A Comparative Performance Analysis of the Phase Recovery Algorithm for Microstructure Reconstruction

A Thesis

Submitted to the Faculty

of

Drexel University

by

Anupama Shankar Kurpad

in partial fulfillment of the

requirements for the degree

of

Master of Science in Computer Engineering

June 2009

© Copyright June 2009
Anupama Shankar Kurpad. All Rights Reserved.

Table of Contents

List of Tables iv

List of Figures v

Abstract vii

1. Introduction 1

2. Microstructure Reconstruction Overview 3

   2.1 The Microstructure Function and 2-point Statistics 3

   2.2 Phase Retrieval Algorithms 5

      2.2.1 Error Reduction Algorithm 6

      2.2.2 Input-Output Algorithm 8

      2.2.3 Phase Recovery for Microstructure Reconstruction 10

3. Current Implementation 13

4. FFT Performance Comparison 16

   4.1 Benchmarking Methodology 16

   4.2 CPU Implementation and Benchmarks 17

   4.3 GPU Implementation 18

      4.3.1 GPU Overview 18

      4.3.2 GPU Hardware 19

      4.3.3 Development on CUDA 21

      4.3.4 GPU FFT Benchmarks 23

   4.4 Comparison of CPU and GPU FFT Performance 23

5. Phase Recovery Performance Comparison 25

   5.1 Benchmarking Methodology 25

   5.2 Phase Recovery Benchmarks 26

   5.3 Performance Optimization Strategies 27

      5.3.1 MKL Implementation 27

5.3.2 NVIDIA CUDA 28
5.4 Limitations 33
6. SPIRAL Implementation 34
6.1 SPIRAL Overview 34
6.1.1 SPL and Ruletrees 35
6.1.2 Vectorization and Parallelization 37
6.2 SPIRAL FFT Benchmarks 40
6.3 Microstructure Reconstruction on SPIRAL 41
6.3.1 Computing the Estimate 43
6.3.2 Error Calculation 46
6.3.3 Computation of Modified Input 47
6.3.4 The Complete Implementation 48
6.4 Phase Recovery Benchmark Results on SPIRAL 49
7. Conclusions and Future Work 51
BIBLIOGRAPHY 52
APPENDIX A: Hardware and Software Specifications 56

List of Tables

5.1 Calculation of Total Operation Count for One Iteration of Phase Recovery 25
5.2 Profiler Output Indicating Uncoalesced Global Memory Loads and Stores for 2D Phase Recovery 31
6.1 Estimated Performance of Vectorized and Parallelized SPIRAL Phase Recovery Implementation 50

List of Figures

2.1 Autcorrelation of a 16 × 16 Microstructure with States 0 and 1 and a Checkerboard Distribution 5

2.2 Laterally Shifted Reconstructed 16×16 Eigen Microstructure 6

2.3 Original 16×16 Eigen Microstructure 6

2.4 Block Diagram of the Error Reduction Algorithm 7

2.5 Block Diagram of the Input-Output Algorithm 9

2.6 Block Diagram of Microstructure Reconstruction 12

3.1 2D Phase Recovery on Matlab 7.6 13

3.2 1D FFT Performance Comparison 14

4.1 2D FFT on Intel Core 2 Quad Q9300 17

4.2 3D FFT on Intel Core 2 Quad Q9300 17

4.3 GeForce 8800 Architecture *Source:* [24] 20

4.4 2D FFT on NVIDIA CUDA with GeForce 9800 GX2 23

4.5 3D FFT on NVIDIA CUDA with GeForce 9800 GX2 23

4.6 Intel MKL and NVIDIA CUDA 2D FFT Comparison 24

4.7 Intel MKL and NVIDIA CUDA 3D FFT Comparison 24

5.1 2D Phase Recovery Comparison 26

5.2 3D Phase Recovery Comparison 26

5.3 CUDA Visual Profiler Summary Plot for 2D Phase Recovery Size: 256×256 31

6.1 The Architecture of SPIRAL *Source:* [35] 35

6.2 The SPL Compiler *Source:* [35] 38

6.3 2D FFT on SPIRAL 40

6.4 3D FFT on SPIRAL 40

6.5 2D FFT Comparison with SPIRAL and Intel MKL 41

6.6 3D FFT Comparison with SPIRAL and Intel MKL 41

6.7 Phase Recovery on Spiral 42

6.8 2D Phase Recovery Comparison between Intel MKL and SPIRAL 49

6.9 3D Phase Recovery Comparison between Intel MKL and SPIRAL 49

Abstract

A Comparative Performance Analysis of the Phase Recovery Algorithm for Microstructure Reconstruction

Anupama Shankar Kurpad
Advisors: Jeremy Johnson, PhD, Prawat Nagvajara, PhD

This thesis explores the high-performance implementation of a phase recovery algorithm for microstructure reconstruction of materials. Implementations on a variety of high-performance computing platforms, including multi-core and Graphics Processing Unit (GPU), were investigated and compared. The phase recovery algorithm is an iterative process requiring multiple Discrete Fourier Transform (DFT) computations each iteration. In order to achieve high-performance, it is necessary to use highly optimized fast Fourier transform (FFT) code to compute the DFTs. In our investigation, several FFT libraries, including FFTW, the Intel® Math Kernel Library (MKL), the CUFFT library for the NVIDIA® GPU, and the SPIRAL generated code, were used and compared. The SPIRAL system provides an extensible framework for generating and automatically optimizing implementations of DSP (digital signal processing) algorithms described using mathematical formulas, and is the most extensible of the platforms investigated here. The phase recovery algorithm intersperses FFT computations with point-wise computations, and while the FFTs are the dominant computation, the point-wise operations can have a significant impact on the overall performance. Therefore, simply relying on the performance of an optimized FFT library is insufficient to obtain optimal performance. Unlike the FFTW, MKL, and CUFFT libraries, the SPIRAL system allows the FFTs to be combined with the point-wise operations and the entire algorithm to be optimized. In this thesis, we obtained a mathematical formula representing the phase recovery algorithm that can be incorporated into the SPIRAL framework and utilize SPIRAL's parallel

and vector code generation and optimization facilities. The SPIRAL code generated in this thesis is sequential. We estimate that with a vectorized and parallelized SPIRAL implementation, it is possible to obtain a 1.5-fold speedup for two-dimensional (2D) phase recovery and 1.88-fold speed up for 3D phase recovery over the MKL implementation.

