Hierarchical N-Body problem on graphics processor unit

Mohammad Faisal Siddiqui

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Hierarchical $N$-Body Problem on Graphics Processor Unit

by

Mohammad Faisal Siddiqui

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by
Dr. Muhammad Shaaban, Associate Professor, Computer Engineering
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
March 15, 2006

Approved By:

M. Shaaban
Dr. Muhammad Shaaban
Associate Professor, Computer Engineering
Primary Adviser

Lawrence Ray
Dr. Lawrence Ray
Adjunct Professor, Computer Engineering

Juan Cockburn
Dr. Juan Cockburn
Associate Professor, Computer Engineering
Title: Hierarchical N-Body Problem on Graphics Processor Unit

I, Mohammad Faisal Siddiqui, hereby grant permission to the Wallace Memorial Library reproduce my thesis in whole or part.

Mohammad Siddiqui
Mohammad Faisal Siddiqui

Date 3/17/2006
Dedication

To mom and dad.
Acknowledgments

Having Dr. Shaaban as my primary adviser has been an amazing experience. His unique and exhaustive teaching style spurred in me the passion for computer architecture. I am overwhelmed with gratitude for his guidance, motivation and support. He gave me enough freedom to explore new ideas, and applied pressure when seemed appropriate. I also admire and respect his integrity both as a professor and a person.

I'd like to thank my committee members, Dr. Lawrence Ray and Dr. Juan Cockburn for giving important suggestions and feedback when necessary. I also appreciate their help and concern throughout my thesis.

I'd like to thank Dr. Stefan Harfst with the Astrophysics Group at Rochester Institute of Technology who helped me through the initial stages of my research during the summer of 2005.

And, finally I would like to thank the entire staff of Department of Computer Engineering at Rochester Institute of Technology for their help and assistance, and making me feel at home during all times.
Abstract

Galactic simulation is an important cosmological computation, and represents a classical N-body problem suitable for implementation on vector processors. Barnes-Hut algorithm is a hierarchical N-Body method used to simulate such galactic evolution systems.

Stream processing architectures expose data locality and concurrency available in multimedia applications. On the other hand, there are numerous compute-intensive scientific or engineering applications that can potentially benefit from such computational and communication models. These applications are traditionally implemented on vector processors.

Stream architecture based graphics processor units (GPUs) present a novel computational alternative for efficiently implementing such high-performance applications. Rendering on a stream architecture sustains high performance, while user-programmable modules allow implementing complex algorithms efficiently. GPUs have evolved over the years, from being fixed-function pipelines to user programmable processors.

In this thesis, we focus on the implementation of Barnes-Hut algorithm on typical current-generation programmable GPUs. We exploit computation and communication requirements present in Barnes-Hut algorithm to expose their suitability for user-programmable GPUs. Our implementation of the Barnes-Hut algorithm is formulated as a fragment shader targeting the selected GPU. We discuss implementation details, design issues, results, and challenges encountered in programming the fragment shader.
Contents

Dedication ......................................................... iv
Acknowledgments .................................................. v
Abstract .......................................................... vi
Glossary .......................................................... xii

1 Introduction ...................................................... 1

2 N-Body .......................................................... 5
  2.1 N-Body Problem ............................................ 5
  2.2 N-Body Methods ............................................ 5
     2.2.1 Particle-Particle Method .............................. 6
     2.2.2 Particle-Mesh Method ................................. 6
     2.2.3 Particle-Particle/Particle-Mesh Method ............ 7
     2.2.4 Hierarchical Methods ................................. 7
  2.3 Barnes-Hut Algorithm ...................................... 8

3 Stream Architecture and Programming Model .................. 13
  3.1 RAW Machines .............................................. 14
  3.2 Imagine Stream Processor ................................. 16

4 Graphics Processor Architecture ................................. 19
  4.1 Graphics Rendering Pipeline ............................... 19
  4.2 Graphics Hardware ........................................ 21
     4.2.1 Geometry Engine .................................... 22
     4.2.2 Channel Processor .................................. 23
     4.2.3 Polygon-Rendering Chip ............................. 23
     4.2.4 High Performance Polygon Rendering ............... 24
4.2.5 Indigo Extreme ........................................... 25
4.2.6 PixelFlow .................................................. 27
4.2.7 RealityEngine ............................................ 28
4.2.8 InfiniteReality .......................................... 29
4.2.9 Graphics Software Libraries ......................... 31
4.2.10 Streaming CPU Extensions ......................... 32
4.3 Programmable Graphics Hardware ....................... 33
  4.3.1 Vertex Processor ..................................... 33
  4.3.2 Fragment Processor .................................. 37
  4.3.3 Graphics Memory ..................................... 38
  4.3.4 GPU Programming Model ........................... 40
4.4 GPU for Non-Graphics Operations ....................... 42

5 Previous Work .............................................. 44
  5.1 Fast Matrix Multiplies using Graphics Hardware ... 45
  5.2 Physically-based Visual Simulation on Graphics Hardware 46
  5.3 GROMACS .................................................. 47
  5.4 Particle-Mesh N-Body Method on Graphics Hardware 48
  5.5 Radiosity on Graphics Hardware ....................... 48
  5.6 Ray Tracing on GPU ...................................... 49
  5.7 Octree Textures on the GPU ............................ 50

6 Implementation ............................................. 52
  6.1 Algorithm Overview ...................................... 53
  6.2 Octree Construction ..................................... 55
  6.3 N^3-tree memory map on the GPU ...................... 57
  6.4 Kernels ..................................................... 60
    6.4.1 Traverse ........................................... 60
    6.4.2 Position ............................................ 64
    6.4.3 Velocity ............................................ 64

7 Results and Discussion ................................... 66
  7.1 Results .................................................... 66
  7.2 Discussion ............................................... 69
  7.3 Difficulties Encountered ................................ 70
  7.4 Debugging Fragment Shader ............................ 74
List of Figures

2.1 Barnes-Hut multipole accessibility criterion. .......................... 8
2.2 The relation s/d for different levels of the tree. ...................... 10
3.1 RAW processor. .............................................. 15
3.2 Imagine processor. ............................................. 17
4.1 Graphics rendering pipeline. .................................... 20
4.2 RealityEngine block diagram. .................................. 28
4.3 InfiniteReality Engine block diagram ............................. 30
4.4 Overall system architecture. .................................... 34
4.5 GeForce 6 Series Vertex shader block diagram. .................. 35
4.6 GeForce 6 Series fragment processor. ........................... 37
4.7 GPU memory hierarchy. ....................................... 39
5.1 Matrix Multiplication .......................................... 45
5.2 CML Performance Comparison .................................... 47
6.1 Algorithm ..................................................... 54
6.2 Octree ....................................................... 55
6.3 Octree mapped to 2D texture ................................... 57
6.4 2D textures on the GPU ....................................... 59
6.5 Tree traversal .................................................. 61
6.6 Position kernel ................................................ 65
6.7 Velocity kernel ................................................ 65
7.1 Position and Velocity kernel timing graph. ........................ 68
7.2 Tree traversal kernel timing graph. ................................ 68
7.3 (Ideal) Position and Velocity kernel timing graph. .............. 72
7.4 (Ideal) Tree traversal kernel timing graph. ........................ 73
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Minimum 2D textures needed to store octree information</td>
<td>67</td>
</tr>
<tr>
<td>7.2</td>
<td>Branch penalties on Nvidia 6 Series GPU</td>
<td>70</td>
</tr>
<tr>
<td>7.3</td>
<td>(Ideal) Minimum 2D textures needed to store octree information</td>
<td>72</td>
</tr>
</tbody>
</table>
Glossary

A

*arithmetic intensity*  Ratio of arithmetic computations to communication.

C

cell  A node of an octree with more than one particle.

F

*fragment processor*  Fragment processors are fully programmable processors, operating in SIMD-parallel fashion on input elements: processing four-element vectors in parallel.

*framebuffer*  Framebuffer is a part of GPU memory which holds final colored pixels for display on the screen.

G

gather  Gather operation occurs when the kernel processing a stream element requests information from other elements in the stream: it “gathers” information from other parts of memory.

*GPU*  A graphics processor unit or GPU is a graphics rendering pipeline used for generating images from geometric models.
K

kernel  Terminology used in stream architecture for describing a computing unit or program that operates on the data.

L

leaf  A node of an octree with only one particle.

M

MIMD  Multiple instruction multiple data lets multiple instructions to operate on multiple data items at any given time.

O

octree  A tree data structure with eight children.

S

scatter  Scatter operation occurs when the kernel processing a stream element distributes information to other stream elements: it “scatters” information to other parts of memory.

SIMD  Single instruction multiple data lets one instruction operate on multiple data items simultaneously.

streams  Terminology used in stream architecture for a data set that needs similar computations are called streams.
Texture memory  Texture memory is the only part of the GPU that is randomly accessible by the fragment processors.

Vertex processor  Vertex processors are fully programmable and operate in either SIMD or MIMD parallel fashion on the input vertices.
Chapter 1

Introduction

The classical $N$-Body problem simulates the evolution of a system, comprising of $n$ discrete bodies under the influence exerted on each body by all other bodies. In the galactic simulation, we study the evolution of a system of particles, where each particle represents a star or sampled to represent collection of stars, solar system, galaxies, cluster of galaxies and other large-scale structures of the universe bounded together by Newtonian gravitational potential.

The hierarchical method proposed by Barnes and Hut [5] builds a tree structured representation of the physical domain of the problem (in our case, space), and then compute interactions by traversing the tree. Such hierarchical tree-based methods reduce the computational complexity to $O(n \log n)$ for non-uniform distributions, or even $O(n)$ for uniform distributions, with parallelism proportional to $n$ and without introducing any significant ambiguity in the results. Applications that use Barnes-Hut method are challenging to obtain effective parallel performance, owing to their non-uniform distribution of bodies over the physical domain, distribution of bodies across the physical domain changes over time, and the need for long range communication. However, we are helped by two properties: first, the system evolves slowly over time, and second, the gravitation interaction falls off quickly with distance. Also, the force computation on any particle can be computed in parallel with other particles, and independent of other iterations. We map the simulation of galactic evolution based on the Barnes-Hut method using these characteristics.
Modern semiconductor technology makes arithmetic inexpensive and memory bandwidth expensive. Both global on-chip and off-chip communication thus becomes the performance limiting factor. Increasing ratio of arithmetic to memory bandwidth called arithmetic intensity and parallelism to use large number of available arithmetic units can offset this shift in cost. Modern multimedia applications emphasize on stream processing programming model. Stream programming model renders support for multi-level parallelism and exposes inherent data locality by partitioning the communication and data storage structures. Stream programming model represents all data as streams and expresses all computational units as kernels. Applications mapped to the stream programming model are constructed by chaining kernels together. These kernels operating on entire streams exposes multi-level parallelism while local data storage within kernel execution units eliminates global communication costs. Computation units are called as kernels and streams are made of identical data elements that require data processing.

The core of the graphics architecture, more commonly referred to as the graphics rendering pipeline or graphics pipeline defines a projection-based rendering algorithm (in real-time) that supports surface-based, volume based or image-based representation, given a virtual camera setting, mathematically modelled 3D objects, position and specification of light sources, lighting models, textures, and more [4]. The graphics rendering pipeline is split into several distinct substages, and more recently vertex and fragment shaders have been introduced as user-programmable stages to improve the functionality and flexibility to perform any task [32]. This conceptual model of graphics hardware and the addition of user-programmable substages makes the graphics hardware rendering pipeline a good match for stream programming model. The vertex and fragment shaders within the GPU can be represented as kernels or computational blocks while the highly localized dataflow between these substages or kernels can be abstracted as streams. The data flow between stages in the graphics pipeline is highly localized with the data produced by one stage consumed by the next stage. Some of the latest GPUs provide IEEE-standard single precision floating-point arithmetic throughout the entire graphics pipeline, for instance Nvidia...
GeForce 6 Series or Nvidia GeForce 7800. The user-defined programs (shader programs) written either in assembly (ARB_fragment_program) or high-level languages [34]; HLSL (High-level shading language); Sh [36] can be loaded into vertex and fragment units at run-time. The application data is stored on the GPU as texture images and can be 1D, 2D or 3D. The fragment or vertex shaders act on this data and write results either to the framebuffer or framebuffer attached objects (framebuffer objects) which can be again used via a feedback loop. Furthermore, with the addition of dynamic flow control (branching) and loop constructs, present generation GPUs can be used for more than one particular with possible gain in performance.

The development of programming components in the GPU has allowed researchers to use them for other applications than just simply rendering polygons. With the advent of programmable GPUs, efforts have been made to also map high-performance computing applications to the GPU, eg. scientific computations like FFTs, linear algebra solvers, differential equations solvers, multi-grid solvers and applications to fluid dynamics, visual simulation, ice crystals; geometric computations like Voronoi diagrams, distance computation, motion planning, collision detection, object reconstruction, visibility; advanced rendering like ray-tracing, radiosity, photon mapping, sub-surface scattering, shadows; and database operations on aggregates, predicates, Boolean combinations, selection queries for large databases [1].

