octree have only bodies as children, the corresponding threads can immediately compute the cell data.

A memory fence is inserted so that no other thread can see the updated mass before it sees the updated center of mass. Unsuccessful threads have to poll until the "ready" flag. For performance reasons, the kernel also caches the child pointers of the current cell in shared memory that point to not ready children.

In addition to calculating the center of mass, this kernel does 2 other performance optimizations. The first one is to count the number of bodies in all subtrees and store the information in cells. This makes kernel 4 faster. Also, it moves the null pointers to the end, so that we can exit the loop whenever first null pointer is encountered without traversing the rest of the subtree.

Pseudo Code:

```
1. Initialize the masses of cells to -1
2. If(missing == 0)
   {
         for(//iterate over existing children)
               if(/*child is ready*/)
                // add its contribution to Center of Gravity
               else
               {
               //cache child index
               missing++;
               }
         }
   }
3. if(missing != 0)
   do {
         if(/*last cached child is ready */)
               //remove from cache and add its contribution to center of
   gravity
               missing—
         }
   } while(missing!=0)
4. if(missing == 0)
   {
         //store center of gravity
         __threadfence()
         //store cumulative mass
         success = true    //computation for cell is done
   }
5. Counting the number of bodies in each subtree
6. Bringing the null child pointers to the end of array to speedup later
   kernels
```

4.3.5 KERNEL 4 : SORT KERNEL

This kernel sorts the bodies in parallel using a top-down traversal. It employs array based traversal technique. Based on the number of bodies in each subtree (calculated in kernel 3) , it concurrently placed the bodies into an array such that spatially close bodies are close to each other in the array.

Doing this speeds up kernel 5 whereby force is calculated.

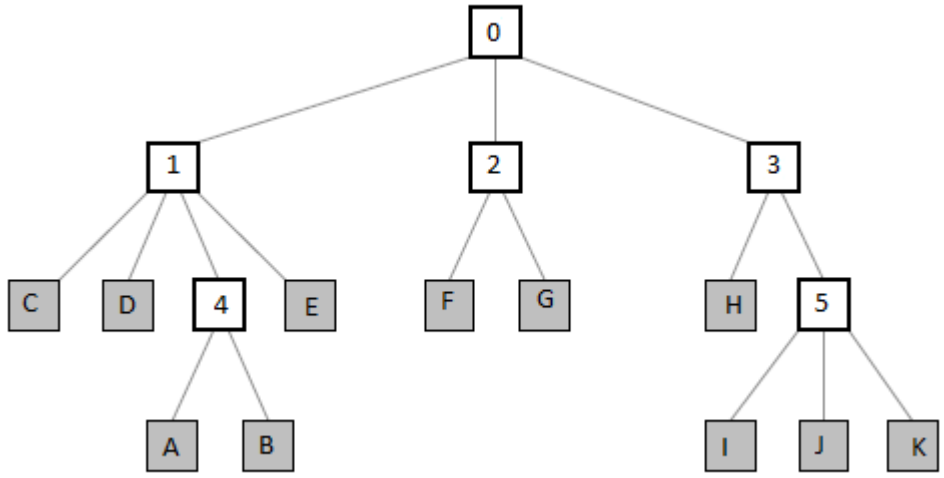


Fig 4.9 Sorting the bodies

Unsorted array:

K	J	I	B	A	H	G	F	E	D	C	5	4	3	2	1	0

Sorted array:

K	J	I	H	G	F	B	A	E	D	C	5	4	3	2	1	0

As can be seen from the above diagram, spatially close bodies are brought together by sorting the child array.

4.3.6 KERNEL 5: FORCE CALCULATION KERNEL

This kernel amounts for vast majority of execution time and hence it requires maximum optimization. It assigns all bodies to blocks and threads within a block in round robin fashion. For each body, the corresponding thread traverses some prefix of the octree to compute the force acting upon this body, as shown in Fig 4.10

These prefixes are similar for spatially close bodies but different for spatially distant bodies. Determining the union's border can be done extremely quickly using the __all thread-voting function as shown in the figure.

Main memory accesses still pose a major performance hurdle because this kernel contains very little computation with which to hide them. Moreover, the threads in a warp always access the same tree node at the same time, that is, they always read from the same location in main memory. Unfortunately, multiple reads of the same address are not coalesced. To remedy this situation, we allow one thread per warp to read pertinent data and then that thread makes the data available to other threads in the same warp by caching the data in shared memory.

Thus, bodies which are far away calculate their interaction only with the center of mass of the far away cell whereas spatially close bodies have to individually calculate the interactions between each pair of bodies in a warp.