1. Introduction

Problem Statement Phase retrieval algorithms, which are widely used in the fields of X-ray crystallography, wave front sensing and image processing, concern recovering the phase of complex valued data when only intensity measurements are made[11, 12, 38, 37]. Microstructure quantification is one such application wherein phase information is lost during the quantification process[19]. The phase recovery algorithm used for microstructure reconstruction is iterative and involves multiple transformations back and forth between the object and Fourier domains. A MATLAB implementation of the algorithm shows a 10-fold increase in computation time for for every 4-fold increase in data size for datasets greater than 2^{16} points, indicating the need for an efficient and scalable implementation of the algorithm. FFT computations constitute $\approx$ 42% of computation time, indicating that performance of the phase recovery algorithm depends on the performance of the library used to compute the FFTs. This thesis addresses the question:

Given the choice of a multitude of platforms, how can an efficient and scalable implementation of the phase recovery algorithm be achieved

Result Summary This thesis provides a comparative analysis of the performance of the phase recovery algorithm on three platforms - the Intel® Math Kernel Library (MKL)[6] on a multi-core CPU with vectorization, NVIDIA® CUDA™ [10, 29] and SPIRAL[3, 35]. This focus of this thesis is on the platform specific adaptation and optimization of an existing algorithm rather than on optimizing the domain specific attributes of phase recovery. We provide an analysis of the performance of the phase recovery algorithm on each of these three platforms. Tuning for performance is a highly platform specific task, and we discuss the methods involved for each of the

three platforms. The SPIRAL code generated for phase recovery as part of this thesis currently is sequential and does not yet utilize multiple cores, vector instructions or the GPU, whereas the MKL implementation is both vectorized and parallelized. We provide an estimate of SPIRAL performance with vectorization and parallelization based on the FFT benchmark results. Our estimate is that it is possible to obtain a speedup of 1.5 times for 2D phase recovery, and 1.88 times for 3D phase recovery over the MKL implementation on the same hardware platform.

Organization of the Thesis This thesis is organized as follows - Chapter 2 begins with an overview of microstructure reconstruction, and describes the algorithms used for recovering phase. Here we present the mathematical formulations of phase retrieval. Chapter 3 discusses the issues with the current implementation and need for an efficient implementation. Chapter 4 provides a performance comparison of DFT libraries, with an overview of the platforms on which the benchmark timings were generated. With the FFT benchmarks as the base, Chapter 5 describes the performance of the phase recovery algorithm on the platforms discussed in Chapter 4. Chapter 6 presents an overview of SPIRAL, and the SPIRAL implementation of phase recovery. We conclude with a summary of our findings and scope for future work in Chapter 7.

2. Microstructure Reconstruction Overview

2.1 The Microstructure Function and 2-point Statistics

The internal structure of a material, called its microstructure, is quantified by means of a list of selected statistical measures such as grain size, orientation distribution and shape distribution[19]. The n-point formalism is one method of microstructure quantification, and is characterized by the microstructure function $m(x, n)$. This function reflects the probability of finding a distinct local state n in the immediate vicinity of position x. The microstructure is assumed to be available on a regular grid in the 3D space that it occupies. The dataset m_s^n denotes all such possible probability distribution functions in the 3D space with n enumerating the set of distinct local states in the system and s enumerating the uniform grid of spatial locations covering the microstructure. If N is the total number of local states and S is the total number of grid points, the microstructure satisfies the property

$$\sum_{n=1}^{N} m_s^n = 1 \tag{2.1}$$

The 1-point statistics indicate the probability of finding a specific local state of interest at a single point thrown randomly into the microstructure.The discretized 1-point statistics is defined by

$$f^n = \frac{1}{S} \sum_{s=0}^{S-1} m_s^n \tag{2.2}$$

At the next hierarchical level of quantification, the 2-point statistics describe the probability of finding specific local states at two ordered points, separated by a specified vector, thrown randomly into the microstructure. This information is obtained for all possible vectors of interest in the microstructure, and the collective dataset is

called the 2-point statistics of the microstructure. The discretized 2-point statistics is given by the convolution

$$f_t^{nn'} = \frac{1}{S}\sum_{s=0}^{S-1} m_s^n m_{s+t}^{n'} \tag{2.3}$$

The superscripts n and n' denote the two local states of interest, and the subscript t enumerates all vectors that can be can thrown randomly into the microstructure. Both points of the vector t lie on the same discretized spatial grid that was used to describe the microstructure function.

The DFT of the microstructure function m_s^n is computed as

$$M_k^n = \mathcal{F}(m_s^n) = \sum_{s=0}^{S-1} m_s^n e^{\frac{2\pi isk}{S}} = |M_k^n| e^{i\theta_k^n} \tag{2.4}$$

Here, $|M_k^n|$ will be referred to as the magnitude of the microstructure function and θ_k^n as its phase. The DFT of the 2-point statistics in 2.3 is computed using the convolution theorem,

$$F_k^{nn'} = \mathcal{F}(f_t^{nn'}) = \frac{1}{S}\sum_{s=0}^{S-1} m_s^n e^{\frac{2\pi isk}{S}} \sum_{z=0}^{S-1} m_z^{n'} e^{\frac{-2\pi izk}{S}} \tag{2.5}$$

$$= \frac{1}{S}|M_k^n| e^{-i\theta_k^n} |M_k^{n'}| e^{i\theta_k^{n'}} \tag{2.6}$$

In equation 2.6, if $n = n'$, the correlations obtained are called the autocorrelations, and

$$F_k^{nn} = \mathcal{F}(f_t^{nn}) = \frac{1}{S}|M_k^n| e^{-i\theta_k^n} |M_k^n| e^{i\theta_k^n} \tag{2.7}$$

$$= \frac{1}{S}|M_k^n|^2 \tag{2.8}$$

It can be seen from 2.8 that the Fourier transform of the autocorrelation is the square of the magnitude of the DFT of the microstructure function indicated in 2.4. All

phase information is lost. The retrieval of this phase information, which is necessary for the reconstruction of the microstructure, is the subject of the following section.

2.2 Phase Retrieval Algorithms

Phase retrieval algorithms typically involve iterative transformations back and forth between the Fourier and object domains; this is a class of problems that has been well studied[38, 12, 20, 37]. In the current problem, the microstructure is assumed to have two local states 0 and 1; such microstructures are called Eigen microstructures.

As an example, consider the reconstruction of a 2-dimensional Eigen microstructure of size 16×16. It must be noted that the starting point for phase recovery is the autocorrelation (equation 2.8) as indicated in Figure 2.1, and not the Eigen microstructure itself.

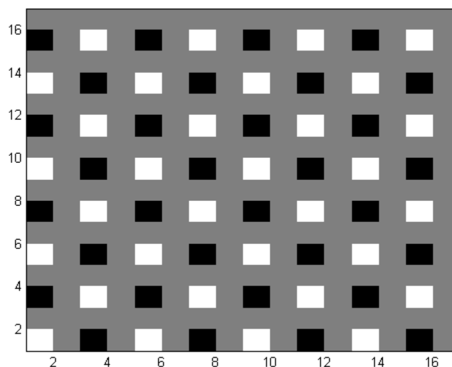

Figure 2.1: Autcorrelation of a 16×16 Microstructure with States 0 and 1 and a Checkerboard Distribution

The microstructure that is reconstructed using the phase recovery algorithm is as shown in Figure 2.2. The original microstructure is shown in Figure 2.3 for reference.