We have developed a complete Barnes-Hut treecode implementation on the GPU. The motivation behind attempting N-Body problem on the GPU is the recent successful implementation of radiosity algorithm on GPUs. The hierarchical subdivision radiosity algorithm is based on the hierarchical N-Body problem. The N-Body problem shares many similarities with the radiosity problem which suggest that these ideas can lead to an increase in performance on the GPU. In both the N-Body and the radiosity problem, there are \( n(n - 1)/2 \) pairs of interactions. Moreover, just as the magnitude of the form-factor between two patches falls off as \( 1/r^2 \), the gravitational or electromagnetic forces also fall off as \( 1/r^2 \). However, there are several differences between the two implementations. One
The major difference between the two problems is the manner in which the hierarchical data structures are formed. The radiosity algorithm begins with a few large polygons and subdivides them into smaller and smaller patches. Our $N$-Body algorithm begins with $n$ particles and cluster them into larger and larger groups. Another difference is that radiosity problem is inherently non-linear because of occlusion where intervening opaque surfaces can block the transport of light between the two surfaces. The $N$-Body problem on the other hand is linear and takes advantage of linear superposition, the principle of superposition states that the potential due to a cluster of particles is the sum of the potentials of the individual particles. Finally, the radiosity problem is based on an integral equation, whereas the $N$-Body problem is based on a differential equation.

The remainder of this document is organized in the following manner:

- **Chapter 2** presents an overview of different $N$-Body methods and explains Barnes-Hut method in detail.

- **Chapter 3** gives a background overview on the stream architecture and programming model, and details two major stream architectures.

- **Chapter 4** presents a detailed account of landmark graphics architectures belonging to different generations, and also presents a detailed explanation on the current generation of programmable graphics hardware.

- **Chapter 5** details some background information in different research areas that provided motivation for the Barnes-Hut treecode implementation on the GPU.

- **Chapter 6** discusses the actual implementation of the Barnes-Hut treecode on the GPU.

- **Chapter 7** discusses the results and provides the discussion on the generated results.

- **Chapter 8**, finally provides the conclusion and suggests some extensions for future work extending this research area.
Chapter 2

$N$-Body

The main objective of our thesis is to implement Barnes-Hut treecode algorithm on the graphics processor unit. In this chapter we explain the $N$-Body problem and present a brief overview of important $N$-Body methods used in astrophysical simulations. Furthermore, we explain Barnes-Hut algorithm in great detail.

2.1 $N$-Body Problem

The $N$-Body problem is stated as follows: Given the initial states, position and velocity of $n$ bodies, compute the states at time $T$. The most widely accepted approach is to perform simulation over a time period. The simulation time period is discretized into hundreds or thousands of time-steps, each time-step computes the forces acting on each particle and updates the position, velocity and other attributes associated with that particle.

2.2 $N$-Body Methods

We briefly describe here different $N$-Body methods commonly used in astrophysical simulations.
2.2.1 Particle-Particle Method

The particle-particle or the direct summation method is the most straight-forward method of the $N$-Body methods. This direct integration approach is flexible but the cost of computation linearly increases with $n$, the number of particles and mostly used with collisional systems. In the case of collisional systems, the evolution of the system is driven by microscopic exchange of thermal energies between different particles. In this case it is not easy to use fast and approximate algorithms to determine the interaction between particles and the cost per timestep is $O(n^2)$, and the total cost of the simulation is $O(n^3)$ [33]. There are two reasons why the use of approximate algorithms for force computation is not plausible. The foremost reason is the need for relatively higher accuracy and secondly, the wide difference in the orbital timescales of particles which means that the particles in the core require much smaller time-steps than particles in the halo. In order to achieve such high computational cost, specialized hardware has been constructed for gravitational interaction, GRAPE (GRAvity piPE) [17]. A GRAPE hardware has specialized pipelines for gravitational force calculation, which is the most expensive part of most $N$-Body simulation algorithms. All other calculations, such as time integration of orbits, are performed on a standard host computer connected to GRAPE.

On the other hand, collisionless systems can be approximated using algorithms such as particle-mesh (PM) method [8], particle-particle/particle-mesh (P3M) method [8], Barnes-Hut tree algorithm [5], fast multipole method [18] or treecode particle mesh (TPM) methods [51].

2.2.2 Particle-Mesh Method

The particle-mesh method treats the force as a field quantity by approximating it on a mesh. The main advantage of the PM method is speed. The number of computations is of order $O(n + n_g log n_g)$ where $n_g$ is the number of grid points on the mesh. However, the PM method becomes unacceptable for studying close encounters between particles and they
have difficulties handling non-uniform particle distributions thus offering limited resolution.

2.2.3 Particle-Particle/Particle-Mesh Method

The particle-particle/particle-mesh method solves the major shortcoming of the PM method - low resolution forces computed for particles near each other. The P3M method creates a combined force function for all \( n \) bodies that is evaluated at regularly spaced points on a 3D mesh. The interaction of \( p \) with other bodies in the system may be decomposed into a number of direct interactions between \( p \) and nearby bodies, and interaction between \( p \) and the combined force interpolated on a mesh. The total number of operations is of order \( O(n + n_g) \), where \( n_g \) is the number of grid points on the mesh. The only disadvantage of the P3M method is that it becomes dominated by the interaction forces between nearby particles.

2.2.4 Hierarchical Methods

Hierarchical based methods like Barnes-Hut tree algorithm and fast multipole method are the best known methods for solving classical \( N \)-Body problems, such as those in astrophysics, electrostatics and molecular dynamics. All the hierarchical methods for classical problems first builds a tree-structured, hierarchical representation of physical space, and then compute force interactions by traversing the tree. The root of the tree represents a space cell containing all the particles in the system. The tree is built by recursively subdividing space cells until some termination condition is met. In three dimensions, every subdivision of a cell results in the creation of eight equally-sized children cells, leading to an octree representation of space, whereas, in two dimensions, each spatial decomposition leads to four equally-sized children.

Treecode particle-mesh methods uses treecode hierarchical decomposition for particles at a farther distance, while particle-mesh strategy is used for closer particles. This enables
us to use best techniques depending on the spatial decomposition of the space.

2.3 Barnes-Hut Algorithm

In the Barnes-Hut tree algorithm, the cubical root node encompassing the total mass is repeatedly divided into eight or fewer daughter nodes of half the side length each, until one ends up with 'leaf' nodes containing single particles or bodies. Forces are then computed on each particle by traversing the tree starting with the root node. A decision is made whether the node provides an accurate enough force acting on the particle. If the answer is affirmative, the multipole force is used to compute the force and the walk along this branch of the tree is terminated, otherwise, the node is traversed further down where we evaluate forces on the particle by considering its daughter nodes.

Figure 2.1: **Barnes-Hut multipole accessibility criterion.** 's' denotes the length of the side of the cell representing the pseudoparticle. 'd' denotes the euclidean distance between the particle and the center-of-mass of the pseudoparticle. The tree traversal is terminated along a branch if the the ratio of $s/d$ is less than $\theta$. 


The tree hierarchical structure provides the means to distinguish between close particles and distant particles without actually computing the distance between every particle. The force between particles that are close enough are computed directly, whereas distant particles are grouped together and are accounted as one giant node or a pseudoparticle. In order to compute forces from these pseudoparticles, several 'multipole acceptance criterion' (MACs) are available which decides whether to accept as it is, or subdivide it into its daughter nodes. The simplest of the criterion is the original s/d introduced by Barnes and Hut Ref. [5] and shown in Figure 2.1. Starting with the root node, the size of the current node s (the side length of the cell comprising the pseudoparticle) is compared with its distance from the particle, d. If the ratio s/d is smaller than some predefined value, \( \theta \), then the node is taken as is and further traversal along that node is terminated. Otherwise, the node is 'opened' into its daughter nodes, each of which is recursively examined according to s/d and, if necessary, subdivided. \( \theta \) is defined as the MAC parameter and can be varied to achieve any desired level of approximation for computing forces on the particles. Usually the MAC proves to be in the range of \( \theta = 0.3 - 1.0 \), depending on the accuracy needed by the application. The simulation of the evolution of a system of \( n \) bodies over time is based on the Newtonian equations of motion. The pairwise interaction potential \( f(i, j) \) between body \( i \) and \( j \) can be expressed as:

\[
f(i, j) = \frac{G.m_i.m_j}{r_{ij}^2}
\]

(2.1)

where \( G \) is gravitational force constant, \( m_i \) and \( m_j \) represent the masses of the two bodies, and \( r_{ij}^2 \) represent the euclidean distance between the two distinct bodies. The total force on a particle \( i \) can be summed as:

\[
f(i) = \sum_{j \neq i} f(i, j).
\]

(2.2)

The Barnes-Hut algorithm is one of the most widely implemented astrophysical simulation algorithm. Efforts have been made to parallelize the algorithm to exploit the parallelism available in the computation part of the algorithm. The core of the force calculation using Barnes-Hut algorithm may be compactly expressed as:
Figure 2.2: The relation $s/d$ for different levels of the tree. The figure illustrates the tree traversal for different values of $\theta$. When $\theta = 1.0$, the force exerted on the remote particle is due to the influence of the entire cell with the side length given by $s$, and computing body-cell force interaction. If $\theta = 0.7$, is assumed, we traverse further deeper within that cell and divide the space into equal spaces, with $s$ again representing the cell side length, and again computing body-cell force interaction. If $\theta = 0.3$ is assumed, we further divide the space evenly and eventually we reach the leaf of the tree, thus computing body-body force interaction.

**Algorithm 1 StepSystem**

1: MakeTree()
2: while loop over all bodies do
3: GravCalc(P(i), Q)
4: end while
Algorithm 2 GravCalc

1: if Q is a leaf then
2:   body-body force interaction
3: else
4:   if P is distant enough from Q then
5:     body-cell force interaction
6:   else
7:     while loop over all daughter nodes do
8:       GravCalc(P, daughter node)
9:     end while
10: end if
11: end if

StepSystem(), as in Algorithm 1 is the routine used for stepping through the simulation. The routine StepSystem() calls MakeTree() which constructs a new octree based on the current particle positions, and then we compute force on each particle/body using the routine GravCalc(), as shown in Algorithm 2. The routine GravCalc() computes the force exerted on the particle $P$ by all other nodes, $Q$. Depending whether the node $Q$ is a particle or a pseudo-particle, we compute the force exerted by that node on the particle $P$.

In our implementation, we used 3 different MACs.

- First, a minimal criterion is used where no cell touching the particle under consideration is accepted.

- Second, the original MAC introduced by Barnes and Hut as mentioned above.

- Third, the MAC introduced by Salmon and Warren to eliminate pathological errors that may occur when using Barnes-Hut MAC [44].

Figure 2.2 shows the effect of $\theta$ on the opening angle criterion. As $\theta$ is taken closer to 0, we traverse deeper down the tree for every node with the force computation more exact as compared with the force computation when $\theta$ is taken closer to 1.

We also eliminated computing quadrupole moments associated with each cell node. By doing so we are able to conserve memory which is limited on the GPU and thus allows
mapping of the octree on the GPU in a memory efficient way. Also it has been observed that using quadrupole or higher cell moments may increase the accuracy of the force computations, but this advantage is compensated by the increased complexity per evaluation and the higher tree construction and the tree storage overhead [48]. Another advantage of using monopole moments arise from the fact that the dynamical tree updates during the simulation can be performed in a very efficient manner.

Barnes-Hut algorithm has been successfully parallelized to exploit data parallelism involved in the force computation, thus making them ideal for high-performance multiprocessors or vector processors. However, such application typically has characteristics that make it challenging to partition and schedule it for effective parallel performance. As clear from the algorithm GravCalc, the force computations on each particle are totally independent of previous simulation steps. Also, we can compute force interactions on different particles simultaneously because the particle positions are only updated at the end of the current simulation step. Thus, the force computations are SIMD and they can be parallelize to achieve faster computation which results in the reduction of the overall simulation time. Recently, stream processors have been proposed that may eventually take place of vector processors. In the next chapter, we explore these stream architectures and study them in detail regarding their usage for SIMD natured applications.
Chapter 3

Stream Architecture and Programming Model

In this chapter we delve into the details of the stream architecture and its programming model. We study two of the ground-breaking stream architectures and their applications to wide ranging areas.

Modern semiconductor technology makes arithmetic inexpensive and memory bandwidth expensive. Both global on-chip and off-chip communication thus becomes the performance limiting factor. Increasing ratio of arithmetic to memory bandwidth called arithmetic intensity and parallelism to use large number of available arithmetic units can offset this shift in cost. Modern multimedia applications emphasize on stream processing programming model.

