UCLA
UCLA Electronic Theses and Dissertations

Title
Efficient Algebraic Representations for Throughput-Oriented Algorithms

Permalink
https://escholarship.org/uc/item/5gs4v9qw

Author
McKinlay, Christopher E.

Publication Date
2013

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

University of California
Los Angeles

Efficient Algebraic Representations for Throughput-Oriented Algorithms

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Mathematics

by

Christopher Ellis McKinlay

2013

© Copyright by
Christopher Ellis McKinlay
2013

Abstract of the Dissertation

Efficient Algebraic Representations for Throughput-Oriented Algorithms

by

Christopher Ellis McKinlay

Doctor of Philosophy in Mathematics

University of California, Los Angeles, 2013

Professor Chris Anderson, Chair

The graphics processing unit (GPU) was initially designed for raster-based graphics computations, but marked improvements in performance and programmability have generated considerable interest in it as a high-performance computing platform. GPU hardware design places a large number of low-power cores on the one chip, as opposed to multiple high-power chips. This configuration has a powerful effect on locality: data can be processed without the throughput costs of spilling it to buffers in the higher levels of the memory hierarchy. However, throughput-oriented algorithms must now explicitly express and optimize memory access.

This dissertation explores the design and deployment of memory-efficient tensor product representations for throughput-oriented processors. These representations lead naturally to algorithms with excellent data-locality and can be used to construct efficient and scalable represenations for hierarchical tensor-based algorithms. In addition to presenting the underlying mathematical framework of tensor product representation, we present applications to several important numerical algorithms.

Our first contribution is a memory-efficient parallel Inverse Discrete Wavelet Transform algorithm. We present test results from an implementation that performs extremely well on commodity desktop GPU hardware. The small memory footprint of our representation

contributes to an efficient data decompression algorithms and ultimately to high quality image analysis software.

We also consider the general problems associated with the efficient deployment of procedures that utilize tensor product representations. We propose an abstract approach to heterogenous cluster application programming, which combines throughput-oriented algorithmic primitives and an integrated work-partitioning mechanism. We then present an experimental OpenCL-based implementation and test it on an N-body simulation in 2-d using the Barnes-Hut algorithm. Our implementation demonstrates good performance on a heterogeneous cluster of GPU-equipped machines.

Finally we present a general framework for creating tensor product representations of integral kernels in d dimensions. We conduct a detailed analysis of the memory complexity characteristics of the Barnes-Hut and Fast Multipole Methods in the general d-dimensional case, again using tensor product representations as primitives. We then present performance benchmarks for the two algorithms, and verify our theoretical results with experimentally measured values.

The dissertation of Christopher Ellis McKinlay is approved.

Alan Laub

Joseph Teran

Alan Willson

Chris Anderson, Committee Chair

University of California, Los Angeles

2013

Table of Contents

1 Introduction **1**

1.1 Throughput-Oriented Algorithm Design 3

1.2 Multilinear Representations 8

1.2.1 Definition of a Tensor 10

1.2.2 Kronecker Products 10

1.2.3 Canonical Decompositions 12

1.2.4 Mode-n Products 13

2 Representations for Image Decompression **17**

2.1 Background 18

2.2 Prior Work 19

2.3 Wavelets 21

2.3.1 Haar Wavelets 22

2.3.2 Daubechies Wavelets 22

2.4 GPU Reconstruction 23

2.4.1 2-d Surface Arrays 24

2.4.2 Packing 26

2.4.3 Reconstruction 28

2.4.4 Rendering 30

2.5 Results 31

2.6 Conclusions 36

3 Representations for Hierarchical Methods - I **39**

3.1 Background . . . 39
3.2 The Barnes-Hut Algorithm . . . 42
3.2.1 Kernel Compression . . . 44
3.2.2 Product Representation . . . 47
3.3 Programing and Partitioning with Primitives . . . 50
3.3.1 Primitives . . . 51
3.3.2 Barnes-Hut Primitives . . . 52
3.3.3 Barnes-Hut on the GPU cluster . . . 54
3.4 Load Partitioning . . . 55
3.4.1 Partitioning for the Map primitive . . . 56
3.4.2 Partitioning for Barnes-Hut . . . 57
3.5 Implementation and Results . . . 58
3.6 Conclusions . . . 58

4 Representations for Hierarchical Methods - II . . . 61

4.1 Background . . . 62
4.2 Volume and Boundary Integral Equations . . . 63
4.2.1 Volume integral equation formulation of the Helmholtz equation . . . 64
4.2.2 Green functions . . . 65
4.2.3 Time-dependent wave equation . . . 65
4.3 Approximate Tensor Rank . . . 66
4.4 Single-Level Representation . . . 68
4.5 Multi-Level Representation . . . 70
4.5.1 Source Tree . . . 71
4.5.2 Target Tree . . . 72

4.6 Multi-Level Algorithm . . . 73
4.7 Algorithmic Complexity . . . 77
4.8 Implementation and Results . . . 80
4.9 Conclusions . . . 83

5 Conclusion . . . 87

6 Appendix A- GPU Hardware . . . 89

6.1 Processors . . . 89
6.2 Memory Management . . . 91
6.3 Cache Alignment and Coalescing Restrictions . . . 93
6.4 Other Device Limitations . . . 95
6.5 Programming Tools . . . 96

References . . . 99

List of Figures

2.1 VAPOR's IDWT 21

2.2 Surface Arrays 25

2.3 Vorticity Reconstruction 27

2.4 Shockwave Reconstructions 32

2.5 Taylor Reconstruction 34

2.6 Haar and Daubechies 37

3.1 Hierachical Grouping 3 42

3.2 Hierachical Grouping 4 43

3.3 Barnes-Hut Partitions 43

3.4 Hierachical Grouping 1 46

3.5 Variable Load Partitioning 59

4.1 Hierachical Grouping 2 67

4.2 Matrix Decomposition 78

4.3 Approximation Error - Size 81

4.4 Barnes-Hut Interactions 82

4.5 Memory Usage- Low Accuracy 84

4.6 Memory Usage- $N = 7 \cdot 10^6$ 85

6.1 GPU Memory Hierarchy 91

List of Tables

2.1 Rendering Time- Array Size 33

2.2 Rendering Time- Window Size 33

2.3 Total Time- Haar 36

2.4 Total Time- Daubechies 38

Acknowledgments

I would first like to thank my advisor, Chris Anderson, for his advice and support. I would also like to thank the kind scientists and administrators CISL laboratory at the National Center for Atmospheric Research. This thesis would not have been possible without the years of tutelage and assistance I received there. Thanks in particular to John Clyne, Rory Kelly and Jose Garcia for their guidance through the rather steep initial learning curve of throughput-oriented algorithm design. Finally, I would like to thank my friends and family for their patience and encouragement.

Vita

2001 A.B. (Chinese), Middlebury College, Middlebury, Vermont.

2008 M.A. (Mathematics), UCLA, Los Angeles, California.

Publications

Lord, J.; Rast, M.; Mckinlay, C.; Clyne, J.; Mininni, P.; *Wavelet Decomposition of Forced Turbulence: Applicability of the Iterative Donoho-Johnstone Threshold.* Physics of Fluids 24-2, 02/12.

Mckinlay, C.; Citro, C.; *Scalable Application of Compressed Linear Operators on GPU Hardware.* 14th SIAM Conference on Parallel Processing for Scientific Computing, 01/10.

CHAPTER 1

Introduction

The utilization of graphics processing units (GPUs) for general purpose numerical computations has recently become a topic of considerable interest. As the growth in computing throughput of scalar microprocessors has stalled, many of the guiding principles behind architecture and algorithm design have evolved. In particular latency-oriented designs, wherein a single processor's utilization is maximized, have been replaced by throughput-oriented designs, wherein incremental performance is maximized as hundreds of cores are employed.

Throughput-oriented processors address problems where parallelism is abundant, which results in hardware designs different from more latency-oriented processors. The most notable difference is that memory latency is extremely high in comparison to processor speed. To fully utilized processor speed, this large disparity needs to be efficiently hidden. Although improving total computational throughput at the expense of increased individual task latency is not always a desirable trade-off (for example in the Linux kernel's task scheduler), it is the optimal parallel hardware configuration for many problem domains, including molecular dynamics, astrophysical simulation, gene sequencing, medical-image analysis, video processing, and real-time computer graphics.

Due to their availability and stable tool chain, GPUs have become the dominant technology in modern throughput-oriented processing. Accordingly, there has been a broad shift towards the increased use of GPUs and other throughput-oriented architectures in parallel scientific computing applications. As the differences between the two architectural paradigms appears to be a persistent reality, this shift towards throughput-oriented computing is having a profound impact on algorithm development in high-performance scientific computing

fields.

Most scientific computing tasks require some form of irregularity (e.g. different input elements may take different lengths of time to process, generate different output quantities/locations, require different data access/communication patterns, etc.), however with irregular data access and usage patterns, few such optimizations can be performed. Therefore high-throughput numerical algorithms must perform complex tasks with irregular data access patterns using hierarchical combinations of data-parallel operations with regular data access patterns. The ability to do so depends fundamentally on equational reasoning: good algebraic representations lead naturally to efficient algorithms.

In the following chapters we present applications of tensor product representations to several important numerical algorithms. In the remainder of this introductory chapter we discuss the throughput-latency trade-off and how this informs fundamental architectural design decisions that differentiate throughput-oriented processors from their more latency-oriented counterparts. We then introduce a memory-efficient tensor product representation, 1.7, which underlies the algorithms in the second and third chapters.

In Chapter 2 we present our representation in an application context, and demonstrate how the memory efficiency of our representation contributes to an efficient data decompression algorithm and ultimately to high quality image analysis software.

In Chapter 3 we consider abstract approaches to load-partitioning that combines throughput-oriented algorithmic primitives and graph-based combinators. Our approach provides a robust mechanism that can can evenly distribute computations between heterogeneous compute nodes in a cluster. We demonstrate the use of our system on a Barnes-Hut-based gravitation simulation with highly asymmetric workloads and dynamic behaviour.

In Chapter 4 we analyze the memory footprints of the Barnes-Hut and Fast Multipole Methods in the general d-dimensional case, again using tensor product representations as primitives. We provide detailed memory complexity derivations, and profile an implementation whose memory bandwidth usage is in good agreement with our expressions.

1.1 Throughput-Oriented Algorithm Design

The two fundamental measures of processor performance are latency (defined as the elapsed time between initiation and completion of a task) and throughput (defined as the total amount of work completed per unit time). A particular processor will be characterized by a number of tradeoffs between latency and throughput optimization, since improving one typically degrades the other. Accordingly, commodity processor designs tend to emphasize either latency or throughput, depending on the workloads they are expected to encounter.

Traditional scalar processors are latency-oriented. Their designs are characterized by hardware optimization techniques such as instruction pipelining, speculative and out-of-order execution, and sophisticated memory caches that have been developed to minimize the running time of a single sequential program. This approach is based on the assumption that any parallelism available in workloads presented to the processor is fundamentally scarce. Single-core scalar CPUs typified by the Intel Pentium IV were aggressively latency-oriented, though modern multi-core CPUs (such as the Intel Core i7) reflect a trend toward hybrid designs that expect a modest amount of parallelism. However general purpose processors such as multi-core CPUs cannot afford to aggressively trade for increased total performance at the cost of single-thread performance. The spectrum of workloads presented to it is too broad, and not all computations are parallel. For computations that are largely sequential, latency-oriented processors typically outperform throughput-oriented processors. On the other hand, a processor specifically intended for parallel computation can accept this trade-off and realize significantly greater total throughput on parallel problems as a result.

Throughput-oriented processors are designed specifically for parallel computation. Their intended use is for computational workloads in which parallelism is abundant. For example, graphics processing units (GPUs) historically evolved to meet the needs of a problem domain real-time computer graphics with tremendous inherent parallelism. Real-time graphics places a premium on the total amount of work that can be accomplished within the span of a single frame (typically lasting 1/30 second). As a result GPUs have emerged as exemplars

of throughput-oriented processor architecture. Their emphasis on throughput optimization and their expectation of abundant available parallelism is more aggressive than many other throughput-oriented architectures.

It is a well-established fact that the energy cost of memory access has continued to increase and is now extremely high in relation to the cost of computation. The energy required to move data between the chip and external dynamic random access memory (DRAM) is an order of magnitude higher than required to operate an on-chip functional unit [1]. Therefore memory access is intentionally restricted in order to ensure that computational throughput per pJ of energy is as high as possible. That is to say that the majority of the total heat generated is a byproduct of direct mathematical computation, rather than book-keeping. Accordingly, the computational throughput of a GPU is much higher than on comparable multi-core designs; peak Floating Point Operations per second (flops) counts in the teraflop (10^{12}) range are now common for single devices.

The time cost of accessing memory is also extremely high in relation to the time cost of computation. This discrepancy is the result of the physical properties of semiconductor technology, therefore it affects both latency and throughput-oriented processors. However, the performance consequences can be more significant for throughput-oriented processors as these processors have a much higher computational-throughput-to-memory-throughput ratio than latency-oriented processors. For example, an Nvidia GTX 280 GPU (see section 6.1) has a computational throughput of nearly 1 trillion floating-point operations per second [nvi], but only roughly 140 gigabytes-per-second of peak throughput to its off-chip global memory.[2]. Furthermore this throughput gap will continue to grow in future systems, where there will be even more processor cores per chip.

Emphasizing computational throughput and power efficiency over task latency and memory flexibility has led to a number of innovations in processor design. Among them, three

[1] For example, in a 45nm silicon process a 64-bit integer addition unit expends roughly 1 picoJoule (pJ), and a 64-bit floating point fused multiply-add (FMA) unit requires around 100pJ. Reading a 64-bit value from external DRAM however requires on the order of 2,000 pJ [Dal09].

[2] computed as an average of ten tests using the Cuda Developer API [cud]

primary architectural trends are typical of throughput-oriented processors: hardware multi-threading, many simple processing elements, and SIMD execution. Throughput-oriented architectures abandon the fast out-of-order superscalar processors characteristic of multi-core architectures in favor of designs that feature increased concurrency with many low voltage cores on the same chip. Simple inorder cores forgo outoforder execution and speculation, and SIMD execution increases the ratio of functional units to control logic. Simple core design and SIMD execution reduce the area and power cost of control logic, leaving more resources for parallel functional units. Furthermore, hardware multi-threading makes managing the expected abundant parallelism cheap. These architectural innovations, combined with a relatively slow clock (1.2-1.4 GHz), create an extremely power-efficient chip design. Furthermore, it is not necessary to run the clock faster in order to make a more powerful chip. The result is that GPU performance is continuing to improve exponentially in accordance with Moore's law.

The memory throughput problem has significant implications in terms of throughput-oriented algorithm design. Throughput-oriented processors seek to tolerate rather than avoid latency. To hide the latency of frequent movement of data to/from main memory requires either more threads or more work per thread. Therefore, strategies for reducing memory footprint and tolerating memory access latency are critical issues in throughput-oriented algorithm design, especially for memory-bound and high-dimensional applications. Throughput-oriented algorithms must expose substantial amounts of fine-grain parallelism, fulfilling the expectations of the architecture. Although exploiting multi-core CPUs requires exposing parallelism as well, the algorithmic role of parallelism on a throughput-oriented processor is substantively different from multi-core. A fourcore CPU can be fully utilized by four to eight threads. Thread creation and scheduling are computationally heavyweight, since they can involve the saving and restoration of processor state and relatively expensive calls to the operating system kernel. In contrast, a GPU typically requires thousands of threads to cover memory latency and reach full utilization, while thread scheduling is essentially without cost.

Algorithm performance is largely determined by the ratio of floating-point calculation to the global memory access operations. With 4 bytes in each single-precision floating-point datum, one can expect to load not more than 35 billion single-precision numbers per second on the GTX 280. Therefore algorithms with a flop-to-access ratio of 1.0 can achieve computational throughputs of not more than 35 billion flops on the GTX 280, only 3.5% of the device's computational throughput. Performing upwards of 20 flops per global memory access is ideal, and recomputing frequently needed values locally is often preferable to storing them in global memory. The best performance is typically achieved when calculation is more common than data access. Consider a simple example of computing a moderately expensive function like $\sin(\theta)$ for 256 unique values of θ. Tabulating all 256 possible values would require little space, but accessing them from external memory would require hundreds of cycles. In the same amount of time, a thread could execute perhaps 50 to 100 instructions that could be used to compute the result and leave the memory throughput available for other uses.

As discussed in the Appendix, GPU architectures have multiple low latency memories in addition to the off-chip DRAM, through which processors have faster access to data that are closer to them in the hierarchy. The characteristics of the various memories available in the GPU are diverse in terms of latency, optimal memory access pattern, and control (either hardware-controlled or software-controlled). This imposes several challenges to effectively reduce memory footprint and hide latency. The optimal access pattern is also dependent on the manner in which threads are mapped for computation and also on the number of threads involved in global memory access as involving more threads would assist in hiding the global memory access latency. Hence, there has to be an optimal thread mapping to ensure optimized memory access. On an algorithmic level, this amounts to maximizing data locality and re-use finding the correct granularity of parallelism.

In conclusion, streamlining memory performance is a critical issue for utilizing the full computational throughput of throughput-oriented processors such as graphics cards. Exploiting the locality in data referencing that an application affords can increase the fraction

of memory references that are satisfied close to the issuing processor, and hence improve performance. Many of the aspects of locality in a multi-processor are no different than on a serial processor, since the part of the memory system that is within a processing node is very similar to a uniprocessor memory hierarchy. The key difference in throughput-oriented architectures is that communication between remote processing nodes is subject to the global memory bottleneck and therefore extremely expensive.

1.2 Multilinear Representations

One of the most significant impediments to solving problems which require high dimensional tensor operations is memory bandwidth requirements that scale exponentially with dimension- the number of entries grows exponentially with d. This makes it extremely expensive in general to store a high-order tensor explicitly. Various data-sparse formats have been developed to address this issue. Depending on the application, these formats may allow for the approximate representation and manipulation of a tensor under dramatically reduced memory bandwidth requirements.

In certain physical problems, such as those arising in computational quantum chemistry, memory costs can be reduced by using tensor product bases. Instead of trying to find a general d-dimensional function $\alpha : \mathbb{C}^d \mapsto \mathbb{C}$ that solves some given integral or differential equation, one only considers sums of functions that can be represented by rank-one tensors:

$$\alpha(x_1, .., x_d) = \prod_{n=1}^{d} \alpha_n(x_n), \tag{1.1}$$

Using the outer product representation (1.1), it is often possible to reduce a d dimensional problem to a system of d one-dimensional equations for the functions$\{\alpha_n(x_n)\}$. For example, in Hartree-Fock and its extensions, one obtains an approximate solution to the original equation using only one-dimensional operations by iteratively solving for each factor α_n. This is an attractive for high-dimensional problems because the complexity scales linearly with the dimension d.

Assuming that all objects in (1.1) have been discretized using n_i points in each dimension, we arrive at the following decomposition of a given order-d tensor into an outer product of vectors (order-1 tensors):

$$\boldsymbol{A} = \bigotimes_{n=1}^{d} a_n. \quad a_1 \in \mathbb{C}^{n_1}, \ldots, a_d \in \mathbb{C}^{n_d} \tag{1.2}$$

Then, instead of the $n_1 \times n_2 \times \cdots \times n_d$ entries of $\boldsymbol{A}$, only the $n_1 + n_2 + \cdots + n_d$ entries

of $a_1, \ldots, a_d$ need to be stored. On a functional level, this corresponds to a separation of variables or an approximation by a separable function such as a multivariate polynomial or product of wavelets. Many important tensors admit bases consisting of controllably small sums of rank one tensor products. This same approach can be profitably used with iterative algorithms for solving integral equations in d dimensions. However instead of using a product representation of the solution as in (1.1), we we assume a product representations of the order-$2d$ tensor generated by a d dimensional integral operator on a rectangular grid; thereby reducing the cost of applying it to a d-dimensional vector. For integral equations where $d >> 1$, this approach results in iterative algorithms with memory bandwidth requirements that scale linearly with d rather than exponentially.

As noted above, this single tensor product decomposition extends naturally to sums of rank-1 tensors:

$$\boldsymbol{A} = \sum_{r=1}^{R} \lambda_r \otimes_{n=1}^{d} A_n^r. \tag{1.3}$$

The main advantage in using the representation (1.3) is that the elementary tensors that we use are of order 1, meaning that order d,$2d$, and $4d$ tensor algebra operations may be performed in the separated representation using fast one, two and four-dimensional algorithms respectively. This presents a tremendous savings in memory bandwidth usage for tensor-vector multiplication-based algorithms. Order-d and $2d$ examples include polynomial and matrix multiplication, modular aritmetic on polynomial and integer rings, cyclic convolution, multi-step finite difference schemes, the Discrete Wavelet transform, and the Fourier transform. Many notable operations of order $4d$ may also be represented as a tensor products, for example matrix-matrix multiplication and its various optimizations (e.g. tiled and Strassen-Winograd schemes [Gro85]).[3] These low-order tensor representations lead naturally to algorithms with excellent data-locality and can be used to construct efficient and scalable represenations for higher order tensor operators.

[3]More generally any BLAS level 1-3 or related operation with a tensor representation can be scaled to generate higher-order operations.

1.2.1 Definition of a Tensor

A multilinear form, or tensor, $\boldsymbol{X} \in \mathbb{C}^{n_1 \times n_2 \times \cdots \times n_d}$ with $n_1, \ldots, n_d \in \mathbb{N}$ is a d-dimensional array with entries $X_{i_1 i_2 \ldots i_d} \in \mathbb{C}$. d is known as the order of the tensor and our methods will target tensors of high order (i.e. $d = 10, d = 10^2$ etc.). A typical scenario is that $\boldsymbol{X}$ represents a d-variate function $f : [0,1]^d \to \mathbb{C}$ sampled on a grid in $\mathbb{C}^d$ or approximated in a tensor product basis.

Let $\mathcal{D} = \{1, \ldots, d\}$ and $\mathbb{F} = \mathbb{R}$ or $\mathbb{C}$. For $n \in \mathcal{D}$, let V_n be a finite dimensional vector space over $\mathbb{F}$ with $dim(V_n) = D_n$. An order $(n, d-n)$ tensor $\boldsymbol{A} \in \bigotimes_{n=1}^{d} V_n$ is defined as a multi-linear map $\boldsymbol{A} : V_1^* \times \cdots \times V_n^* \times V_{n+1} \cdots \times V^d \mapsto \mathbb{F}$, where V_k^* is the dual space of V_k. For our purposes $n = 0$, hence we will drop the co-variant index and simply refer to a tensor as having order d.

By taking an order 2 tensor $\boldsymbol{X}$ and setting the co-variant and contra-variant indices equal to one, we see that the basic tensor product acts on the space of linear endomorphisms of V $End(V) \cong T_1^1(V)$ as well as on V . Specifically, given two elements $\alpha, \beta \in T_1^1(V)$, the tensor product $\alpha \otimes \beta \in T_2^2(V)$ is a linear tranformation: $\alpha \otimes \beta : V \otimes V \mapsto V \otimes V$ defined by:

$$(\alpha \otimes \beta)(v_1 \otimes v_2) = \alpha(v_1) \otimes \beta(v_2). \tag{1.4}$$

The map (1.4) extends naturally to products of d maps [HGG04]:

$$\otimes_{n=1}^{d} \alpha^n : \otimes_{n=1}^{d} T_1^1(V) \mapsto T_d^d(V)$$

.

1.2.2 Kronecker Products

The classical Kronecker tensor product construction is commonly used in tensor-based computations. However, as we will demonstrate, it is unsuitable for throughput-oriented computations. We present a more memory-efficient construction, which we then use as a data primitive throughout the rest of this thesis.

With a standard choice of basis, (1.4) has an explicit matrix representation known as the Kronecker product.

Definition 1.1. Let $A \in T_1^1(V_m)$, $B \in T_1^1(V_n)$ The Kronecker product $A \otimes B$ is the $D_m D_n \times D_m D_n$ block matrix representation of the tensor product $\alpha_m \otimes \alpha_n$ of the two corresponding linear maps α_m, α_n using the standard basis:

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1D_m}B \\ \vdots & \ddots & \vdots \\ a_{D_m 1}B & \cdots & a_{D_m D_m}B \end{bmatrix} \tag{1.5}$$

The following Lemma (proved in [HJ91]) uses the Kronecker product to express matrix multiplication as a linear transformation on matrices.

Lemma 1.2. *Let $A \in T(V_m, V_m)$, $B \in T(V_n, V_n)$ and $Y \in T(V_n, V_m)$. The matrix equation*

$$\boldsymbol{X} = AYB$$

is equivalent to the system of $D_m D_n$ equations in $D_m D_n$ variables given by:

$$vec(\boldsymbol{X}) = (B^T \otimes A)vec(Y)$$

, where $vec(\boldsymbol{X})$ is the vectorization of the matrix $\boldsymbol{X}$ formed by stacking all of the columns of $\boldsymbol{X}$ into a single column vector.

The above lemma implies that the $(D_m D_n)^2$ operation $(B^T \otimes A)vec(Y)$ can be computed using only $D_m D_n (D_m + D_n)$ operations. Thus the Kronecker product is not work-optimal. Moreover, when viewed as a tensor storage format it posesses poor data locality characteristics. This is evident from (1.5), since the same matrix A is used in each of the N^2 $N \times N$ sub-blocks of $B^T \otimes A$. Therefore, the Kronecker product is not a scalable solution to the high-order tensor multiplication problem. In Theorem 1.7, below, we show how Lemma 1.2 can be generalized to tensor equations of arbitrary order. This provides a mathematical foundation for development of the memory-efficient algorithms discussed in *chapter* 2.