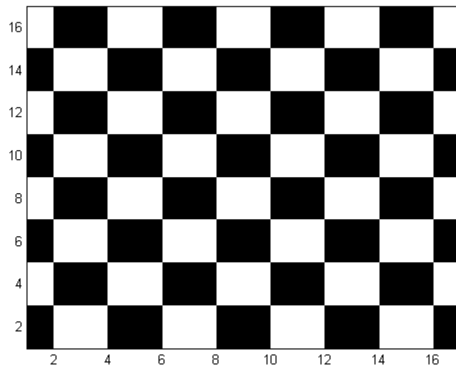

Figure 2.2: Laterally Shifted Reconstructed 16×16 Eigen Microstructure

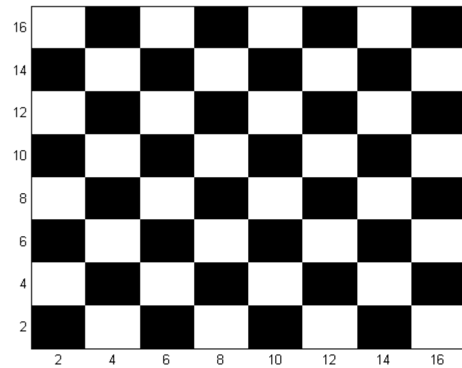

Figure 2.3: Original 16×16 Eigen Microstructure

The 2-point statistics filter out translations and inversions of the microstructure. In other words, microstructure functions m_s^n and $m_{\pm s+a}^n$ produce identical sets of 2-point statistics. Therefore, any reconstruction from a given set of 2-point statistics can at best recover the original microstructure to within an arbitrary translation and/or an inversion[19, pg.945]. This translation can be seen in the reconstructed microstructure in Figure 2.2. The following sections describe the general approaches to recovering phase, and the stages involved in the retrieval process specific to microstructure reconstruction.

2.2.1 Error Reduction Algorithm

The generalized error reduction algorithm for phase recovery, also known as the Gerchberg-Saxton algorithm[20] consists of the following basic steps

1. Fourier transform an estimate of the object
2. Replace the magnitude of the computed Fourier transform with known Fourier modulus

3. Inverse transform the estimate of the Fourier transform

4. Modify the new estimate of the object to satisfy object domain constraints

These steps are performed iteratively until the estimate obtained is a satisfactory estimate of the object; in this case, the microstructure function. The stopping criterion for the algorithm is determined by computing a root mean squared error as indicated in Equations 2.9 and 2.10. If the computed error indicates that the estimated value of the object is not an accurate estimate when compared to the desired result, another iteration is performed. The input to this iteration is the output of the previous iteration, modified so as to satisfy the required object domain constraints. Figure 2.4 shows the algorithm as applied to the general case.

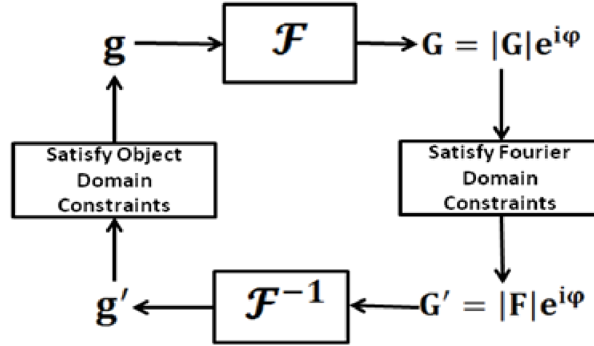

Figure 2.4: Block Diagram of the Error Reduction Algorithm

Error Computation At the kth iteration, the input to the error reduction algorithm is $g_k(x)$ and the corresponding output is $g'_k(x)$. The Fourier domain representations of these quantities are $G_k(u)$ and $G'_k(u)$ respectively. We can see that $G'_k(u)$ was formed from $G_k(u)$ by substituting the Fourier modulus.

The squared error in the Fourier domain can be expressed as

$$E_{Fk}^2 = \frac{1}{N^2} \sum_u [|G_k(u)| - |F(u)|]^2 \tag{2.9}$$

In the object domain, this error is -

$$E_{Ok}^2 = \sum_x [|f(x)| - |g_k'(x)|]^2 \tag{2.10}$$

The algorithm is said to have converged when the error reaches a specified small value[19, pg. 948]. It has been shown that the error either reduces every iteration or remains the same. The error typically plateaus after a certain number of iterations, and with persistence, one can go beyond the plateau region and make progress towards a solution[38, pg. 2760][13]. For single intensity measurements, the number of iterations can be large, necessitating modifications to the algorithm for efficiency. The Input-Output algorithm is result of these modifications.

2.2.2 Input-Output Algorithm

The first three steps of the Input-Output algorithm are similar to those in the Error Reduction algorithm. However, the difference is that the input $g(x)$ need not be an estimate of the object. While this allows a greater flexibility in choosing the input, it requires that the output of each iteration be modified so that the estimate is moving in the direction of the required result. Figure 2.5 shows the dataflow in the Input-Output algorithm. It has been shown that a small change in the input results in a change in the output in the same general direction as the change in the input[38, 36, pg. 2763]. This is of particular importance to the Input-Output algorithm because the chosen input is not the current best estimate of the object as in the case of the Error Reduction algorithm. Hence it becomes necessary to ensure that the algorithm

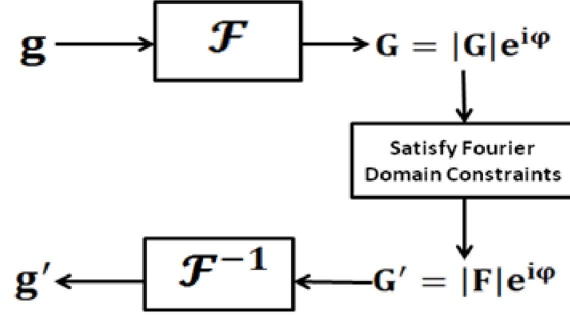

Figure 2.5: Block Diagram of the Input-Output Algorithm

incrementally proceeds towards the desired result with every iteration. At the kth iteration, the desired change in the output is defined as

$$\Delta g_k(x) = \begin{cases} 0 & x \notin \gamma, \\ -g'_k(x) & x \in \gamma. \end{cases} \tag{2.11}$$

where γ is the set of all points where the output violates the object domain constraints. This equation indicates that there is no need for a change of the output at those points where the constraints are satisfied. But where the constraints are violated, the desired change of the output, in order to have it satisfy the object domain constraints, is one that drives it to a value of zero (and, therefore, the desired change is the negative of the output at those points). Therefore, a logical choice for the next input is

$$g_{k+1}(x) = g_k(x) + \beta \Delta g_k(x) \tag{2.12}$$

Substituting for $\Delta g_k(x)$ from 2.11,

$$g_{k+1}(x) = \begin{cases} g_k(x) & x \notin \gamma, \\ g_k(x) - \beta \Delta g'_k(x) & x \in \gamma. \end{cases} \tag{2.13}$$

One of the properties of this algorithm is that if $g'(x)$ is applied as the input to the next iteration, it passes through the system unchanged, implying that the output of the next iteration is also $g'(x)$ [11, pg. 2763]. From this point of view, another logical choice for the input to the next iteration is

$$g_{k+1}(x) = g'_k(x) + \beta \Delta g_k(x) \tag{2.14}$$

Substituting 2.11 into 2.14,

$$g_{k+1}(x) = \begin{cases} g'_k(x) & x \notin \gamma, \\ g'_k(x) - \beta \Delta g'_k(x) & x \in \gamma. \end{cases} \tag{2.15}$$

Choosing the next input as a combination of equations 2.13 and 2.15,

$$g_{k+1}(x) = \begin{cases} g'_k(x) & x \notin \gamma, \\ g_k(x) - \beta \Delta g'_k(x) & x \in \gamma. \end{cases} \tag{2.16}$$

2.16 indicates the change that needs to be applied to the output of the kth iteration to obtain the input to the $(k+1)^{st}$ iteration.