Stream architecture still in its infancy converges these underpinning issues effectively. Stream architecture divides the application to expose the inherent locality and concurrencies in applications by partitioning the communication and storage structures and provides explicit interconnect (communication) between functional units and register files. Stream processing architecture achieves support for multi-level parallelism also known as concurrency by exposing instruction-level parallelism (instruction overlap in execution across multiple functional units), data parallelism (exposing SIMD within same kernel) and task parallelism (exposing MIMD). Stream architecture partitions the storage structures into
different (explicit) levels of storage forming a data-bandwidth hierarchy (register hierarchy), thus reducing the distance traveled by the operands and limiting the communication cost. The higher the arithmetic intensity, the better the application is suited to stream processing architecture. Media applications including 3D computer graphics, image compression and signal processing have abundant parallelism to take advantage of stream architecture. Follows below are some of the representative architectures based on stream processing.

### 3.1 RAW Machines

The MIT RAW machines (Figure 3.1) provided efficient parallel, pipelined support for multimedia applications. The RAW microprocessor executes 16 different load, store, integer, or floating-point instructions every clock cycle, controls 25 Gbytes/s of I/O bandwidth and has 2 Mbytes of on-chip L1 SRAM providing 57 Gbytes/s of memory bandwidth [50]. The RAW microprocessor addresses the issue of wire delays and stream-based multimedia computations by exposing the hardware architecture’s low-level details to the compiler. The RAW microprocessor renders support in its software (ISA) for explicit communication to transfer data values between the computational portions of the tiles on a switched, point to point interconnect, and thus helps modeling wire delays. The RAW microprocessors also differ from current superscalar architectures by distributing the register files, and memory ports, thus rendering support for data locality. Much of the hardware support for register renaming, instruction scheduling, and dependency checking has been rendered in software, hence making room for more memory (distributed) and computational logic to be placed on chip. This results in the increase in the arithmetic intensity rendering support for data parallelism, maximizing the number of tiles that can be fit on a chip and increasing the chip’s achievable clock speed. Multiple processes can run simultaneously with operations being assigned to tiles in a manner that minimizes congestion. The RAW microprocessor provides an architecture that is scalable and provides high-bandwidth, fine-grained and
low-latency communication for efficient execution of scientific compute-intensive and media applications. The applications were written in a portable, high-level language called StreamIt to execute on the RAW microprocessor architecture. The basic unit of computation in StreamIt is the filter. Work forms the core of the filter unit and performs most fine-grained execution step in the steady state, thus exposing parallelism. In order to support explicit communication between different computation blocks or filters, StreamIt provides high-bandwidth I/O channels (FIFO queues), which can operate in pop, peek and push modes.

RAW processors not only run conventional scalar codes, but also word-level computations that require so much performance that they have been assigned to custom ASIC solutions. RAW processors have been used to implement Gigabit Internet protocols, video and audio processing, filters and modems, I/O protocols (RAID, SCSI, FireWire) and communication protocols for cellular phones, multiple channel cellular base stations, high-definition TV, Bluetooth, and IEEE 802.11a and b.
3.2 Imagine Stream Processor

The Imagine processor (Figure 3.2) [24] represents a programmable processor designed to operate directly on streams. The Imagine stream-programming model makes communication explicit, and exposes inherent data parallelism available in multimedia applications. The Imagine architecture proposes novel register-file architecture and supports 48 floating-point arithmetic units in eight arithmetic clusters. Each cluster can be mapped to operate on a different process by the stream-programming model using VLIW instruction (SIMD fashion). The stream program organizes data as streams and expresses computations as kernels (in media applications each kernel is composed of hundreds of computations), thus exploits data parallelism through streams and explicit communication (between streams) between kernels. A kernel consumes a set of input streams of data and generates a set of output streams of data. In order to capture data locality, Imagine (architecture) divides the register hierarchy into local-register files (LRFs), and stream-register files (SRFs). Data streams flow between different computation kernels is achieved through SRFs and intra-kernel storage/retrieval of results is done through LRFs (allowing explicit communication). As each kernel performs many computations (hundreds of computations), LRFs help maintain high data bandwidth needed for reading/writing data by different ALUs and handles the bulk of the data during kernel execution. StreamC and KernelC (both uses C-like syntax) are used for writing kernels and applications for Imagine processor. StreamC code compiled by stream scheduler on the host-CPU sends stream instructions to Imagine’s (processor) stream controller using static and dynamic runtime mechanism. KernelC code running on Imagine processor is compiled statically using VLIW kernel and performs kernel computations on the set of input streams and uses communication scheduling for explicit communication between operands [35].

Imagine is an on-going research project at Stanford University and they have successfully used it for varied range of applications. They have used it for audio and image compression and decompression. More recently, Imagine processors have been utilized to construct a supercomputer named as Merrimac. Merrimac uses stream architecture and
advanced communication interconnection networks to give an order of magnitude more performance per unit cost than cluster-based scientific computers [13].

![Diagram of Imagine processor]

Figure 3.2: Imagine processor. An Imagine stream processor consists of a 128-Kbyte stream register file (SRF), 48 floating-point arithmetic units in 8 arithmetic clusters controlled by a microcontroller, a network interface, a streaming memory system with 4 SDRAM channels, and a stream controller. However, the Imagine processor is controlled by a host processor which sends instructions to it.

We have seen above the needs for designing new architectures and programming models to meet the performance and flexibility requirements of media applications. We also saw that streams provide a powerful programming abstraction that have been suitable for efficiently implementing complex and high-performance media applications. In the last few years, some important work has been done to employ stream architecture and programming model for computer graphics application [40]; [23]. [40] focused on the design and implementation, and analysis of a computer graphics pipeline on a stream architecture using the stream programming model. Imagine was used as the stream architecture, and an OpenGL-like pipeline was completely implemented on it to illustrate that the stream processing architectures are well suited for graphics applications. [23] constructed scalable graphics architectures (called as Chromium) from multiple graphics accelerators housed in nodes.
of a cluster of off-the-shelf workstations. Chromium uses the notion of stream processing to achieve scalability while retaining maximum flexibility. Both works argued that stream architectures can provide the necessary scalability and flexibility needed by future graphics and multimedia applications to achieve higher performance rate.

Today’s graphics processor unit or GPU (VPU) can process over tens of million triangles and over a billion of pixels per second, owing primarily to their stream architecture which enables all additional transistors to be devoted to increasing computational power. On a positive side note, this also has led to the growing use of graphics processor as a general-purpose stream processing engine. Researchers are exploring the idea and porting more and more applications to use GPU as a general-purpose co-processor to achieve higher performance than on a CPU. In the next chapter we study some of the landmark graphics architectures from different generations, and also detail their recent evolution as a stream processing engine.
Chapter 4

Graphics Processor Architecture

In the last chapter we studied about the stream architecture and programming model, and presented that the current generation of graphics processor are modelled after stream architecture. In this chapter, we study some of the important graphics architectures from different generations, and also introduce the latest generation of graphics processor, more commonly referred as GPU or VPU. We begin with an overview on the classical graphics rendering pipeline.

4.1 Graphics Rendering Pipeline

The classic computer graphics-rendering pipeline enumerates a process that geometric objects must experience on their way to becoming pixels on the screen. These objects can be in the form of points, lines or primitives in the shape of polygons and expressed in homogeneous coordinate system. The operations performed on the primitives consist of transformations, divisions (w-division), and clipping as in Figure 4.1. An object processed through the pipeline sees the coordinate spaces in the following order:

- Object Space
- Universe Space or World Space
- Eye Space or View Space
Figure 4.1: **Graphics rendering pipeline.**

- Perspective Space
- Clip Space
- Normalized Device Coordinates Space
- Pixel Space

The transformations form the object space to the universe space is a combination of translation, rotation, and scaling operations. This is done to fit the primitives in the universe space. Universe to eye space conversion consists of pure translation and rotation so that the viewing camera or user is placed at the origin and the direction of viewing being set to +z axis. Then eye to perspective space conversion is done in order to provide mapping from 3-D space to 2-D space. However, the conversion results in technically 3-D space representation with z-axis values stored in the Z-buffer and used later in the pipeline to enable visible and non-visible effects to the object being rendered on the screen. The objects in this coordinate space are represented as 2-D objects with corresponding z-axis values stored in the Z-buffer. Clip space represents the view volume where the objects lying
outside the boundary coordinates of this volume are rendered invisible whereas objects seen to lie within the boundary coordinates are made visible on the screen. This space is chosen to make the clipping arithmetic simpler. It maps the desired visible chunk of perspective space to a fixed region. Later, we perform clip to normalized device coordinate space transformation in order to represent the objects on an internal pixel independent dimensions. From this internal pixel independent representation, we transform the objects in the hardware pixel space from where they are rendered onto the screen. This space represents the screen real state. The whole rendering process can be performed through four major steps:

- Transform to clipping space
- Clip
- Divide by w (real clipping space as dictated by the screen)
- Transform to pixel space

### 4.2 Graphics Hardware

According to one classification of computer graphics architectures [2], various generations can be classified by the capabilities for which the architecture was primarily designed (target capabilities with maximized performance) and not by the scope of the capabilities. Another classification categorizes them on the basis of technology (ASIC, DSP, RISC CPU, and CPU extensions), arrangement (SIMD or MIMD), and programmability (end-user programmability and the relative ease with which they were programmed) [32]. We use the latter classification method to discuss the motivation behind various (representative) graphics architectures.

The advent of VLSI technology and the need for both functional and data parallelism put forward the desire and need for developing special-purpose chips for geometry and image processing Ref. [9]. The motivation behind VLSI graphics architectures was to cut
down on system costs. We describe some of the landmark graphics architectures, and present the latest generation of GPU.

- **Geometry Engine**, 1982
- **Channel Processor or CHAP**, 1984
- **Polygon-Rendering Chip or PRC**, 1986
- **High Performance Polygon Rendering Architecture**, 1988
- **PixelFlow**, 1992
- **Indigo Extreme**, 1993
- **RealityEngine**, 1993
- **InfiniteReality**, 1997
- **Programmable Graphics Architecture**, 2001

### 4.2.1 Geometry Engine

One of the earliest design efforts was the development of the Geometry Engine [10] for Geometry Graphics System (IRIS Graphics System). The Geometry Engine was a special-purpose, four component vector, floating-point processor for accomplishing three basic floating-point operations: matrix transformations, clipping and viewport transformations in computer graphics applications. The Geometry Engine provided limited flexibility and could be configured to act as a part of the matrix subsystem, clipping subsystem and scaling subsystem in the graphics rendering pipeline. The Geometry System was a slave processor and could only accept or yield one format of data and instructions. The graphical primitives were stripped into individual coordinates or vertices and each component processed in parallel by the Geometry System. Twelve copies of the Geometry Engine were arranged
in a pipeline order to achieve performance of the order of 5 million floating-point operations per second. The programming model of the graphics system was used as a hardware subroutine to the IRIS processor/memory system and thus was specific to the platform and architecture of the host/controlling processor.

4.2.2 Channel Processor

The system called Lucasfilm Compositor used the channel processor or CHAP [30], a special-purpose, four component vector, programmable pixel processor to provide digital processing capabilities for special-effects film production. The CHAP performed all the computations and controlled the flow of pixels (made of four components; red, green, blue and alpha) in the Compositor. Scratchpad memories provided for program data storage could be referenced for four or one operand by any arithmetic operation. Vector arithmetic units connected to the scratchpad memories through a crossbar network thus allowed identical or different operations to be performed on all the four pixel components in parallel. The CHAP’s datapath was optimized for 16-bit integer operations. In order to render rich images for film applications, for the first time framebuffer memory banks were configured with 12-bits per channel. The CHAP performed single instruction, four component processing (SIMD). One of the unique features of CHAP was the support for disabling some subset of processors over a range of operations like clamping. Applications required skilled programming efforts and could be complicated by the pipeline delays built into the hardware modules. Pixar’s CHAP could provide interactive usage on limited resolution images while acceptable performance for high resolution, movie quality images required offline rendering.

4.2.3 Polygon-Rendering Chip

Image rendering because of its compute-intensive nature represents the major performance bottleneck in many computer graphics systems. In order to alleviate this problem, all the
necessary components needed for real-time image rendering were put on one-chip VLSI implementation. The chip known as polygon-rendering chip or PRC was designed primarily for mechanical engineering CAD applications [49]. CAD applications require interactive manipulation of solids (shaded polygons). In order to render real-time performance, PRC facilitated pipelining and internal bandwidth for computation and communication. The performance was maximized by pipelining different parts of the fill and shading algorithms and running different functional blocks in parallel. This made the architecture unique in the sense that data could be moved from one functional block to another in one clock cycle via register-register transfers. The architecture also used pixel cache so framebuffer writes and Z-buffer accesses could be done in parallel.