1.2.3 Canonical Decompositions

Hitchcock [Hit27] originally proposed the idea of the polyadic form of a tensor, i.e., expressing a tensor as the sum of a finite number of rank-one tensors in 1927. Seventeen years later Cattell [Cat44] proposed the idea of multiple axes for analysis of circumstances, objects, and features, as well as methods for parallel proportional analysis. The concept finally became established after its third introduction by Carroll and Chang as the canonical decomposition [CC70].

The canonical decomposition decomposes a tensor into a sum of component rank-one tensors. For example, given a third-order tensor $\boldsymbol{X} \in \mathbb{C}^{I \times J \times K}$, we wish to write it as

$$\boldsymbol{X} = \sum_{n=1}^{r} a_n \otimes b_n \otimes c_n \tag{1.6}$$

where r is a positive integer, and $a_n \in \mathbb{C}^I, b_n \in \mathbb{C}^J$ and $c_n \in \mathbb{C}^K$ for $n = 1, \ldots, r$. The value r is called the rank of the tensor $\boldsymbol{X}$.

Definition 1.3. The rank $rank(\boldsymbol{X})$ of a tensor $\boldsymbol{X}$, is defined as the smallest number of rank-one tensors (see 1.4) that sum to $\boldsymbol{X}$.

In other words, this is the smallest number of components in a canonical decomposition. We can extend this definition to all tensors by saying that the rank of $\boldsymbol{A}$ is the minimum number r such that $\boldsymbol{A}$ is expressible as the sum of r rank-one tensors in the given field.

Definition 1.4. A nonzero tensor $\boldsymbol{A}$ is said to be of rank one if there are d vectors $a^n \in \mathbb{C}^{m_n}$ such that

$$\boldsymbol{A} = \bigotimes_{n=1}^{d} a^n \tag{1.7}$$

Definition 1.4 corresponds to the classical notion of outer products for vectors.

We mention in passing that there are close connections between the canonical decomposition in the $d = 1$ case and the Singular Value Decomposition (SVD). The rank of an

order $(1,1)$ tensor is equal to the rank of the corresponding matrix, a fact easily seen by considering that the SVD of the matrix is generated by such outer products. In fact, both tensor and matrix rank are defined as the minimal number of such outer products needed to express the operator as a sum [Bou89, Hal74, Kru97, Kie00]. These and other issues related to the SVD and its tensor generalizations have been studied extensively [LMV00a].

When the variable r in (1.6) is fixed, there are many algorithms for finding such a decomposition using L_2 and L_1 error norms [DM07, GZ01, Har70, LMV00b]. By increasing r, an approximate solution can be made as accurate as desired. Canonical decompositions appear in computational quantum chemistry, for example, through the use of tensor product basis as well as auxiliary basis sets (such as Gaussian-type orbitals [BER73]); both approaches may be considered as tensor product approximations in the sense of (1.13), where r corresponds to, respectively, the number of tensor products of single-particle wave functions, and the total number of primitive Gaussian-type orbitals in the auxiliary basis set.

When $d >> 1$, the canonical decomposition becomes expensive to compute. Fortunately many commonly used algorithms involve the repeated application of one or more static linear operators to a series of changing arguments; consequently these algorithms require only a single factorization computation done during the preprocessing stage. Examples from linear algebra include the (inverse) power method for determining the (smallest) largest eigenvector and eigenvalue of an operator, Schulz iteration for constructing inverses, and the various Krylov subspace-based methods for solving linear systems.

1.2.4 Mode-n Products

The extension of Lemma 1.2 to order-d tensors requires generalizations of the notions of vectorization and matrix multiplication. The former is known as tensor matricization [Kie00, LMV00a].

Definition 1.5. Let $\boldsymbol{X} \in \bigotimes_{n=1}^{d} V_n$ be an order-d tensor. Let the ordered sets $\mathcal{I} = \{i_1, \ldots, i_K\}$ and $\mathcal{J} = \{j_1, \ldots, j_L\}$ be a partition of the set $\mathcal{D} = \{1, \ldots, d\}$. Then the

matricized tensor $X_{\mathcal{I}\times\mathcal{J}}$ is defined as the $\prod_{n\in\mathcal{I}} D_n \times \prod_{n\in\mathcal{J}} D_n$ with elements:

$$X_{(\mathcal{I}\times\mathcal{J})ij} = x_{n_1\ldots n_d}, \tag{1.8}$$

where

$$i = 1 + \sum_{l=1}^{K}(n_{r_l} - 1)\prod_{l'=1}^{l-1} D_{r'_l} \quad ; \quad j = 1 + \sum_{m=1}^{L}(n_{r_m} - 1)\prod_{m'=1}^{m-1} D_{r'_m}$$

.

The modes $\{1,\ldots,d\}$ are partitioned into two sets, $\mathcal{I}$ and $\mathcal{J}$, which are mapped to the columns and rows of $X_{(\mathcal{I}\times\mathcal{J})}$, respectively. The above definition of matrix vectorization is a special case where $\times$ is of order-2 and all of the modes are column modes:

$$vec(\boldsymbol{X}) \equiv X_{\mathcal{D}\times\{\}}$$

. Another important special case for what follows is $\mathcal{I} = \{n\}$ and $\mathcal{J} = \{1,\ldots,n-1,n+1,\ldots,d\}$. In this case, known as mode-n matricization, the one-dimensional elements the n-th mode are aligned as the columns of the resulting matrix $X_{\{n\}\times\{1,\ldots,n-1,n+1,\ldots,d\}}$, which we abbreviate by $X_{\{n\}}$.

The second notion we require is the mode-n product [LMV00a], which defines multiplication of a tensor by a matrix in mode-n.

Definition 1.6. The mode-n matrix product of an order-d tensor $\boldsymbol{Y} \in \bigotimes_{n=1}^{d} V_n$ with a matrix $A \in \mathbb{F}^{C\times D_m}$, written $\boldsymbol{Y} \times_n A$, produces an order-d tensor with dimensions $D_1 \times \cdots \times D_{n-1} \times C \times D_{n+1} \times \cdots \times D_d$ with elements

$$(\boldsymbol{Y} \times_n A)_{j_1\ldots j_{n-1} i j_{n+1}\ldots j_d} = \sum_{j_n=1}^{J_n} y_{j_1\ldots j_d} a_{ij_n}$$

A useful property that follows immediately from Definition 1.6 is that if $A \in T_1^1(D_n)$, then

$$\boldsymbol{X} = \boldsymbol{Y} \times_n A \iff X_n = AY_n. \tag{1.9}$$

The following generalization of Lemma 1.2 provides the necessary connection between mode-n matrix products, matricization, and Kronecker products.

Theorem 1.7. *Let $\boldsymbol{X}, \boldsymbol{Y} \in \bigotimes_{n=1}^{d} V_n$ be an order-d tensor. If $A^n \in T_1^1(V_n)$ for all $n \in \mathcal{D} = \{1, \ldots, d\}$, then the system of equations:*

$$\boldsymbol{X} = \boldsymbol{Y} \times_1 A^1 \cdots \times_d A^d \tag{1.10}$$

is equivalent to the system:

$$vec(\boldsymbol{X}) = \left(\bigotimes_{n=1}^{d} A^n \right) vec(\boldsymbol{Y}), \tag{1.11}$$

that is, $vec(\boldsymbol{Y} \times_1 A^1 \cdots \times_d A^d) = (\otimes_{n=1}^{d} A^n) vec(\boldsymbol{Y})$. Furthermore, both systems posess unique solutions if and only if A^n is invertible for all $n \in \mathcal{D}$.

Proof. We use an inductive argument to construct the necessary equivalence. For $N \in \mathcal{D}$, let $\Theta_N : T_1^1(V_N) \times T_1^1(V_1 \times \cdots \times V_{N-1}) \mapsto T_1^1(V_1 \times \cdots \times V_N)$ be given by:

$$\Theta_N(A, B) \mapsto \left(Y \mapsto A Y_{(N)} B^T \right). \tag{1.12}$$

By Lemma 1.2, we have that:

$$vec\left((\Theta_n(A, B))(\boldsymbol{Y}) \right) = (B \otimes A) vec(\boldsymbol{Y})$$

for $\boldsymbol{X}, \boldsymbol{Y} \in \bigotimes_{n=1}^{N} V_n$. Using (1.9), we have that:

$$A Y_{(N)} B^T = (A \times_N Y)_{(N)} B^T,$$

which proves the inductive step. The base case is given by Lemma 1.2 as stated. Uniqueness of solutions follows from the following elementarty property of Kronecker products: $rank(\mathbf{A} \otimes \mathbf{B}) = rank\mathbf{A}\, rank\mathbf{B}$. □

Theorem 1.7 establishes that the application of any tensor operator of the form $\bigotimes_{n=1}^{d} A^n$ can be computed with a sequence of mode-n matrix multiplications in each mode $n \in \mathcal{D}$. Mode-n matrix multiplication in $\boldsymbol{A}$ only requires access to the nth factor matrix $A^{(n)}$, and the work is just a matrix-matrix multiply. As noted above, this presents an enormous savings in

memory bandwidth usage over Kronecker product-based algorithms on throughput-oriented processors.

The extension of Theorem 1.7 to canonical decompositions of R Kronecker products

$$\sum_{r=1}^{R} \lambda_r \bigotimes_{n=1}^{d} A_n^r \tag{1.13}$$

is trivial, although for data-locality reasons it is more efficient in practice to compute this as

$$\boldsymbol{A} \times_0 \lambda = \lambda \boldsymbol{A}_{(0)} \quad \text{for}$$

$$\boldsymbol{A} \in \mathbb{F}^R \otimes \left(\bigotimes_{n=1}^{d} T_1^1(V_n) \right)$$

where $\lambda \in \mathbb{F}^R$ is stored as the innermost mode.

As noted in the introduction to this section, tensors in the form (1.6) have storage requirements linear in d as they use only $R(1+\sum_{n=1}^{d} D_n^2)$ elements as opposed to the $\prod_{n=1}^{d} D_n^2$ elements required for a general order (d, d) tensor. This results in a dramatic reduction memory bandwidth usage and increased scalability in high-dimensional algorithms. Furthermore, tensor-vector multiplication requires only $rdN^{(d-1)}N^2 = rdN^{(d+1)}$ memory accesses. This is the main factor behind the performance of the inverese Discrete Wavelet Transform algorithm in chapter 2; it is also foundational to the more complex hierarchical operator decomposition algorithms in chapter 4 below.

CHAPTER 2

Representations for Image Decompression

In this chapter we present our first application of a multilinear product representation. We demonstrate how the representation given by Theorem 1.7 enables the creation of a memory-efficient Inverse Discrete Wavelet Transform algorithm. We present test results from such an implementation using CUDA [cud] that performs extremely well on commodity desktop GPU hardware. This implementation has since been incorporated into CISL's VAPOR project [VAP].

The movement of information during an algorithm deserves careful scrutiny in a parallel algorithm. For example, consider a two dimensional fast Fourier transform, which is the tensor product of one-dimensional fast Fourier transforms. With regard to data movement, the two dimensional fast Fourier transform on an $n \times n$ data set is equivalent to n simultaneous one-dimensional trans forms along one dimension, followed by n simultaneous one-dimensional transforms along the other dimension. If the data set is arranged so that one dimension is across thread groups [1] and the other is within thread groups, then the first step involves executing the transform in parallel on data contained entirely within local memory. No transfer of data between thread groups is required. The next step requires n more transforms, and these may be done successively, using a parallel transform where each transform takes one datum from each thread group. This entails on the order of $\log_2 n$ inter-group memory transactions per transform, which is extremely expensive. A better alternative is to transpose the data and repeat the transform within thread groups. This is the approach we take with the Inverse Discrete Wavelet Transform in section 2.4 below.

[1] see the appendix 6 for more information about thread groups, memory hierarchy and other aspects of GPU programming

2.1 Background

A common method for storing data resulting from large-scale scientific computations is to sample the results on grids with high resolution. While it is relatively easy to store the data in this format, it is difficult to work interactively with it because the memory requirements exceed the capabilities of most commidity desktop computers. This situation occured in atmospheric modelling applications with the National Center for Atmospheric Research's VAPOR [2] application for example. This chapter presents a successful soltution to the problem that inserts a GPU accelerated Inverse Discrete Wavelet Transform (IDWT) into the standard graphics pipeline. Our method reduces bandwidth requirements to modest levels and makes the real-time inteactive exploration of large gas and fluid dynamics datasets with VAPOR a possibility on desktop UNIX and Windows systems equipped with modern 3D graphics cards.

In general, wavelet compressed data need to be reconstructed using an inverse DWT before further processing by application-dependent rendering algorithms. Although shared memory-based wavelet reconstruction methods can be sufficient for simple wavelets such as the Haar wavelet, in general it is too slow for real-time computation using wavelets that achieve higher compression ratios and better image quality. Several recent studies [HE99b, HE00, WWH02] have reconstructed wavelets using graphics hardware. However, they have mostly focused on 2-d images, and the limitations of the available graphics hardware have only recently made reconstruction using low-latency surface memory competitive against shared memory.

With the advent of programmable graphics hardware and high-level shading languages, 3-d wavelet reconstruction can now be fully realized faster and easier than before. High-precision surface formats with full floating-point pipelines and pbuffers for off-screen rendering give the necessary ingredients to implement robust fragment programs capable of

[2] VAPOR (Visualization and Analysis Platform for Ocean, Atmosphere, and Solar Researchers) [VAP] is a visual data discovery environment tailored towards the specialized needs of the astro and geosciences CFD community. It provides advanced interactive 3D visualization and data analysis capabilities suitable for terascale size data sets.

carrying out the tasks for wavelet reconstruction. For example, VAPOR's GPU rendering algorithm [VAP] performs real-time 3-d wavelet reconstructions using GPUs by mapping surfaces that contain wavelet coefficients and by evaluating wavelet reconstruction formulae at each fragment. This allows for real-time voxel rendering and user interaction, even when working with large datasets on commodity laptop hardware. Although our rendering primitives are 2-d quadrilaterals, by virtue of their vectorized datatypes they are able to efficiently reconstruct a 3-d right rectangular prism (array) of voxels. Since it is necessary to be able to access elements at different locations within the array, we use 2-d surface memory, as described in section 6.2. 2-d surface arrays have proven to be highly flexible for rendering schemes that produce 3-d surface coordinates [KE02, LKH03] . We then incorporate this building block into a multi-resolution hierarchy to render a wavelet octree [GWG02, Cly03] .

The remainder of this chapter is organized as follows: section 2.2 presents previous work for wavelets and GPU programming, section 2.3 reviews the Haar and Daubechies wavelets, section 2.4 describes our reconstruction algorithm, which uses Theorem 1.7 and 2-d surface arraying to maximize memory efficiency. Finally section 2.5 presents results from various VAPOR test images, and section 2.6 summarizes with our conclusions and possibilities for future work.

2.2 Prior Work

Volume rendering applications typically involve large amounts of data, therefore compression and multi-resolution schemes that allow for visualization of combined levels-of-detail are attractive tools. Many visualization algorithms have been proposed using the support of 3-d surface arrays to streamline various aspects of the rendering pipeline. For example to avoid decompression [NH93, SW03], or to fetch arrays from multi-resolution octrees based on viewing metrics [LHJ99].

Multi-resolution analysis provides a robust foundation for 3-d filter banks and volume

rendering [SDS96, SN96]. Westermann describes one multi-resolution framework for volume rendering [Wes02, Cly03] . Other authors have also used the features of decomposition to efficiently render wavelets [IP99, Rod99] and extract features [GRB09, LRM12]. Among the different types of wavelet, the Haar wavelet and the Daubechies wavelet [Dau92] are typical in the computer graphics literature. The first because it is very fast to calculate, and the second because it retains more visual detail, albeit with a slightly slower reconstruction time.

Most studies implement the IDWT reconstruction in shared memory (see Guther et. al. [GWG02] for example), however recent improvement to hardware caching mechanisms have increased the bandwidth of read-only surface memory and thus motivates the effort to develop efficient wavelet transforms that use surface memory. On most current devices a surface fetch costs one device memory read only on a cache miss, otherwise it will cost only one read from the surface cache. The surface cache is optimized for 2-d spatial locality, so threads of the same warp that read surface addresses that are close together will achieve best performance. Surface memory is also designed for streaming fetches with a constant latency; that is, a cache hit reduces DRAM bandwidth demand, but not fetch latency.

Surface memory-based implementations of wavelet transformations have mainly focused on images rather than volumes, mainly due to the hardware emphasis on 2-d spatial locality. However since wavelets can be regarded as combinations of down-sampling, up-sampling and filtering operations, Hopf et. al. solved them by applying convolution operators using OpenGL extensions [HE99a, HE99b, HE00]. Wang et.al. follow Murakis [Mur92] approach to transform wavelets on the GPU for RGB images [WWH02]. Here we organize the data in a 2-d layout that allows the transformation of 3-d surface coordinates into 2-d counterparts [KE02, LKH03] and then apply reconstruction formulae that follow the tensor product given by 1.7.

The range of applications for computer graphics has widened over the past decade, and complex operations at vertex and fragment levels are now available for GPUs [FHW04] . Shading languages complete the package to simplify much of the programming obstacles that one faced with fixed pipelines and assembly code [MGA03, Ros04]. In our work, we

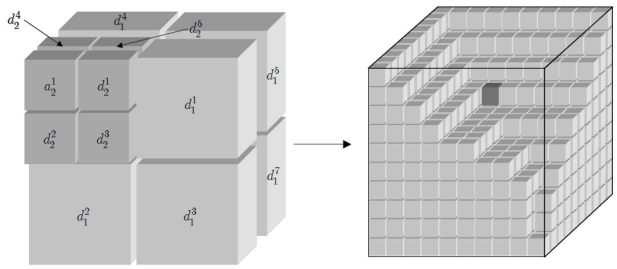

Figure 2.1: VAPOR's GPU volume rendering algorithm is based on a arrayed variable-depth 3-d IDWT. Each 3-d array has symmetric boundary conditions and is handled separately by the GPU.

apply both in order to evaluate the reconstruction formulae.

2.3 Wavelets

Before presenting our surface memory wavelet reconstruction algorithm, we briefly overview the elements from wavelet theory that are relevant to the various stages of the process. Wavelets are defined basis functions in relation to pairs of finite impulse response filters that divide a set of original values (hereafter referred to as A-values) into two parts: the averages or low-frequency coefficients (L values) and the details or high-frequency coefficients (H values). The two filters can now be applied to the L values to create a mulit-resolution analysis [Mal88, Dau92, SN96]. The DWT is a rank one tensor operator, hence multi-dimensional data can be decomposed by applying the wavelet filters successively on each dimension as in (1.1). 2.3 shows the different coefficients resulting from 1, 2 and 3 dimensional decompositions.

Since there are many choices for the basis functions, there are many ways to reconstruct the values. For purposes of comparison, we use the widespread Haar (Daubechies-2) and Daubechies-4 wavelet bases.

2.3.1 Haar Wavelets

Haar wavelets are the simplest to compute, however unfortunately they suffer a larger penalty on image quality when dropping high-frequency coefficients, which happens in multi-resolution environments or time critical conditions. The Haar reconstruction formulae are given in the following equations:

$$\begin{aligned} A_{2i} &= \frac{L_i}{\sqrt{2}} + \frac{H_i}{\sqrt{2}} \\ A_{2i+1} &= \frac{L_i}{\sqrt{2}} - \frac{H_i}{\sqrt{2}} \end{aligned} \tag{2.1}$$

2.3.2 Daubechies Wavelets

Daubechies wavelets are more expensive to compute, however they have superior image quality at a given level of resolution. As opposed to Haar wavelets, quantizing them into integer pipelines results in large loss of precision, which is problematic for solutions that attempt to use the framebuffer as the reconstruction target. The Daubechies reconstruction formulae are the following:

$$\begin{aligned} A_{2i} &= h_0 L_i + h_2 L_{i1} + h_3 H_i + h_1 H_{i1} \\ A_{2i+1} &= h_1 L_i + h_3 L_{i1} - h_2 H_i - h_0 H_{i1} \end{aligned} \tag{2.2}$$

where

$$\begin{aligned} h_0 &= \frac{1+\sqrt{3}}{4\sqrt{2}} \quad , \quad h_1 = \frac{3+\sqrt{3}}{4\sqrt{2}} \\ h_2 &= \frac{3-\sqrt{3}}{4\sqrt{2}} \quad , \quad h_3 = \frac{1-\sqrt{3}}{4\sqrt{2}} \end{aligned} \tag{2.3}$$

The presence of L_{i1} and H_{i1} requires special handling at boundaries. This can be dealt with by zeroing, mirroring or replication. In our implementations, we choose the latest, because it is known to reduce reconstruction artifacts. Finally, these formulae are part of the known Daubechies-4 (db4) wavelet. Other kinds, such as Daubechies-6 (db6) or Daubechies-12 (db12), follow the same reconstruction logic as db4, however they require

more coefficients to reconstruct, which would imply more surface lookups as we will see in the following section.

2.4 GPU Reconstruction

The input to a reconstruction algorithm typically comes as a series of 3-d arrays that represent different kinds of wavelet coefficients (L and H values). A standard approach would fetch them from disk, arrange them into a convenient layout and, by using shared memory, reconstruct into the original data (the A values). The reconstructed volumes are later uploaded into surface memory and rendered. This shared memory-based approach incurs high computation overhead when used together with interactive applications such as surface-based 3-d volume rendering, because there is a lot of memory copying and reordering. To minimize the overhead, our algorithm moves the reconstruction stage into the GPU, where the 3-d L and H coefficients are uploaded into surface memory. By rendering quadrilaterals bound to those surfaces with a specialized fragment program, the A values can be computed and stored into an intermediate surface that is directly accessible by the rendering fragment program without the copying and the reordering.

Assume each coefficient array is a three-dimensional entity that has d elements along each dimension, making it a d^3 array of voxels. As shown in 2.3, there will be 8 arrays of coefficients, LLL, LLH, LHL, LHH, HLL, HLH, HHL and HHH, coming from the wavelet transform of the original $2d^3$ volume. To reconstruct the A values from those arrays, a simple GPU solution is to upload into 3-d surfaces two arrays at a time that represent the corresponding low-frequency and high-frequency coefficients in the z direction (those pairs with mismatching H and L in the left-most letter, that is, LLL and HLL, LLH and HLH, LHL and HHL, and LHH and HHH). One would then render the quadrilaterals using the same z slice from each of the two coefficient arrays as surfaces and combine the values together using (2.1) (for Haar wavelets), or (2.2) (for Daubechies wavelets) to produce the z-reconstructed intermediate values. The vales are copied into another surface. When all z slices are pro-

cessed, we have 4 new arrays of $d \times 2d$. The process then repeats for the y-dimension and later for the x-dimension for which the quadrilaterals must be aligned accordingly, producing a reconstructed array of $2d \times 2d \times 2d$ original values. While straightforward, this solution has the overhead of surface copying and involves too many reconstruction passes. This is essentially what previous approaches have suggested [HE99b] .

To minimize the reconstruction overhead we present here a different method that reduces memory bandwidth usage. As noted in section 1.2, correct index ordering is essential in order to ensure that data locality is preserved. r-major ordering with moderate values of r allows for a higher granularity of parallelism and improved computational throughput.

Our method represents the initial 8 arrays as 2 matricized tensors composed of 4-value elements and stores them in floating-point RGBA surfaces. We then use 2-d surfaces to represent the compacted arrays by further flattening the two 3-d compacted RGBA arrays into two 2-d RGBA surface arrays, where each tile in the surface array represents a slice in the 3-d array. This amounts to representing the IDWT tensor with the matricization $\boldsymbol{X}^{m}_{\{1\}\times\{2,3\}}$. The reconstruction formulae are applied to these RGBA surface arrays to produce the reconstructed surface array, which is the input for the rendering stage. The surface array layout enables us to use off-screen rendering which avoids surface copying and/or reordering and the associated performance penalty. Even though we have a 2-d surface array, we are still able to perform 3-d volume rendering with view-aligned slicing by transforming 3-d surface coordinates into 2-d counterparts. 2.2 shows the whole reconstruction process. The blue boxes represent fragment programs that will be explained in the following sections.