2.2.3 Phase Recovery for Microstructure Reconstruction

A Hybrid Approach: The hybrid approach for microstructure reconstruction combines features from these two algorithms to ensure faster convergence[19, 7]. More specifically,

1. The initial input to the algorithm $g(x)$ is chosen to be random
2. The output of the kth iteration is modified as indicated in 2.16

3. The output of the kth iteration is modified to satisfy object domain constraints, i.e. $g'_k \leq 1$

4. The basis of determining convergence is the squared error calculated at every iteration

5. β is chosen to be 1.1

It can be seen that steps (1) and (2) follow the Input-Output algorithm, and steps (3) and (4) are along the lines of the Error Reduction algorithm.

Error Calculation and Convergence: The Error Reduction algorithm suggests that the squared error become an arbitrarily small value as the metric for convergence. But as mentioned earlier, the error reaches a plateau after a certain number of iterations. For the current problem of microstructure reconstruction, we consider the algorithm to have converged after the first plateau has been reached[13, pg. 1900]. Figure 2.6 is a pictorial representation of all of the stages involved in phase retrieval.

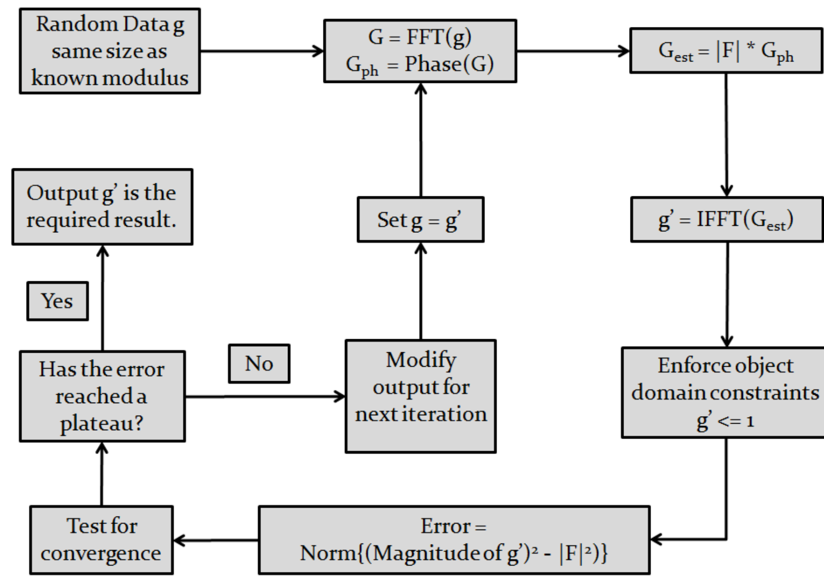

Figure 2.6: Block Diagram of Microstructure Reconstruction

3. Current Implementation

Figure 3.1 shows the performance of the algorithm on MATLAB 7.6[1] with computation time being represented on a logarithmic scale. It can be seen that computation time increases 10-fold for data sizes greater than 2^{16}.

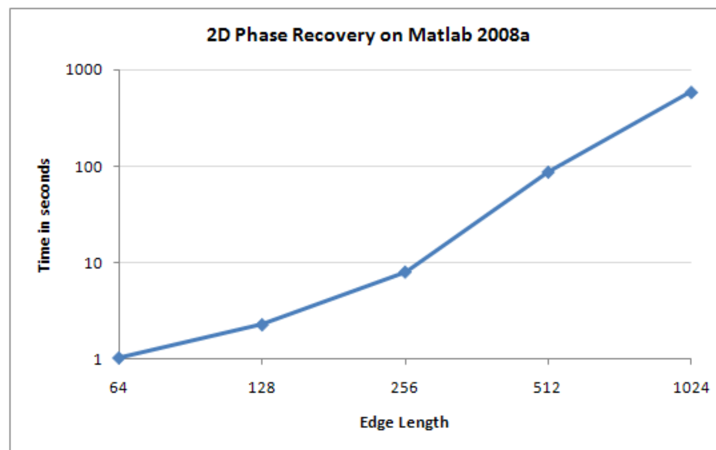

Figure 3.1: 2D Phase Recovery on Matlab 7.6

The microstructures in question are typically 3-dimensional, and of sizes $2^9 \times 2^9 \times 2^9$ and higher. The current implementation becomes infeasible for these sizes, given the increase in computation time seen in Figure 3.1. This clearly indicates the need for an efficient implementation of phase recovery that will allow for microstructures to be reconstructed in real time. In order to optimize the current implementation, it is first necessary to identify "hotspots" in the program - an analysis that can be easily performed by using a profiler. The MATLAB profiler summary shows that for data sizes greater than 2^{16}, 42-45% of runtime is constituted by the DFT computations.

[1]MATLAB code provided by Stephen Niezgoda, Department of Material Science and Engineering, Drexel University

While it is true that better performance can be achieved with implementations on platforms other than MATLAB, it must be noted that the version of MATLAB used in this implementation calls FFTW 3.0, a highly optimized, compiler C library, to compute the DFTs. Since the DFTs, which constitute the bulk of the computation time, are being computed by a highly tuned library, the performance indicated here provides a fair base reference against which future optimizations can be compared. Performance of the algorithm depends on that of the underlying DFT computations, and hence, optimal FFT performance is crucial for obtaining optimal phase recovery performance.