4.2.4 High Performance Polygon Rendering

A novel graphics subsystem was developed to meet the demands for displaying realistic solid objects and support for real-time image rendering and independent windows. The graphics subsystem was partitioned into four special-purpose pipelined subsystems; geometry, scan-conversion, raster and display subsystem with each performing a dedicated task [3]. Each of these subsystems were pipelined into various functional blocks. The Geometry subsystem comprising of Geometry Engines was organized as a single pipeline of floating-point processors with each executing a specific subset of geometric transformations. Geometry Engines operated independently, thus rendering support for SIMD. When taken together, goemetry subsystem achieves an efficiency of 50% not seen by vector processors used in other graphics systems present at the same time. Scan-conversion subsystem responsible for converting vertex data (from geometry subsystem) to individual pixels comprised of Polygon processor, Edge processor and five Span processors (operating parallel) arranged in pipeline with each composed of fixed-point engines. Five Span processors again operating in SIMD acheived an aggregate fill rate of 40 million pixels per second. Raster subsystem was pipelined into twenty Image Engines, each of which operated independently and controlled a section of the frame-buffer memory (providing interleaved
operation). They resulted in extremely powerful and flexible pixel fill operations not seen by other graphics systems at the same time. And finally the Display subsystem interpreted pixels from the frame-buffer memory and routed it to the DAC for display. The graphics subsystem produced some brilliant operating numbers (rendering 100,000 Gouraud shaded and Z-buffered polygons per second) and represented a unique graphics system skeleton that was followed by subsequent graphics architectures. Some of the special features supported were pan and zoom, window ID masking, real-time video, alpha blending and antialiased lines.

4.2.5 Indigo Extreme

One of the limiting factor of the High Performance Polygon Rendering design was its choice for the host processor. This choice made the design only suited for one particular kind of platform and thus rendered limited programmability and portability to other hardwares. Other disadvantages include the throughput of various subsystems being dictated by the speed of the slowest processing step. This results in a non-linear performance gain. Considerable overhead is added with each processor in subsystem stages in the form of sequencer, microcode store, globals data store and interface logic between different pipeline stages resulting in the loss of performance.

In order to eliminate the above mentioned design limitations and disadvantages and to circumvent the bottlenecks seen in the floating-point compute-intensive units used in the geometry subsystem, compute-intensive arithmetic units used in the scan-conversion subsytem, the limited fill rate and the pixel bandwidth as seen into the frame-buffer, SGI introduced a unique computer graphics architecture of the Indigo Extreme aimed at maximizing performance while minimizing size [20]. The decision was made to combine the per-vertex and slope calculations into one unit. The slope calculations were implemented in a microcoded processor because of relatively complex algorithms involved. The advantages achieved through this approach resulted in the increase in vertex and slope processing while minimizing size and complexity. In order to maximize performance, custom-based
floating-point functional units were implemented. Geometry Engine design approach followed the fundamental design principle of maximizing the performance of most expensive functional units. This was achieved by providing custom design data paths to the operands and support for multiple threads of the same algorithm active simultaneously. Multi-port registers were used to render support for multiple simultaneous threads of calculation. More than one Geometry Engines were used in SIMD parallel approach instead of single pipeline order to attain a linear performance gain with minimal overhead required to implement it. This resulted in the processing of eight polygons (primitives) in parallel. Hyper-pipelining was used for designing Raster Engines instead of using multiple copies in an SIMD parallel approach. This was done to minimize the area requirements while achieving the desired performance. Memory bandwidth was increased by using an interleaved frame-buffer across multiple memory banks. The resulting architecture was capable of over 500,000 Gouraud shaded Z-buffered polygons per second and a fill rate of 80 million pixels per second. However, the design was particularly suited to one kind of application (CAD programs). Any deviation from this fixed-function implementation resulted in significant performance loss.

So far we have mentioned architectures that used special-purpose, fixed-function chips (ASICs) for geometric transformations and image rendering arranged in either fine-grained or course-grained SIMD arrangement. These graphics architectures processed graphical primitives as independent vertices or coordinates. Some of the disadvantages have been outlined with the description of each architecture. SIMD architectures do not scale linearly with the number of geometric engines or image renderers. Several important architectures based on MIMD (multiple instruction multiple data) arrangement were introduced to resolve these issues. MIMD arrangements processed the vertex stream as a part of a geometric primitive thus enabling operations like culling and reducing the processing time and increasing parallelism. Here we discuss some of these important architectures and outline their benefits and limitations.
4.2.6 PixelFlow

PixelFlow architecture [38] overcame the scaling limitation as well as the geometric transformations and frame-buffer bottlenecks by using the technique of image composition. Image composition designed for real-time (30Hz) 3D graphics algorithms and applications achieved high rendering performance by applying object parallelism throughout the graphics rendering pipeline. The rasterization processors (renderers, shaders and frame-buffers) were associated with a portion of the primitives (occupying 128x128 pixel regions of the screen), rather than a portion of the screen which is termed as image parallelism. Each renderer computes pixel values for its primitives regardless of their viewport association. The computed pixel values are then transmitted over a network to compositing processors, which resolve the visibility of pixels from each renderer. Some of the unique architectural features offered by PixelFlow were in the form of supersampling antialiasing and deferred shading mechanism. This distribution mechanism of primitives is also classified as sort-last mechanism [37]. Sort-last mechanism issues some outstanding features in the form of linear scalability and being less prone to load imbalances. PixelFlow can be scaled linearly to achieve tens of millions of polygons per second. PixelFlow could be configured to operate as a part of workstation or parallel computer. One distinct feature was that all the components of PixelFlow were general-purpose, programmable and offered a simple programming model. PixelFlow rendered complex primitives such as spheres, quadrics and supported real-time illumination models, procedural and image-based texturing, environment mapping and restrictive antialiasing. Some major limitations were that the image-composition network must support very high bandwidth communication to transfer 128x128 pixel region of pixel data. PixelFlow suffered significant performance loss in case of load imbalances. PixelFlow could also render the final image once all the primitives corresponding to the logical tiles had been transformed and sorted. This limitation could significantly increase the system latency.
4.2.7 RealityEngine

RealityEngine is a massively parallel computer graphics architecture developed by SGI. SGI classified RealityEngine (Figure 4.2) as their third-generation design primarily for real-time (15-30 Hz) rendering of textured mapped, antialiased triangles. The Geometry Engine performed multiple instruction, multiple component processing and required hand-coded assembly code to extract the greatest performance. Texture mapping renders images that are highly visually interesting and looked more realistic. PixelFlow rendered texture mapped images but RealityEngine provided expansive support in the hardware to generate such detailed images. Fragment Generators incorporated were fixed-function pipeline unit and performed the initial generation of fragments, generation of the coverage mask, texture address generation, texture lookup, texture sample filtering, texture modulation of the fragment color and fog computation and blending. RealityEngine also supported 1- and 3-dimension texture maps and thus could be used to render volumetric images. Antialiasing becomes mandatory when rendering texture-mapped images. Antialiasing could

Figure 4.2: **RealityEngine block diagram.** Board-level block diagram of an intermediate configuration consisting of 8 Geometry Engines, 2 raster memory boards, and a display generator board.
be performed either by single-pass (PixelFlow did not support this method) or multi-pass antialiasing methods. Unlike PixelFlow, RealityEngine separated the transformation and fragment generation rates. This made possible the fine tuning of the architecture for unbalanced rendering requirements. Image Engines were assigned a fixed subset of the pixels in the framebuffer. Unlike PixelFlow, all the different functional blocks running in parallel could complete the final image as soon as the last primitive was rendered by the Image Engine. The color component resolution was increased from 8-bits to 12-bits to eliminate visible banding and support for substantial framebuffer composition. RealityEngine used both object parallelism and image parallelism to achieve very high performance rates. Object parallelism implies partitioning either the geometric description of the scene where vertices of a geometric primitive are processed independently by different processors in parallel or the associated object space where different geometric primitives are processed independently by different processors in parallel. Image parallelism splits the image space as fragments where each fragment is processed independently by different processor in parallel.

4.2.8 InfiniteReality

The InfiniteReality [39] graphics system architecture shown in Figure 4.3 was designed with the primary goal of real-time application performance. The architecture was developed as a superset of RealityEngine graphics system. Unlike previous mentioned designs where display list processing had been handled by the host processor, InfiniteReality introduced local display list processing by the graphics subsystem to eliminate host to graphics I/O bottlenecks. RealityEngine and previous architectures distributed primitives among different Geometry Engines using round-robin mechanism. As the Geometry Engines operated in an MIMD arrangement, InfiniteReality extended the support for least-busy distribution mechanism which offered performance advantage over ubiquitous round-robin mechanism. To achieve maximum performance, the Geometry Engines were custom designed
Figure 4.3: **InfiniteReality Engine block diagram.** Board-level block diagram of InfiniteReality Engine with maximum support for 4 Geometry Engines, 4 Raster Memory boards, and a Display Generator board with 8 output channels.

with four stage pipelined arithmetic units executing pixel-components in SIMD arrangement. The screen-space primitives were then input to the Raster Memory board which comprised of a Fragment Generator and eighty Image Engines. Unlike previous architectures, the Fragment Generators in InfiniteReality assembled screen-space primitives (triangle setup) and computed parameter slopes. The performance of InfiniteReality graphics system made possible the use of very large texture databases by introducing a representation called clip-map, and multipass rendering techniques to implement reflections, shadows and spotlights effects. A clip-map can significantly reduce the storage requirements.
for very large textures while multipass rendering techniques can enhance visual realism. The display generator board provided support for flexible and software controlled video architecture that could drive up to eight autonomous display output channels. To counteract fill-rate bottlenecks, InfiniteReality graphics system introduced dynamic video resizing where the scene is rendered to the frame-buffer at a potentially reduced resolution such that primitives are drawn in less than one frame time. Despite the success of MIMD architectures, SIMD systems offered a simpler programming model. This is owing to the fact that MIMD systems imposes a burden on applications and operating systems, which must be able to cope with the arrival of data at unpredictable intervals and in arbitrary order. This often resulted in the design of complex and cumbersome communication and buffering protocols rendering software overheads. SIMD systems operating in the lock-step fashion virtually eliminates these overheads. Also, it is often possible to structure algorithms as several distinct functional blocks, each of which operates on a uniform data type. The rendering pipeline maps naturally onto this structure, and the regularity of the data structures within each phase leads to uniform operations, providing a good fit with the SIMD programming paradigm. Finally, SIMD architectures usually contain thousands of simple processing elements. Because of their sheer numbers, good performance can often be achieved even though processors may not be fully utilized [12].

4.2.9 Graphics Software Libraries

The realm of computer graphics applications was growing but there was no real coordinated effort to develop hardware independent graphics software libraries that could be used to render simple or complex scenes efficiently. All the above mentioned graphics system designs were developed independently without any coercion towards industry standard compliance. These graphics systems could not be implemented on a variety of platforms with a range of graphical capabilities. There existed several independent standards most notably PHIGS (Programmer’s Hierarchical Interactive Graphics System), PHIGS+, PEX
(PHIGS extension to X window system) [15] that could render 2D or 3D objects on limited graphics systems with limited graphical capabilities making program portability problematic. Also we have mentioned that it required considerable knowledge and expertise to program graphical applications efficiently and was mostly carried out by the systems’ designers. OpenGL developed by SGI is an emerging graphics standard. It is a low-level graphics library specification that provides advanced rendering features while maintaining a simple and flexible programming model. OpenGL is rendering-only, so it is independent of the methods by which the user input and other window or non-window system functions are achieved, making the rendering portions of the graphical program that uses OpenGL platform and operating system independent.

4.2.10 Streaming CPU Extensions

PixelFlow, RealityEngine, InfiniteReality, etc represented massively parallel graphics architectures with varied range of capabilities. However, the above mentioned graphics architectures were accessible at very high cost for special-purposes. Technological advancements in the VLSI technology made possible visual realism experience for desktop personal computers by incorporating SIMD arrangement in the general-purpose processor designs. Intel Corporation introduced MMX technology for multimedia applications. MMX technology exploited SIMD, where a single instruction operates over multiple data elements in parallel. MMX technology extended the scalar integer instructions into SIMD versions and included add, subtract, multiply, compare and shift instructions to process data in parallel. AMD extended the SIMD concept by inventing 3DNow! technology to reduce the bottleneck for floating point intensive multimedia applications. 3DNow! technology introduced a shared multimedia unit which included a single instruction, multiple-data (SIMD) floating-point adder, a shared SIMD MMX and 3DNow! multiplier, a reciprocal or a square-root approximation unit, and an MMX shifter [41]. Motorola developed Altivec technology providing a richer set of floating point SIMD instructions set than previously available in
desktop computers. Motorola's Altivec design offered 128-bit datapath, Permute unit offering more complex data re-organization and thus enabling higher data parallelism (SIMD), more arithmetic functional units for simultaneous execution, data prefetch for hiding memory latency and special instructions like multiply-add which is widely used in DSP and media applications. Intel further added extensions like SSE, SSE2 and later SSE3 to achieve higher performance when running floating-point compute intensive calculations.