2.4.1 2-d Surface Arrays

A surface array is our primary data structure inside the GPU and it represents a matricized order 3 tensor. These arrays are implemented with 2-d surfaces, and we refer to them as either input or output surface arrays. The input arrays hold the L and H coefficients and are implemented with the OpenGL function `glTexImage2D`. The output arrays, on the other

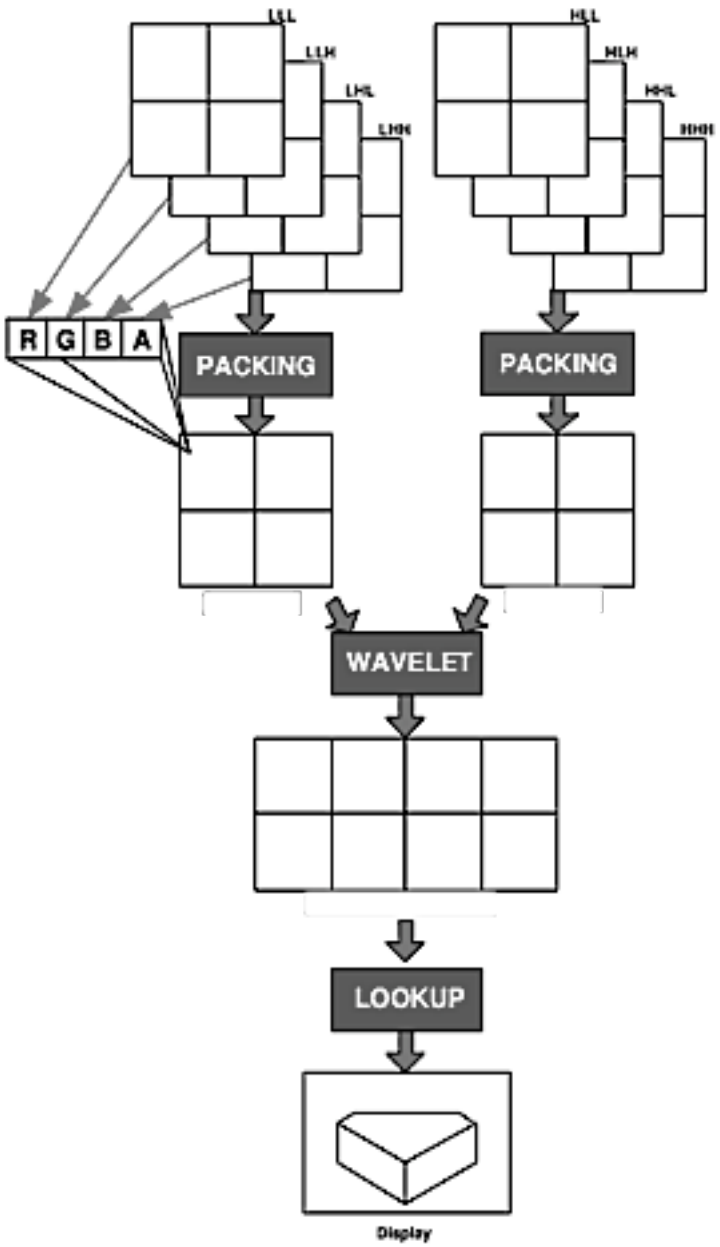

Figure 2.2: An overview of the reconstruction process. Each coefficient array, after being retrieved from disk, is uploaded into a separate surface array; then the 8 surface arrays are separated into 2 groups. Each group is packed into an RGBA surface array such that each color channel corresponds to one of the individual arrays of the group. The 2 new surface arrays (L- and H- arrays) serve as input for the reconstruction stage, which generates an array with double the number of tiles. This surface array serves as input for rendering. When rendering, 3-d surface coordinates must be converted into 2-d counterparts to access the reconstructed array.

hand, hold the reconstructed values and are implemented with pbuffers [FHW04] . The dimensions of a surface array (T_W, T_H) and the number of tiles in each dimension (N_x, N_y) are derived from the dimensions of a array with d^3 coefficients as follows:

$$\begin{aligned} N_x &= 2^{\log_2 \sqrt{d}} \quad ,N_y = 2^{\log_2 \sqrt{d}} \\ T_W &= d \times N_x \quad ,T_H = d \times N_y \end{aligned} \tag{2.4}$$

Since each array has d slices, the surface array has d tiles and they are arranged into a board of $N_x \times N_y$ tiles, where each tile has $d \times d$ elements. We will keep in mind that the are derived from a array with $2d^3$ voxels. An example of a reconstructed surface array is shown in 2.3.

With such a surface array layout, all elements from the original coefficient array are placed in one 2-d surface array, hence a single surface id is sufficient to bind it with fragment programs. Also, accessing a specific element is straightforward even in the case of rendering using 3-d surface coordinates. Details for surface coordinate transformations will be presented in a later section.

The 2-d surface arrays provide the foundation for packing, reconstruction and rendering, which are now presented.

2.4.2 Packing

Before reconstruction, the original volume as a whole is first divided into small volume blocks of $2d \times 2d \times 2d$ voxels. each voxel is then transformed via a Discrete Wavelet Transform into 8 arrays containing $d \times d \times d$ wavelet coefficients.

The ability to reduce the number of surface lookups during reconstruction can effectively accelerate the calculations. This is due to the fact that multiple coefficients in different arrays but at the same location in memory are needed to evaluate the reconstruction formulae presented below. In most of the graphics hardware, a single surface lookup can retrieve four color channels simultaneously with an optimized performance, so it will be beneficial if we

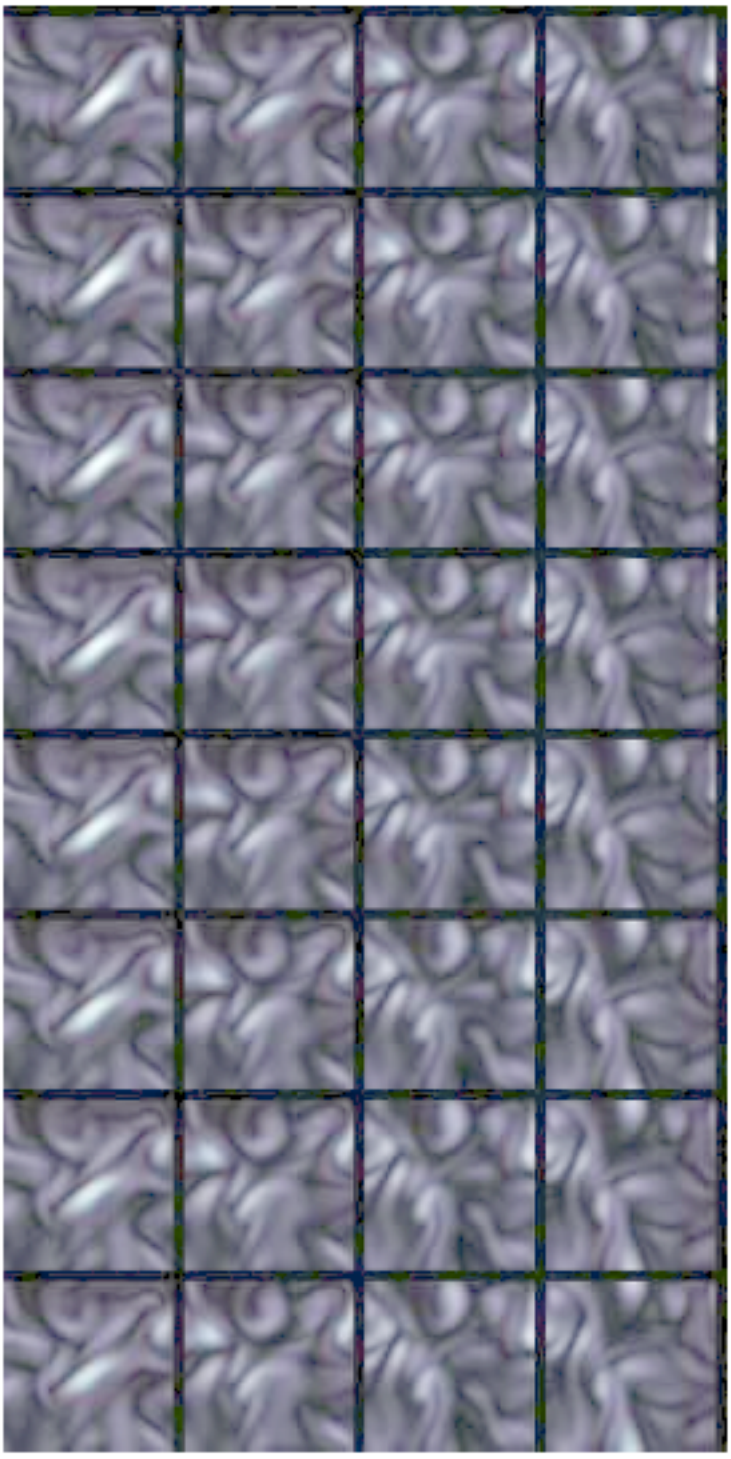

Figure 2.3: A partial view of the reconstructed surface array for the 128^3 vorticity dataset. A single tile contains 128^2 voxels, and the entire surface array is a 2048×1024 2-d texture.

can have four coefficients into a single RGBA surface. We exploit this property by packing our coefficient arrays into RGBA surface arrays at the beginning of the reconstruction stage. Initially, we will retrieve the 3-d arrays from disk and upload each one into its respective one-component surface array. Then, using a fragment program, we look up each single-component surface array surface and compact them into two RGBA surface arrays: one surface array will hold the L coefficients in Z, i.e., LLL, LLH, LHL, and LHH, and the other surface array will hold the H coefficients (HLL, HLH, HHL, and HHH).

In the compacted surface array, each floating-point color channel is used to store one coefficient array. For example, in the first compacted surface array, the red channel stores the coefficients from LLL; the green channel stores the coefficients from LLH; the blue channel stores the coefficients from LHL, and the α channel is used to store the coefficients from LHH. The 2 compacted surface arrays are known henceforth as the L-surface array and the H-surface array. The separate uploading for each coefficient array at the beginning of the packing process also significantly reduces bandwidth requirements. This is due to the fact that uploading is not performed for unneeded arrays (i.e. the array is hidden in the current rendered image, the array has zero coefficients, etc.). However, since the reconstruction process still needs to access those zero coefficients, a zero surface array is used for binding rather than the coefficients from the discarded array. This artificial surface array is initialized once and has the same dimensions as a coefficient array. Another advantage is the flexibility to incorporate methods that upload compressed surfaces, such as the work of Kraus et.al. [KE02] , whose adaptive surfaces can easily be integrated. With the L- and H-surface array ready, the reconstruction stage can now proceed to evaluate the equations given in the next section.

2.4.3 Reconstruction

The goal of the reconstruction stage is to transform the values stored in the L- and H-surface array into a new surface array that will hold the A values for rendering. A specialized fragment program will perform this task by rasterizing a quadrilateral that uses the two

input surface arrays to produce the new surface array. The output array will then have $2d$ tiles with each tile having a total of $2d \times 2d$ voxels, and the input array has d tiles with $d \times d$ coefficients. When rasterizing the quadrilateral in our GPU code, each fragment receives coordinates (c_x, c_y) that correspond to a location in the output array, where A values are written out. Therefore (c_x, c_y) vary from $(0, 0)$ to (T_W, T_H), which are the dimensions of the output array. In order to retrieve the coefficients from the input arrays, since the size of the arrays are different, it is necessary to transform the output coordinates into the coordinate space of the input surface arrays before we can reconstruct. It is also necessary to determine which which reconstruction equations to use for each dimension. First, given the surface coordinates (c_x, c_y), we need to determine which tiles i and j this fragment corresponds to in the input and output surface arrays respectively, because all calculations are performed inside their dimensions. We calculate i and j as follows

$$\begin{array}{rcl} j & = & \lfloor c_x/2d \rfloor + \lfloor c_y/2d \rfloor N_x \\ i & = & \lfloor j/2 \rfloor \end{array}, \tag{2.5}$$

where N_x is the number of tiles in the x dimension of the output surface array. We also determine (c_x, c_y)s relative position (cj_x, cj_y) inside the tile j by:

$$\begin{array}{rcl} cj_x & = & c_x \% (2d) \\ cj_y & = & c_y \% (2d) \end{array}, \tag{2.6}$$

where % is the mod operator. The tiles in each surface array are essentially linearly even though conceptually the surface array has a two-dimensional layout. The 3-d coordinates (cj_x, cj_y, j) decide the reconstruction formulae to apply in each dimension. If a value for a corresponding dimension is even, we use the formula for A_{2i}; otherwise, we use the formula for A_{2i+1}.

Finally, to retrieve the coefficients from the input arrays we need to find the location (ci_x, ci_y) inside of tile i. We proceed to find values of i beginning (o_x, o_y) and then offset the address pointer by $(cj_x, cj_y)/2$. The calculations using N_x of the input arrays are as follows:

$$
\begin{aligned}
o_x &= +i\%N_x \times d \\
o_y &= +\lfloor i/N_x \times d \\
ci_x &= o_x \lfloor cj_x/2 \rfloor \\
cj_x &= o_y \lfloor cj_y/2 \rfloor
\end{aligned}
\tag{2.7}
$$

With all locations calculated, we reconstruct according to the type of wavelet. In the case of Haar, looking up both the L- and H-surface array with the same surface coordinates (ci_x, ci_y)fetches 8 coefficients (2 RGBA values).

The j index decides between equations 1 and 2 to apply to the 2 RGBA tuples, thus reconstructing in the z-dimension and leaving a new RGBA tuple. cj_y decides the reconstruction of two more values based on this new tuple: one combining its red and blue channel, and the other combining its green and alpha channel. Finally, cj_x decides how to reconstruct the remaining two values for rendering. In the case of Daubechies, the calculations are not straight-forward, however it follows the same logic. The 8 values fetched are insufficient to evaluate equations 3 or 4 because of the L_{i1} and H_{i1} terms. Therefore, we need an additional coordinate, which we derive from i. In the case that $i = 0$, then we derive it from d. This completes the requirements but only for the z dimension. For the sake of simplicity, we perform a separate pass to reconstruct the x and y dimensions, which uses the output of the first pass. For every position in this second pass, 4 lookups with 4 values each are performed, yielding the 16 coefficients to reconstruct an A value. The 0 condition in the x and y passes is enforced inside of a tile, moving to the last row or last column as needed.

2.4.4 Rendering

The last stage performs the traditional 3-d volume rendering algorithm [FCC94] on a reconstructed surface array. The volume is sliced by a set of polygons that are composited back-to-front and clipped to the boundaries of the volume. However, the surface array is a 2-d surface that represents a 3-d object. Therefore a fragment program must convert the surface coordinates before lookup of the value can take place. The following does the

conversion from 3-d coordinates $(c3_x, c3_y, c3_z)$ to 2-d coordinates $(c2_x, c2_y)$ using N_x of the reconstructed surface array:

$$\begin{aligned} c2_x &= (\lfloor c3_z \rfloor \% N_x) \cdot d + c3_x \\ c2_y &= (\lfloor c3_z \rfloor / N_x) \cdot d + c3_y \end{aligned} . \tag{2.8}$$

The integer part of the z-component is decomposed into the indices that indicate the row and the column of the target tile in the reconstructed surface array. Multiplying the indices by the tile resolution $2d$ results in the target tiles beginning. Finally, the xy-components are added together to form the 2-dth coordinate of the surface array. This emulates the option GL NEAR. To emulate GL LINEAR, linear filtering is turned on for the reconstructed surface array, and a second co- ordinate is calculated by using the xy-components on tile $\lfloor c3_z \rfloor + 1$. After performing both bilinear lookups, we interpolate between the retrieved values with the fractional part of the z-component. 2.4 shows GL_NEAR and GL_LINEAR emulation for the shockwave dataset.

2.5 Results

Our datasets, either consisting of a single array or multi-resolution hierarchies, were transformed into the wavelet coefficients with an off-line decomposition program. For single array testing, the following 128^3 volumes were used: the vorticity dataset, which contains 8-bit integer voxels, and the 8-bit shockwave dataset. For multi-resolution testing, the dataset tested was the rotating convective Taylor columns dataset, where we extracted a 480^3 subvolume and converted the 12-bit values to floating-point values.

All our testings were performed on an Nvidia GeForce G80 graphics card. The fragment programs were written using noted in section 2.4, the wavelet filtering is done in two passes: one for the Z-dimension and the other for the XY-dimensions. Finally, our shared memory implementation followed the process described in section 2.3. surface arrays were implemented with 32-bit floating-point channels and Nvidias GL TEXTURE RECTANGLE NV.

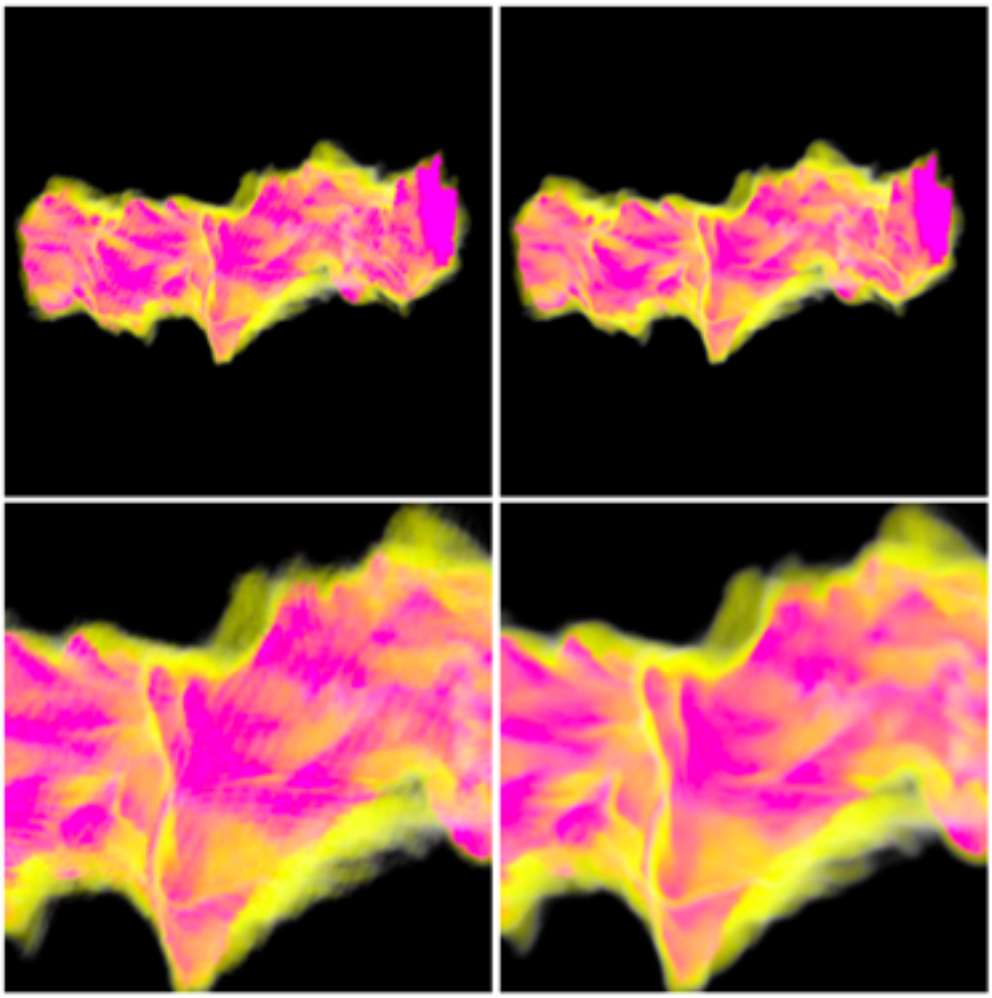

Figure 2.4: Image samples of the Shockwave dataset. The left-hand column contains the reconstructed surface array rendered using nearest neighbor interpolation. The right-hand column contains the reconstructed surface array rendered using simulated trilinear interpolation.

However, reconstructed surface arrays used 8-bit channels when interpolation was turned on, because floating-point interpolation was not supported for this card at the time of writing.

Table 2.1: Total rendering timings for single varying array size under Haar and Daubechies GPU reconstruction using the vorticity dataset. The numbers in parenthesis are times for the uploading and the reconstruction stages. The window size for all tests was 256^2 pixels.

Brick Size (voxels)	GPU Haar Total (Up/Rec) (msec)	GPU Daub Total (Up/Rec) (msec)
128^3	102.89 (44.12/15.24)	124.61 (43.89/37.34)
64^3	33.28 (5.78/13.20)	33.44 (5.64/16.48)
32^3	16.64 (0.94/0.12)	16.80 (0.93/0.12)
16^3	16.64 (0.12/0.08)	16.72 (0.10/0.08)

Table 2.2: Timings of the rendering algorithm under varying window size using the vorticity dataset.

Brick Size (voxels)	256^2 (msec)	512^2 (msec)	1024^2 (msec)
128^3	43.20	121.25	419.85
64^3	16.80	64.22	210.47
32^3	16.56	63.28	209.12
16^3	16.64	61.87	210.33

We first present our results for solitary arrays. We registered the average timings of an animation that rotates the array with all stages turned on, which means that for every frame we uploaded, reconstructed and rendered. The Haar wavelets are indeed more computationally than the Daubechies, however array size does affect the packing and rendering stages. From 2.1, we see that the uploading stage scales as expected with array size. There a sudden increase in performance durning the reconstruction stage when processing arrays of 16^3 and 64^3 voxels. This is likely related to the manner in which surface memory is organized and the parallel execution of the fragment program. When the hardware is rendering, we see that array sizes that are greater than 64^3 voxels experience overhead.

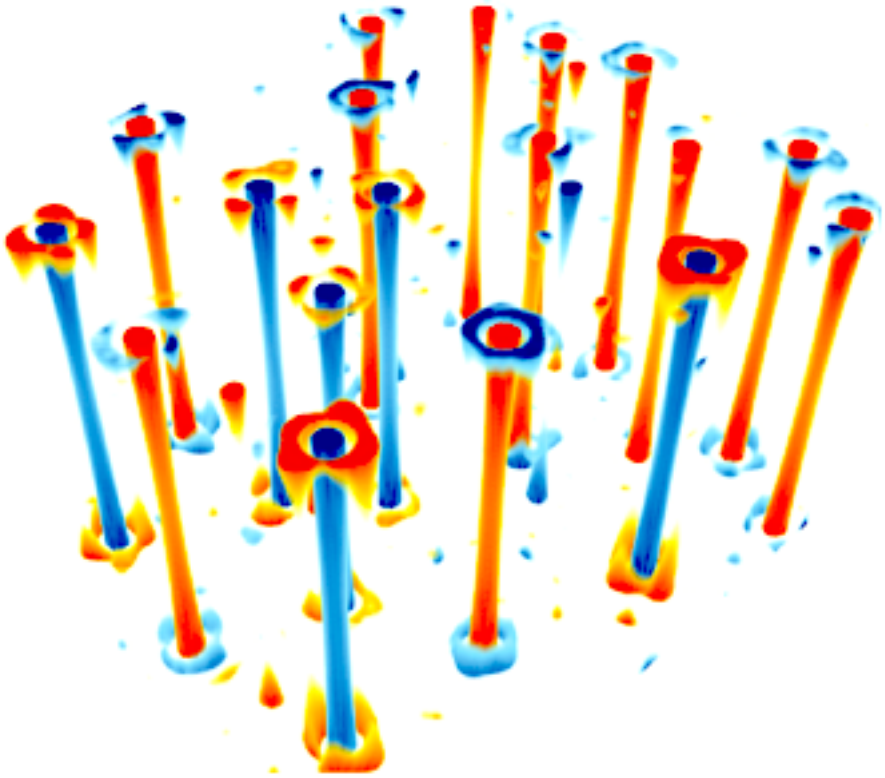

Figure 2.5: A full-resolution reconstruction of the rotating convective Taylor columns dataset. The 480^3 dataset is reconstructed from 5 resolution levels using 585 Haar wavelet nodes.[3]

In 2.2, we show the performance of the rendering stage alone with varying window size. The packing and reconstruction was done only for the first frame and the results were reused for the rest of the animation. When we increas the window size to generate more rendering fragments, we see that the transformation from 3-d surfaces coordinates to 2-d counterparts is transparent to the user and scales properly for all but the largest array. The 128^3 array size makes a 2048×1024 surface that is well within the maximum surface size of 4096, however the graphics hardware makes more effort to access the texels, which may be related to cache coherency and surface memory organization. We therefore choose the 64^3 array size to decompose large volumes into multi-resolution hierarchies for most VAPOR applications.

We decomposed the rotating convective Taylor columns dataset into 64^3 overlapping

arrays and generated 5 levels of detail with 585 reconstruction nodes. Since it is our intention to see the performance of the reconstruction, we rendered all nodes, disregarding viewing metrics and empty spaces. 2.5 shows the image at full resolution. 2.3 and 2.4 show the performance under Haar and Daubechies DWTs respectively. In these tests, the time to retrieve all arrays from the disk fluctuated between 1 and 0.80 seconds. When compared against our shared memory implementation that uses 3-d surfaces for rendering, the difference in performance is notable. Furthermore, the packing stage shows it's utility when we start to discard high frequency coefficients, which in a multi-resolution scenario is very likely to happen. From 2.3 and 2.4, we also see that if we systematically drop coefficient arrays, it affects the timings of the reconstruction, which includes uploading and packing. The timing of the rendering stage remains the same since the window size is kept at 512^2 pixels. The numbered lines indicate that we are reconstructing with that many arrays, 8 arrays for full reconstruction and 1 array for no uploading. However the manipulation of the arrays comes at a visual cost, fig. 2.3 shows the gradual loss of quality when discarding high-frequency coefficient arrays for the vorticity dataset. The Daubechies-4 wavelet retains the features of the data better than Haar wavelet, reconstructing most of the features at half-resolution. However the Haar wavelet exhibits the well-known blocking pattern at half-resolution.

Shared memory counterparts cannot take advantage of the discarding of arrays since they always upload a array to a 3-d surface with the same dimensions. Moreover, in shared memory implementations, there is an additional step for overhead: once the parent reconstructs its array, children extract parts of it, for which a copying is performed. Our packing stage, which does the interleaving and formatting into RGBA surfaces, takes care of this extraction by calculating correct surface coordinates given the childs initial offsets, so no additional copying is introduced.