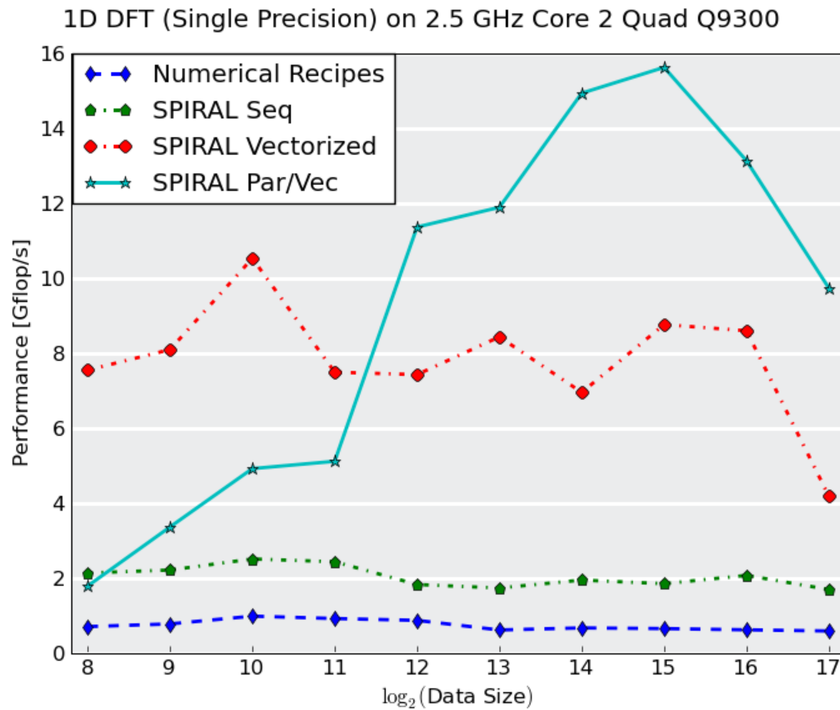

Figure 3.2: 1D FFT Performance Comparison

One way of achieving optimal FFT performance is by making use of a highly optimized, state-of-the-art library such as the FFTW, the Intel Math Kernel Library (MKL), or the AMD Performance Library (APL), all of which offer up to a 20-fold speedup over conventional FFT implementations. The CUFFT library implemented on NVIDIA's CUDA platform is a recent addition to the group of high performance libraries. Figure 3.2 is an indication of the performance improvement that can be expected by using an optimized library; the comparison is between a base Numerical Recipes[4] implementation and SPIRAL. It can be seen that the sequential implementation of SPIRAL is 3-4 times faster than the Numerical Recipes performance. The SPIRAL implementation with vectorization and parallelization is ≈ 23.7 times faster than the base implementation.

In this thesis, we first compare FFT performance, and then implement the phase recovery algorithm using those libraries that promise optimal performance. It must be noted that all optimizations performed are from the perspective of reducing computation time, and not the number of operations. The latter approach requires manipulation of the domain-specific attributes of the algorithm, and is beyond the scope of this thesis.
The CUDA implementation with CUFFT is discussed in a separate section since this required development on a computing platform that was significantly different from the rest. In the following chapter, we compare the performance of the MKL and FFTW libraries in order to determine the best CPU implementation for phase recovery.

4. FFT Performance Comparison

In this chapter, we present a comparison of FFT performance between three optimized libraries. The first section compares of the FFTW and Intel MKL libraries on the Intel Core™ 2 Quad Q9300 processor. The subsequent sections provide overviews of the NVIDIA GeForce 9800 GX2 hardware and CUDA platforms, and a comparison of CUFFT performance with Intel MKL on the Intel Core™ 2 Quad Q9300 processor.

4.1 Benchmarking Methodology

The benchmarking methodology for the DFTs was the same as used in the FFTW benchmarks [1]. The DFTs were first executed an arbitrary number of times to determine the number of iterations needed to get a consistent timing. The DFTs were then timed over the obtained number of iterations. The time per iteration is simply the total runtime divided by the number of iterations. This process of obtaining the time per iteration was repeated 8 times. The best run from these 8 timing values was chosen to be the required time per iteration. This repetition is needed in order to minimize the effects of random interference from a variety of sources including those from the operating system.

DFT performance was measured in terms of pseudo GFlops/s which was calculated as

$$(5 * Nlog_2(N) * 10^9)/(Time\ per\ iteration\ in\ sec) \quad (4.1)$$

where N is the total number of elements. This method has been used to benchmark all of the libraries being evaluated in this thesis. The hardware and software

specifications of all platforms/libraries used is listed in Appendix A.

4.2 CPU Implementation and Benchmarks

FFTW is a highly optimized C library for computing DFTs in one or more dimensions, for both real and complex valued data[18, 1]. FFTW can be configured for sequential, vectorized and/or threaded operations at the time of installation. We provide a comparison of the performance of the library in all modes of operation.

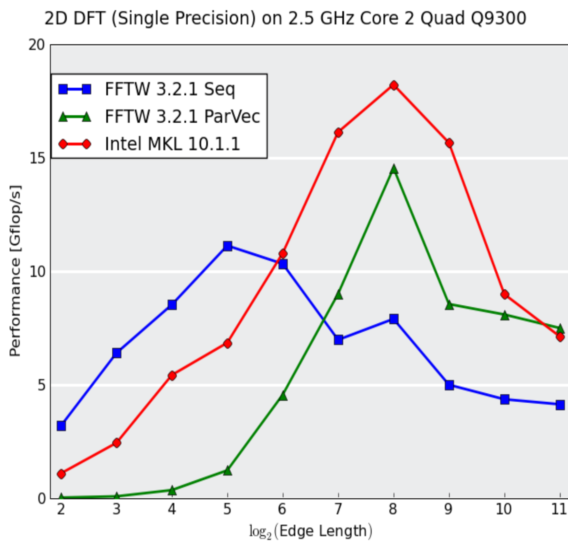

Figure 4.1: 2D FFT on Intel Core 2 Quad Q9300

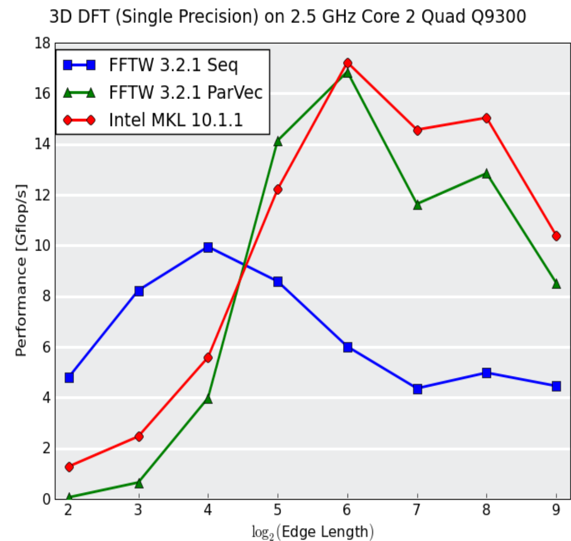

Figure 4.2: 3D FFT on Intel Core 2 Quad Q9300

The Intel Math Kernel Library (Intel MKL) provides developers of scientific, engineering and financial software with a set of highly optimized linear algebra routines, fast Fourier transforms, and vectorized math and random number generation functions. A detailed description of the library's functionalities can be found in the docu-

mentation [6]. The FFT library is both vectorized and parallelized, and is available as a distributed version for execution on clusters. Unlike FFTW, the Intel MKL cannot be configured for sequential operation, and so only the vectorized and parallelized performance is presented here. The FFT library is thread-safe with threads being called and managed within the library, and does not require threads to be explicitly created by the calling application.