All these different graphics architectures were a part of fixed-function graphics rendering pipeline offering limited user-programmability. A fixed-function graphics rendering pipeline could only implement a limited set of predefined algorithms. In order to provide the developers the ability of creating more visually enhanced realistic images, the fixed-function pipeline was replaced by the programmable graphics hardware. Present generation graphics hardware has fully programmable vertex and fragment processors. The following block diagram as in Figure 4.4 gives an illustration of the computer system architecture.

In the next section we give a detailed explanation of programmable graphics hardware.

## 4.3 Programmable Graphics Hardware

In this section we describe the user-programmable vertex and fragment engine. We also describe the GPU programming model and discuss the design in the areas of input, output, data path, and instruction set architecture.

### 4.3.1 Vertex Processor

Vertex shaders provide a programmable way to modify values associated with each polygon's vertex, such as its color, texture coordinates, and position. A typical vertex shader seen in Nvidia 6 Series GPUs is shown in Figure 4.5. The need for such a component arose from the need of generating more visual realistic graphically rendered objects. The vertex shader replaces the transform-lightning section of the original fixed function graphics
Figure 4.4: **Overall system architecture.** The diagram illustrates different components of the PC hardware with the associated data flow. GPU communicates with the CPU via the North Bridge. Recent advances in the bus architecture in the form of PCI-X and PCI Express has enabled higher data rate flow between the GPU and the CPU.

pipeline. Vertex shader can be written in assembly language in order to render maximum performance or in an high-level shading language, for instance, Cg [34], Sh [36], or HLSL. The instructions of vertex shader program have access to four different memory locations: the per-vertex data of an incoming vertex, constant memory, temporary registers, and per-vertex output-registers. Each one of these memory locations stores a collection of four-dimensional vectors; each component of such a vector is a 32bit, signed floating-point number. These float-vectors represent a combination of positional data (xyzw), texture coordinates (uvwq), colors (rgba), or simply a collection of four scalars (abcd). The per-vertex data memory contains per-vertex data such as model-space vertex-coordinates, vertex color, and texture coordinates. It is only a read-only memory where only the application can write. Constant memory typically stores world and per object data that changes.
Figure 4.5: **GeForce 6 Series Vertex shader block diagram.** Vertex processor has a separate functional unit for scalar and vector data. The vertex texture fetch unit helps to fetch data from any of the memory locations: constant memory, temporary registers, and per-vertex output registers. The recent addition has been the Branch unit which enables the vertex processor to perform scatter operation where it can write results to computed memory address locations.

per-frame and/or per-object, for example, light properties, transformation matrices, or material properties. Temporary registers are used to store intermediate results generated by vertex shader. Per-vertex output registers contain results generated by the vertex shader. Clip-space vertex-coordinates, diffuse and specular color, and texture coordinates are typical outputs. These per-vertex output registers are write-only and feed into the rest of the graphics pipeline. The vertex shader allows us the possibility of exploiting SIMD as computations for every vertex is independent of the others. Vertex shaders can increase the speed of algorithms that compute more elaborate lighting models. They can calculate values that slowly change over the surface, and the hardware will then interpolate such values.
With the flexibility of the vertex shaders developers are able to perform more advanced operations including:

- Procedural geometry
- Advanced vertex blending for skinning and vertex morphing
- Texture generation
- Advanced keyframe interpolation
- Particle system rendering
- Real-time modifications of the perspective view (lens effects, underwater effect)
- Advanced lighting models
- Initial steps in displacement mapping procedure

Some of the recent changes to the vertex processor (Nvidia GeForce 6 Series) instruction set can be briefly enumerated as:

- Increased instruction count.
- More temporary registers.
- Support for instancing.
- Dynamic flow control.
- Vertex texturing.
- Advanced mathematical functions like exponential, reciprocal and trigonometric functions.
Figure 4.6: **GeForce 6 Series fragment processor.** The figure illustrates the fragment processor and texture pipeline operating in concert to apply a shader program to each fragment independently.

### 4.3.2 Fragment Processor

Fragment processors (Figure 4.6) provides a flexible way of creating realistic geometric models with many different effects. A program or fragment shader is created with set of instructions that operate on a set of constants, interpolated values, and retrieved texture values to generate a pixel color, and an optionally alpha value. More recently, they can also provide the ability of general dependent texture reads where the texture coordinates are computed and then used by the fragment or pixel shader to fetch the data from the texture memory, also known as gather operation. Fragment processors operate in an SIMD fashion, where processing is performed on four-element vectors (input elements) in parallel. Fragment processors are not capable of scatter operation, where data is written to a
computed memory address by the fragment shader or program. A pixel or fragment shader (program) is written to execute on these fragment processor. In a typical fragment program, it has three sets of inputs: the interpolated diffuse and specular colors and alphas, eight or more constants, and four or more texture coordinates. Each of these input is a vector of upto four elements. After performing the arithmetic operations, the results from the fragment shader are written on the framebuffer or on the framebuffer object attached texture images for feedback purposes.

Some of the recent changes to the vertex processor (Nvidia GeForce 6 Series) instruction set can be briefly enumerated as:

- Increased instruction count.
- Multiple render targets.
- Dynamic flow control or branching.
- Indexing of attributes.
- Upto ten full-function attributes.
- Centroid Sampling.
- Support for floating point 32 and 16 internal precision.

Next we describe the graphics memory which is fundamentally different than the system main memory, and has played a vital role in using GPU for general-purpose computations.

4.3.3 Graphics Memory

Another component of the graphics rendering pipeline that has evolved over the last several years is the texture memory unit (Figure 4.7). Texture unit is the read-only memory interface to the vertex and fragment processors. Output can be written to the memory on the GPU in three different ways: first, written to the framebuffer memory that can be displayed; second, render-to-texture also known as write-only memory interface mechanism
to implement feedback of GPU output to input without going back to the host processor; third, as an indirect feedback in the form of copy-to-texture, which requires a copy from one location in the GPU’s memory to another. Memory reads and writes are fundamentally different on the GPU. Memory reads are provided by the texture unit while memory writes are provided through render-to-texture unit. Also, GPU memory can be accessed only in the form of streams. The three types of streams available to the GPU programmer are vertex streams, framebuffer streams, and texture streams. There is another stream type called as the fragment stream, but it is produced and consumed entirely within the GPU. Over the last couple of years, we have seen video graphics cards providing ample memory storage space to render some quality images. The texture memory is capable of storing images in 1D, 2D or 3D formats. Recently, supported has been added to support the image formats for non-power-of-two (NPOT) textures. These NPOT find exclusive use in scientific and engineering applications implemented on the GPU.

Figure 4.7: GPU memory hierarchy. GPU has its own set of caches, registers and mass storage area to accelerate data access during computation. However, in order to operate the data, the programmer needs to copy the data from the system memory to the GPU texture memory also known as the video RAM or VRAM. The texture memory can support 1D, 2D and 3D data formats. The fastest data access is achieved with 2D data format as the texture memory has native support for this format.
4.3.4 GPU Programming Model

So far we have described main programmable components of the GPU. Here we present the description of the programming model and discuss the data flow within the GPU.

Precision and Data Type

The GeForce FX and GeForce 6 Series GPUs support 32-bit and 16-bit floating point formats (called float and half, respectively). The float data type is like IEEE single precision floating point format, with an s23e8 format. The half is also IEEE like, in an s10e5 format. Normally, in graphics applications, half data type delivers higher performance than float data type. However, the introduction of float data type is also a major reason of porting more scientific and engineering applications to the GPU. The common data type in 3D graphics are 3 and 4 component vectors, for example position, normal, texture coordinates and colors. Therefore, the basic data type is written as quad-float vector in the form of \((x, y, z, w)\).

Scalar and Vector Handling

GPU programming model provides a novel technique of packing similar data types as vectors and then operating on them to exploit data parallelism. This is also known as scalar packing and often within the graphics pipeline, scalar and vector are mixed together and operated upon at the same time. In order to grant more flexibility and achieve better performance, swizzle operations were also allowed were scalar components can be arbitrarily rearranged or replicated, and also within a vector, only a part of the vector or components can be operated upon while the rest can keep their original values intact. But poor packing can also lead to severe performance cuts and makes it harder for the compiler to optimize the code.
**Data Flow**

The GPU data flow can be compactly described as:

- First, commands, textures and vertex data are received through the shared buffers from the host CPU. The CPU sends these commands and data that initializes the GPU state.

- The vertex processor fetch unit parses these commands and reads the vertices referenced by the commands.

- Vertices are then grouped into primitives, which are points, lines or triangles. Also part of the primitives that are not visible are removed and the data is setup for the rasterization stage.

- The rasterization stage computes which pixels are covered by each sample, and associates appropriate depth and color information with each sample or fragments.

- The data or fragments from the rasterization stage is passed to the fragment processors which provide the final level of detail to each pixel mapped on the screen.

- In the final pass, the pixels pass through the fog stage which is often used in graphics applications to render the image highly realistic. The final image is either written to the framebuffer which could be the screen or stored in the framebuffer attached texture memory object. This framebuffer attached texture memory objects can be used for reading back the data as in a feedback loop.

So far we have studied GPU architecture, when used as a graphics pipeline. Over the last few years, there has been tremendous effort by the researchers of using GPU as a stream co-processor owing to the availability of large amount of programmable floating point power and memory bandwidth, which can be exploited for compute intensive scientific and engineering applications. In the next section, we focus on using GPU for non-graphical applications, which is the main theme of our thesis.
4.4 GPU for Non-Graphics Operations

As stated in the previous section, that current generation of GPU can be viewed as having two user-programmable stages. These two blocks: vertex processor and fragment processor are designed to exploit instruction-level parallelism, data parallelism and task parallelism. Both the computing blocks offer support for floating point 32 and 16 precision.

In order to use the GPU for non-graphics applications, we need to write our application either as a vertex or fragment program. This means that performing a general-purpose computation leading to a numerical result can be viewed as computing a color value (also known as a shading) of a vertex or pixel. The vertex and fragment both needs access to the data in the same way data is accessed on the CPU. This is provided by the texture unit which act as a random-access data fetch unit and provides very high memory bandwidth. The transfer of data from the main stem memory to the GPU texture memory is a cause of concern for GPUs, as it is hindered by the AGP or PCI Express data-bandwidth bottleneck. This makes programmable GPUs ideal for applications with high arithmetic intensity. However, there are several issues when performing general-purpose computations on the GPU and they can be summarized as:

- Data transfer between CPU and GPU is hindered by the data-limited bandwidth.
- Current GPUs provide IEEE like precision and this could be trickier when implementing scientific computations on them.
- The developer needs to be aware of graphics pipeline when writing general-purpose applications on the GPU.
- Also lack of standardization makes the job of the developer even more cumbersome.

This chapter discussed different GPU architectures and the current generation of user-programmable GPU architecture. It detailed about the different computing blocks present
in the latest generation of GPUs and provides an explanation why GPUs are being exploited for general-purpose computations. In the next chapter we discuss some of the major general-purpose applications ported to GPUs and presents the background work that inspired us to implement Barnes-Hut algorithm on the GPU.
Chapter 5

Previous Work

In recent years, there has been an upsurge of developing high-performance general-purpose scientific and engineering applications on programmable graphics hardware [1], [1]. The goal of these efforts is to accelerate the simulation of the particles on the GPU to achieve performance gain in terms of real-time speed, while maintaining physical accuracy. In the last chapter we saw that GPUs exposes stream processing architecture, which renders support for multi-level parallelism (concurrency) and also exposes inherent data locality by partitioning communication and data storage structures. With the advent of programmable GPUs, efforts have been made to map high-performance computing applications to the GPU, e.g. scientific computations like FFTs, linear algebra solvers, differential equations solvers, multi-grid solvers and applications to fluid dynamics, visual simulation, ice crystals; geometric computations like Voronoi diagrams, distance computation, motion planning, collision detection, object reconstruction, visibility; advanced rendering like ray-tracing, radiosity, photon mapping, sub-surface scattering, shadows; and database operations on aggregates, predicates, Boolean combinations, selection queries for large databases Ref. [1].

In this chapter we visit some of these applications and look at the work that motivated us to implement Barnes-Hut $N$-Body method on the GPU.
Figure 5.1: Matrix Multiplication. Performance of multiplying square matrices on different hardware.

5.1 Fast Matrix Multiplies using Graphics Hardware

Matrix-matrix multiplication was performed using a parallel processing algorithm on low-cost graphics hardware. The parallel processing algorithm performs matrix-matrix multiplication using SIMD. In this paper [25], the authors visualize this algorithm on the graphics hardware. Texture maps are used to store elements of matrix A and B. Multi-texturing is used to represent multiplication between elements of A and B. Finally we use blend operation to sum all the individual element multiplication. This final sum matrix is displayed on the screen (stored in framebuffer) from where it is transferred to the main memory in order to be analyzed for results. Thus this paper outlines general-purpose parallel processing on GPUs. However, the computation only provided integer arithmetic owing to the hardware limitation. Owing to this limitation, serious comparisons were not made, however this computation on GPUs can provide enormous performance gains as they don’t
suffer from limitations of memory system. However, more recent work [14] (Figure 5.1), showed that huge performance gains can be achieved using GPUs for matrix multiplication.