Finally, the speedup between shared memory and hardware, with full 8-array reconstruction, is 5.83x for Haar, and 6.76x for Daubechies. This makes GPU-based 3-d wavelet reconstruction quite efficient in comparison to shared memory counterparts.

Table 2.3: Total rendering, loading and reconstruction time for the vorticity dataset using Haar wavelets and a 512^2 window size. Full GPU reconstruction achieves a 5.83x speedup with respect to the shared memory implementation, and, by dropping coefficient arrays, the GPU gains further speed in the reconstruction stage.

Resolution	Total (sec)	Speedup	Rendering (sec)	Haar (sec)
3-d Surfaces & CPU	29.40	1.00	0.29	28.11
8	5.04	5.83	0.29	3.90
7	4.60	6.39	0.27	3.46
6	4.16	7.06	0.27	2.92
5	3.87	7.59	0.28	2.68
4	3.30	8.90	0.28	2.13
3	2.80	10.50	0.29	1.66
2	2.34	12.56	0.28	1.18
1	1.90	15.47	0.27	0.75

2.6 Conclusions

We have devised an algorithm that can successfully utilize GPUs to reconstruct wavelet coefficients in the form of single arrays or arrays from a multi-resolution hierarchy. We construct a pipeline divided into 3 stages: packing, reconstruction and rendering. All stages are accessing or rendering to surface arrays that represent a 3-d array, even though, they are in fact 2-d surface entities. The surface arrays are directly input to our hardware-based volume renderer with fragments programs that transform 3-d surface coordinates into 2-d counterparts. The flexibility of the surface array allows for rendering to surfaces through pbuffers, which have comparitively low memory access latency.

This solution has allowed atmospheric scientists working on GPU-equipped desktop machines to use VAPOR to represent large 3-d sections of the Earth's atmosphere with extremely high resolution in order to make accurate predictions about macro-level phenomena

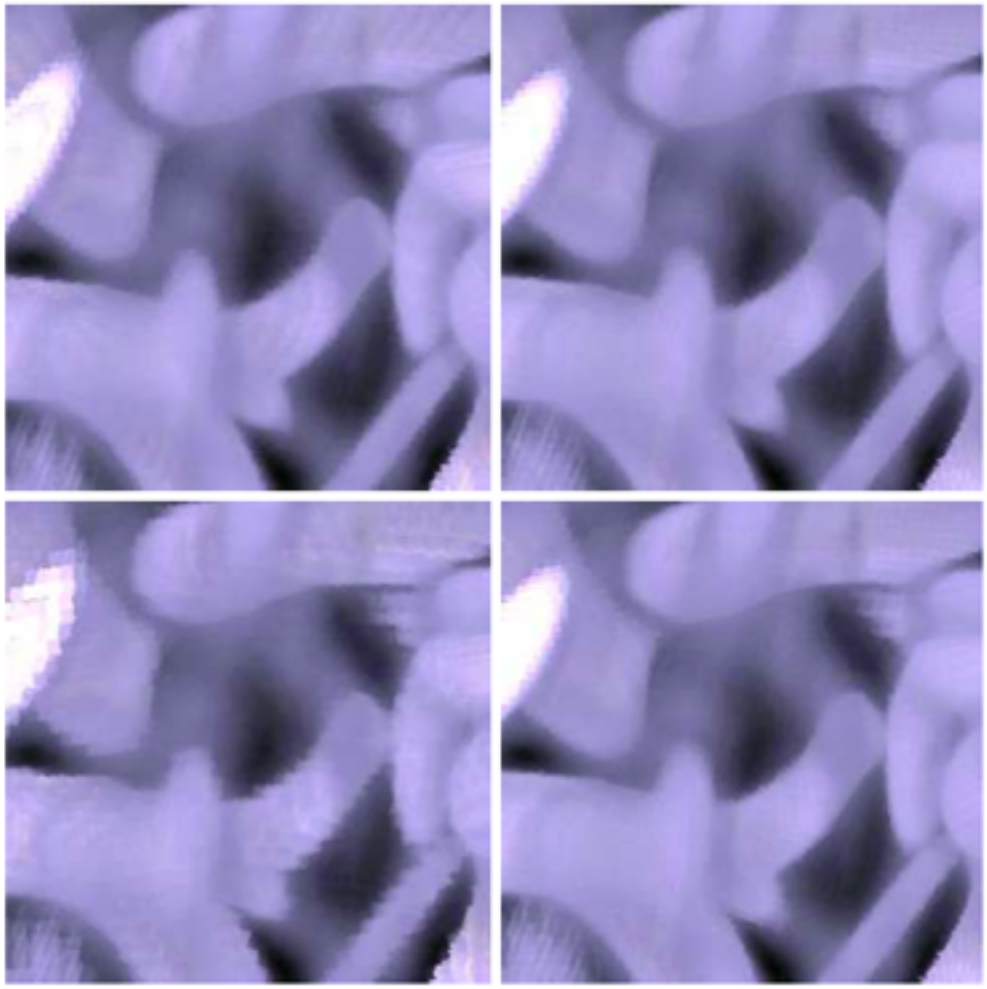

Figure 2.6: A comparison between Haar (left column) and Daubechies (right column) wavelets for the vorticity dataset. Haar shows artifacts as coefficients are dropped. The images on the top have dropped the four H** coefficient arrays; while those at the bottom have dropped everything but LLL and LLH. At this point, Haar shows the familiar blocking pattern.

Table 2.4: Total rendering, loading and reconstruction time for the vorticity dataset using Daubechies wavelets and a 512^2 window size. In this case the gains are better than those for the Haar wavelet because the corresponding shared memory implementations are slower.

Resolution	Total (sec)	Speedup	Rendering (sec)	Daub (sec)
3-*d* Surfaces & CPU	55.31	1.00	0.29	53.79
8	8.18	6.76	0.25	6.92
7	7.56	7.31	0.25	6.48
6	7.09	7.80	0.25	5.99
5	6.60	8.38	0.30	5.39
4	6.15	8.99	0.30	4.94
3	5.72	9.66	0.29	4.56
2	5.44	10.16	0.29	4.27
1	5.15	10.73	0.29	4.04

[CMN07, NCC08]. This would not be possible without the aid of efficient data compression and multi-resolution volume rendering to reduce the computation requirements. Within the scope of VAPOR's applications, the GPU accelerated Inverse Discrete Wavelet Transform (IDWT) described above has proven to be critical for the exploration of large gas and fluid dynamics datasets on commodity hardware.

CHAPTER 3

Representations for Hierarchical Methods - I

In this chapter we demonstrate the suitability of tensor product representations to an N-body simulation in 2-d using the Barnes-Hut algorithm. This relatively simple problem serves as an introduction to the more complex analysis of hierarchical methods in arbitrary dimensions discussed in Chapter 4. It also provides a useful testbed for the other main goal of this chapter, which is to formulate high-level abstractions that hide the complexity of parallelisation and work load distribution from the application programmer. These abstractions provide fundamental building blocks for designing efficient GPU cluster applications.

In section 3.3, we present an abstract approach to programming heterogeneous GPU clusters, which combines throughput-oriented algorithmic primitives and graph-based combinators. We present an implementation of this approach using OpenCL, which provides application programmers with a set of efficient, reusable components (henceforth called primitives), which are customisable for particular applications by means of data and code parameters. We demonstrate the use of our system on a non-trivial case study with a complex, dynamic behaviour the Barnes-Hut (BH) algorithm for N-body simulation. The BH algorithm's n-bodies can be conveniently represented as a vectorization of an order-$2n$ tensor (see section 1.2 above). It is therefore an excellent candidate for middleware abstraction.

3.1 Background

$\mathcal{O}(n)$ or $\mathcal{O}(n\log(n))$ hierarchical algorithms that can solve a wide range of physical problems are increasingly being used in large scale science and engineering applications [GL06].

These algorithms are based on the Barnes-Hut and Fast Multipole methods, and are often referred to as hierarchical N-body methods. They are based on two common properties that many natural systems exhibit: rapid decay of the high-frequency components of the forces involved, and a concomitant range of scales in terms of information requirements. A body in such a system generally requires less information and requires it less frequently from more distant parts of the system. Hierarchical N-body methods exploit this property in order to efficiently propagate information from region to region throughout the domain. In astrophysical applications for example, the bodies are stars or planets in a galaxy, and the governing interaction law is gravitational. In molecular dynamics the bodies are molecules or atoms, and the interaction is Coulombic. Similarly, in vortex methods for incompressible flow the interactions among the vortices have a Coulombic character.

The applications listed above are all classical N-body problem domains. In these cases, the system being simulated actually consists of physical entities (planetary bodies, electrons, molecules, etc.). However, the same hierarchical algorithms have had notable success as numerical methods in their own right. For example, integral equations resulting from boundary value problems can be solved rapidly by such methods, where N is the number of nodes in the discretization of the boundary (see section 4.2 and [Kil]). Many problems in the field of numerical complex analysis can be reduced to computing a Cauchy integral, which can itself be viewed as equivalent to an electrostatic problem. Other examples include wavelet basis methods (see chapter 2), multigrid methods, domain decomposition methods, multi-level preconditioners, and adaptive mesh-refinement algorithms.

The hierarchical N-body methods mentioned above afford substantial parallelism, hence applications that use them are frequent clients on high-performance multiprocessors. Such applications typically have characteristics that make their workloads challenging to partition effectively for parallel performance, however, owing to their nonuniform, dynamically changing characteristics and their need for long-range communication. In particular, the workload distribution and communication patterns across the logical units of parallelism (the bodies, for example) are both nonuniform and may also change as the computation proceeds, thus

complicating the simultaneous incorporation of load partitioning and data locality.

GPU cluster programmers are faced with the challenge of developing applications that can run distributed across several heterogeneous hosts. Programs must use cluster resources efficiently and incorporate flexible load-partitioning strategies in order to distribute tasks among hosts that are not known at compile time. The success of GPU cluster technology will depend on creating suitable programming models and middleware to liberate application programmers from the complex and low-level details that have to be taken into account during cluster software development.

These heterogeneous GPU clusters offer enormous amounts of potential computational resources. However, actually exploiting that potential is a nontrivial challenge. Both parallelism and, especially, heterogeneous parallelism are major challenges for software development. Tackling them has created renewed interest in both academia and industry for developing high-level programming models, such as OpenCL [ope] and OpenMP [DCA08]. High-level programming models provide two major advantages over low-level interfaces such as threads and locks. First, they provide constructs that make it easier for programmers to express parallelism. Second, their runtimes and schedulers manage concurrency and communication, simplifying these burdens for the programmer. High-level programming models achieve their advantages by placing the application-programming model boundary closer to application constructs than hardware interfaces. This gives them more insight into application structure, patterns, and semantics. In exchange, they must impose assumptions about the chief types of applications and developers that they target.

Our approach differs from other programming frameworks for clusters in that it provides predefined computation patterns with built-in load-partitioning strategies. van Nieuwpoort [NKB01] also provides load-partitioning, but it was limited to divide-and-conquer applications.

In the following sections we show how the use of high-level algebraic representations can hide most of the complexity of distributing computations over a heterogeneous GPU cluster from the application programmer; also how the load-partitioning mechanisms incorporated

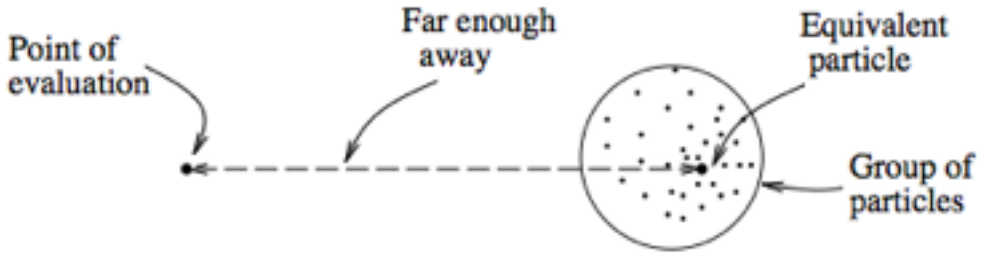

Figure 3.1: Approximation of a group of bodies by a single equivalent particle.

in our system can evenly balance work between heterogeneous compute nodes in a cluster. We conclude the chapter by reporting experimental results for the BH algorithm on a GPU testbed.

3.2 The Barnes-Hut Algorithm

In classical N-body problems, the evolution of the interaction of acollection of N distinct bodies is studied. Computing the interactions among bodies can be expensive, especially when long-range interactions cannot be approximated. For example, if all pairwise interactions are computed directly, the resulting computationaly complexity is $\mathcal{O}(N^2)$, which is prohibitive for the large systems typical in applications. Hierarchical N-body methods reduce the algorithmic complexity to $\mathcal{O}(N \log N)$ (for the Barnes-Hut algorithm [BH86]), or to $\mathcal{O}(N)$ (for the Fast Multipole Method [Rok85, GR87]).

Both algorithms take advantage of the fact that high-frequency components of interactions between bodies fall off rapidly with distance. Therefore the effect of a large group of bodies may be approximated by an equivalent body whose first few moments match those of the particle group distribution, provided the group is far enough away from the affected location (see 3.1). The farther away the bodies, the larger the group that may be approximated by a single body (see 3.2). Furthermore, this insight may be applied in a hierarchical fashion to clusters of groups.

As noted above, the Barnes-Hut (BH) algorithm is a widely used approach to comput-

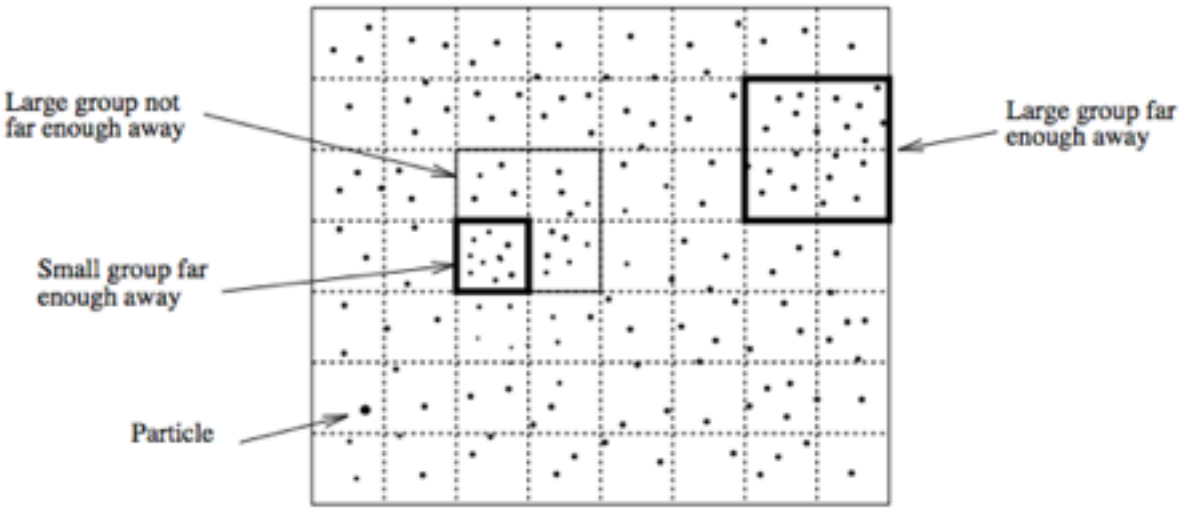

Figure 3.2: Hierarchical approximation of cells by equivalent bodies.

ing force interactions of bodies based on their attractive pull and position in space. At each timestep, the pairwise interactions of all bodies have to be calculated, which implies a computational complexity of $\mathcal{O}(n^2)$ for n bodies. However the BH algorithm reduces the complexity to $\mathcal{O}(n \log n)$, by grouping distant particles: for a single particle in the BH algorithm, distant groups of particles are considered as a single object if the ratio boxsize/distance is smaller than a simulation-specific coefficient θ chosen by the user.

For efficient access to the plethora of possible groups in a simulation space with a large

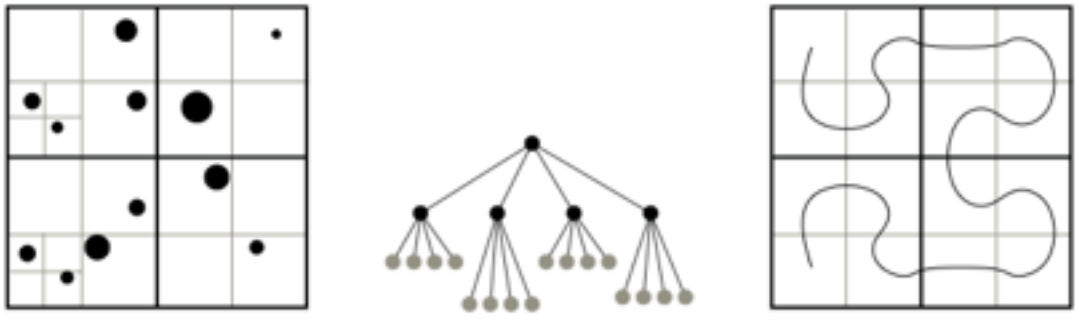

Figure 3.3: Barnes-Hut partition of the simulation space using (from left to right) space partitioning, a quadtree, and Peano-Hilbert ordering

number of objects, the BH algorithm subdivides the 3D simulation space using a hierarchical quadtree with four child cubes for each node. The trees leaves contain single particles, parental nodes represent the particle group of all child nodes and contain the groups center and aggregated mass. The force calculation at the location of a single particle then is performed by a depth-first traversal of the tree. Figure 3.3 depicts an example partition and the resulting quadtree for the 2D case (see [BH86] and 4 below for further details and cost considerations).

For computing the force interactions in parallel, the particles are distributed among participating hosts. Our implementation uses the Peano-Hilbert order (the right-hand image in figure 3.3), which provides a total linearisation of a two- or three-dimensional space [GKS94]. In the resulting vector of particles, adjacent objects are placed close together. Thus, a block-wise distribution of the particles vector among hosts results in a contiguous particle space assigned to each host. This leads to reduced communication between hosts, because particles that are close together (and therefore need to exchange information often) are likely to be on the same host.

3.2.1 Kernel Compression

This section provides sufficient conditions for the application the tensor product representation described in Theorem 1.7 above to the BH algorithm in the two dimensional case. Our goal is to use Lemma 3.3 and the results in section 3.2.2 below to prove the following theorem concerning the kernel G_0.

Theorem 3.1. *For any positive integer n, sets P and Q each consisting of at most n points in $\mathbb{R}^2$, and positive real number ϵ , the kernel G_0 defined in* (4.10) *has a tensor rank of at most a universal constant times* $\ln(T/\epsilon)$*, where T is defined in* (4.28) *in section 4.5.1.*

The relevant matrix has a numerical rank of at most a universal constant times $\ln(T/\epsilon)$ since, as we will see, 3.3 represents the corresponding potentials as linear combinations of at most a universal constant times $\ln(T/\epsilon)$ functions.

We begin with a lemma (3.3 below), which is needed for the proof of 3.1. Lemma 3.3 represents as a linear combination of $2m+1$ functions $g_0, g_1, g_2, \ldots, g_{m-1}, g_m$, and $\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_{m-1}, \tilde{g}_m$ (known as multipoles) the action of the kernel G_0 , as acting from an arbitrary finite number of points in $\mathbb{R}^2$ to any point separated by a factor S from a disc containing the source points. For convenience, we first prove 3.3 for a single source point (i.e., with $n = 1$ in 3.3).

Lemma 3.2. *Suppose R, S are real numbers, and ρ, z_0, z_1 are complex numbers. Let $R > 0, S > 1$ and $\|z_1 - z_0\| \leq R$. Then*

$$|\rho \ln(z - z_1) - (\rho \ln(z - z_0) + \sum_{j=1}^{m} c_j g_j(z) + \sum_{j=1}^{m} \tilde{c}_j \tilde{g}_j(z))| \leq \frac{|\rho|}{(S-1)S^m}$$

For $m \in N$ and $z \in C$ satisfying $|z - z_0| \geq SR$.

$$\begin{aligned} c_j = f_j \rho \quad & f_j = -\tfrac{1}{2j}(\tfrac{z_1 - z_0}{R})^j \\ \tilde{c_j} = \tilde{f}_j \rho \quad & \tilde{f}_j = -\tfrac{1}{2j}(\overline{\tfrac{z_1 - z_0}{R}})^j \\ & g_j(z) = (\tfrac{R}{z - z_0})^j \\ & \tilde{g_j}(z) = (\tfrac{R}{\overline{z - z_0}})^j \end{aligned}$$

Proof. Define w by the formula $w = \frac{z_1 - z_0}{z - z_0}$ and observe that $\ln|z - z_1| = \ln|z - z_0| + \ln|1 - w|$. Furthermore $|z - z_0| \geq SR$ and $\|z_1 - z_0\| \leq R$ imply that $|w| \leq S^{-1}$. Hence we may expand a Taylor series

$$\begin{aligned} \ln|1 - w| &= \textstyle\sum_{j=1}^{\infty} \frac{-w^j}{j} \\ |\ln|1 - w| - (\textstyle\sum_{j=1}^{m} \frac{-w^j}{j} + \sum_{j=1} \frac{-\bar{w}^j}{j})| &\leq \textstyle\sum_{j=m+1}^{\infty} \frac{|w|^j}{j} \end{aligned}$$

The right hand side is in turn bounded by

$$\sum_{j=m+1}^{\infty} \frac{|w|^j}{j} \leq \sum_{j=m+1}^{\infty} |w|^j = \frac{|w|^{m+1}}{1 - |w|}$$

Since $|w| \leq S^{-1}$ the desired bound follows immediately.

□

We can now generalize 3.2 to sums of kernels $\sum_{k=1}^{n} \rho_k \ln|z - z_k|$ provided that the z_j are separated from z. See 3.2.1. This is a slight modification of a lemma in [GR87].

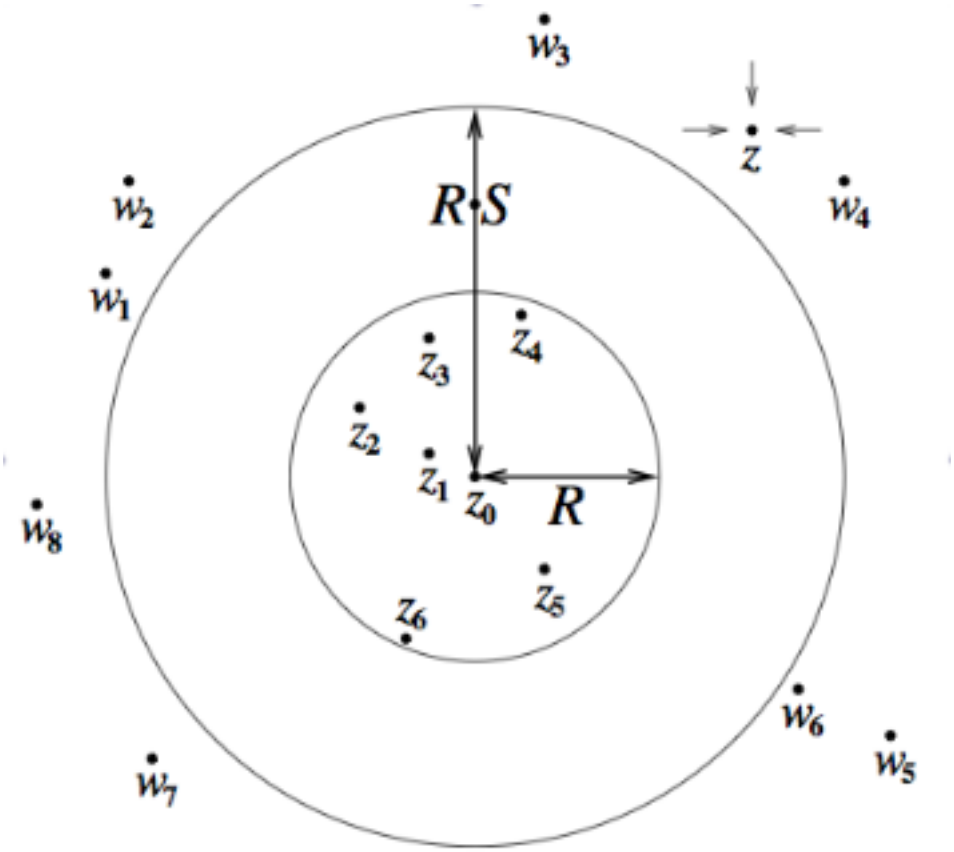

Figure 3.4: The points $z_1, z_2, ..., z_{n?1}, z_n$ are all closer than R units to z_0. The points $z, w_1, w2, ..., w_{l?1}, w_l$ are all farther than RS units from z_0.