The peak performance of the CPU used in this implementation is 40 GFlops/s (obtained as *number of cores * vector length * CPU operating frequency* = 4 * 4 * 2.5 GHz). Figures 4.1 and 4.2 show the 2-dimensional and 3-dimensional DFT benchmarks using the Intel MKL 10.1.1 and FFTW 3.2.1 libraries. It can be seen that the FFT peak performance is roughly 17 GFlops/s, which is about 42.5% of the performance that can be obtained on this platform.

The performance of the FFTW and MKL libraries is comparable in the case of both 2D and 3D data, but the Intel MKL shows better performance for 2D data. Hence, the MKL was used in the CPU implementation of the phase recovery algorithm. The following section discusses yet another high-performance library and computing platform.

4.3 GPU Implementation

4.3.1 GPU Overview

Commodity Graphics Processing Units (GPUs), found on video cards, are high-performance many-core processors that perform graphics rendering and texturing operations. Graphics applications are typically comprised of data-parallel computations with high arithmetic intensity (the ratio of arithmetic operations to memory

operations). GPU hardware is designed particularly for parallel computations and so can function as efficient co-processors for non-graphics applications that are highly data-parallel, such as general signal processing applications, matrix algebra, real time physics simulations, and applications related to computational biology and computational finance. Until recently, the use of GPUs for non-graphics applications required a detailed understanding of graphics APIs; a task typically associated with a steep learning curve[26]. General Purpose computing on Graphics Processing Units, or GPGPU, is a development in computing that aims at exploiting the high computation and data throughput that GPUs offer towards improving performance of non-graphics applications[2, 26]. A more recent effort at standardizing general-purpose parallel programming of heterogeneous systems is OpenCL™ created by the Khronos group[5]. Compute Unified Device Architecture, or CUDA™ , developed by and proprietary to NVIDIA® Corporation, is an example of a programming model that exposes the GPU's parallel capabilities to non-graphics applications without requiring programmers to use graphics APIs [29, 10]. The following sections provide a descripton of the CUDA hardware and software models, and present the FFT performance of the GPU using the CUFFT library.

4.3.2 GPU Hardware

The NVIDIA GPU architecture is described in detail in the NVIDIA documentation [29, 21, 9, 24, 27, 28]. Figure 4.3 shows the organization of the shader core[24]. The shader core on the GeForce 8800 GTX is made up of 8 clusters called **Texture Processor Clusters (TPC)** that is in turn made up of a texture unit and two **Streaming Multiprocessors (SM)**. The 8800 GTX has 8 TPC units and therefore 16 SM's. The front end of the texture unit reads, decodes and issues instructions. The streaming multiprocessors constitute the backend and each consist of 8 **Scalar**

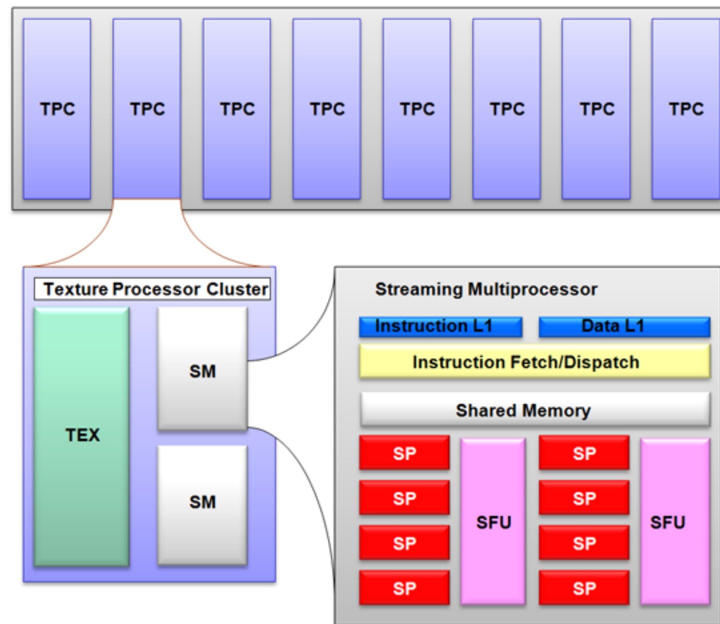

Figure 4.3: GeForce 8800 Architecture *Source:* [24]

Processor cores (SP), two special function units for transcendentals and on-chip shared memory. The backend operates at double the frequency of the front end, which makes the SPs appear as 16-way SIMD units instead of 8-way ones. The SM creates, schedules and manages threads that run in batches of 32 called warps. At each cycle, an instruction is issued to the warp selected by the front end. The backend requires four cycles to issue the instruction to all 32 threads in the warp, but it operates at double the frequency of the front end, and hence considers only two cycles to have been executed. The SPs are clocked at a default of 1.35 GHz. When running CUDA applications, each SP issues one multiply-add (MAD) instruction per cycle. This gives each SM a peak performance of 21.6 GFLOPS, and the GeForce 8800 GTX with 16 SMs, an aggregate performance of 345.6 GFlops/s. Compared to the CPU peak of 40 GFlops/s, the GPU indicates an 8-fold increase in performance.

Threads in a warp access data from multiple memory spaces during the time of

their execution. These memory spaces are

- Per thread registers (Read-Write)
- Per thread local memory (Read-Write)
- Per grid global memory (Read-Write)
- Per grid constant memory (Read-Only)
- Per grid texture memory (Read-Only)

Fast barrier-synchronization together with low thread creation overhead provide support for very fine-grained parallelism.[29, pg.9-11].

The GPU implementation of phase recovery uses the GeForce 9800 GX2 GPU which features a dual-GPU architecture. The hardware architecture described for the 8800 GTX applies to each card of the 9800 GX2 as well. Of the two GPUs available on the 9800 GX2, only one (device 0) was used for all our benchmarking. The main reason is that SLI must be disabled in order to be able to run CUDA on both GPUs. However, in doing so there can be no direct data transfers between the two GPUs,which means that data must be transfered to the CPU from the first GPU and then copied onto the second GPU. For operations such as the DFT where *all* input data points are needed to calulate even a single output value, using both devices is not an efficient approach.