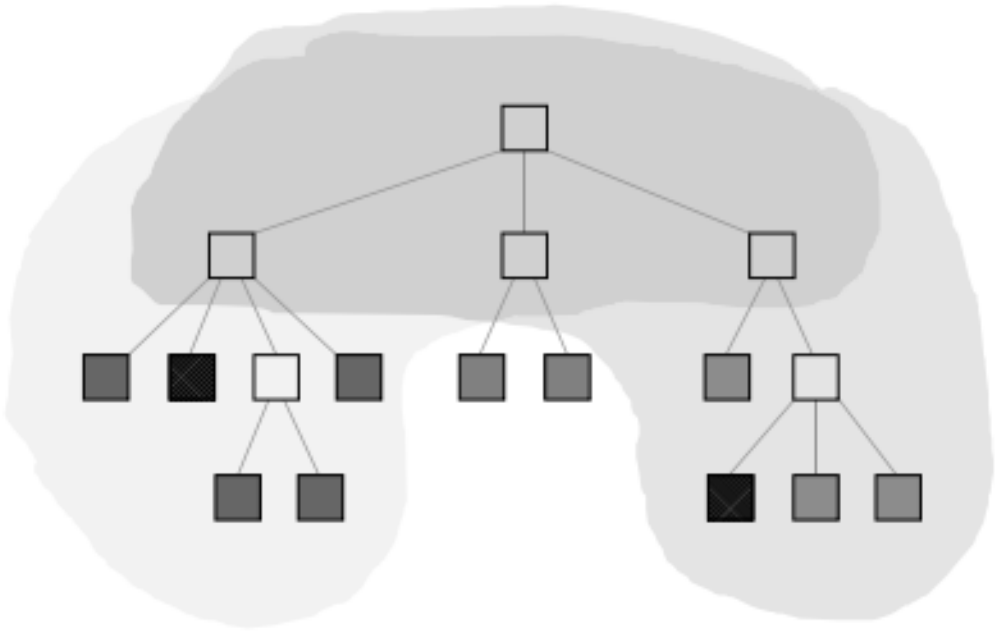


Fig 4.10 Tree prefixes that need to be visited for two marked bodies

Pseudo Code:

```
1. Determine first thread in each warp
2. For(/* sorted body indexes assigned to me*/)
   {
         //cache body data
         //initialize iteration stack
         depth = 0
         while(depth >=0)
         {
               while(more nodes to visit)
               {
                     if(I am the first thread in warp)
                     {
                           //move on to next node
                           // read node data and put in shared memory
                     }
```

```
                __threadfence()
                if(node is not null)
                {
                      get node from shared memory
                      compute distance to node
                      if(node is a body) || __all(distance >= cutoff)
                      {
                            //compute force
                      }
                      else
                      {
                            depth++
                            if(I am the first thread in warp)
                            {
                                  //push node's children onto stack
                            }

                      }

                      __threadfence()

                }

                else

                {

                      depth = max(0,depth -1)

                }

          }

    }

    depth—

    }
```

3. update body data

4.3.7 *KERNEL 6: INTEGRATION KERNEL*

This kernel updates the velocity and position of each body based on the computed force. It is straightforward and fully coalesced, non-divergent streaming kernel as shown in figure. Threads are assigned in blocks in a round robin fashion.

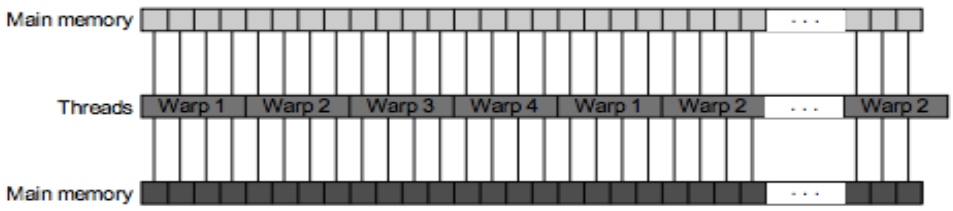


Fig 4.11 Fully coalesced streaming updates of the body position and velocity

5 PARALLEL OPTIMIZATIONS

This section provides a summary of all the optimization described above. Following are the major optimization principles.

5.1 MAXIMIZING PAARALLELISM AND LOAD BALANCE

To hide long latencies, it is important to run a large number of threads and blocks in parallel. By parallelizing every step of our algorithm across threads and blocks as well as by using array elements instead of heap objects to represent tree nodes, we were able to assign balanced amounts of work to any number of threads and blocks.

5.2 MINIMIZING THREAD DIVERGENCE

To maintain maximum parallelism, it is important for the threads in a warp to follow the same control flow. Grouping similar work together reduces thread divergence especially in irregular codes like tree structures.