Lemma 3.3. *Suppose R, S are real numbers, and $\{\rho_k\}_{k=1}^n, z, \{z_k\}_{k=1}^n$ are complex numbers. Let $R > 0, S > 1$ and $\|z_k - z_0\| \leq R$. Then*

$$|\rho \sum_{k=1}^{n} \ln|z - z_k| - (\rho \ln|z - z_0| + \sum_{j=1}^{m} c_j g_j(z) + \sum_{j=1}^{m} \tilde{c}_j \tilde{g}_j(z))| \leq \frac{\|\rho\|_1}{(S-1)S^m}$$

For $m \in N$ and $z \in C$ satisfying $|z - z_0| \geq SR$.

$$\begin{aligned} c_j = \textstyle\sum_{k=1}^n f_{j,k}\rho \quad & f_{j,k} = -\frac{1}{2j}(\frac{z_k - z_0}{R})^j \\ \tilde{c_j} = \textstyle\sum_{k=1}^n \tilde{f_{j,k}}\rho \quad & \tilde{f_{j,k}} = -\frac{1}{2j}(\frac{\overline{z_k - z_0}}{R})^j \\ & g_j(z) = (\frac{R}{z - z_0})^j \\ & \tilde{g_j}(z) = (\frac{R}{\overline{z - z_0}})^j \end{aligned}$$

Proof. Apply theorem 3.2 for to z, z_k for each k. □

A straight-forward application of 3.3 then proves 3.1 above. It is also important to note that the separable representation of $\rho \sum_{k=1}^n \ln|z - z_k|$ in 3.3 is numerically stable, since all relevant quantities are at most 1 in magnitude for any $z \in \mathbb{C}$, $j = 1, 2, \ldots, m1, m$, and $k = 1, 2, \ldots, n-1, n$. The following lemma interprets 3.3 in linear-algebraic terms and provides an upper bound on rank $2m+1$ approximation error.

Theorem 3.4. *Suppose R, S are real numbers, and $\{z_k\}_{k=1}^n, \{w_i\}_{i=1}^l$ are complex numbers. Let $R > 0, S > 1$, $\|z_k - z_0\| \leq R$ and $|w_i - z_0| \geq SR$. Let A be the ln matrix with entries $A_{i,k} = \ln|w_i - z_k|$. Then the $(2m+2)^{nd}$ greatest singular value $\sigma_{2m+2}(A)$ satisfies*

$$\sigma_{2m+2}(A) \leq \frac{\sqrt{nl}}{S^m}$$

For any positive integer m with $2m + 1 < min(l, n)$.

Proof. Apply 3.3 repeatedly to $\{w_i\}_{i=1}^l$ and sum. □

3.2.2 Product Representation

In this section we generalize Lemma 3.4 above and establish sufficient conditions for the use of the product representation in Theorem 1.7. Specifically, we show that for any set

of n continuous functions on a compact space S, there exist numerically stable n-point interpolation formulae.

Let Ω denote a compact space, let $m \leq n$ be positive integers and $f_1, \ldots, f_{m-1}, f_m$ denote bounded complex-valued functions on Ω (all results apply equally well to real-valued functions, however). For any n points $x_1, x_2, \ldots, x_{n-1}, x_n$ in Ω, let $A = A(x_1, x_2, \ldots, x_{n-1}, x_n)$ be the m by n matrix with entries $a_{j,k} = f_j(x_k)$.

Lemma 3.5. *Let k be a positive integer, E be an arbitrary set containing at least l points, and $\{f_1, f_2, \ldots, f_{l-1}, f_l\}$ be complex-valued functions on E that are linearly independant. Then, there exist l points $\{x_1, x_2, \ldots, x_{l-1}, x_l\} \in E$ such that the vectors $\{u(x_1), u(x_2), \ldots, u(x_{l-1}), u(x_l)\}$ are linearly independent, where, for any $x \in E$, $u = u(x)$ is the $l \times 1$ column vector with the entry $u_i = f_i(x)$.*

Proof. We apply Gaussian elimination with row pivoting (see, for example, [DB74]) to the set of all $1 \times l$ row vectors $u(x)^t$ for all $x \in S$, while ensuring that all pivot vectors are non-zero via appropriate row-pivoting. □

Lemma 3.6. *Let E be a compact space, l and m be positive integers with $l \leq m$, and $\{f_1, f_2, \ldots, f_{m-1}, f_m\}$ be continuous complex-valued functions on E, such that at most l of $\{f_1, f_2, \ldots, f_{m-1}, f_m\}$ are linearly independent. Then, there exist l points $\{x_1, x_2, \ldots, x_{l-1}, x_l\} \in E$ and l functions $\{g_1, g_2, \ldots, g_{l-1}, g_l\}$ on E such that*

$$|g_i(x)| \leq 1 \tag{3.1}$$

for any $x \in E$ and all $1 \leq i \leq l$. Furthermore, any complex linear combination f of the functions $\{f_1, f_2, \ldots, f_{m-1}, f_m\}$ may be written:

$$f(x) = \sum_{i=1}^{i=m} f(x_i) g_i(x) \tag{3.2}$$

for any $x \in E$.

Proof. Without loss of generality, we may assume that $\{f_1, f_2, \ldots, f_{m-1}, f_m\}$ are linearly independent (i.e. that $l = m$). Choose the points $\{x_1, x_2, \ldots, x_{l-1}, x_l\}$ in E by max-

imizing the function $D(x_1, x_2, \ldots, x_{l-1}, x_l) = |det(\boldsymbol{A}(x_1, x_2, \ldots, x_{l-1}, x_l))|$. Where $\boldsymbol{A} = \boldsymbol{A}(x_1, x_2, \ldots, x_{l-1}, x_l))$ is the $l \times m$ matrix with entries:

$$A_{j,k} = f_j(x_k) \tag{3.3}$$

for $j, k = 1, 2, \ldots, l-1, l$. Since E is compact and D is continuous, there exist l points $\{x_1, x_2, \ldots, x_{l-1}, x_l\}$ in E where D attains its maximum. If we define

$$B = \sup_{x_1,\ldots,x_l \in E} D(x_1, x_2, \ldots, x_{l-1}, x_l) \tag{3.4}$$

Hence there exist points $\{x_1, x_2, \ldots, x_{l-1}, x_l\} \in E$ such that

$$D(x_1, x_2, \ldots, x_{l-1}, x_l) = B \tag{3.5}$$

Furthermore $B > 0$ by lemma 3.5.

Define functions $\{g_1, g_2, \ldots, g_{m-1}, g_m\}$ on E by the formula:

$$g_i(x) = \frac{det(\boldsymbol{A}(x_1, x_2, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_{l-1}, x_l))}{det(\boldsymbol{A}(x_1, x_2, \ldots, x_{l-1}, x_l))} \tag{3.6}$$

We then obtain (3.2) by applying (3.5) and the Cramer rule to the linear system

$$\boldsymbol{A}\boldsymbol{v} = \boldsymbol{u} \tag{3.7}$$

where $\boldsymbol{v} = \boldsymbol{v}(x)$ is the $l \times 1$ column vector with entries

$$v_i = g_i(x) \tag{3.8}$$

and $\boldsymbol{u} = \boldsymbol{u}(x)$ is the $l \times 1$ column vector with entries

$$u_i = f_i(x) \tag{3.9}$$

Finally, it follows from (3.4) that

$$D(x_1, x_2, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_{l-1}, x_l)) \leq B \tag{3.10}$$

for any $x \in E$. Combining (3.3),(3.5),(3.6),and (3.10) yields (3.1). □

The procedure described in lemma 3.6 is numerically stable due to (3.1).

An immediate corallary of 3.6 is that if $f_1(a_1, \ldots, a_l), \ldots, f_m(a_1, \ldots, a_l)$ are multivariate polynomials with integer coefficients, then there is a nonzero polynomial $g(x_0, \ldots, x_l)$ with integer coefficients such that $g(f_1(a1, \ldots, a,), \ldots, f_m(a_1, \ldots, a_l) = 0$ for all real $a_1, \ldots, a_l$. Hence any polynomial chain with l parameters has at most l degrees of freedom.

3.3 Programing and Partitioning with Primitives

As noted in chapter 1 most scientific computing problems require some form of irregularity. Irregularity can take many forms: different input elements may take different lengths of time to process, generate different numbers of outputs or different communication patterns, etc. At the same time, coherency enables amortization: it is more efficient to make one scheduling decision for one thousand items than one for each; to use Throughput-oriented programming execution to process a batch of items at a time; or to group work by access pattern for memory system locality. Finally, many workloads generate significant quantities of temporary/intermediate data during computation that can be consumed incrementally as it appears. Not only does this allow parallelism of production and consumption, it has a powerful effect on locality: the data can be processed without the bandwidth and storage costs of spilling it to buffer in the higher levels of the memory hierarchy.

In our model, each cluster GPU provides a set of generic throughput-oriented programming components that we call primitives. When a program is executed on a client, calls to resource intensive primitives are delegated to the GPUs. The client program either calls primitives directly or uses meta-primitives which combine several primitives of the same type running on different GPUs and present themselves as a single primitive to the client. While each primitive comes with a prepackaged efficient implementation on a GPU, meta-primitives are implemented locally on the client; they distribute computations and perform load-partitioning, coordination and monitoring of the distributed execution.

Throughput-oriented programming models (described above in section 6.5) all emphasize

data-parallelism. They express programs as a sequence (directed acyclic graph or pipeline) of implicitly data-parallel kernels and generally run one kernel at a time, with the scheduler creating and managing enough independent copies of the kernel to utilize the available hardware resources and mask memory latency. Throughput-oriented algorithms initially arose in the context of vector processors such as the early Cray and CM supercomputers [HS86], and were exemplified by programming models/languages such as C* [RS87]. The archetypal current Throughput-oriented programming models are those arising from GPGPU programming (using GPUs for non-graphics general purposes), such as OpenCL and CUDA [ope, cud].

3.3.1 Primitives

In this section we present a basic repository of primitives, which will be used for implementing the Barnes-Hut case study, using a functional notation:

- Map: Apply a unary function f to all elements of a list:

$$map(f, [x_1, ..., x_n]) = [f(x_1), ..., f(x_n)] \tag{3.11}$$

- Sort: Sort all elements of an input list according to a given order $\prec$:

$$sort(\prec, [x_1, \ldots, x_n]) = [x_{i1}, \ldots, x_{in}] \tag{3.12}$$

  where $x_{ij} \prec x_{ik}$ for all $j < k$.

- Reduce: Compute the sum of a list using a binary associative operator $\oplus$:

$$reduce(\oplus, [x_1, ..., x_n]) = [x_1 \oplus x2 \oplus x3 \oplus \cdots \oplus x_n] \tag{3.13}$$

In addition to these data-parallel primitives, the system also provides personalised all-to-all communication facilities to transfer data directly between GPUs.

Our primitive-based cluster programming system shown in Fig. 1 is implemented on top of OpenCL , mostly for reasons of portability. Primitives are offered as OpenCL device

kernels, implemented in an architecture-specific way on different GPUs. All primitives operate on single objects or arrays which can be distributed blockwise over several GPUs. For each primitive, the system provides an interface which is implemented both by client-based meta-primitives and on the GPUs. To avoid unnecessary remote communication, primitives work as much as possible with remote references, which are pointers to the actual data on the GPUs. Primitives are executed asynchronously, returning such a remote reference immediately when called. This remote reference can be passed to the next GPU, while the first GPU is still executing. Upon completion of the first primitive, the actual data is sent directly to the second GPU.

On a heterogeneous GPU cluster, it is critical to distribute data and computations efficiently across the potentially heterogeneous hosts. In our approach, we integrate load-partitioning policies into primitive and meta-primitive implementations, thereby hiding the complexity of load-partitioning from the application programmer. Load-partitioning is realised on two levels in the system:

- Each primitive running on a single parallel GPU balances the load between the processors of the GPU.
- The meta-primitive is responsible for distributing data and balancing work between different GPUs. On both levels, primitive-specific load-partitioning strategies are used, which distribute data based on the primitive structure. On the GPU level, implementation and architectural aspects can also be taken into account.

3.3.2 Barnes-Hut Primitives

The first step in our high-level programming approach is to express the desired application using the primitives contained in the repository of 3.3.1. First, the particle vector is partitioned into p segments which are distributed among the p GPUs taking part in the computation. Then, the following stages are performed iteratively:

1. `bb = reduce(boundaries, particles); //bounding box`

2. `map(PHIndex, particles); //Peano-Hilbert index`

3. `sort(LessEqual, particles); //sorting`

4. `tree = reduce(treebuild(bb), particles); //treebuild`

5. `map(interact(tree), particles); //interaction`

6. `map(update, particles); //update`

Each stage is described as follows:

1. Calculation of the total boundary of the simulation space: In order to build the tree, the size of the simulation universe is calculated by the reduce operation in line 1. The function boundaries compares two particles or a particle and a bounding box, and returns the new bounding box.

2. Sorting: Sorting involves two steps: (1) computing an index for each particle according to the Peano-Hilbert order (using the map primitive in line 2), and (2) sorting the particles vector in ascending order using this index (sort in line 3).

3. Building the quadtrees: For each remote cluster host, the local quadtree is built in a bottom-up fashion by combining neighbouring particles into trees of depth one. Neighbouring trees are combined into larger trees until a single tree is formed, using a merging algorithm similar to the one described in [SHT95] . Line 4 of the code expresses this process as a reduction primitive, using the tree merging operation as a parameter, where bb is the bounding box computed in the first step.

4. Force computation: The particle interactions are computed using the map-primitive as depicted in line 5. For each particle in particles, the operation interact traverses the quadtree and adds the force effects of the current node to the velocity vector of the particle if $boxsize/distance < \theta$. If this criterion is not yet met, the four child nodes are processed recursively.

5. Particle update: Line 6 uses a map invocation to update the current particle position according to the new velocity vector. For each particle, the unary function update adds the velocity, multiplied by the length of time for each iteration, to the current particles position.

The six-line primitive code of the BH algorithm has a clear structure, where the details of the parallelisation are hidden in the primitive implementations. The application programmer is, therefore, liberated from low-level considerations.

3.3.3 Barnes-Hut on the GPU cluster

The critical problem when bringing the BH algorithm on to the cluster is that it is infeasible to make the entire particle tree available on every host by replication: this would require each host to send the updated values for its own particles to all other hosts after each iteration, thus increasing communication time. Therefore, a distribution among several hosts requires a more elaborate algorithm. We adopt a solution similar to [GKS94]: Each host constructs a locally essential tree by requesting all tree nodes from other hosts that are relevant to the force computations for its own local particles. The goal is to minimise information exchange between nodes during the interaction phase, by propagating the necessary tree information to all nodes once in advance. Each host executes three steps:

1. Send a description of the sub-space with local particles to all other hosts.

2. For each sub-space received from another host, traverse the local tree recursively and add all nodes matching the $boxsize/distance < \theta$ criterion to the result vector. Send the result vectors to all other hosts.

3. Build the locally essential tree by incorporating the particles and tree nodes received from other hosts.

The primitive pseudocode for building the locally essential tree is as follows:

For the amount of data communicated between hosts in the essential tree building phase, there is a trade-off between the accuracy of space approximation sent to other GPUs and the remote tree-nodes received in reply. Our approach (whose details we omit due to the lack of space) is to describe local sub-spaces by several boxes of varying sizes, where the number of boxes used for the sub-space approximation can be adjusted before the start of a simulation

```
for each host:
    host.localtree = host.reduce(treebuild, host.particles);
    host.subspace = host.exec(constructSubspace, host.particles);
otherSubspaces = allToAll(subspaces);
for each host:
    relevantParticles = host.map(selectParticles(host.localtree),
host.otherSubspaces);
alltrees = personalisedAllToAll(relevantParticles);
tree = reduce(treebuild, alltrees);
```

run.

3.4 Load Partitioning

Our approach is to aid the application programmer in the task of load-partitioning by providing generic load-partitioning strategies for primitives, which can be adapted to the application. In our cluster programming system, each single-GPU primitive implementation is responsible for distributing work equally among all processors of the GPU, and meta-primitives distribute work among all participating GPUs. Load-partitioning within a single GPU depends on the GPUs hardware and is considered a black-box from the viewpoint of the application and the meta-primitive.

Load-partitioning at the meta-primitive level is done done in two steps before the primitive execution is started:

1. A primitive specific load-partitioning function computes an optimal distribution of data according to a predefined load-partitioning strategy.

2. A generic redistribution method is responsible for the actual communication required to distribute the input data according to the new distribution.

In general, load-partitioning for clusters considers two factors:

1. The amount of work required for an input element (application-specific).

2. The amount of work that a host can process per second (host-specific).
Therefore, our load-balancing functions have both application- and hardware-specific parameters.

As an example, we will discuss the load-partitioning strategies implemented for the map primitive (both single-GPU and meta-primitive), using this approach.

3.4.1 Partitioning for the Map primitive

In order to balance the work among processors, it is necessary to assess the load generated by each element of the input list in advance, i.e. we require a load-prediction function. For our BH implementation we use the work necessary for the previous iteration as an estimate for the work of the next iteration.

Single-GPU Map primitive. On a single GPU with homogeneous processors, the map primitive implementation has to distribute the load equally among all processors. This is done in two steps: in the first step, for each element, the partial sum of the work required for all elements up to the current one is computed (accumulated load). The total work (i.e., the value computed for the last element) is divided by the number of processors to obtain the amount of work to be assigned to each processor (processor share). Then, the elements of the input list are assigned to processors in a blockwise fashion, assigning elements to one processor until the total work required for completing the share assigned to the processor equals the processor share computed in the first step. Thus the first data element assigned to processor p is the first element for which the partial sum of work exceeds the share that should be assigned to processor $p-1$.

Map Meta-primitive. For computing an efficient load distribution between several cluster hosts, we take into account the heterogeneity of the hosts. We introduce a performance factor c_p for each host p, which is proportional to the number of load-units that host p can process per second. The amount of work w_p allocated to GPU p, is proportional to the performance factor of that GPU and computed as follows: $w_p = c_p/\Sigma_i c_i$. The load is then

distributed according to the computed distribution w_p using the same method as for the single-GPU case.

3.4.2 Partitioning for Barnes-Hut

The particle interaction phase which computes force interactions for each particle is the most time consuming phase of the BH algorithm: it accounts for well over 90% of the overall runtime. Compared to the interaction phase, all other phases account for only very little time, so that the overhead for balancing load for these phases can be expected to outweigh the gain in performance. Additionally, the locally essential particle-trees constructed for a particular host in the tree-build phase are specific to the particles on that host. Thus, the particles assigned to a particular host need to be the same in the tree-build and the particle interaction phase, impeding load balancing for the tree-build phase. Therefore we only do load-partitioning for this phase, which is expressed as a map primitive. Because the distribution of the particles must not be changed between the tree building and the interaction phase, redistribution of the particles is inserted between lines 3 and 4 in the algorithm above.

We use the map primitives load-partitioning function presented in the previous sections, which requires two parameters: a load predictor which estimates the amount of work necessary for each particle, and a performance factor for each cluster host (amount of work per second). For the performance factors, we use the factors obtained a-posteriori from the previous primitive iteration. For the first iteration, the performance factor of each host is set to the number of processors available on that host. The load predictor for the BH algorithm returns for each particle the number of interactions recorded for that particle during the last interaction phase. For the first iteration, the load predictor returns 1 for each particle, assuming the same amount of work for all particles.

Note that the initial performance factors and load predictions are not very accurate and may lead to a considerable load imbalance. However, load is quickly balanced for later

iterations, as demonstrated in our experiments.

3.5 Implementation and Results

We measured the performance of our skeletal Barnes-Hut implementation using three shared-memory GPUs at the NCAR CISL Laboratory. Processors 1 and 2 are components of the Tesla machine described in section 6.1. Processor 3 is a separate Tesla K20 workstation within the lab. The K20 architecture consists of 2496 cores with a peak single precision floating point performance 3.52 Tflops, and a memory bandwidth 208 GB/sec [K20]. The client is a standard workstation also within the lab with a dual-core, 2.6 GHz processor. All experiments were done using Nvdia's CUDA SDK. Client and GPUs are connected by two LANs within the lab, the measured bandwidth of which was approximately 3.2MB/s at the time of testing.

To evaluate the load-partitioning strategy used for map, we measured the performance for 10^4 particles with ($\theta = 0.25$). The runtimes for each of 15 iterations are shown in Figure 3.5(a), and the partition of the particles between the hosts is shown in Figure 3.5(b). For iterations 5 through 8, we have artificially decreased the performance of Processor 2 (starting other time-consuming applications on that host). The figure shows that the work is rebalanced in iteration 6, due to the decreased performance of Processor 2. The performance decrease seen in Figure 3.5a at iteration 5 is thus compensated in the following iterations. After Processor 2 is fully available again in iteration 9, work is rebalanced for iteration 10, obtaining the same performance as in the first iterations. This shows that the implemented load-partitioning mechanisms are able to adapt to varying performance on different hosts.

3.6 Conclusions

We have shown that high-level throughput-oriented abstractions allow to implement relatively complex applications, such as the Barnes-Hut algorithm, while hiding the complex-

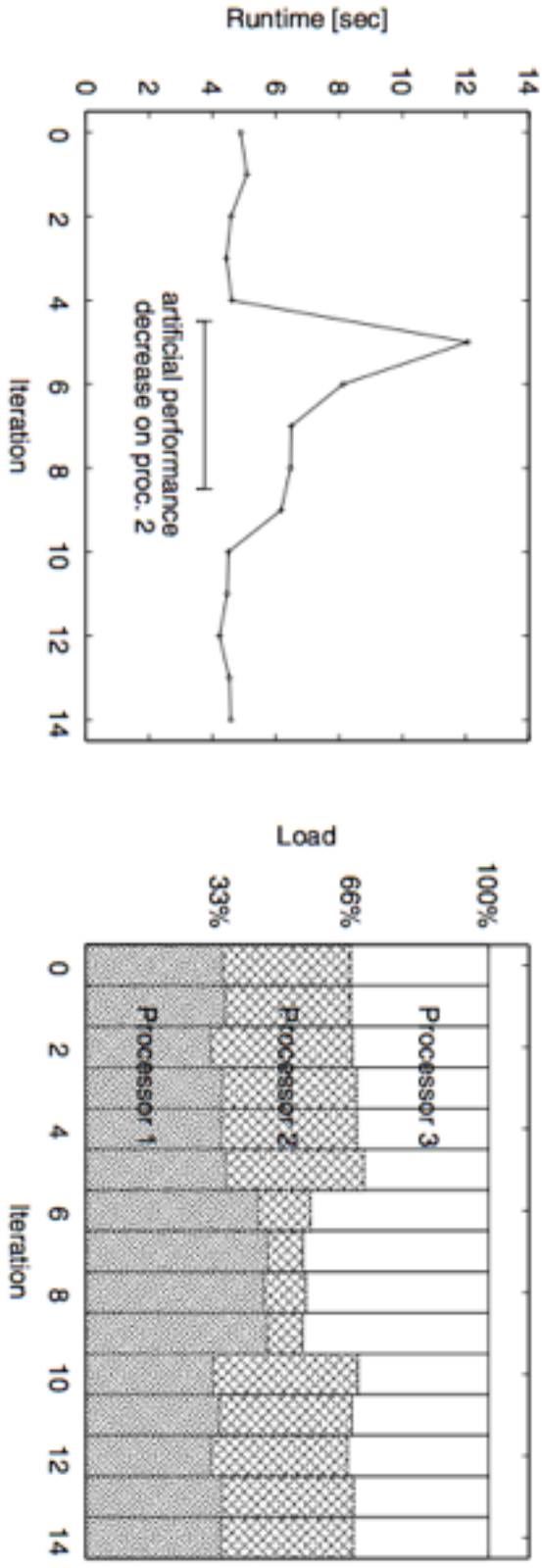

Figure 3.5: load-partitioning on three processors with varying load.

ity of throughput-oriented parallelisation from the application programmer. The numerous previous BH implementations (e.g. [GKS94, SHT95, SLW01]), mostly targeted parallel or homogeneous distributed architectures, while our implementation aims to be executed in heterogeneous clustered systems. Our experiments demonstrated good performance both on one GPU and across several GPUs. Load-partitioning strategies for heterogeneous clusters have been widely studied, and the strategy we used for the map primitive is similar to [KRW96] and [CQ93]. We have shown how to abstract these strategies using a functional paradigm, achieving good performance in a dynamic cluster setting.

CHAPTER 4

Representations for Hierarchical Methods - II

In this chapter we study the memory complexity characteristics of the Barnes-Hut method [BH86], and Greengard and Rokhlins Fast Multipole Method (FMM) [Rok85, GR87] in the general d-dimensional case. These two important methods are best known in the context of classical N-body problems, such as those in astrophysics and electrostatics. However, they have also been extended by Rokhlin [JMR94, GR02] to efficiently perform tensor operations associated with the solution of boundary integral equations in high dimensions. We consider them here in this more general context so that we may apply tools from section 1.2 to our analysis and implementations.

There have been numerous attempts to implement hierachical operator compression algorithms on parallel machines. However, there has been little work comparing throughput-oriented implementations of the Barnes-Hut and FMM algorithms from a practical standpoint in terms of both error and memory bandwidth usage. In section 4.8, we addresses the issue in the following ways. Firstly, we compare basic performance benchmarks for the two algorithms. Secondly, we consider both electrostatic and gravitational distributions of data, and show that the algorithms have quite different characteristics with the two types of forces. Thirdly, we verify the theoretical results above comparing our formulae to experimentally measured values for the expected number of source-target interactions and memory transactions for uniform particle distributions.

In the following two sections we discuss the background and various applications of this representation, notably solutions to common boundary value problems involving separable integral kernels. We then present the detailed memory complexity analysis required to obtain

effective memory throughput-oriented performance for the Barnes-Hut method and the Fast Multipole Method.

4.1 Background

As noted in section 1.2, there are many important operators in the computational sciences that admit product representations. Examples include well-known 'natively' factorizable operators such as the d dimensional Discrete Fourier and the Discrete Wavelet Transforms (discussed in chapter 2), as well as non-factorizable operators which are well-approximated (either analytically or numerically) by sums of products. Examples of this latter sort include the Green's functions discussed in section 4.2 below for the Helmholtz and Maxwell equations at sufficiently low frequencies [BM02], as well as with the heat equation and the Laplace equation.