4.3.3 Development on CUDA

CUDA is a scalable parallel programming model which consists of extensions to the standard C programming language[31]. A typical CUDA application involves a heterogeneous implementation which uses both the CPU and GPU; serial portions

of applications are run on the CPU, and parallel portions are offloaded to the GPU. CUDA can be incrementally applied to existing C applications, i.e. data-parallel portions can be identified and only these need to be modified for the GPU. The number of threads to be created is specified in the *execution configuration* when launching the kernel. The CPU and GPU are treated as separate devices that have their own memory spaces. This configuration also allows simultaneous computation on both the CPU and GPU without contention for memory resources[29, 30]. The CUDA software stack consists of a device driver, an API and its runtime environment, and the CUFFT[33] and CUBLAS[32] libraries for FFT and linear algebra computations respectively. A CUDA application can be developed using the driver or the runtime API environment, but not both at the same time. While the runtime environment is easier to use, the driver offers more control in terms of context and stream management. For the current application, the CUDA runtime was used for development. Listing 4.1 shows the structure of a typical CUDA program.

```
//Allocate memory on the host
float *hostPtr;
hostPtr = (float*) malloc (sizeof(float)*data_size);

//Allocate memory on the device
float *devPtr;
cudaMalloc( (void**) & devPtr, sizeof(float)*data_size);

//Copy data from host to device
cudaMemcpy(devPtr, hostPtr, sizeof(float)*data_size, cudaMemcpyHostToDevice);

//Call kernel
kernel<<<gridSize, blockSize>>>(devPtr);

//Copy result back to host from device
cudaMemcpy(hostPtr, devPtr, sizeof(float)*data_size, cudaMemcpyDeviceToHost);
```

Listing 4.1: A Typical CUDA Program

4.3.4 GPU FFT Benchmarks

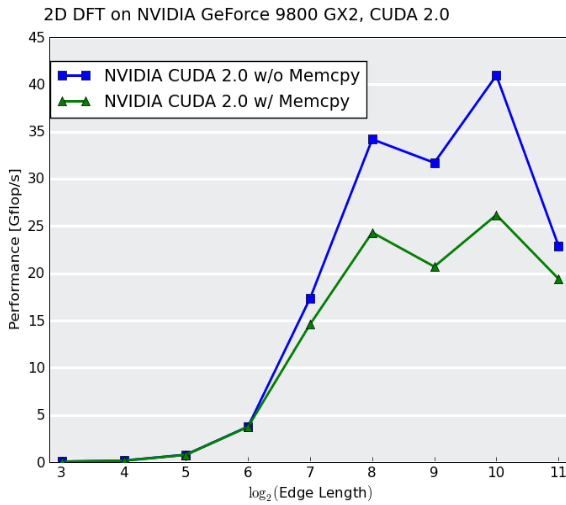

Figure 4.4: 2D FFT on NVIDIA CUDA with GeForce 9800 GX2

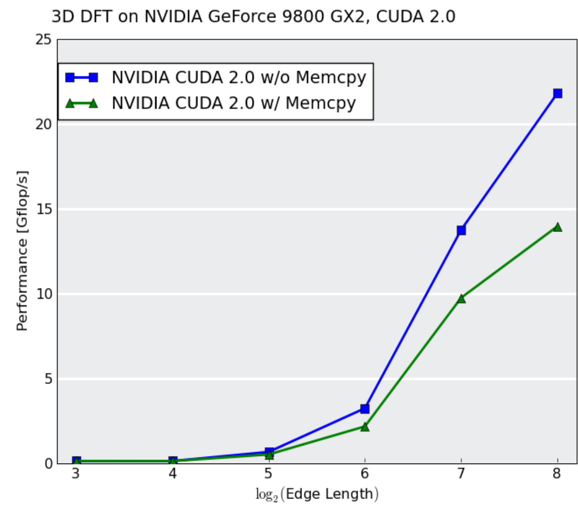

Figure 4.5: 3D FFT on NVIDIA CUDA with GeForce 9800 GX2

The DFT performance of the CUFFT library is shown in Figures 4.4 and 4.5. It can be seen that there is a performance drop of roughly 37.5% in the case of the 2D DFT of size 2^{20} when memory transfers are included in the timings. This is an indication that host-device memory transfers can be potential bottlenecks and should hence be minimized.

4.4 Comparison of CPU and GPU FFT Performance

Figures 4.6 and 4.7 summarize the performance of the libraries investigated so far. It can be seen that the CPU performance becomes comparable to that of the GPU when host-device memory transfers are included in the timings. This further emphasizes the importance of minimizing such data transfers in the GPU implementation.

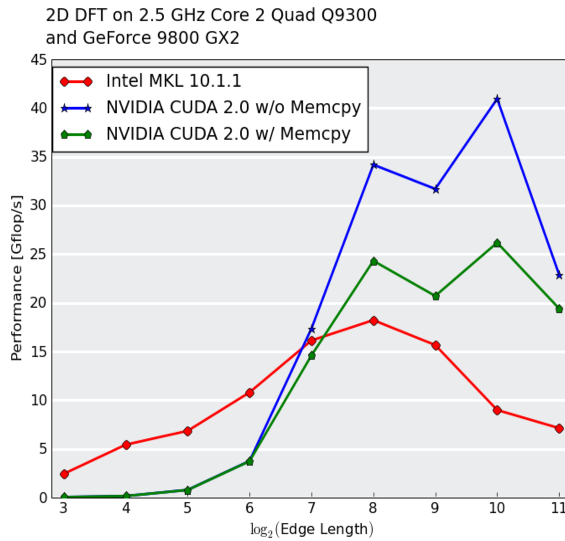

Figure 4.6: Intel MKL and NVIDIA CUDA 2D FFT Comparison

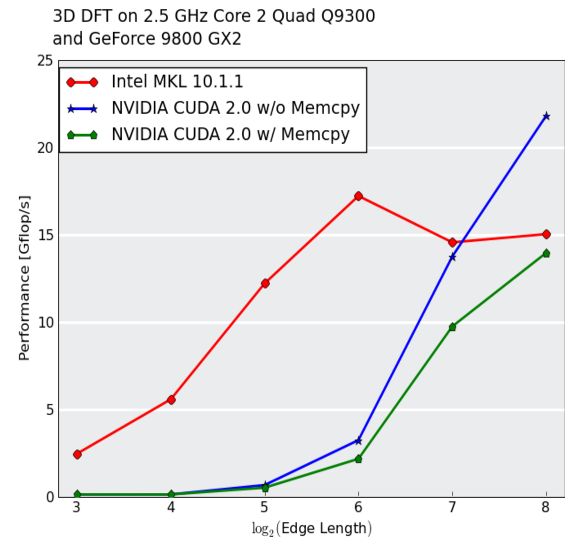

Figure 4.7: Intel MKL and NVIDIA CUDA 3D FFT Comparison

Based on the performance of the FFT libraries investigated so far, the phase recovery algorithm was implemented using two libraries - one with the Intel MKL on the CPU, and the other using the CUFFT on the GPU. The following chapter discusses these implementations.

5. Phase Recovery Performance Comparison

This chapter provides a comparison of phase recovery performance using the Intel MKL and NVIDIA CUFFT platforms. We discuss the tuning strategies used on both platforms and provide a summary of both implementations.

5.1 Benchmarking Methodology

The phase recovery timings were also obtained as the best of 8 runs. The pseudo GFlop/s count was calculated based on an approximation of the total operation count per iteration. Table 5.1 indicates the breakdown of operation count for one iteration of phase recovery.