5.3 MINIMIZING MAIN MEMORY ACCESS

The most important optimization is to minimize main memory access because it can take upto 1000s of cycles to access the main memory. Therefore, by using caches, we reduce main memory accesses.

5.4 USING LIGHTWEIGHT LOCKS

If locking is necessary, a few items only should be locked to avoid serialization. The locking is done using an atomic operation and unlocking is done with a faster memory fence store operation. For maximizing speed, we use existing data fields instead of separate locks. These are called lightweight locks.

5.5 COMBINING OPERATIONS

Because pointer chasing operations are expensive on GPUs, it is probably worthwhile to combine tasks to avoid additional traversals. For example, our code computes mass and center of gravity, counts all the bodies in subtrees and moves null pointers to the end in a single traversal.

5.6 Maximizing coalescing

By carefully allocating node data and spreading the fields over multiple scalar arrays, we managed to attain coalescing in several kernels.

5.7 Avoiding CPU/GPU transfers

Data has to be transferred to the GPU only once from the CPU before starting the computation. Thus, we are able to eliminate the data transfers over a slow PCI bus from CPU to GPU avoiding ping-ponging. Moreover, we used constant memory to transfer kernel parameters.

6 EVALUATION

6.1 SETUP

6.1.1 IMPLEMENTATIONS

In addition to our parallel CUDA version of Barnes Hut algorithm, we wrote two more implementations for comparision:

- A serial C version (CPU Baseline)
- A parallel CUDA version of the $O(N^2)$ algorithm using Tiling

The $O(N^2)$ computes all pair wise forces and is therefore more precise than the approximate Barnes Hut Algorithm. Though the margin of error was found out to be only about 1%.

6.1.2 INPUTS

We generated 7 input datasets with 1000, 5000, 10000, 50000, 100000, 500000 and 1000000 N-bodies. We use a random generator for generating the positions of the N-bodies. The masses of all N Bodies have been assumed to same for the sake of computation.

6.1.3 SYSTEM

We evaluated the performance on the GEM Cluster which have (Tesla C2050/C2070) devices and a compute capability of 2.0. The CPU have the following configuration:

```
model name : Intel(R) Xeon(R) CPU           X5680  @ 3.33GHz
stepping   : 2
cpu MHz         : 1596.000
cache size   : 12288 KB
```

6.1.4 COMPILERS

The CUDA codes were compiled with a nvcc compiler and the "-O3 –arch=sm20" flags. For the serial code, we used the gcc compiler with the "-O3" flag.

6.1.5 METRIC

We have the best runtime amongst the three experiments. We measured only the timestep loop, that is generating the input and writing the output is not included in runtimes.

6.1.6 VALIDATION

We compared the output (positions) of the three codes to verify that they are computing the same result. However the results are not entirely identical because the $O(N^2)$ algorithm is more accurate than the Barnes Hut algorithm and because the CPU's floating-point arithmetic is more precise than GPU's.

6.2 RESULTS:

Following are the results for all the three implementations:

Nbody	Serial	$O(N^2)$ on GPU	Barnes-Hut
1000	23.768	0.313	6.3
5000	598.89	1.884	17.2
10000	2404.344	5.622	27.8
50000	60508.89	104.271	163.3
100000	242378.26	410.687	375.2
500000	612258.57	10160.527	2343.6
1000000	2297531.15	40509.186	5128.6

Table 1: Execution times of all implementations in ms

The values have been plotted for 2 timesteps of the algorithms.

The plots for the same are as follows:

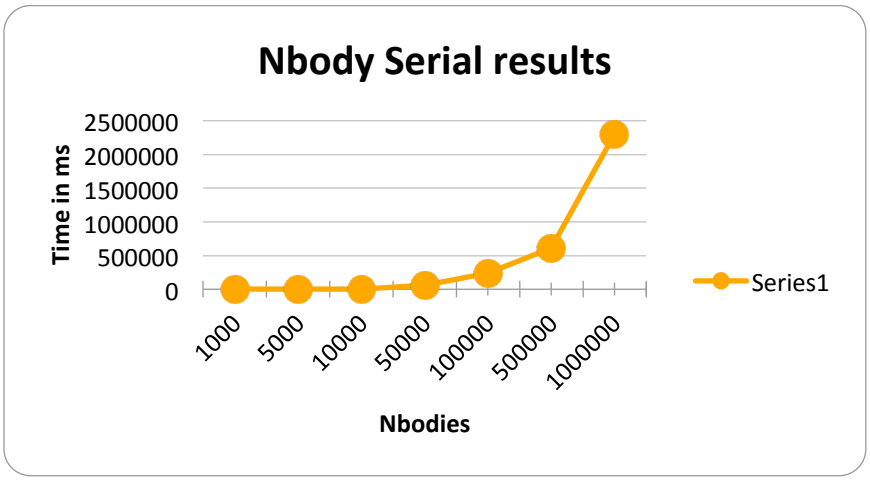


Fig 6.1 Variation of execution time of CPU Baseline Version with N-bodies

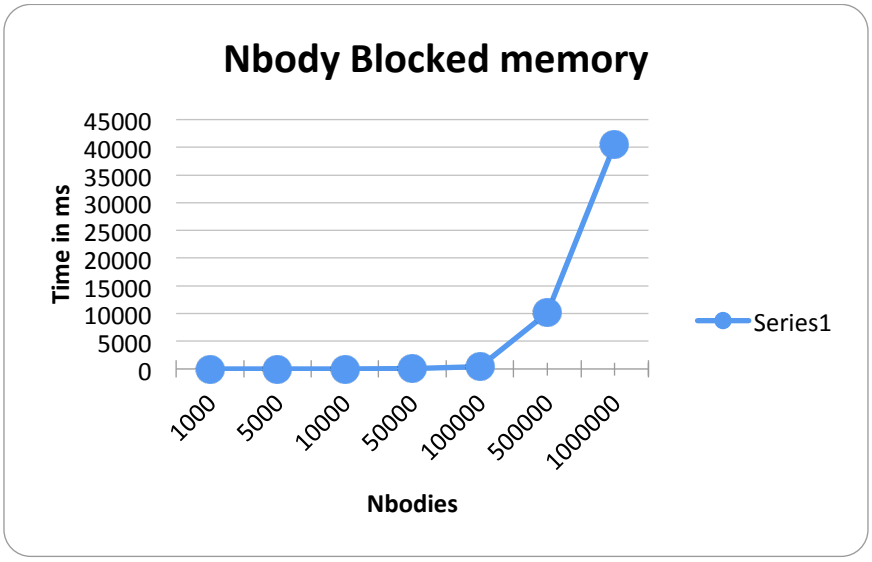


Fig 6.2 Variation of execution times of Tiled O(N^2) algorithm on GPU in ms (naïve implementation)

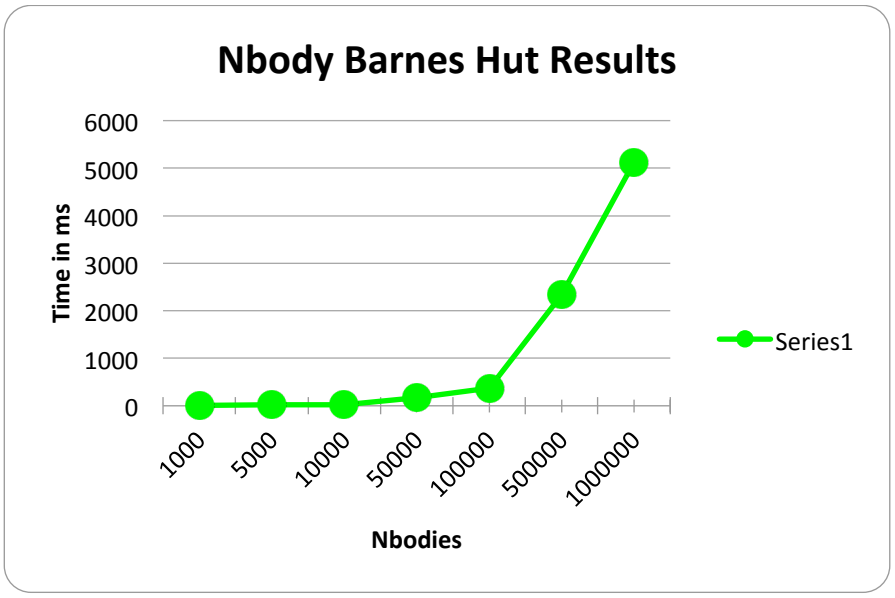


Fig 6.3 Variation of execution times of Barnes Hut Algorithm on GPU in ms

Following figure indicates a quantitative comparision of all three implementations.