In cases where the operator does not permit a global approximation by products or sums of products, a profitable strategy has been to form hierarchical sums of local product approximations. There are several well-known approaches of this type that work with different classes of integral operators. Examples include the Fast Multipole method of Greengard and Rokhlin [Rok85, GR87, JMR94, GR02], the mosaic-skeleton approximation method of Tyrtyshnikov [GTZ97], the panel-clustering method [HN89, Hac91], and the Hierarchical matrix approach [Hac99, HK00] of Hackbusch. All of these methods are based upon the idea of splitting discrete matrices generated by the integral operator into a set of blocks with small rank. Each block is then decomposed into either numerical factors by Singular Value Decomposition-like approximations, or analytical factors of the form (1.1) by special functions (e.g. multipole expansions in the fast multipole method).

Several works have proposed extensions to tensors for two and three dimensional problems [Ibr02, HKT05]; however they do not treat arbitrary dimension d and furthermore rely on the Kronecker product representation, which is unsuitable for large problems in higher dimensions due to its data redundancy properties (see theorem 1.1). Other throughput-oriented

algorithms research has thus far focused solely on utilizing hierarchical decompositions for classical N-body-type problems, such as accelerating particle simulation [GD08], radiance calculations [PDC03], N-body force calculations [NHP07], and game physics [PDC03].

4.2 Volume and Boundary Integral Equations

In this section we present applications for the case of n-body systems and high-dimensional partial differential equations (PDEs). In a typical application, the tensor representing the solution, X, arises from the discretization of a high-dimensional or parameter-dependent partial differential equation and is only given implicitly as the solution to a typically large linear system or eigenvalue problem. Consider for example a linear operator A mapping a tensor in $\mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ to a tensor in $\mathbb{R}^{m_1 \times m_2 \times \cdots \times m_d}$. The aim is to use a low-rank approximation of a tensor X defined implicitly through a linear system or eigenvalue problem

$$A(X) = B, \tag{4.1}$$

where B has low rank, or

$$A(X) = \lambda X, \tag{4.2}$$

with $X \neq 0$. Typically, the linear operator A results from the discretization of a PDE on a tensor grid, and has a Kronecker product representation:

$$vec(A(X)) = \sum_{j=1}^{s} (A_d^j \otimes \cdots \otimes A_1^j) vec(X), \tag{4.3}$$

with $A_k^j \in \mathbb{R}^{k \times k}$ and a small number of summands s.

In the following subsections we reformulate several common differential equations as integral equations, in order to facilitate their solution by means of the methods described in this chapter. Many of the partial differential equations encountered in physics and in mathematics reduce to the equations of the type considered here, namely the Helmholtz equation and its variants.

4.2.1 Volume integral equation formulation of the Helmholtz equation

Given a complex number k, a positive integer d, and a function V on $\mathbb{R}^d$, we would like to solve the following pair of equations for ψ_{scat}:

$$(\nabla^2 + k^2 \boldsymbol{I})\psi_{tot} = V\psi_{tot} \tag{4.4}$$

and

$$\psi_{tot} = \psi_{in} + \psi_{scat}, \tag{4.5}$$

where ψ_{tot}, ψ_{in} and ψ_{scat} are functions on $\mathbb{R}^d$. In (4.3) and (4.5) , we assume that ψ_{in} in satisfies an analogue of (4.3) in which V is identically zero, namely,

$$(\nabla^2 + k^2 \boldsymbol{I})\psi_{in} = 0. \tag{4.6}$$

Formula (4.3) is known as the Helmholtz equation (or time-harmonic wave equation) in the presence of the potential V ; ψ_{scat} is known as the scattered field, and ψ_{in} is known as the incident or incoming field (which satisfies an analogue (4.4) of (4.3) with an identically zero potential).

Inserting (4.5) into (4.3), and then using (4.4), we obtain that

$$(\nabla^2 + k^2 \boldsymbol{I})\psi_{scat} = V\psi_{in} + V\psi_{scat} \tag{4.7}$$

Applying the operator $(\nabla^2 + k^2 \boldsymbol{I})^{-1}$ to both sides of (4.5), we obtain the Rayleigh-Lippmann-Schwinger integral equation

$$\psi_{scat} = (\nabla^2 + k^2 \boldsymbol{I})^{-1} V\psi_{in} + (\nabla^2 + k^2 \boldsymbol{I})^{-1} V\psi_{scat}, \tag{4.8}$$

which represents ψ_{scat} in terms of a distorted version of itself, plus a distorted version of ψ_{in}. In the following section, we will express $(\nabla^2 + k^2 \boldsymbol{I})^{-1}$ explicitly as an integral operator.

4.2.2 Green functions

In accordance with the general definition of a Green function, for any positive integer d, and complex number k with a nonnegative imaginary part, the Green function G_k for $\nabla^2 + k^2\boldsymbol{I}$ is the function on $\mathbb{R}^d \times \mathbb{R}^d$ such that

$$(\nabla^2 + k^2\boldsymbol{I})G_k(x, y) = \delta(x, y) \tag{4.9}$$

for all $x, y \in \mathbb{R}^d$, where δ is the Dirac delta and $\nabla^2 + k^2\boldsymbol{I}$ acts with respect to x. In $\mathbb{R}^2$ for example, the solution is of the form

$$G_k(x, y) = \begin{cases} \frac{1}{2\pi} \ln|x - y| & , \quad k = 0 \\ \frac{-i}{4} H_0^{(1)}|x - y| & , \quad \text{k}\neq 0 \end{cases} \tag{4.10}$$

for any $x, y \in \mathbb{R}^2$ with $x \neq y$, where $H_0^{(1)}$ is the Hankel function of the first kind of order 0. In $\mathbb{R}^3$ the solution is of the form

$$G_k(x, y) = \begin{cases} \frac{-1}{4\pi|x-y|} & , \quad k = 0 \\ \frac{-exp(ik|x-y|)}{4\pi|x-y|} & , \quad \text{k}\neq 0 \end{cases} \tag{4.11}$$

for any $x, y \in \mathbb{R}^3$ with $x \neq y$.

For any positive integer d, and complex number k with a nonnegative imaginary part, applying $\nabla^2 + k^2\boldsymbol{I}$ to the expressions on both sides of the following identity and using (4.9) verifies that

$$(\nabla^2 + k^2\boldsymbol{I})^{-1} f(x) = \int_{\mathbb{R}^d} G_k(x, y) f(y) dy \tag{4.12}$$

for any $x \in \mathbb{R}^d$, and any complex-valued function f on $\mathbb{R}^d$.

4.2.3 Time-dependent wave equation

The Helmholtz equation is also closely related to the time-dependent wave equation. Given a positive integer d, a positive real number c, and a function s on d such that $0 < s(x) \leq c$

for any $x \in \mathbb{R}^d$, we would like to solve the following equation for the function $\psi = \psi(x,t)$ on $\mathbb{R}^d \times \mathbb{R}$:

$$\frac{\partial^2}{\partial t^2}\psi = s^2 \nabla^2 \psi \tag{4.13}$$

Formula (4.13) is known as the time-dependent wave equation; the number c is known as the speed of wave propagation in the ambient space, and the function s is known as the local speed of wave propagation. To solve the PDE, we choose an appropriate nonzero complex number k with a nonnegative imaginary part, and consider the ansatz

$$\psi(x,t) = \psi_k(x) e^{ikct} \tag{4.14}$$

for any $x \in \mathbb{R}^d$, and $t \in \mathbb{R}$, where $\psi_k(x)$ is a function on $\mathbb{R}^d$.

Combining (4.13) and (4.14), we obtain the Helmholtz equation

$$(\nabla^2 + k^2 \boldsymbol{I})\psi_k = V_k \psi_k, \tag{4.15}$$

where V_k is the function defined on $\mathbb{R}^d$ via the formula

$$V_k(x) = k^2 \left(\boldsymbol{I} - \frac{c^2}{s(x)^2} \right). \tag{4.16}$$

Note that combining (4.16) and the fact that $s(x)c$ for any $x \in \mathbb{R}^d$ yields that $V_k(x) \leq 0$ for any $x \in \mathbb{R}^d$. This latter fact guarantees that (4.14) does not have any resonances (i.e., bound-states). It is also typical in applications that $s(x) = c$ in free space, that is, outside of objects that scatter the wave field. Therefore, $V_k(x)$ defined in (4.16) vanishes outside of objects that scatter the wave field.

4.3 Approximate Tensor Rank

We now describe in detail the hierarchical compression of an integral kernel $G(x,y)$ with $(x,y) \in \mathbb{R}^d \times \mathbb{R}^d$. The bottom level of our hierarchy consists of a single pair of source and

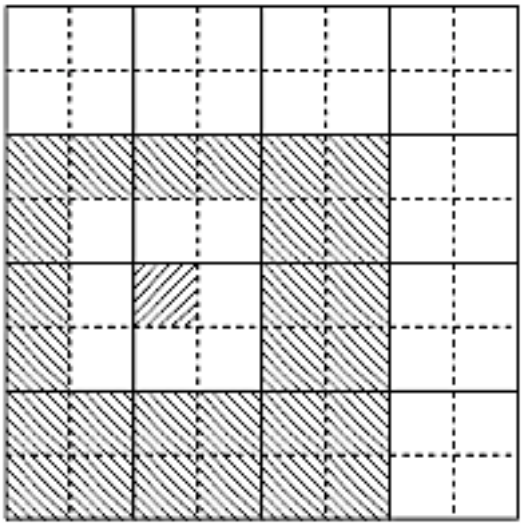

Figure 4.1: The shaded squares that surround the isolated one are all the source squares that are of the same size as the isolated square that do not touch it, but that do lie in the source squares of the same size as the parent of the isolated square which touch the parent.

target bricks, which we take to be compact rectangular prisms in $\mathbb{R}^d$ containing either source or target points. We would like to compress a kernel G mapping from a source brick P to a target brick Q.

For any brick C in $\mathbb{R}^d$, we define S_C to be the set of all points in P that lie outside of both C and the bricks of the same size as C that just touch C. For any brick C in $\mathbb{R}^d$, we define T_C to be the set of all points in Q that lie outside of both C and the bricks of the same size as C that just touch C. Figure 4.3 shows an example in $\mathbb{R}^1$.

For any function G on the Cartesian product $P \times Q$ of P and Q, and any positive real number ϵ ,we say that G has a tensor rank(with respect to d, P, Q and ϵ)of k to mean that G has the following two properties:

- For any source brick C, any set of strictly more than k of the functions f_y, with y ranging through T_C, is linearly dependent to within precision ϵ, where f_y is defined on the set of points that belong both to P and to C via the formula

$$f_y(x) = G(x, y). \tag{4.17}$$

- For any target/test brick C, any set of strictly more than than k of the functions e_x, with x ranging through S_C, is linearly dependent to within precision ϵ , where e_x is defined on the set of points that belong both to Q and to C via the formula

$$e_x(y) = G(x, y). \tag{4.18}$$

4.4 Single-Level Representation

In this section, we discuss a single-level algorithm for applying a discretized tensor rank-k kernel efficiently when every point in P is separated from all points in Q by at least the diameter of P.

Given a kernel G on $P \times Q$ and an initial particle strength function ρ on P , we would like to compute the following sum efficiently, for every $y \in Q$:

$$\sum_{x \in P} \rho(x) G(x, y) \tag{4.19}$$

We assume throughout that G has a tensor rank of k. We use 3.6 to construct a subset U of at most k points from P, and at most k functions g_z, with z ranging through U, such that

$$f_y(x) = \sum_{z \in U} f_y(z) g_z(x) \tag{4.20}$$

for any $x \in P$, and any $y \in Q$, where f_y is the function defined on P via the formula

$$f_y(x) = G(x, y). \tag{4.21}$$

It follows from (4.20) that

$$\sum_{x \in P} \rho(x) f_y(x) \approx \sum_{z \in U} \rho_P(z) f_y(z) \tag{4.22}$$

for any $y \in Q$, where ρ_P is the function defined on U via the formula

$$\rho_P(z) = \sum_{x \in P} \rho(x) g_z(x). \tag{4.23}$$

Therefore for any $y \in Q$, we may approximate the sum

$$\sum_{x \in P} \rho(x) f_y(x) \tag{4.24}$$

with the sum

$$\sum_{z \in U} \rho_P(z) f_y(z) \tag{4.25}$$

for any $y \in Q$.

Suppose that m and n are positive integers such that there are m points in P and n points in Q. Since there are m points in P, and at most k points in U, it follows that evaluating (4.23) for all points in U costs $\mathcal{O}(km)$. Since there are n points in Q, and at most k points in U, it follows that, once we have evaluated (4.23) , evaluating (4.25) for all points in Q costs $\mathcal{O}(kn)$. Overall, then, obtaining an approximation to (4.24) via (4.22) , by evaluating (4.23) and (4.25) , costs $\mathcal{O}(k(m+n))$. In contrast, there are m points in P and n points in Q, so that evaluating (4.24) directly for all $y \in Q$ costs $\mathcal{O}(mn)$. When m and n are much larger than k, clearly $k(m+n)$ is much less than mn, so that evaluating (4.24) directly costs far more than obtaining an approximation to it via (4.22) .

Rather than requiring that every point in P be separated from all points in Q by at least the diameter of P , we can instead require that every point in Q be separated from all points in P by at least the diameter of Q. In the latter case, a construction analogous to that used in the present section yields a similarly fast algorithm for approximating (4.24).

For example, the kernel associated with the gravitational potential generated by the large collection of point particles that constitute the moon, as acting on the large collection of point particles that constitute the Earth, the algorithm of the present section simply takes advantage of the observation that only a few parameters are needed to accurately describe the gravitational effects of the moon upon the Earth.

4.5 Multi-Level Representation

In the remainder of this chapter, we discuss a multilevel algorithm for applying a discretized tensor rank-k kernel efficiently when the sets P and Q are arbitrary (but finite).

For motivation, we consider the case when P and Q both consist of points in $\mathbb{R}$, specifically the numbers $1, 2, \ldots, n-1, n$, where n is some positive integer power of 2. Figure 4.2 depicts the matrix arising from a discretization of a low-tensor rank kernel for these particular sets P and Q. Each of the blocks of matrix entries indicated in 4.2 that does not touch the diagonal has a rank of at most k, according to the definition of tensor rank. The multilevel algorithm efficiently applies the whole matrix to a vector by applying to the vector each of the low-rank blocks, and summing up the results.

We proceed with the construction as follows. First, section 4.5 sets up tree structures. Second, section 4.5.1 organizes P in a tree, associating to the subset of points of P contained in each brick a numerical representation of the field generated by those points in accordance with applying the kernel G; Third, section 4.5.2 organizes Q in a tree, associating to the subset of points of Q contained in each brick a numerical representation of the field detected at those points in accordance with applying the kernel G. Fourth, section 4.6 provides a multilevel algorithm for applying a low-rank kernel rapidly. Finally, section 4.7 counts the memory operations and words of memory required by the algorithm described in section 4.6.

In the remainder of the present section, as well as in section 4.5.1 and section 4.5.2, we set notation and construct explicit low-rank representations for the blocks shown in 4.2. Section 4.6 utilizes this notation and these representations to construct a multilevel algorithm for

applying a low-tensor rank kernel efficiently.

We define the largest source brick to be a brick which contains all of the source points, the points in P. We then form a tree structure by recursively dividing any source brick containing more than m points into 2^d source bricks (which we call children), each with half the sidelength of the parent.

Similarly, we define the largest target/test brick to be a brick which contains all of the target/test points, the points in Q. We then form a tree structure by recursively dividing any target/test brick containing more than m points into 2^d target/test bricks (also called children), each with half the sidelength of the parent. For any target/test brick C, we say that B is a client of C if B is a source brick of the same size as C that does not touch C, but that does lie in a brick of the same size as the parent of C which touches the parent. Figure 4.3 above displays an example with a particular target brick C in $\mathbb{R}^2$, as well as all the clients of C.

4.5.1 Source Tree

We organize $\mathbb{R}^d$ in a tree for P , as in section 4.5. We begin by marking all source bricks as unprocessed. We sweep through all of the source bricks multiple times, processing during every sweep each unprocessed brick whose children have all been processed (on the first sweep, we process each brick which does not have any children). To process a source brick C which does not have any children, we define P_C to be the set of all points that lie in both P and C, i.e.,

$$P_C = P \cup C. \tag{4.26}$$

To process a source brick C whose children have already been processed, we define P_C to be the union of all the sets U_B defined below, with B ranging over all children B of C, i.e.,

$$P_C = U_B, \tag{4.27}$$

where the union is taken over all children of C. After defining P_C, we construct a subset U_C of at most k points from P_C, and at most k functions $g_{C,z}$, with z ranging through U_C, such that

$$\sum_{z \in U_C} f_y(x) \approx f_y(z) g_{C,z}(x) \tag{4.28}$$

for any $x \in P_C$, and any $y \in T_C$, where f_y is the function defined on P_C via the formula

$$f_y(x) = G(x, y). \tag{4.29}$$

For each source brick C, the set U_C contains at most k points and the set P_C contains at most $2^d k$ points.

4.5.2 Target Tree

We now reverse the roles P and Q played in section 4.5.1 by gathering the points in Q according to the bricks to which they belong and organizing $\mathbb{R}^d$ in a tree for Q, as in section 4.5. We begin by marking all target/test bricks as unprocessed. We then sweep through all of the target/test bricks multiple times, processing during every sweep each unprocessed brick whose children have all been processed (on the first sweep, we process each brick which does not have any children).

To process a target/test brick C which does not have any children, we define Q_C to be the set of all points that lie in both Q and C, i.e.,

$$Q_C = Q \cap C. \tag{4.30}$$

To process a target/test brick C whose children have already been processed, we define Q_C to be the union of all the sets VB defined below, with B ranging over all children B of C, i.e.,

$$Q_C = \bigcup_B V_B, \tag{4.31}$$

where the union is taken over all children of C. After defining Q_C , we construct a subset V_C of at most k points from Q_C, and at most k functions $h_{C,z}$, with z ranging through V_C, such that

$$\sum_{z \in V_C} e_x(y) \approx e_x(z) h_{C,z}(y), \tag{4.32}$$

for any $y \in Q_C$, and any $x \in S_C$, where e_x is the function defined on Q_C via the formula

$$e_x(y) = G(x, y). \tag{4.33}$$

For each target/test brick C, the set V_C contains at most k points and the set Q_C contains at most $2^d k$ points.

4.6 Multi-Level Algorithm

In this section, we discuss a multilevel algorithm for applying a discretized tensor rank-k kernel efficiently. First, we discuss the rationale behind the algorithm. We assume throughout that G has a tensor rank of k. The algorithm consists of five steps, described below. We break up the sum that we want to compute as follows:

$$\sum_{x \in P} \rho(x) f_y(x) = \sum_C \sum_{x \in P_C} \rho(x) f_y(x), \tag{4.34}$$

where the first sum in the right-hand side is taken over all source bricks that do not have any children, and where f_y is defined in (4.29) , and P_C is defined in (4.26) . We define the function ϕ_C on U_C (defined in section 4.5.1) via the formula

$$\sum_{x \in P} \rho(x) f_y(x) = \sum_C \sum_{x \in P_C} \rho(x) g_{C,z}(x), \tag{4.35}$$

where P_C is defined in section 4.5.1, and $g_{C,z}$ is defined in (4.28). Combining (4.28), (4.34), and (4.35) then yields that

$$\sum x \in P) f_y(x) \approx \sum_C \sum_{x \in U_C} \rho_C(x) f_y(x), \tag{4.36}$$

where the first sum in the right-hand side is taken over all source bricks that do not have any children, and where f_y is defined in (4.29) , and U_C is defined in section 4.5.1. Thus, knowing the values of ϕ_C defined in (4.35) , we can use (4.36) to reconstruct the sum

$$\sum_{x \in P} \rho(x) f_y(x), \tag{4.37}$$

where f_y is defined in (4.29) . Step 1 involves calculating ϕ_C defined in (4.35) . Step 2 repeats the construction involved in Step 1 recursively. For any target/test brick C, we define the function ϕ_C on V_C (defined in section 4.5.2) via the formula

$$\phi_C(y) = \sum_{x \in S_C} \rho(x) e_x(y), \tag{4.38}$$

where x is defined in (4.33) . For any target/test brick C that does not have any children, we obtain from (4.38) that

$$\sum_{x \in P} \phi(x) e_x(y) \approx \phi_C(y) + \sum_{x \in P S_C} \rho(x) e_x(y), \tag{4.39}$$

for any $y \in Q_C$, where again e_x is defined (4.33) , ϕ_C is defined in (4.38) , and Q_C is defined in (4.30) . Thus, knowing the values of ϕ_C defined in (4.38) , we can use (4.39) to reconstruct the sum

$$\sum_{x \in P} \phi(x) e_x(y), \tag{4.40}$$

where e_x is defined (4.33) . Step 5 uses (4.39) , together with (4.28) and (4.35) . In order to calculate ϕ_C defined in (4.38) efficiently, we do the following. For any tar- get/test brick C, we define the function ψ_C on Q_C (defined in (4.30) and (4.31)) via the formula

$$\psi_C(y) = \sum_B \sum_{x \in U_B} \phi_B(x) G(x, y), \tag{4.41}$$

where the leftmost sum in the right-hand side is taken over all clients of C, and where ϕ_B is defined in (4.35) , and U_B is defined in section 4.5.1. Combining (4.32) , (4.38) , and (4.41) yields that

$$\rho_C(y) \approx \psi_C(y) + \sum_{z \in V_B} \phi_B(z) h_{B,z}(y), \tag{4.42}$$

for any $y \in Q_C$, where B is the parent of C, ϕ_C and ϕ_B are defined in (4.38) , ψ_C is defined in (4.41) , $h_{B,z}$ is from (4.32) , V_B is defined in section 4.5.2, and Q_C is defined in (4.30) and (4.31). Step 3 involves calculating ψ_C defined in (4.41) . Step 4 uses (4.42) to obtain ϕ_C defined in (4.38) .

The following five steps therefore comprise the complete multilevel algorithm. Steps 1 and 2 aggregate source points, Step 3 maps from source points to target/test points, and Step 4 and 5 disaggregate target/test points.

- (1) For each source brick C that has no children, we compute the values of the function ϕ_C defined on U_C (defined in section 4.5.1) via the formula

$$\rho_C(z) = \sum_{x \in P_C} \rho(x) g_{C,z}(x), \tag{4.43}$$

  where $g_{C,z}$ is from (4.28), and P_C is defined in (4.26).

- (2) We start by marking all source bricks that have no children as processed and all other bricks as unprocessed. Then, we sweep through all source bricks multiple times, until all are processed. During every sweep, for each source brick C whose children have all been processed, we compute the value of the function ϕ_C defined on U_C via the formula

$$\rho_C(z) = \sum_B \sum_{x \in U_B} \rho_B(x) g_{C,z}(x), \tag{4.44}$$

where the leftmost sum in the right-hand side of (4.44) is taken over all children of C, and where $g_{C,z}$ is from (4.28) , and U_C and U_B are defined in section 4.5.1. We then mark the brick C as processed.

- (3) For each target/test brick C, we compute the values of the function ψ_C defined on Q_C (defined in (4.30) and (4.31)) via the formula

$$\psi_C(y) = \sum_B \sum_{x \in U_B} \rho_B(x) G(x, y), \tag{4.45}$$

where the leftmost sum in the right-hand side of (4.45) is taken over all clients of C, and where ϕ_B is defined in (4.43) and (4.44) , and U_B is defined in section 4.5.1.

- (4) We start by defining the function ϕ_C on Q_C (defined in (4.30) and (4.31)) for the largest target/test brick C, via the formula

$$\phi_C(y) = \psi_C(y), \tag{4.46}$$

where ψ_C is defined in (4.45) . Then, we mark the largest target/test brick as processed and all other bricks as unprocessed. Finally, we sweep through all target/test bricks multiple times, until all are processed. During every sweep, for each target/test brick C whose parent has been processed, we compute the value of the function ϕ_C defined on Q_C (defined in (4.30) and (4.31)) via the formula

$$\phi_C(y) = \psi_C(y) + \sum_{z \in V_B} \phi_B(z) h_{B,z}(y), \tag{4.47}$$

where B is the parent of C, ψ_C is defined in (4.45) , $\mathrm{h}_{B,z} is from (4.32), and \mathrm{V}_B$ is defined in section 4.5.2

- (5) For each target/test brick C that has no children, we compute the value of the function ϕ_C defined on Q_C (defined in (4.30)) via the formula

$$\Phi_C(y) = \phi_C(y) + \Sigma_1 + \Sigma_2, \tag{4.48}$$

where ϕ_C is defined in (4.46) and (4.47) , and where

$$\Sigma_1 = \sum_B \sum_{x \in U_B} \rho_B(x) G(x, y), \tag{4.49}$$

with the the leftmost sum in the right-hand side of (4.49) being taken over all childless source bricks that do not touch C but that do lie in the bricks of the same size as C that touch C, with ρ_B defined in (4.43), and

$$\Sigma_2 = \sum_B \sum_{x \in P \cap B} \rho(x) G(x, y), \tag{4.50}$$

with the the leftmost sum in the right-hand side of (4.50) being taken over both C itself and all childless source bricks which are not separated from C by at least a brick of their own size.