5.2 Physically-based Visual Simulation on Graphics Hardware

Coupled Map Lattice (CML) is a standard practice for solving real-world dynamic phenomena. CML uses a set of simple local operations to model complex global behavior. CML is a transformation from the continuous real-world (dynamic) state to discrete nodes on the lattice that interact with other nodes on the lattice according to pre-determined rules. Graphics hardware (GPU) not only offers a natural environment for rendering such a behavior because lattice nodes can be mapped to array indices of texture maps but also renders fast such a dynamic behavior. In this paper [22] (Figure 5.2), authors provide a detailed implementation of some commonly used CML techniques (numerical algorithms for initial-value partial differential equations) needed to render dynamic phenomena (cloud formation, boiling water simulation, etc) on the graphics hardware. In the implementation, the framebuffer is used to store intermediate values and the textures serve as main memory for state storage. The CML operation is divided into 3 stages: setup graphics hardware, one-to-one mapping from texture to screen and final state stored in texture called as render-copy (divided into 4 steps: Neighbor Sampling, Computation on Neighbors, New State Computation, and State Update). Thus, this paper outlines using graphics hardware for solving partial differential equations to model dynamic state phenomena with performance gain of up to 25 times. However, limitations observed were due to limited floating point arithmetic support (in vertex and pixel shader) and rounding-off errors generated due to the conversion of unsigned numbers into signed numbers for computation purposes.
<table>
<thead>
<tr>
<th>Resolution</th>
<th>Software</th>
<th>GeForce 3</th>
<th>GeForce 4</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>266.5</td>
<td>1252.9</td>
<td>1752.5</td>
<td>4.7 / 6.6</td>
</tr>
<tr>
<td>128x128</td>
<td>61.8</td>
<td>679.0</td>
<td>926.6</td>
<td>11.0 / 15.0</td>
</tr>
<tr>
<td>256x256</td>
<td>13.9</td>
<td>221.3</td>
<td>286.6</td>
<td>15.9 / 20.6</td>
</tr>
<tr>
<td>512x512</td>
<td>3.3</td>
<td>61.2</td>
<td>82.3</td>
<td>18.5 / 24.9</td>
</tr>
<tr>
<td>1024x1024</td>
<td>.9</td>
<td>15.5</td>
<td>21.6</td>
<td>17.2 / 24</td>
</tr>
<tr>
<td>32x32x32</td>
<td>25.5</td>
<td>104.3</td>
<td>145.8</td>
<td>4.1 / 5.7</td>
</tr>
<tr>
<td>64x64x64</td>
<td>3.2</td>
<td>37.2</td>
<td>61.8</td>
<td>11.6 / 19.3</td>
</tr>
<tr>
<td>128x128x128</td>
<td>.4</td>
<td>NA</td>
<td>8.3</td>
<td>NA / 20.8</td>
</tr>
</tbody>
</table>

Figure 5.2: CML Performance Comparison. A speed comparison of our hardware CML boiling simulation to a software version.

5.3 GROMACS

GROMACS is a molecular dynamics method that provides extremely high performance compared to all other molecular dynamics programs. Molecular dynamics predicts how proteins assemble themselves, which determine how they operate in biological systems by simulating the dynamics of large molecules using Newton’s equation of motion for the atoms. The non-bonded forces computation on a CPU accounts for 90 – 95% of the total simulation runtime. The above mentioned method modified the original algorithm to perform all non-bonded force calculations on the GPU using Brook, a high-level GPU programming language to achieve twice as much computation as the CPU implementation. The atoms are grouped together as molecules and a common neighbor list is created for such molecules. These neighbor lists are then transferred over to the GPU where non-bonded force computations are performed.
5.4 Particle-Mesh N-Body Method on Graphics Hardware

An N-Body simulation uses numerical integration techniques (based on Newton’s laws of motion) to approximate the evolution of a system of N bodies whose mutual interactions are governed by a potential function. Harris et al. [1] propose a new method similar in approach to the particle-mesh strategy to calculate potential function on every body with half-precision floating-point values. The new method relies on the notion of array addition for superposition of potential fields, linear interpolation for arbitrary, and partial derivatives to determine the force from the potential field. The basic idea behind this approach is to sample the potential field of a single particle on a regular grid. However, the biggest disadvantage of using mesh-based methods are that they have difficulties handling non-uniform particle distributions and thus offers limited resolution.

In this thesis, we explore N-Body method for galaxy evolution based on treecodes as the tree structures partition the mass distribution into a hierarchy of localized regions, so that when calculating the force on a given particle, the tree region near the particle in consideration is explored in more detail than the more distant regions. Furthermore, treecodes are gridless and have no preferred geometry or orientation and totally relies on the available particles distribution.

5.5 Radiosity on Graphics Hardware

Radiosity is a widely used technique for global illumination and recently, it has been implemented on the GPU to run at interactive rates [11]. [11] exploited the computational power and programmability of modern graphics hardware.

The hierarchical subdivision radiosity algorithm is based on the hierarchical N-Body problem. The N-Body problem shares many similarities with the radiosity problem which suggest that these ideas can lead to an increase in performance on the GPU. In both the N-Body and the radiosity problem, there are $n(n-1)/2$ pairs of interactions. Moreover, just as the magnitude of the form-factor between two patches falls off as $1/r^2$, the gravitational or
electromagnetic forces also fall off as $1/r^2$. However, there are several differences between the two implementations. One major difference between the two problems is the manner in which the hierarchical data structures are formed. The radiosity algorithm begins with a few large polygons and subdivides them into smaller and smaller patches. Our $N$-Body algorithm begins with $n$ particles and cluster them into larger and larger groups. Another difference is that radiosity problem is inherently non-linear because of occlusion where intervening opaque surfaces can block the transport of light between the two surfaces. The $N$-Body problem on the other hand is linear and takes advantage of linear superposition, the principle of superposition states that the potential due to a cluster of particles is the sum of the potentials of the individual particles. Finally, the radiosity problem is based on an integral equation, whereas the $N$-Body problem is based on a differential equation.

## 5.6 Ray Tracing on GPU

Purcell et al. [42] showed that the entire raytracing process can be implemented on a programmable GPU using a uniform grid acceleration data structure arguing that the uniform grid enables constant-time access to the grid cells and takes advantage of coherence using the blocked memory system of the GPU. Work by Foley et al. [16] challenged the idea of using uniform grid acceleration structures for GPU-based raytracers. They demonstrated kd-tree traversal algorithms suitable for GPU implementation and integrated it into a streaming raytracer. Lars et al. [46] did an elaborate study of GPU-based raytracing methods. They compared GPU based traversal of kd-tree and uniform acceleration grids with a novel bounding volume hierarchy traversal scheme. All these works have similar motivation to that of our thesis, in that they translate the high-performance computation from the CPU to the GPU. The data structure is created on the CPU and buffered over to the GPU for traversal and computations. These works have been possible because of the increasing flexibility and programmability of the GPU. The problem of galaxy evolution is one of the grand challenge problems, and typically implemented on vector machines
or parallel processors. We describe an implementation of the Barnes-Hut treecode on the GPU, a hierarchical \textit{N-Body} method based on the stream programming model. Our work takes advantage of the fragment programs, dynamic flow control, single precision floating point operations, the floating point texture and the multiple floating point render targets (framebuffer objects) of the latest generation GPUs.

\section*{5.7 Octree Textures on the GPU}

Texture mapping is a very efficient technique of enriching visual appearance of polygonal models with details. Textures not only store color information, but also other necessary details like normals for bump mapping and other attributes to create appealing surface effects. However, texture mapping on complex meshes leads to distortions and artifacts due to 2D parametrization where a 2D texture coordinate is associated with every mesh vertex. The 2D parameterization errors can be eliminated by defining the texture inside a volume enclosing the object, thus storing color where the surface intersects the volume. 3D hierarchical data structures named octree textures have been used to efficiently store color information along a mesh surface without texture coordinates \cite{19}; \cite{7}. \cite{27} presented a new interactive method to texture complex geometries using sprites stored in a 3D hierarchical structure called octree textures surrounding the object’s surface. The authors were able to render complex geometries at very high resolution, while using little memory and without the need for a global planar parametrization. They have also implemented octree textures on the GPU to demonstrate a surface painting application where they store color information along a mesh surface, and a non-physical simulation of liquid flowing along a surface in which they demonstrate using an octree structure to simulate on GPU liquid flowing along a mesh. The authors also argue the importance of converting octree texture into a standard 2D texture as GPU natively supports 2D textures.

In this chapter we saw wide range of areas where GPUs have been successfully used to enhance performance. We also introduced other works that motivated us to implement
Barnes-Hut $N$-Body method on the graphics hardware. In the next chapter, we finally explain our strategy to implement Barnes-Hut on the GPU. We give a detailed account of our software implementation of Barnes-Hut algorithm on the GPU.
Chapter 6

Implementation

We now explain how to map the hierarchical octree structure in the texture memory, and how to traverse it on the GPU. We provide a complete Barnes-Hut treecode solution on the GPU. This is implemented in a fragment program or pixel shader executed per-pixel on the GPU.

Hierarchical data structures such as octrees that form the basis of the Barnes-Hut treecode, is most commonly obtained by a recursive subdivision of space. A cubical root node is used to encompass the full mass distribution, which is repeatedly subdivided into eight daughter nodes of half the side-length each, until one finally ends up with the leaf nodes containing lone particles. The force is computed on each particle by traversing the tree, i.e. starting at the root node, the multipole accessibility criterion is checked. If the answer to this criterion is 'yes’, the multipole force is used and the walk along this branch of tree can be terminated, otherwise, the node is opened, which means that its daughter nodes are considered in turn. This process is followed till we traverse the entire tree structure. In the next section we describe our complete Barnes-Hut treecode algorithm on the GPU. We present an overview of our algorithm which is then followed by an elaborate explanation for each component in subsequent sections.
6.1 Algorithm Overview

We summarize the overall Barnes-Hut treecode algorithm on the GPU in this section. The complete Barnes-Hut treecode simulation of the evolution of galaxies is depicted in the flowchart drawn in Figure 6.1. Simulating Barnes-Hut treecode on the GPU follows these steps:

- We flatten our octree structure into a 2D texture [31]; [29]; [21].

- We pack the relevant information from the octree structure in two separate 2D 128-bit 4-component floating point texture keeping the neighboring relationship of the original model. We also create 128-bit 4-component 2D textures to store particles position and velocity coordinates.

- We attach three 128-bit 4-component floating point textures, and bind them as to serve as logical buffer, or render target buffer to store new particle position, velocity and acceleration coordinates. In order to render to a texture, we use the EXT_render_target extension, which is simpler to use and more efficient in terms of performance and speed. These attached framebuffer textures along with the position and velocity coordinates textures switch back and forth (ping-pong buffers) between them to store the results of the time-step during the simulation.

- During the simulation, textures are updated dynamically at every time-step by binding to the framebuffer.

- In the present generation GPUs, it is most efficient to use RGBA texture. Each RGBA texel has 4 channels, hence can store uptp 4 scalars or a vector with upto 4 components. The fragment program performs on each pixel stored in the position coordinate texture and traverses the tree to generate new particle particle position and velocity coordinates every time-step.
Figure 6.1: Algorithm. Simulation of evolution of galaxies algorithm overview on the GPU.
Figure 6.2: Octree. Starting with the root node, the node is divided every time it contains more than one particle into equal spaces. The process is repeated until we reach the desired tree depth or if there is only one particle present inside that node/cell.

6.2 Octree Construction

Our octree data structure is built offline on the CPU shown in Figure 6.2. Creating an octree or any hierarchical data structure directly on the GPU is very difficult, mainly because of dynamic memory allocation and pointer creation which is not yet available in the GPU programming model. In a dynamic tree creation process, we only specify the maximum tree depth which in our case is up to 32 levels. We then dynamically allocate memory to provide space for loading particles. The primary reason for dynamic memory allocation is to conserve memory space as static allocation is not only an unlikely viable solution, but also totally inefficient. Any static allocation using arrays would require space of the order of at least $8^{31}$ for an octree with maximum depth of 32 levels. Also, the pointer arithmetic requires a scatter operation, which is not possible on today’s GPU (fragment processors). Fragment processors on the GPU are only capable of gather operation. Furthermore, the tree creation process is a recursive (sequential) process and does not fit well on the GPU.
(stream programming model) as they are designed to take advantage of data level parallelism.

The primary data structure used in the code is an eight-way tree composed of bodies and cells (Appendix A). Bodies form the leaves of the tree while the internal nodes are represented by cells. Node structure is common to the bodies and cells, and contain information like mass and position vector for both bodies and cells. Body structure contains velocity and acceleration vector, and potential force for each body. Cell structure represents the internal eight-way branchings of the tree and stores information like the critical radius of the cell, number of particles in the cell, cell size, cell centers of all the cells.