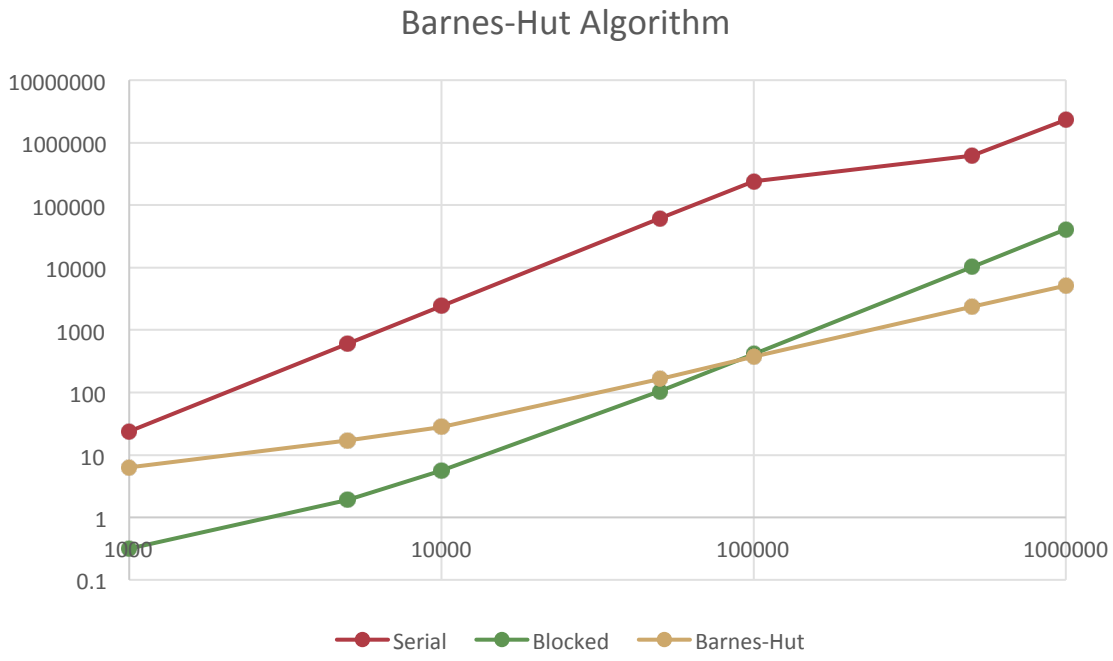


Fig 6.4 Quantitative Comparision of All three Implementations

Note: The Y axis is on log scale

The Barnes Hut algorithm on GPU is almost 1000 times faster than CPU implementation with higher input sizes. It is slower than the Tiled CUDA code for smaller input sizes but as input size grows beyond 50,000 we observe a speedup of almost 10x with this algorithm.

The algorithm is slower for smaller datasets because the overhead of tree traversal is large for smaller parallelism to extract enough parallelism from the GPU. But as datasets grow larger, the overheads decrease as we combine more work together and hence results in increased speeds.

As shown in the related work section, there exist other algorithms to do this same problem but we could not evaluate this because no code was available for those implementations and also lack of time.

7 CONCLUSIONS

The main conclusion of our work is that GPUs can be used to accelerate irregular codes like tree data structures. However, a great deal of effort is required to parallelize the code. The major hurdles being irregular memory accesses to global memory.

In particular, we had to redesign the tree data structure using arrays, convert all recursive code into iterative code, manage the caches and shared memory efficiently and parallelize the implementation. Some of the tuning efforts that we did were to ensure memory coalescing and reduce thread divergence. Also, we used only one thread to load data from memory and share them with other threads without the need for synchronization. Similarly, because barriers are implemented in hardware on GPUs they are extremely fast and we used them to reduce wasted work and main memory accesses.

We obtained a speedup of almost 10x with this implementation as compared to our naïve implementation and a mind boggling speedup with respect to the serial CPU Baseline implementation.

8 FUTURE WORK

We plan to implement the algorithms such as FMM and Parallel Multipole Tree Algorithms as indicated in Related Work and test its performance and scalability. Also, we plan to improvise on the existing Barnes Hut algorithm by using newer much more efficient GPUs which provide many efficient features than the current ones on GEM.

9 ACKNOWLEDGMENTS

We would like to thank Dr. Hwu for his lucid explanations of various concepts and algorithms throughout the semester and for providing feedback on our implementation. We would also like to thank the TA Izzat El Hajj for his continuous guidance and feedback on forum as well as by email.

Also, we would like to thank our Mentor and Professor Dr. Zhou for his guidance and feedback for this project.

10 REFERENCES

[1] J.Barnes, P. Hut, A hierarchical O(*n log n*) force-calculation algorithm, Nature, 324 (4) (1986) 446-449

[2] A practical comparison of N-body Algorithms- Guy Blelloch and Girija Narlikar-Parallel Algorithms, Series in Discrete Mathematics and Theoretical Computer Science, volume 30, 1997

[3] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html

[4] http://www.cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/nbody-problem.pdf