For an alternative approach using the Singular Value Decomposition in place of interpolation, see Gimbutas and Rokhlin [GR02]. 4.2 displays the bricks in the matrix representation of a linear operator formed by the Cartesian products of the pairs of source and target/test bricks involved in the third step (3), as well as by the pairs of source and target/test bricks involved in the fifth, last step (5). Obviously, the Cartesian products of the pairs of bricks from the third step (3) and fifth step (5) cover the matrix representation exactly once, with no overlap.

4.7 Algorithmic Complexity

In this section, we tabulate the number of memory operations and words of memory required by the algorithm described in section section 4.6.

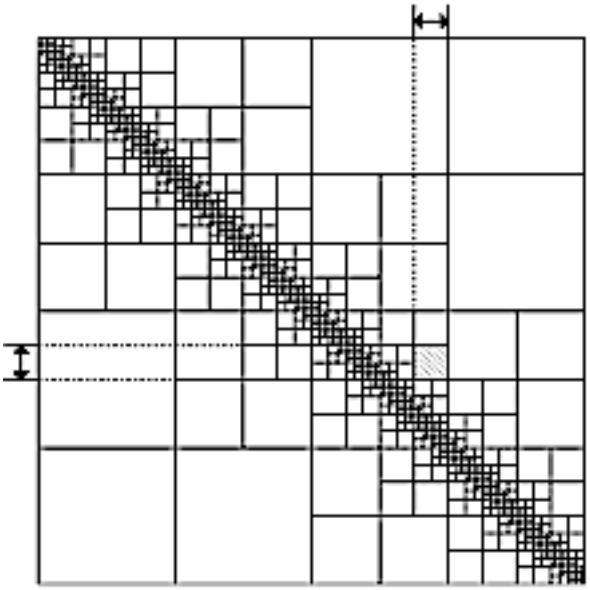

Figure 4.2: Illustration of the matrix decomposition associated with the case in in which both P and Q consist of the numbers $1, 2, \ldots, n-1, n$, for some n that is a positive integer power of 2. The shaded box in the matrix decomposition lies at the intersection of rows corresponding to one possible target/test brick C and columns corresponding to one possible client of C.

First, we estimate the number of source and target bricks. Suppose that there are n source points, i.e., P consists of n points. Then, it follows from the fact that the parent of any source brick that has no children contains at least m points, that there are at most n/m such parent bricks. Therefore, since any such parent brick has at most 2d children, there are at most $2^d n/m$ source bricks that do not have any children.

Note that there are at most $\log_2(1/\epsilon)$ levels in the 2^d-ary tree of source bricks. Combining the facts that there are at most $2^d(n/m)$ source bricks that have no children, and each such brick has at most $\log_2(1/\epsilon)$ parents, yields that there are at most

$$n_P = 2^d(n/m)\log_2(1/\epsilon) \tag{4.51}$$

source bricks in total. Next, suppose that there are n target/test points, i.e., Q consists of n points. Suppose also that there are at most $\log_2(1/\epsilon)$ levels in the $2d$-ary tree of target/test bricks. Then, using (4.29) and (4.33), we have that there are at most

$$n_Q = 2^d(n/m)\log_2(1/\epsilon) \tag{4.52}$$

target/test bricks in total.

If G has a tensor rank of k, then we obtain the following costs for the corresponding steps of the algorithm described in section section 4.6.

- There are at most $2^d n/m$ source bricks that have no children, at most k points in each U_C, and at most m points in each P_C. Thus, Step 1 costs at most $\mathcal{O}(2^d kmn/m)$.
- There at most nP source bricks, at most 2^d children of any brick, and at most k points in any U_C. Thus, Step 2 costs at most $\mathcal{O}(2^d k^2 n_P)$.
- There are at most n_Q target/test bricks, at most $2^d k$ points in each Q_C, at most k points in each U_B, and at most $6^d 3^d$ clients of C. Thus, Step 3 costs at most $\mathcal{O}((6^d 3^d)2^d k^2 n_Q)$.
- There at most n_Q target/test bricks, at most $2^d k$ points in each Q_C, and at most k points in each V_B. Thus, Step 4 costs at most $\mathcal{O}(2^d k^2 n_Q)$.

- For any source brick C that has no children, there are at most 3d bricks of any fixed size at least as large as C, for which C lies in a brick of that same fixed size that touches one of the 3^d bricks. Therefore, since there are at most $2^d(n/m)$ source bricks that have no children, and at most $\log_2(1/\epsilon)$ different sizes of bricks, it follows that there are at most $3^d\ n_P$ source bricks in total involved in the sums (4.49) . Similarly, there are at most $3^d\ n_P$ source bricks in total involved the sums (4.50) . Moreover, there are at most $2^d k$ points in each Q_C, at most k points in each U_B, and at most m points in each $P \cap B$. Thus, Step 5 costs at most $\mathcal{O}(3^d 2^d k(k+m)n_P)$.

Overall, if we take $m = k$, then we find that the whole procedure costs $\mathcal{O}((6^d 3^d)4^d kn \log_2(1/\epsilon))$. Suppressing the dependence on the dimension d, we find that the whole procedure costs

$$\mathcal{O}(kn \log_2(1/\epsilon)). \tag{4.53}$$

In many circumstances, k is proportional to $\log_2(1/\epsilon)$. See Nabors, Korsmeyer, Leighton, and White [NKL94] for a serial algorithm that has similar operation counts without the assumption that there are only $\log_2(1/\epsilon)$ levels in the 2^d-ary trees of source and target/test bricks. Also note that for throughput-oriented architectures in general, it is more efficient to move some of the computations associated with Step 5 above to earlier steps. However, this higher throughput algorithm is somewhat more complicated.

4.8 Implementation and Results

We have implemented two versions of the Barnes-Hut algorithm, one in Euclidean coordinates, and one in spherical coordinates. Both versions can use an arbitrary number of terms in the expansion, increasing the accuracy and the cost of computing the additional terms. For completeness, we have also included a hybrid of Barnes-Hut and FMM algorithms, which recursively subdivides target bricks until they contain no more than m particles (instead of one particle as in the case of the Barnes-Hut algorithm). The particles in the target brick interact directly with particles in all source bricks that are not well separated from it. The

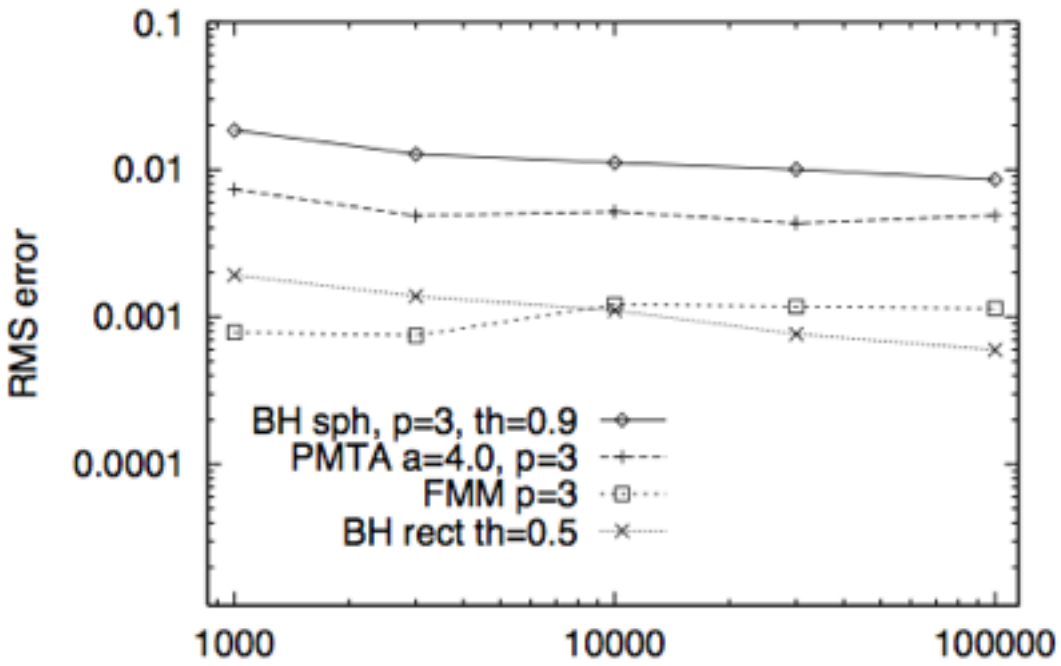

Figure 4.3: Approximation error in computed gravitational forces on up to 10^5 uniformly distributed points. Analogous tests were performed for other paramater values on both graviational and electrostatic forces. In all cases the approximation errors scaled similarly with problem size.

number of terms p in the multipole expansion, and the separation parameter α can both be varied to control accuracy.

Before presenting our memory throughput results, we first verify that the algorithms scale appropriately. Table 4.3 shows the dependence of error with problem size N with fixed values of the accuracy parameters in the algorithms, namely θ in the case of Barnes-Hut, α in the hybrid case, and the number of terms p in the multipole expansion for all three algorithms. The RMS error values shown are empirically derived by comparing the tree versions with a brute force method.

We compared the work performed by the algorithms for two levels of accuracy. For uncharged distributions at a lower accuracy (RMS error on the order of 10^{-5}, where $\theta = 0.55$, $\alpha = 0.8$ and $p = 4$, all three make comparable use of the availabe memory bandwidth, which

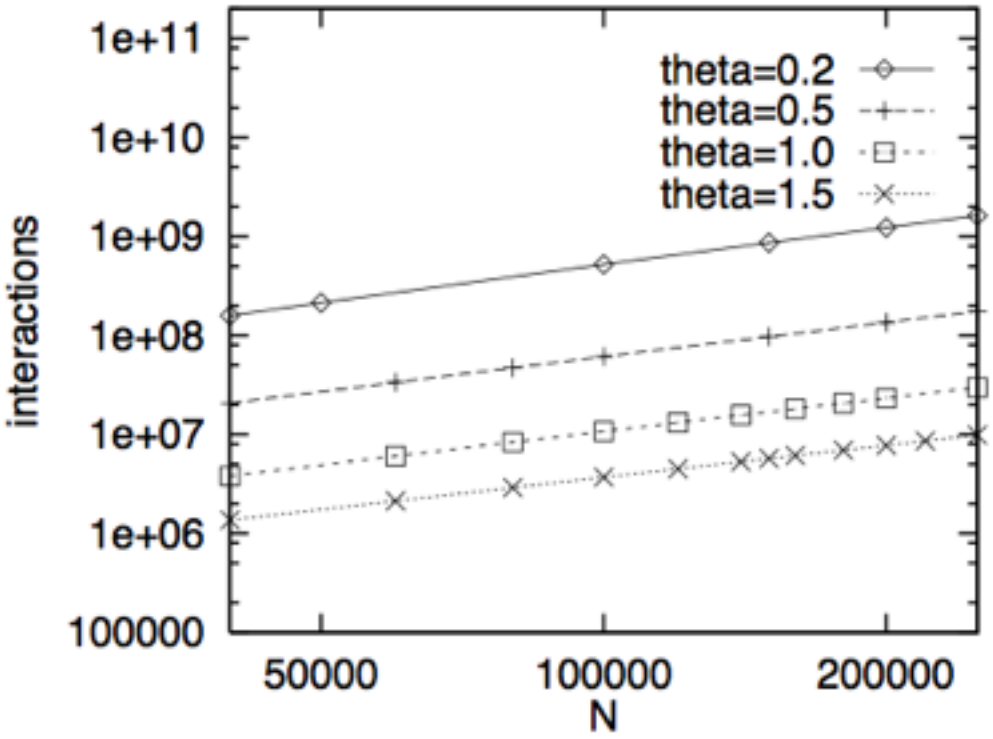

Figure 4.4: Computed number of interactions for several different values of θ

Euclidean Barnes-Hut having the smallest memory footprint for N up to $2*10^8$. This is to be expected, as Euclidean Barnes-Hut is a purely gather-based algorithm and thus reveals the highest amount of data parallelism. Figure 4.5 shows the memory footprint of each algorithm running at low accuracy for chargeless distributions. Figure 4.6 summarizes our results for both levels of accuracy, for both charged (electrostatic) and uncharged (gravitational) kernels of size $N \approx 10^7$. At lower accuracy the Euclidean Barnes-Hut again has the smallest memory footprint for gravitational kernels. For electrostatic kernels however, the FMM and hybrid algorithms both out-perform the Barnes-Hut versions at low accuracy. Finally, at high accuracy the FMM outperforms both of the other algorithms. The Euclidean Barnes-Hut has superior performance for gravitational distributions because the lower order terms dominate. This is not the case for electrostatic distributions, where more than three terms are needed for accurate potential and force computations.

4.9 Conclusions

In this chapter we have outlined a general family of methods for compressing integral kernels in n dimensions, which can be applied to all of the operators mentioned in section 4.2 above. We then profiled four specific algorithms from this family and compared their performance for several different compression parameters and inputs.

All three algorithms reveal substantial data parallelism, with Euclidean Barnes-Hut exposing the most: even relatively small problems ($N \approx 1000$) achieve 45-fold speedups on Cerberus with double-precision arithmetic and 118-fold speedups using single precision arithmetic. However, due to the severe restrictions on memory bandwidth [1], it is advantageous to expose less fine-grained parallelism by using larger target bricks and higher-order mulitpole expansions. Thus the hybrid and FMM algorithms exhibit superior performance for the more complex electrostatic kernel. Barnes-Hut in spherical coordinates does prove to

[1] The Tesla C870 GPU used for these experiments has 76.8 GB/s of throughput to its 1536 MB off-chip global memory. With computational resources supporting nearly 400 Gflops of performance and each FP instruction operating on up to 12 bytes of data, all three algorithms easily saturate that bandwidth. See the appendix 6 for more detailed information.

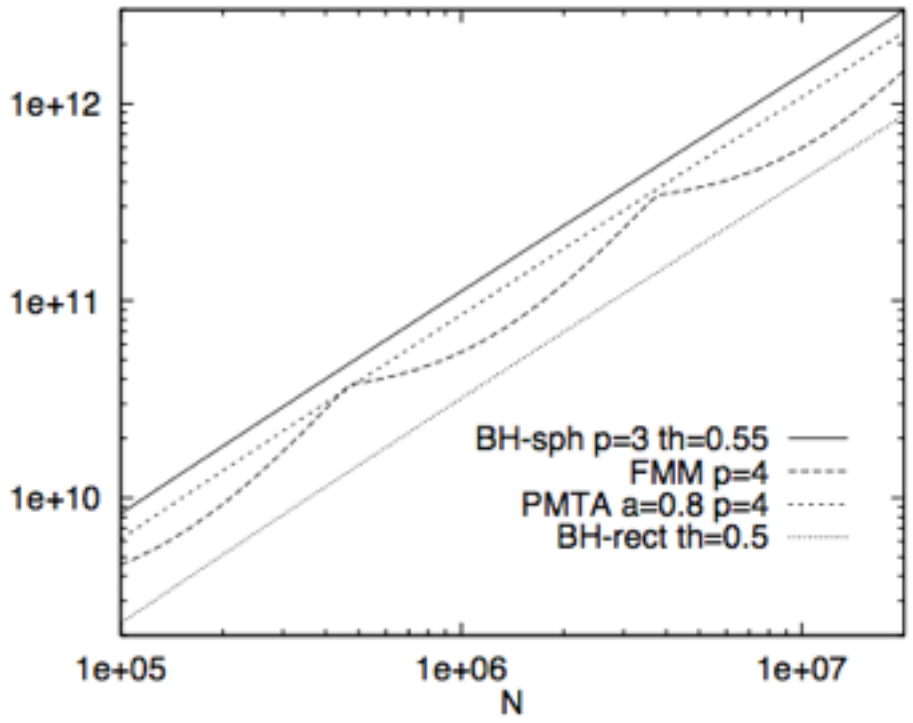

Figure 4.5: Algorithm memory usage for gravitational kernels at low accuracy (RMS error $\approx 10^{-3}$. Y-axis values idicate the total number of bytes read from or written to global memory. The bumps in the FMM curve are due to the changes in optimal octtree depth as N increases.

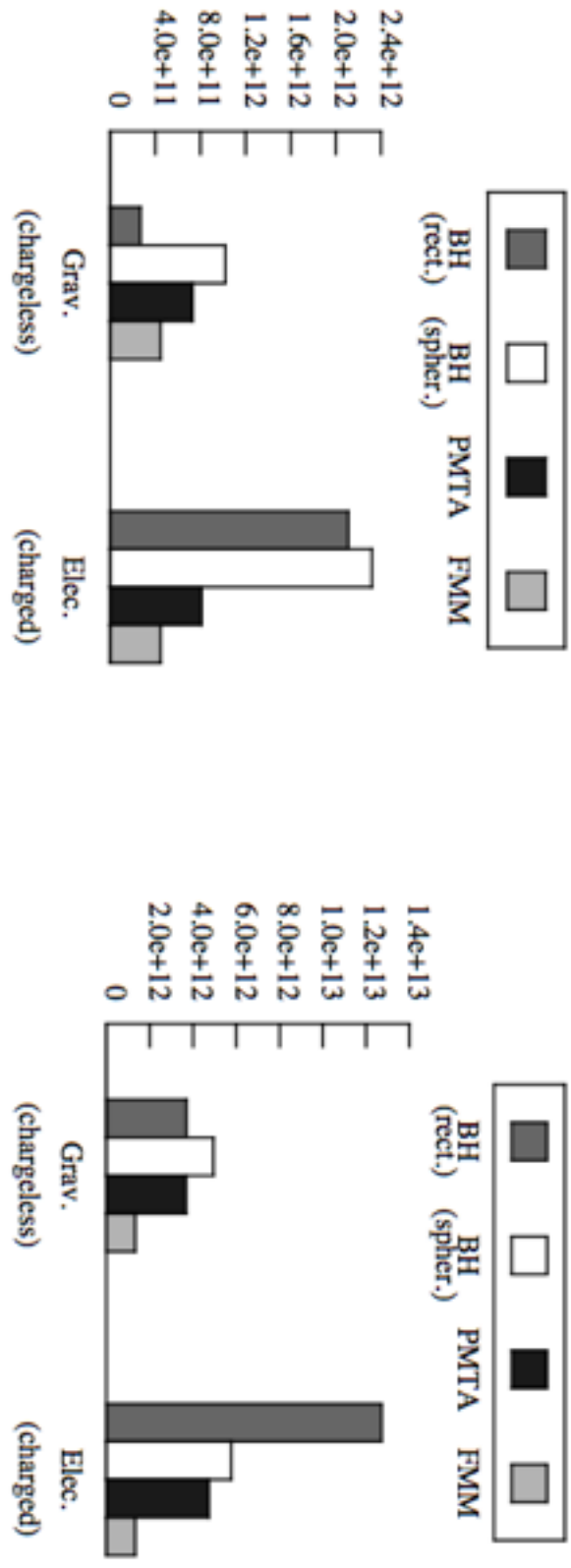

Figure 4.6: Algorithm memory usage for electrostatic and gravitational kernels with $N = 7 \cdot 10^6$, at low accuracy (top) and high accuracy (bottom). Y-axis values idicate the total number of bytes read from or written to global memory.

be more useful than Euclidean Barnes-Hut for electrostatic distributions at high accuracy, largely due to the fact that we have only used up to the quadrupole moment in the rectilinear version. Quadrupole approximation is effective for the gravitational kernel because the first few multipole terms dominate. The electrostatic kernel, on the other hand, requires higher order terms. This is because opposite charges tend to negate each other in the calculation of the monopole term.

CHAPTER 5

Conclusion

The modern graphics processing unit (GPU) was initially designed for raster-based videogame graphics, but its marked improvement in performance and programmability has generated considerable interest in it as a high-performance computing platform. The fact of multiple cores on the same chip, as opposed to multiple chips has a powerful effect on locality: the data can be processed without the throughput and storage costs of spilling it to buffer in the higher levels of the memory hierarchy. However, throughput-oriented algorithms and numerical methods must now explicitly express and optimize data movement and other high-energy operations.

This dissertation explored the design and deployment of memory-efficient tensor product representations for throughput-oriented processors. We have presented applications to several important numerical algorithms. In Chapter 2 we demonstrated how the representation given by Theorem 1.7 enables the creation of a memory-efficient Inverse Discrete Wavelet Transform algorithm. We presented test results from such an implementation using CUDA [cud] that performs extremely well on commodity desktop GPU hardware. In Chapter 3 we presented high-level representations for performing load distribution for an N-body simulation in 2-d using the Barnes-Hut algorithm. Finally, in Chapter 4 we conducted a detailed analysis of the memory complexity characteristics of the Barnes-Hut and Fast Multipole Methods in the general d-dimensional case, again using tensor product representations as primitives. We then presented performance benchmarks for the two algorithms, and verified our theoretical results with experimentally measured values.

Although many of the fundamental building blocks of throughput-oriented algorithms

have been created, there is much significant work still to be done in virtually every area of GPU algorithm design, especially at the application level. We fully anticipate that efficient algorithms as well as future hardware innovations will pull these co-processors closer to the CPU over time until, as with earlier co-processors, they disappear into the CPU itself.

CHAPTER 6

Appendix A- GPU Hardware

This appendix briefly discusses Cerberus, the machine used for much of the profiling presented aboive, and the GPU architecture in general.

6.1 Processors

Many of the performance metrics given in this dissertation were computed with Cerberus, a linux-based desktop machine with two GPUs.[1] The first GPU is an Nvidia GeForce 8 Series G80, and the second is an Nvidia Tesla Series C870. Both GPUs have simimlar architectures, which we outline for the G80 below.

Nvidia's GeForce G80 consists of a set of 16 multi-processor units called the streaming multi-processors (SMs). Each SM contains a set of 8 processor cores called streaming processors (cores), which run at 1.35 GHz, and one instruction fetch/decode unit. This implies that all 8 cores must simultaneously execute the same instruction. This is why divergence in the device code should be avoided as much as possible.

Each SM has 8,192 registers and a 16KB on-chip shared memory which, as noted above, has extremely low latency provided that memory bank conflicts are avoided. Each core has a lone arithmetic unit that performs single-precision floating point arithmetic and 32 bit integer operations. In addition, each SM has two special functional units, which perform more complex floating point operations such as inverse square root and trigonometric functions with high throughput. Both the cores and the two special functional units are fully pipelined.

[1] Cerberus was made available to me by kind permission of John Clyne and the NCAR CISL laboratory.

Thus, each of the 16 SMs can perform 18 flops per clock cycle (1 multiply-add operation per SP and one complex operation per special functional unit), yielding 388.8 Gflops of peak theoretical performance.

The G80's processors are virtualized in software and organized in a three-level hierarchy. At the highest level, a program (called a kernel) is executed by many threads on the GPU. Each kernel creates a single grid, which consists of many thread blocks. Thread blocks are co-scheduled groups of threads that can share data through on-chip shared memory and synchronize their execution using barrier instructions. A thread block consists of up to 512 threads; the number is usually chosen statically by the developer for each kernel. The hardware initiates thread block execution dynamically as resources become available. Each compiled code for each thread block is sent to a single SM, which executes it in its entirety. [2]

Threads within a block are further divided into sub-groups of 32 threads called warps. Each warp executes a single instruction simultaneously, with the SM's instruction unit broadcasting the instruction to the each of eight cores on four consecutive clock cycles. If threads within a warp take different control paths (called branch divergence), it is assumed that multiple iterations with blocking of threads on divergent paths are necessary to execute the algorithm. SMs can efficiently interleave warps on an instruction-by-instruction basis to hide the latency of global memory transactions and high-latency arithmetic operations. If one warp stalls, the SM can quickly switch to a new warp with ready operands available in the same thread block, or to a new warp in some other thread block assigned to the same SM. The SM stalls only if there are no warps available to execute within any of its assigned thread blocks. Scheduling freedom is high in many applications because threads in different warps are independent with the exception of explicit synchronization among threads in the same thread block. Therefore the number of active warps in a block is an important factor when it comes to hiding the latency form global memory transactions.

[2] It is important to note however that each SM can execute multiple thread blocks simultaneously.

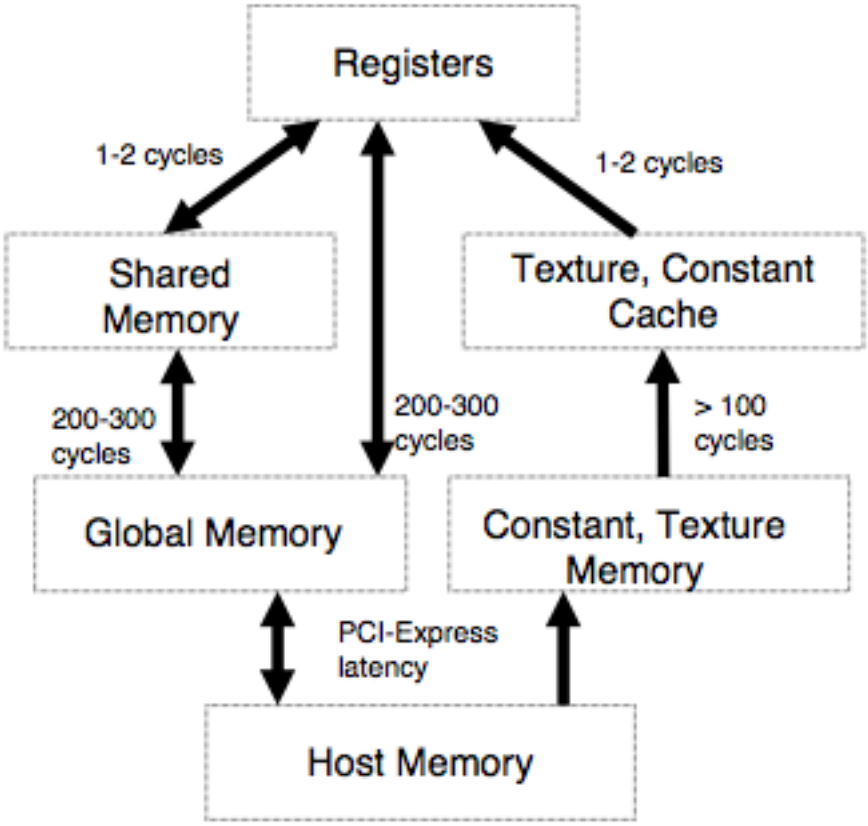

Figure 6.1: GPU memory hierarchy with typical access latencies.