The initial particle position, particle velocity and their masses can be either generated using the Plummer model or loaded from a data file. Plummer model is a special case for distributing bodies (globular clusters or heavenly bodies) in space such that no two bodies occupy the same position or have the same set of coordinates. The current positions of the particles are then used to determine the dimensions of the root cell which encompasses all the particles. Since particles move between time-steps, the bounding box or root cell dimensions may change and needs to be computed every time-step. Now, the tree is constructed by adding particles one by one into the initially empty root cell, and subdividing a cell into its daughter nodes as soon as it contains more than a single particle [6]. The resultant tree structure contains internal nodes which are space cells and leaves contain individual particles. Each cell may hold the addresses of up to 8 daughter nodes, which may be either cells or particles. The tree is then completed together to form a directed graph, which could be traversed starting from the root cell out towards the leaves. Since $n$ particles are loaded into an empty tree, and since the expected height of the tree when the $ith$ particle is loaded is $logi$, the expected computational complexity of the tree construction process is $nlogn$. The expected computational complexity of this phase is $O(n)$ [47]. This whole process of tree creation and computing the cell centers takes about 5 - 7% of the overall execution time of the Barnes-Hut treecode simulation.
6.3 $N^3$-tree memory map on the GPU

Figure 6.3: Octree mapped to 2D texture. The diagram pictorially represents how an octree is converted into a 2D mesh structure which is then stored in the GPU texture memory. The different colors in the diagram represent nodes belonging to different levels of the octree. For instance, the first black colored square/node represents the first daughter node of the root cell. This daughter node then branches off to its daughter node which are highlighted as grey in color. The second node in this series branches off to its daughter nodes depicted by purple colored nodes. Similarly, all the different colored nodes represent nodes belonging to different levels of the tree. Nodes which are of the same color belong to the same parent and are neighbors or siblings in relation to each other. The figure represents mapping of the octree depicted in Figure 6.2.

The octree created on the CPU is a dynamic sparse data structure as it is updated every time-step by the CPU. This can be stored on the GPU in several different ways as illustrated by [27], [43], [29], [28], [26]. In order to make the most efficient use of the limited GPU memory, and to minimize the bandwidth cost of the kernels, we propose to map our octree on the GPU as a flattened 2D array using several packed data structures as shown in Figure 6.3. We preferred to store our data structures as 2D textures maps on the GPU memory because:
- Barnes-Hut treecode simulation requires at least single precision floating point support.

- Better storage efficiency as compared to 3D textures.

- 2D texture arrays allow us to update the entire octree structure in a single rendering pass.

As implemented (Appendix B), we only store relevant information needed for tree traversal on the GPU. We create 4 128-bit 2D floating point textures to store this relevant information on the GPU as shown in Figure 6.4. Each node in MassAddressNode represents a cell in the octree on the CPU. MassAddressNode contains all the relevant information needed to traverse to the next level on the GPU. mass determines if the node is a cell or a body. parent as the name signifies determines the parent of the current node. child determines the first child of the current cell. neighbor determines the sibling of the current node (could point to a cell or a body). parent, child, neighbor are the indices of the array instead of pointers (as on CPU). We attempt to pack these variables in the same texel to reduce the number of textures to be activated and the number of texture fetches. This also improves data locality as well as the cache coherence of the textures.

These array indices are known when mapping the tree onto implicit array representation. This array representation is also known as sequential representation because it allows a tree to be implemented in a contiguous block of memory (an array) rather than via pointers connecting widely separated nodes. We modify this array representation to allocate an array element if and only if it represents a node of the tree. We create this 2D array first on the CPU and then map it onto the GPU. “PositionCritNode” (Appendix B) has one-one correspondence with “MassAddressTextureNode” (Appendix B). “xaxis”, “yaxis”, “zaxis” forms the position vector of the cell center or the center-of-mass the cell. In case, the current node is a body, xaxis, yaxis, xaxis forms the position vector of the body or particle. “rcrit2” is the critical radius of the cell or zero if it points to a body or particle. We also create textures to store particle position vector, potential and velocity vector.
Figure 6.4: **2D textures on the GPU.** The figure shows different data structures mapped on the GPU. MassAddressTexture and PositionRcritTexture contains information as stored in the octree on the CPU. The information is divided into two data structures as the pixels can only store worth 128-bits on the GPU in the texture memory. MassAddressTexture and PositionRcritTexture has one-to-one direct mapping with the octree on the CPU, which means that the data cells in each of the data structures can be mapped to nodes on the octree. This is pictorially represented by painting cells by identical colors. ParticlePhiTexture and VelocityTexture contains information specific to the particles or leaves stored in the octree. They do not contain information related with the internal nodes on the octree. Again they are divided into separate data structures as the pixels can only store 128-bit information. ParticlePositionPhiTexture and VelocityTexture are painted with identical colors to represent that cells with similar colors store information for the same leaf in the octree.
6.4 Kernels

6.4.1 Traverse

GPUs have multiple fragment processors. They are fully programmable and operate in SIMD mode on the input elements, processing four element vectors in parallel. For general-purpose applications on the GPU, the fragment processors are typically used more heavily than the vertex processors because: first, there are more fragment processors than vertex processors; second, the output of the fragment processors can be stored directly into the memory; third, the values stored in the texture memory can be read directly by the fragment processors. The stream programming model allows GPU to execute kernels in parallel, thus process many data elements simultaneously. This data parallelism ensures that the computation on one stream element cannot affect the computation on another element in the same stream. As a result of this, the only values that can be used in the computation of a kernel are the inputs to that kernel and global memory or texture memory reads. This data parallelism is the most fundamental reason to the speedup offered by GPUs over serial processors. Current GPUs have restrictions in their programming model. But slowly, GPU vendors are relaxing these restrictions and providing options to utilize them for tasks other than simply feed-forward triangle rendering. Some of the recent developments in the GPU programming model are full-pixel branching support, support for multiple render targets, and infinite length pixel programs. In a typical implementation on the CPU, the force on a particle can be approximated by making a recursive traverse of the tree structure starting at the root cell and exploring different parts of the tree at different levels of resolution. The force computation using recursive algorithm can be expressed as:

- The recursive requires the use of stack to keep successive generations of local variables and parameters.
- Each time that a recursive function is entered, a new allocation of its variables is pushed on top of the stack.
Figure 6.5: Tree traversal. Tree traversal kernel starts with the root node stored in the MassAddressTexture. It traverses to the first daughter node and based on the MAC traverses further down that node if it is a cell, else traverses to its neighbor. Once the entire tree traversal is completed, the tree traversal kernel comes back to the root node (starting point).
• Any reference to a local variable or parameter is through the current top of the stack.

• When the function returns, the stack is popped, the top allocation is freed, and the previous allocation becomes the current stack top to be used for referencing local variables.

GPU programming model is still not completely developed. Features supporting more general-purpose computations are regularly added to its programming model. Currently, GPU programming model does not support stack programming. We cannot write and read the same value during the same rendering pass, thus making recursive algorithms inefficient. Once the tree is structured as an implicit array representation in texture memory, we access it from a fragment program. The tree is traversed for every particle by the fragment program. Details about different data structures storing the information about the octree can be referred from Appendix B. Here we describe in detail the tree traversal using the fragment program written using ARB_fragment_program:

• Suppose we traverse the tree for particle \( p \) (RGBA value) stored in the “ParticlePositionPhiTextureA/B”. The tree traversal starts from the root node and successively visits all the nodes in the tree depending on multipole accessibility criterion (Figure 6.5).

• The tree traversal is initialized with \( \text{Root} = (0,0) \) which corresponds to the tree root index and is used to fetch information stored as RGBA value from the texture “MassAddressTexture”. “MassAddressTexture” is 128-bit 4-component floating-point texture. This texture contains information like mass of the current node which decides if the current node is a cell or a particle. The first node of the “MassAddressTexture” corresponds to the root of the octree structure. Mass of this node corresponds to mass of all the particles in the octree.

• Starting with this root node, we visit the first child pointed to by the child field stored within this node on “MassAddressTexture”. The address of the child field is stored
as 1D array index. We use address translation to convert 1D array index into a 2D texture address to visit the child node in “MassAddressTexture” using DECODEINDEX fragment program routine. We rely on dependent texture lookups or texture indirection to return data from the texture. This requires the hardware to support an arbitrary number of dependent texture lookups, which is the case in most modern GPUs.

- The current child node of the tree has three distinct possibilities:
  
  - First, if the current node is a particle, calculate the interaction between p and the current node.
  
  - Second, if the current node is a cell for which we can accept multipole accessibility criterion, calculate the interaction between p and the current node and the tree is not traversed further down that node.
  
  - Third, if the current node is a cell for which we cannot accept multipole accessibility criterion, we traverse further down that node to calculate the interaction between p and the subcells of the current node.

- To compute interaction between the current node and the particle p, we fetch data (RGBA) from “PositionRcritTexture”. This texture contains the position coordinates of the particle or the cell center of mass, and the cell critical radius (zero in case the current node is a particle).

- The interaction between the particle and the current node is calculated using GRAVSUM fragment program routine. GRAVSUM computes acceleration of the particle p based on the gravitational potential force exerted on that particle by the current node. Since recursion is not possible on current GPUs, we use iteration to traverse down the tree.

- We use WALKTREE fragment program routine to visit the next node in the array which could be either a sibling of the current node, child of the current node, or the
parent of the current node if we have visited all the subcells of the current node. The tree lookup ends when a leaf is reached. In order to implement iteration on the current hardware, we use loop statements which allow fixed number of iterations. We used nested loop statements in our shader to handle tree traversal for any depth, which is dependent on the opening angle $\theta$ of multipole accessibility criterion. The resultant acceleration of the particle $p$ is known once the entire tree structure is traversed, which is stored in the framebuffer attached texture image called “AccelerationTexture”.

6.4.2 Position

The position kernel is a very simple kernel of the Barnes-Hut treecode on the GPU as shown in Figure 6.6. Given, the new acceleration coordinates of the particle, it computes the new particle position and advances the simulation by one time-step. The fragment program is invoked for each pixel on the screen and the new particle position coordinates are computed. However, we discard the fragments that do not hold valid particle coordinates. This kernel also advances velocity coordinates of the particle by half time-step.

6.4.3 Velocity

The velocity kernel illustrated in Figure 6.7, finally advances the velocity of the particle $p$ by another half time-step, thus advancing it by a full time-step (computed by the position kernel). Again, we discard the fragments that do not represent valid particle velocity coordinates. Once we have advanced the velocity coordinates by a time-step, we have completed running a simulation step.
Figure 6.6: **Position kernel.** Position kernel on the GPU which traverses the simulation by half time-step. Particle position and velocity coordinates computed in the previous time-step, and the new acceleration coordinates computed during the current tree traversal are used to compute the new particle position coordinates.

Figure 6.7: **Velocity kernel.** The velocity kernel completes another time-step to finally step the simulation. Here we compute the new velocity coordinates of the particles by using the intermediate particle velocity coordinates computed during the half-step traversal done using Position kernel and the acceleration particle coordinates computed during the tree traversal.
Chapter 7

Results and Discussion

7.1 Results

We benchmark our GPU based N-body algorithms to analyze the relative performance merits with the typical solution running on a general-purpose processor or a CPU. The experiments were performed on a host machine running an AMD Athlon XP 1800+ processor with the core speed of 1.54 GHz, and 128 KB of L1 cache and 256 KB of L2 cache and 512 MB of RAM. We tested our GPU algorithms on the Nvidia GeForce 6600GT (AGP) running at a core speed of 500 MHz. GeForce 6600GT features 8 pixel pipelines with a peak fill rate of 2 Gpixels/sec. The memory bus interface operates at 900 MHz with 128-bit bus width and supports a peak memory bandwidth of 14.4 GB/sec. We tested our algorithm specifically on the Nvidia cards owing to the fact that they support floating-point precision, support for non-power-of-two textures, support for framebuffer attached objects like texture memory objects, support for branching and looping constructs, and also because of relative more programming documentation available on the internet.

The GPU programming model is still in its infancy for its support for general-purpose computation especially for computations that involve branching and looping constructs. Our research is a forward looking endeavour and we are able to unravel some of the issues related with the GPU programming model. As we mentioned earlier in the “Implementation” chapter, we divide the overall computation on the GPU in three distinct kernels: Traverse,
Position, and Velocity kernels. The Traverse kernel is the main fragment shader that performs the tree traversal on the GPU and we analyze here the results obtained when running this shader on the GPU. We seek to specifically evaluate the computational performance of our GPU implementations and so explicitly exclude the overhead of repacking input data in system memory, transferring them to the graphics card, and initially loading and binding shader programs. However, the timing results are still not accurate as there is no counter available on the GPU which could capture the exact run time analysis for general-purpose shader programs. The timing analysis is generated by passing the command from the host processor to the GPU to execute the shader program and once the shader execution is completed, the host processor receives the acknowledgment signal from the GPU indicating that the shader execution is completed. So, in reality we are calculating the overall time which includes the time taken by the command to go from the CPU to the GPU, the execution time of the shader and the time taken by the signal to return from the GPU to the CPU indicating the end of process.

We generate results for simplified MAC (as mentioned earlier in chapter Barnes-Hut Algorithm). Figure 7.1 represents results obtained for the simplified MAC method used for tree traversal. Figure 7.2 represents the results obtained when using GPU to compute the new position and velocity coordinates of the particles. Table 7.1 illustrates the size of the textures used to store the octree information on the GPU.

<table>
<thead>
<tr>
<th>nbody</th>
<th>Texture dimensions (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>14x14</td>
</tr>
<tr>
<td>200</td>
<td>19x19</td>
</tr>
<tr>
<td>300</td>
<td>143x143</td>
</tr>
<tr>
<td>400</td>
<td>377x377</td>
</tr>
<tr>
<td>500</td>
<td>495x495</td>
</tr>
</tbody>
</table>

Table 7.1: Minimum 2D textures needed to store octree information.
Figure 7.1: **Position and Velocity kernel timing graph.**

Figure 7.2: **Tree traversal kernel timing graph.**
7.2 Discussion

As clear from the above results, our tree traversal shader hits various complications when simulating particles of the order of $N > 100$. There are still various limitations imposed when implementing fragment shaders on the GPU. Some of the most common limitations that impact the performance of the fragment shader on the Nvidia GeForce 6 Series GPU are:

- **Number of instructions.** Fragment shaders are only capable of supporting 65,535 static instructions and 65,535 dynamic instructions. We thought that we might be running out of the total dynamic instructions available on the fragment shader. But we reject this notion for the problem as the correct results are generated when we increase the framebuffer attached texture memory objects.

- **Branching and looping limitations.** Branching and looping constructs are supported on the latest end Nvidia video graphics cards. Currently, they provide limited support for their usage within the shader programs and also suffer from overhead to flow-control operations. Branching is supported up to only four nested levels, while the looping is supported up to 65,536 iterations. Also the number of loop iterations needed to be known at compile time otherwise the GPU programming model throws an error and fails to execute.

- **Texture memory layout.** Texture memory layout on the GPU and the specialized processing power of the GPU. But since the required textures layout sizes are typically small, and thus this doesn’t seem to be the cause for junk results.

- **Number of texture indirections.** We checked our tree-traversal shader if we were exceeding the limited number of texture indirections available on the GPU. Current GPUs limits that only four consecutive texture indirections could be performed in any given sequence.
- **Proper sampling.** We also verified that we are correctly sampling pixels from the data stored in the texture memory. We ratified this by retrieving all the pixel values from the GPU and writing them on a file, which was then compared with the original data (when stored on the GPU).

- **Appropriate viewport size.** In general-purpose computations on the GPU, we are typically processing every element of a rectangular stream of fragments representing the grid. So, we verified that we are creating a proper size quadrilateral on the GPU so that the computation can be invoked on every pixel that the quadrilateral covers.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cost (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If/endif</td>
<td>4</td>
</tr>
<tr>
<td>If/else/endif</td>
<td>6</td>
</tr>
<tr>
<td>Call</td>
<td>2</td>
</tr>
<tr>
<td>Ret</td>
<td>2</td>
</tr>
<tr>
<td>Loop/endloop</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 7.2: Branch penalties on Nvidia 6 Series GPU.

However, our shaders are not limited by none of the above mentioned limitations. Apparently, the real issue with the availability of those features depends how stable and bug-free are the shader compilers. In the following we describe all the difficulties that we encountered while programming fragment shaders to compute force interactions using Barnes-Hut treecode.

### 7.3 Difficulties Encountered

Our first encounter with problems related with programming the graphics card video started when we were developing our application on the Linux platform using Cg shader language. Though, Cg developed by Nvidia as a high-level shading language to assist developers in
creating visually enriched graphical images and objects, its compiler is not completely error free when using them to create general-purpose applications. On the Linux platform we were not able to simulate our application correctly when we created the framebuffer size smaller than 16x16. The Cg compiler release documentation do not list any such size limitation, and it was indeed very frustrating as to predict why is this happening. So, we started writing my shaders using the ARB_fragment_program and Nvidia_fragment_program2 assembly language constructs. But even after rewriting our entire shaders in GPU assembly language, the problem existed and we were again clueless. We completed debugged our code to find any hidden bugs that might be yielding incorrect results in the framebuffer. It was only after several weeks of rigorous debugging sessions, and trying different Linux flavors, different host machine configurations, and a different video graphics card that we decided to move over to Windows platform (WindowsXP SP2). And we found out that the framebuffer minimal size limitation only exist on Linux platform as the application ran fine when simulated on the Windows platform after porting the entire code from the Linux platform to the Windows platform.

Apparently, there is a bug with the Cg compiler 1.4 on Linux platform which results in garbage framebuffer results when we create the framebuffer size smaller than 16x16. Also we realized that there are various other tools available from Nvidia that probably would assist in developing general-purpose applications on the video graphics card when the development platform is Windows, for instance, NVemulate where you can select from several different graphics card configurations to test if there is an inherent problem with the original graphics card available in your host machine. NVemulate is only available for Windows platform and comes in handy when there is not enough hardware resources in the form of video graphics card available for experiment purposes. However, our problems with programming video graphics card didn’t end here.

The biggest problem we encountered was the odd behaviour illustrated by the framebuffer attached objects when we simulate our application for particles $N > 100$. Figure 7.3 and Figure 7.4 illustrates the ideal timing results for position and velocity kernel, and tree
traversal kernel on the GPU, respectively.

<table>
<thead>
<tr>
<th>nbody</th>
<th>Texture dimensions (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>14x14</td>
</tr>
<tr>
<td>200</td>
<td>18x18</td>
</tr>
<tr>
<td>300</td>
<td>22x22</td>
</tr>
<tr>
<td>400</td>
<td>25x25</td>
</tr>
<tr>
<td>500</td>
<td>28x28</td>
</tr>
</tbody>
</table>

Table 7.3: (Ideal) Minimum 2D textures needed to store octree information.

![Ideal Position and Velocity kernel timing graph.](image)

Figure 7.3: (Ideal) Position and Velocity kernel timing graph.

These ideal characteristics were obtained by simulating the Barnes-Hut tree algorithm on the GPU without comparing the final results with the results obtained when simulating the algorithm on the CPU. These ideal behaviour can be thus taken as representative or approximate timing behaviors for Barnes-Hut tree algorithm on the GPU. Table 7.3 represents the minimum 2D textures needed to store octree information on the GPU. As we mentioned
in the “Implementation” chapter that we map our octree structure in a 2D array representation, and we store this array representation in the graphics card texture memory. The tree traversal algorithm traverses this data structure stored in the texture memory to compute the forces exerted on each particle, which are then written to the framebuffer attached texture memory objects. Framebuffer attached objects are a recent addition to the OpenGL graphics pipeline and provide a mechanism to render to destinations other than those provided by the window system. However, the support for this feature is not yet completed in the graphics card drivers as they are in the beta stage of the drivers supporting this new feature. Table 7.3 shows the ideal dimensions of the texture memory sizes when simulating for different number of particles (N) on the GPU. Table 7.1 suggests the minimum dimensions of the texture dimensions required to simulate the Barnes-Hut algorithm correctly on the GPU. As clear from the table 7.1, we are allocating space that is not part of the computation and thus resulting not only in waste texture memory space on the GPU, but also our shaders are doing unnecessary work which eventually affect their performance (apparent from the timing sections for different shaders). This issue is further compounded by the lack of
hardware details about the underlying hardware and also the lack of sufficient debugging tools.

### 7.4 Debugging Fragment Shader

We discussed about various fragment program errors in the “Discussion” section of this chapter and we found that the real error is with the inability of the graphics drivers to handle complex algorithms. We also investigated within the driver itself, the real cause of concern was caused by the branch unit (within the Nvidia video graphics card driver).

We determined that the driver fails after executing a branch instruction in our tree-traversal shader when we create textures of sufficient or optimum sizes after executing certain number of iterations. We check that the values in the registers of the GPU memory are correct just before we execute this branch instruction. And after we execute this branch instruction, we check the registers for various variables values, and the results are all incorrect values. We found this to happen whenever we create textures of dimensions smaller than a random size, which clearly has no possible relationship with the viewport settings on the GPU. We also try to look for any such relationship between the viewport settings and the dimensions of the textures in the Nvidia driver documentations and other GPU programming manuals. Also there is no mention in any of the above documentations that branch unit only supports limited number of branch instructions, which we really doubt the case as the simulation proceeds flawless if we increase the framebuffer attached texture objects dimensions. This is the primary reason why we were not able to perform any substantial performance comparison of the Barnes-Hut algorithm on the GPU with the typical solution running on a CPU as we hit the limitation in terms of the available texture memory space available on the GPU for simulating particles \( N > 500 \). The Barnes-Hut algorithm will really prove advantageous in terms of the performance when we are able to simulate for particles sizes, \( N > 1000 \) because we can then see the benefits of available parallelism in the underlying algorithm. The processors in the GPU are optimized for a
different set of computations and exploits the parallelism available in the given application. However, current generation of GPUs must perform a framebuffer read operation to implement a write operation which is again written on the framebuffer attached texture memory objects, whereas the CPU can simply read the memory and perform a write operation on the registers.

Thus, when the graphics hardware provide complete implementation of the framebuffer attached texture memory objects with the tested branch unit support in the video graphics driver, we believe that the Barnes-Hut algorithm can be very competitive against special processor machines like GRAPE where they employ a special piece of hardware to perform computations in the inner loop.
Chapter 8

Conclusion and Future Work

8.1 Conclusion

Overall, this work was a forward looking research and demonstrated that highly complex algorithms such as Barnes-Hut algorithm can be implemented on the present generation of graphics hardware. Comparisons were not made with the special-purpose hardware, GRAPE because of the relative inadequacy of the video driver graphics card to handle branches effectively. However, we still argue that once the drivers support is there, the GPU implementations will provide competitive solutions against special-purpose hardware specifically designed to handle such algorithms or computations. Some of the reasons why we believe our approach will be advantageous are:

- We are performing the entire computation on the GPU and not just performing the inner loop computations on it.

- The graphics hardware is designed to exploit parallelism available in the application.

- The memory bandwidth (between the GPU processors and the texture memory) is much higher than the available bandwidth between the system memory and the special-purpose processor, for instance, GRAPE.

- With the availability of multiple GPUs on the same board, we can perform simulation and visualization of the results simultaneously without any delay. Multiple GPUs
can be helpful to further parallelize the application.

8.2 Future Work

We believe that this research work can be used in many ways to explore the capability of graphics hardware for implementing complex algorithms. Some of the extensions to our work could be any of the following:

- There remains significant amount of analysis that can be done on our work once the graphics hardware provide more tested and stable drivers that can handle branching (instructions) effectively. The performance comparisons can then be made between special-purpose solutions like GRAPE and GPU solutions.

- Multiple GPUs (Nvidia SLI Technology) can be explored to devise parallel solutions that may challenge contemporary solutions typically implemented on the vector machines.

- Our work could serve as the basis for implementing other similar algorithms as seen in the fluid dynamics, molecular dynamics and medical and imaging applications.


Bibliography


Appendix A

Octree Data Structures on CPU

In this appendix, we describe the body of different data structures needed to create octree on the CPU.

typedef struct _Node
{
  short type; // node or cell
  bool type update; // update flag
  real mass; // mass of the node
  vector pos; // position coordinates of the node
  struct _Node* next; // pointer to the next node
} Node, *NodePtr;

typedef struct _Body
{
  Node BodyNode; // contains specifics for the BodyNode
  vector vel; // velocity coordinates of the particle
  vector acc; // acceleration coordinates of the particle
  real phi; // potential of the particle
} Body, *BodyPtr;
typedef struct _Cell
{
    Node CellNode;    // contains specifics for the CellNode
    real rcrit2;      // critical radius of the cell
    NodePtr More;     // pointer to the first daughter node
    union
    {
        NodePtr SubP[NSUB];  // array contains all daughter
                               // nodes
        matrix quad;          // quadruple moment force of the cell
    } Sorg;
    int nparticles;     // number of particle in the cell
    real cellsize;      // cell side length
    vector cellcenter;  // cell center
} Cell, *CellPtr;
Appendix B

Octree Data Structures on GPU

In this appendix, we describe the data structures required to map octree (created on the CPU) on the GPU. We only list here the different data structures that were created to contain all the vital information needed to traverse octree on the GPU.

typedef struct
{
    float mass;       // mass of the cell or body
    float parent;     // parent of the cell or body
    float child;      // child of the current cell
    float neighbor;   // neighbor of the current cell or body
} MassAddressNode;

typedef struct
{
    float xaxis;       // position of the cell center or center-of-mass
    float yaxis;       // position of the cell center or center-of-mass
    float zaxis;       // position of the cell center or center-of-mass
    float rcrit2;      // cell critical radius
} PositionRcritNode;