6.2 Memory Management

Modern GPUs such as the G80 and C870 exhibit a complex memory organization with multiple low latency on-chip memories in addition to the off-chip DRAM. The access latencies and the optimal access patterns of each of the levels vary significantly, posing a significant challenge to devise techniques that optimally utilize the various memories to tolerate the latency and improve the memory throughput.

The various memories are organized in a hybrid cache and local-store hierarchy. Figure 6.1 illustrates these memories along with their hierarchical order and access latencies. The memories are as follows:

1. on-chip registers

2. on-chip shared memory

3. off-chip surface memory (with on-chip L-2 cache)

4. off-chip constant memory (with on-chip L-2 cache)

5. off-chip local memory

6. off-chip global memory

The on-chip registers are described in section 6.1 above. The other memories are detailed as follows:

Shared Memory. Shared memory is present in each SM and is organized into banks. When multiple addresses belonging to the same bank are accessed at the same time, it results in bank conflict. To achieve high memory throughput for concurrent accesses, shared memory is divided into equally sized memory modules (banks) that can be accessed simultaneously. Hence any n-address memory transaction that spans n distinct memory banks may be executed simultaneously, yielding an effective throughput that is n times higher than the throughput for a single bank. However, if multiple addresses in a transaction map to the same bank, the accesses will be serialized. The hardware splits the bank conflict into as many separate conflict-free requests as necessary, decreasing the effective throughput by a factor equal to the number of separate memory requests. Therefore accesses to shared memory should be designed to avoid serializing requests due to bank conflicts.

Constant,Surface and Local Memories. The constant and surface (or texture) memories are readonly regions in the global memory space and they have onchip readonly caches. For example, the GeForce G80 has a 64KB, offchip constant memory, and each SM has an 8KB constant memory cache. The constant cache is lowlatency, but it has only a single port and hence simultaneous accesses of different addresses will yield stalls. However, when multiple threads access the same address during the same cycle, the cache broadcasts the address's value to those threads with the same latency as a register access. Texture cache has a much higher latency than constant cache, but it does not suffer greatly when memory read accesses are irregular and is also beneficial for accessing data with 2-d spatial locality. Texture latency is per thread, whereas global latency is per transaction. Texture memory is a throughput

enhancing cache designed for streaming fetches with a constant latency; that is, a cache hit reduces DRAM throughput demand, but not fetch latency. Thus a texture cache hit has a higher latency on average per thread than a coalesced global load.
Finally, part of the off-chip memory, called the local memory, is used for storing variables that are spilled from registers.
Global Memory. Global memory provides the main storage for applications on the GPU. It consists of off-chip DRAM and has very low throughput compared to the chip's computational resources. Global memory supports read/write transactions from 2^0 to 2^4 bytes in size. Any access (via a variable or a pointer) to data residing in global memory will execute as a single global memory transaction if and only if the size of the data type is $2^0 - 2^4$ bytes and the data is aligned with cache boundaries (i.e. its address is a multiple of that size).

Providing named references to the different levels of memory improves device scalability by effectively forcing the issue of cache management up the abstaction ladder and into the software domain. In order to better manage memory usage, programmer interfaces (such as Cuda [cud], Stream [amd], and OpenCL [ope], described in section 6.5 below) expose a tiered memory hierarchy in software. Effective use of this memory hierarchy is critical to highthroughput algorithm design as it allows an application to reduce its demands for offchip memory throughput by exploiting data locality as noted above.

6.3 Cache Alignment and Coalescing Restrictions

The Tesla C870 GPU has 76.8 GB/s of throughput to its 1536 MB off-chip global memory. With computational resources supporting nearly 400 Gflops of performance and each FP instruction operating on up to 12 bytes of data, applications can easily saturate that bandwidth. To fully use the available global throughput, the 16 threads in the first or last half of a warp must each concurrently access contiguous, aligned blocks of memory so that the 16 accesses will be coalesced into a single largeword access; otherwise, more bandwidth is consumed as the hardware services each access separately.

When a group of threads executes an instruction that accesses global memory, it coalesces the memory read/writes of the threads within the warp into one or more of these memory read/writes depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. For example, if a 32 byte memory transaction is generated for each thread's 4 byte access, throughput is divided by 8.

As noted above, global memory resides in device memory and device memory is accessed via 32, 64, or 128 byte memory read/writes. These memory read/writes must be naturally aligned: Only the 32, 64, or 128 byte segments of device memory that are aligned to their size (i.e. whose first address is a multiple of their size) can be read or written by memory read/writes. In particular, this means that an array whose width is not a multiple of this size will be accessed much more efficiently if it is actually allocated with a width rounded up to the closest multiple of this size and its rows padded accordingly. [3] To maximize global memory throughput, it is therefore important to ensure that all memory transactions coalesce by:

1. Using data types that meet the size and alignment specifications required by the target hardware. If these requirements are not fulfilled, the memory read/writes will compile to multiple transactions with interleaved access patterns that prevent these instructions from fully coalescing. It is therefore necessary to use data types that meet this requirement [4] for data that resides in global memory.

2. Using aligned access patterns (i.e. the kth thread accesses the kth word in a segment). Perfomance degrades rapidly devices can degrade when successive threads access memory locations that have nonunit strides (i.e. the kth thread accesses the N*kth word in a segment). This pattern occurs naturally when dealing with multidimensional data or matrices; for example, when a half warp of threads accesses matrix elements columnwise and the ma-

[3]GPU programming interfaces provide memory (de-)allocation and copy functions in order to enable programmers to write non-hardware-dependent code that conforms to these constraints.

[4]programmer API's such as Cuda provide such data types as a matter of course

trix is stored in rowmajor order.

3. Padding high dimensional arrays in cases where lengths in one or more dimensions do not align. In particular it is unwise to allocate a linear array and calculate the required addresses in source code.

With judicious use of several on-chip memories an application can exploit data locality and data sharing to reduce its demands for offchip memory throughput. For example, the G80 GPU has a 64 KB, off-chip constant memory, and each SM has an 8KB constant memory cache. Because the cache is single ported, simultaneous accesses of different addresses yield stalls. However, when multiple threads access the same address during the same cycle, the cache broadcasts the address value to those threads with the same latency as a register access. Each SM also has a 16 KB writable shared memory that is useful for software managed data reuse or communication among threads in a thread block.

6.4 Other Device Limitations

Several other notable programming limitations exist for GPU software. For example, recursion and static variables are not allowed. These limitations do not apply to the host code however; in fact recursive fuctions inside of the host code constitute a powerful technique, since the recursion-related stack frames are naturally separated from the floating-point intensive part of the program. Although recursive divide-and-conquer algorithms are naturally cache efficient [Gus97], they are a poor fit for GPU architectures due to their recursive nature and do not perform competitively. Cuda has no such limitations because its recursion stack resides on the CPU and does not interfere with the GPU kernel.

Code optimization on a GPU is a tedious job with many obstructions. Performance on a GPU is often more fragile than performance on a CPU. It has been observed that small changes can cause huge effects on the performance [RRB08]. For example, in the optimized GEMM routine of Volkov [VD08], each thread block is 16×4 and each thread uses 32 registers. This means that 256 (8192/32) threads and 4 (256/64) thread blocks can be active

at the same time on each SM. Since one warp requires 4 cycles to execute an instruction and there are two warps per thread block, a latency of 32 (8*4) cycles can therefore be completely hidden. In the case that an extra variable is required, the compiler has two options. It can either spill the variable out to local memory and keep the register count intact, or or it can increase the register usage per thread by one. In the latter case, the number of thread blocks active on the chip is reduced to 3. This introduces a 25% reduction in parallelism, but may still out-perform the former option if the kernel has few instructions because access to a local variable will introduce a one-time latency penalty of 200-300 cycles. Therefore it is possible to drastically slow down the computation by adding just one extra line. Whichever option is chosen, it is obvious that the performance is not robust.

Another obstruction, discussed in detail above in **??**, is bandwidth optimization. In Cuda, peak bandwidth can only be achieved through memory coalescing, i.e., by making consecutively numbered threads access consecutive memory locations. One can heavily underutilize the GPU bandwidth by failing to fully coalesce memory transactions. Furthermore, the coalescing mechanism is rather counter-intuitive. For example, suppose one wants to scan a $16 \times N$ matrix stored in row-major order. On a symmetric multi-processing system with 16 cores, the most bandwidth-friendly method is to use each processor to scan a different row of the matrix. Each processor then makes at most N/B cache misses, which is optimal. However, on a GPU contiguous threads do not access contiguous range of memory addresses and this method will create multiple global memory read/writes per warp.

6.5 Programming Tools

The massively parallel nature of GPUs makes them capable of yielding theoretically much higher GFlops rates than current state-of-the-art CPUs. GPU performance also grows much faster than CPU performance due to specialized explicit parallelism. The amount of computational power to be harvested has also attracted the high-performance computing community, and we have seen many scientific applications successfully implemented with significant

performance gains on the GPU [BFG03, SHZ07].

As more and more CPU-bound applications are being reimplemented to run on the GPU, a collection of techniques (called general-purpose computing on graphics processing units (GPGPU)) has begun to coalesce. Both Nvidia and AMD offer programming interfaces for making GPGPU accessible to programmers who are not experts in computer graphics [amd, cud]. Nvidias Compute Unified Device Architecture (Cuda) offers a higher level C-like interface, whereas AMDs Close-to-Metal (CTM) allows the programmers to access lower levels of hardware. As opposed to CTM, the Cuda platform is unified in the sense that it has no architectural division for vertex and pixel processing. In addition the Kronos group has released an open specification OpenCL [ope].

As noted above, GPUs offer tremendous amounts of bandwidth and single-precision floating point arithmetic computation rates. Program execution on a GPU is expressed in terms of kernels launched on large input data sets with the fine-grained divisions onto hardware resources handled internally by a scheduler of the kind commonly found on GPUs. A program is composed of two parts: A host (CPU) code that makes kernel calls, and a device (GPU) code that actually implements the kernel. The host code is conceptually a serial C program, but the device code must be high-throughput in order to harness the power of the GPU.

This hierarchical notion of execution (executing one kernel at a time with barriers between stages/passes) is straight-forward and highly effective for most algorithms in computer graphics (the original problem domain). For example, throughput-oriented programming models excel at shader support. However, throughput-oriented programming models do have two weaknesses: footprint and load-balance in the presence of irregularity. Scheduling streams is antithetical to the server-client model: the entire output of one kernel must be accumulated and buffered before any can be used by the next kernel, therefore no pipeline parallelism can be used to overlap stages that cannot by themselves fill a machine.

Moreover, implementing GPGPU applications requires significant expertise, even with the recently introduced C-like APIs such as Cuda [LNO08]. The key to performance is to

hide the data access latency by having many threads on the fly, however GPGPU algorithm performance is usually fragile and requires careful craftmanship from the algorithm designers side. It is up to the designer to ensure that the registers and other levels of cache are neither under nor over-utilized. Much research has been devoted to the issue of achieving the optimal balance on GPUs. For example, [RRB08, VD08] rely on novel methods that are not necessarily intuitive to the existing HPC programmer.

References

[amd] "AMD Stream SDK." `http://ati.amd.com/technology/streamcomputing/`.

[BER73] E. Baerends, D. Ellis, and P. Ros. "Self-consistent molecular Hartree-Fock-Slater calculations part I- The computational procedure." *Chemical Physics*, **2**(1):41–51, 1973.

[BFG03] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid." *ACM Transactions on Graphics*, **22**(3):917–924, 2003.

[BH86] J. Barnes and P. Hut. "A hierarchical $\mathcal{O}(N \log N)$ force-calculation algorithm." *Nature*, **324**(4), 1986.

[BM02] G. Beylkin and M. Mohlenkamp. "Numerical operator calculus in higher dimensions." *Proceedings of the National Academy of Sciences*, **99**(16), 2002.

[Bou89] N. Bourbaki. *Elements of mathematics and Algebra I.* Springer-Verlag, 1989. II and Ch. 7 and No. 8.

[Cat44] R. Cattell. "Parallel proportional profiles and other principles for determining the choice of factors by rotation." *Psychometrika*, **9**:267–283, 1944.

[CC70] J. Carroll and J. Chang. "Analysis of individual differences in multidimensional scaling via an N-way generalization of Eckart-Young decomposition." *Psychometrika*, **35**:283–319, 1970.

[CGM05] H. Cheng, Z. Gimbutas, P.-G. Martinsson, , and V. Rokhlin. "On the compression of low rank matrices." *SIAM J. Sci. Comput.*, **26**:1389–1404, 2005.

[Cly03] J. Clyne. "The multiresolution toolkit: Progressive access for regular gridded data." In *Proceedings of Visualization, Imaging, and Image Processing*, pp. 152–157, 2003.

[CMN07] J. Clyne, P. Mininni, A. Norton, and M. Rast. "Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation." *New Journal of Physics*, **24**(8), 2007.

[CQ93] P. Crandall and M. Quinn. "Block data decomposition for data-parallel programming on a heterogeneous workstation network." *HPDC*, pp. 42–49, 1993.

[cud] "CUDA SDK." `http://developer.nvidia.com/object/cuda.html`.

[Dal09] W. Dally. "Power efficient supercomputing." `http://www.lbl.gov/cs/html/Manycore_Workshop09/GPUMulticoreSLAC2009/dallyppt.pdf`, 2009.

[Dau92] I. Daubechies. *Ten lectures on Wavelets.* SIAM Press, 1992.

[DB74] G. Dahlquist and A. Bjorck. *Numerical Methods.* Dover Publications, 1974.

[DCA08] A. Duran, J. Corbalan, and E. Ayguade. "Evaluation of OpenMP task scheduling strategies." In *Fourth International Workshop in OpenMP.*, 2008.

[DM07] P. Drineasa and M Mahoney. "A randomized algorithm for a tensor-based generalization of the singular value decomposition." *Linear Algebra and its Applications*, **420**(2-3):553–571, 2007.

[FCC94] J. Foran, B. Cabral, and N. Cam. "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware." In *Workshop on Volume Visualization*, 1994.

[FHW04] R. Fernando, M. Harris, M. Wloka, and C. Zeller. "Programming Graphics Hardware." In *EUROGRAPHICS*, 2004.

[GD08] N. Gumerov and R. Duraiswami. "Fast Multipole Methods on Graphics Processors." *Journal of Computational Physics*, **227**(18), 2008.

[GKS94] A. Grama, V. Kumar, and A. Sameh. "Scalable parallel formulations of the barnes-hut method for n-body simulations." *IEEE Supercomputing*, pp. 439–448, 1994.

[GL06] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 3 edition, 1006.

[GR87] L. Greengard and V. Rokhlin. "A fast algorithm for particle simulations." *Journal of Computational Physics*, **73**:325–348, 1987.

[GR02] Z. Gimbutas and V. Rokhlin. "A generalized fast multipole method for nonoscillatory kernels." *SIAM Journal of Scientific Computing*, **24**, 2002.

[GRB09] K. Gruchalla, M. Rast, E. Bradley, J. Clyne, and P. Mininni. "Visualization-driven structural and statistical analysis of turbulent flows." In *Advances in Intelligent Data Analysis*, volume 8, pp. 321–332, 2009.

[Gro85] H. De Groote. *Lectures on the complexity of bilinear problems*, volume 245 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[GTZ97] S. Goreinov, E. Tyrtyshnikov, and N. Zamarashkin. "A theory of pseudoskeleton approximations." *Linear Algebra and Its Applications.*, **261**, 1997.

[Gus97] F. Gustavson. "Recursion leads to automatic variable blocking for dense linear-algebra algorithms." *IBM J. Res. Dev.*, **41**(6):737–756, 1997.

[GWG02] S. Guthe, M. Wand, J. Gonser, and W. Strasser. "Interactive rendering of large volume data sets." In *IEEE Visualization*, 2002.

[GZ01] G. Golub and T. Zhang. "Rank-one approximation to high order tensors." *SIAM Journal of Matrix Analysis and Applications*, **23**(2):534–550, 2001.

[Hac91] W. Hackbusch. "The panel clustering algorithm." *The Mathematics of Finite Elements and Applications.*, **7**, 1991.

[Hac99] W. Hackbusch. "A sparse matrix arithmetic part I: Introduction to H-matrices." *Computing*, **62**, 1999.

[Hal74] P. Halmos. *Finite-dimensional Vector Spaces*. Springer-Verlag, 1974.

[Har70] R. Harshman. "Foundations of the PARAFAC procedure: Models and conditions for an explanatory multi-modal factor analysis." Technical report, University of California, Los Angeles, 1970.

[HE99a] M. Hopf and T. Ertl. "Accelerating 3d convolution using graphics hardware." In *IEEE Visualization*, 1999.

[HE99b] M. Hopf and T. Ertl. "Hardware based wavelet transformations." In *Workshop on Vision, Modeling, and Visualization.*, 1999.

[HE00] M. Hopf and T. Ertl. "Hardware accelerated wavelet transformations." In *EG/IEEE TCVG Symposium on Visualization.*, 2000.

[HGG04] M. Hazewinkel, N. Gubareni, N. Gubareni, and V. Kirichenko. *Algebras and Rings and Modules.* Springer-Verlag, 2004.

[Hit27] F. Hitchcock. "The expression of a tensor or a polyadic as a sum of products." *Journal of Mathematics and Physics*, **6**:164–189, 1927.

[HJ91] R. Horn and C. Johnson. *Topics in Matrix Analysis.* Cambridge University Press, 1991.

[HK00] W. Hackbusch and B. Khoromskij. "A sparse H-matrix arithmetic part II: application to multi-dimensional problems." *Computing*, **64**, 2000.

[HKT05] W. Hackbusch, B. Khoromskij, and E. Tyrtyshnikov. "Hierarchical Kronecker tensor-product approximations." *Journal of Numerical Mathematics 13*, 2005.

[HN89] W. Hackbusch and P. Nowak. "On the fast matrix multiplication in the boundary element method by panel clustering." *Numerische Mathematik*, **54**, 1989.

[HS86] W. Hillis and G. Steele. "Data parallel algorithms." *Communications of the ACM*, **29**(12), 1986.

[Ibr02] I. Ibraghimov. "Application of the three-way decomposition for matrix compression." *Numerical Linear Algebra With Applications*, **9**(551), 2002.

[IP99] I. Ihm and S. Park. "Wavelet-based 3d compression scheme for interactive visualization of very large volume data." *Computer Graphics Forum*, **18**(1), 1999.

[JMR94] P. Jones, J. Ma, and V. Rokhlin. "A fast direct algorithm for the solution of the Laplace equation on regions with fractal boundaries." *Journal of Computational Physics*, **113**, 1994.

[K20] `http://www.nvidia.com/object/personal-supercomputing.html`.

[KE02] M. Kraus and T. Ertl. "Adaptive texture maps." In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.*, 2002.

[Kie00] H. Kiers. "Towards a standardized notation and terminology in multiway analysis." *Journal of Chemometrics*, **14**:105–122, 2000.

[Kil] M. Kilian. "On the Riemann-Hilbert problem." Master's Thesis, Technical University of Berlin.

[Kru97] J. Kruskal. "Three-way arrays: rank and uniqueness of trilinear decompositions and with application to arithmetic complexity and statistics." *Linear Algebra Applications*, **18**:95–138, 197.

[KRW96] M. Kaddoura, S. Ranka, and A. Wang. "Array decompositions for nonuniform computational environments." *Journal of Parallel and Distributed Computing*, **36**:91–105, 1996.

[LHJ99] E. Lamar, B. Hamann, and K. Joy. "Multiresolution techniques for interactive texture-based volume visualization." In *IEEE Visualization*, 1999.

[LKH03] A. Lefohn, J. Kniss, C. Hansen, and R. Whitaker. "Interactive deformation and visualization of level set surfaces using graphics hardware." In *IEEE Visualization*, 2003.

[LMV00a] L. Lathauwer, B. Moor, and J. Vandewalle. "A multilinear singular value decomposition." *SIAM Journal on Matrix Analysis and Applications*, **21**(4):1253–1278, 2000.

[LMV00b] L. Lathauwer, B. Moor, and J. Vandewalle. "On the best rank-1 and rank-$(R_1 and R_2 and \ldots and R_N)$ approximation of high-order tensors." *SIAM Journal on Matrix Analysis and Applications*, **21**(4):1324–1342, 2000.

[LNO08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. "Nvidia Tesla: A unified graphics and computing architecture." *IEEE Micro.*, **28**(2):39–55, 2008.

[LRM12] J. Lord, M. Rast, C. Mckinlay, J. Clyne, and P. Mininni. "Wavelet decomposition of forced turbulence: Applicability of the iterative Donoho-Johnstone threshold." *Physics of Fluids*, **24**, 2012.

[Mal88] S. Mallat. "Multiresolution representations and wavelets." PhD dissertation, University of Pennsylvania, 1988.

[MGA03] W. Mark, R. Glanville, K. Akeley, and M. Kilgard. "Cg: A system for programming graphics hardware in a C-like language." *ACM Transactions on Graphics 22*, 2003.

[Mur92] S. Muraki. "Approximation and rendering of volume data using wavelet transforms." In *IEEE Visualization*, 1992.

[NCC08] A. Norton, Y. Chen, and J. Clyne. "Analysis and Visualization of High-Resolution WRF Hurricane Simulation using VAPOR." Technical report, National Center for Atmospheric Research, 2008.

[NH93] P. Ning and L. Hesselink. "Fast volume rendering of compressed data." In *IEEE Visualization*, 1993.

[NHP07] L. Nyland, M. Harris, and J. Prins. *Fast n-body simulation with CUDA.*, volume 3 of *GPU Gems*. Addison-Wesley, 2007.

[NKB01] R. van Nieuwpoort, T. Kielmann, and H.E. Bal. "Efficient load balancing for wide-area divide-and-conquer applications." In *Proceedings of the ACM SIGPLAN.*, 2001.

[NKL94] K. Nabors, F. Korsmeyer, F. Leighton, and J. White. "Preconditioned and adaptive and multipole-accelerated iterative methods for three-dimensional first-kind integral equations of potential theory." *SIAM Journal of Scientific Computing*, **15**, 1994.

[nvi] "Nvidia." `http://www.nvidia.com`.

[ope] "OpenCL." `http://www.khronos.org/opencl/`.

[PDC03] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan. "Photon mapping on programmable hardware." In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2003.

[Rod99] F. Rodler. "Wavelet based 3d compression with fast random access for very large volume data." In *Proceedings of the 7th Pacific Conference on Computer Graphics and Applications*, 1999.

[Rok85] V. Rokhlin. "Rapid solution of integral equations of classical potential theory." *Journal of Computational Physics*, **60**, 1985.

[Ros04] R. Rost. *OpenGL Shading Language.* Addison-Wesley, 2004.

[RRB08] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA." In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*, pp. 73–82, 2008.

[RS87] J. Rose and G. Steele. "C*: An extended C language for data parallel programming." In *Second International Conference on Supercomputing*, volume 2, 1987.

[SDS96] E. Stollnitz, T. Derose, and D. Salesin. *Wavelets For Computer Graphics: Theory and Applications.* Morgan Kaufmann, 1996.

[SHT95] J. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. "Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut and fast multipole and and radiosity." *Journal of Parallel and Distributed Computing*, **27**:118–141, 1995.

[SHZ07] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. "Scan primitives for GPU computing." In *ACM Graphics Hardware.*, volume 22, pp. 97–106, 2007.

[SLW01] Y. Sun, Z. Liang, and C. Wang. "Distributed particle simulation methods on adaptive collective systems." *Future Generation Computer Systems*, **18**:79–87, 2001.

[SN96] G. Strang and T. Nguyen. *Wavelets And Filter Banks.* Wellesley-Cambridge, 1996.

[SW03] J. Schneider and R. Westermann. "Compression domain volume rendering." In *IEEE Visualization*, 2003.

[VAP] VAPOR. `http://www.vapor.ucar.edu/`.

[VD08] V. Volkov and J. Demmel. "LU and QR and Cholesky Factorizations using Vector Capabilities of GPUs." Technical report, University of California, Berkeley, 2008.

[Wes02] R. Westermann. "A multiresolution framework for volume rendering." In *ACM Symposium on Volume Visualization*, 2002.

[WWH02] J.J. Wang, T.T. Wong, P.A. Heng, and C.S. Leung. "Discrete wavelet transform on GPU." In *ACM Workshop on General Purpose Computing on Graphics Processors*, 2002.