Table 5.1: Calculation of Total Operation Count for One Iteration of Phase Recovery

Operation	Constituent Operations	Scaling Factor
DFT (4 per iteration)		$4 * 5 * Nlog_2(N)$
$\|G\|$ Calculation	2 Multiplications + 1 Addition + 1 Square Root	$4N$
g' Calculation	2 Multiplications + 4N from $\|G\|$	$6N$
Autocorrelation of g	2 Multiplications + 1 Addition	$3N$
Error Calculation	1 Subtraction + 1 Square Root	$N + 1$
Modification of g'	1 Subtraction + 1 Multiplication	$2N$

The total operation count per iteration, C is calculated as

$$C = 16 * N + 20 * N * log_2(N) + 1 \tag{5.1}$$

. The pseudo GFlops value was calculated as

$$C * 10^9/(Time\ per\ Iteration\ in\ seconds) \tag{5.2}$$

It must be noted from 5.2 that performance of the phase recovery algorithm is $O(Nlog_2(N))$, which is intuitive because the performance of the algorithm is dominated by that of the DFTs.

5.2 Phase Recovery Benchmarks

The MKL implementation of phase recovery is straightforward, with calls to the library to compute the forward and inverse transforms. The remaining sections of the algorithm were vectorized and parallelized using the auto-parallelization and vectorization options available in the Intel® C++ Compiler. The specific compiler flags and options used are listed in Appendix A. Figures 5.1 and 5.2 show performance of

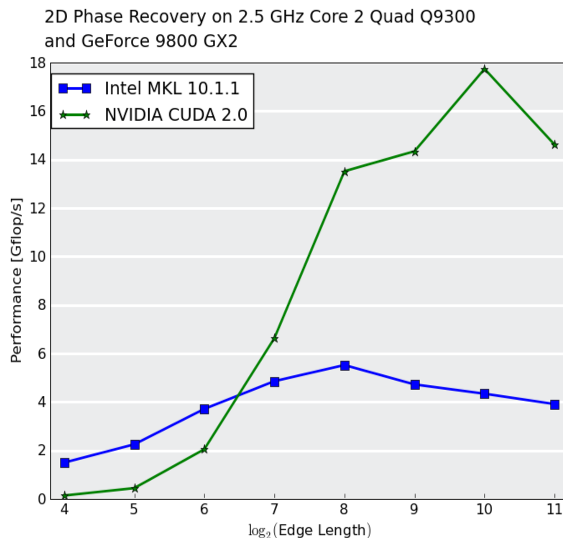

Figure 5.1: 2D Phase Recovery Comparison

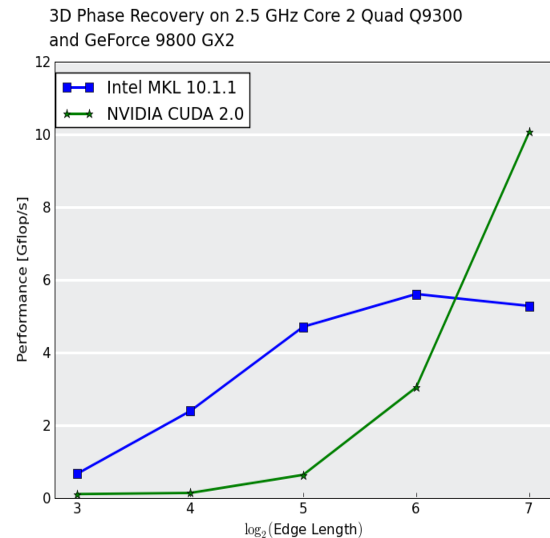

Figure 5.2: 3D Phase Recovery Comparison

the phase recovery algorithm on the CPU and the GPU.

5.3 Performance Optimization Strategies

5.3.1 MKL Implementation

DFT operations in the MKL implementation occur within the call to the library, and are not accessible to the application. Hence, the optimizations described here are with regards to the point-wise operations that occur in between the FFT calls.

The Intel VTune Performance Analyzer™ is a profiler that evaluates applications running on Intel processors. Analysis of the VTune profile results for this implementation showed that other than the DFTs, the "**cabsf**" function which is used to compute the modulus of the DFTs, was consuming the most computation time. The "**cabsf**" function contains a square root, and this is an $O(N)$ operation in this implementation. A potential solution would be to eliminate the square root altogether, and as a result, change the scaling factor operating on the estimate. This would in turn require manipulating the thresholding function to ensure that function constraints are still met. While this approach can improve performance, it requires manipulation of the domain specific attributes of the phase recovery process, and is beyond the scope of this thesis.

The Intel C++ Compiler provides numerous options that can be experimented with to determine the best combination of compiler flags for the implementation. However, this implementation will be slower because the extra linear passes through the data cannot be combined with the FFT code. The extra passes through the data can have a significant impact on the overall performance, especially when the data does not fit in the cache.

5.3.2 NVIDIA CUDA

The DFT computations in the phase recovery algorithm were computed using NVIDIA's CUFFT library. While the source code for CUFFT is not available, a general overview and documentation can be found in[33]. The CUFFT library provides for *batching* of FFT calls, wherein one can compute the DFT of two or more independent datasets simultaneously. However, this feature could not be utilized while implementing the phase recovery algorithm. The reason is that operations such as enforcing function constraints which are interspersed with FFT computations create data dependence between FFT calls. However, there were a number of CUDA-specific features that aided in tuning for optimal performance, which are discussed next.

Shared Memory vs Global Memory: Accesses to global memory can entail a latency of 400-600 cycles[29, pg.51-53]. This is in addition to the 4 cycles needed to issue each instruction for a warp. In contrast, accesses to the shared memory space are as fast as accessing a register as long as there are no bank conflicts[29, pg. 60-67]. A typical shared memory implementation would involve the following steps

- Load one block of data from global memory to shared memory
- Issue a __syncthreads() to ensure that all threads have finished reading data
- Do the required computations on data in shared memory
- Write results back to global memory

As an example, the global and shared memory implementations of the "scaleElements" function for 2D data are as shown in Listings 5.1 and 5.2.

```
1 unsigned int threadIDx = blockIdx.x * blockDim.x + threadIdx.x;
2 unsigned int threadIDy = blockIdx.y * blockDim.y + threadIdx.y;
3 unsigned int index = threadIDx + threadIDy * row;
4
```

```
5  if(threadIDx < row && threadIDy < col) {
6         in_data[index] = in_data[index]*scale);
7  }
```

Listing 5.1: Accessing GPU Global Memory

Lines 1 and 2 in Listing 5.1 show the indexing of threads within one block of a 2D block. The *blockIdx* keyword identifies the position of a block in a grid, which is also two dimensional in this case. The *threadIdx* keyword indexes a thread within a thread block. Line 3 shows the absolute position of a thread within all thread blocks in the grid. It is this value that is used to index data in the *in_data* array. There is no need for a for loop to iterate over all elements in the input array since multiple blocks are executed simultaneously. Instead, *if* statements (line 5) are used to check for boundary conditions - a necessary step because for smaller data sizes, the number of threads created may be greater than the number of data elements being processed.

In contrast, the indexing variables in line 7 of listing 5.2 access elements only within a single block. Data is loaded into shared memory as indicated in lines 8 and 9. Line 10 shows the address translation from shared memory to global memory, i.e. from a block index to an absolute index in the grid. Lines 15 and 16 show processed data being written back to global memory.

```
1         unsigned int in, out, b_index;
2         __shared__ blockx[BLOCK_SIZEX];
3         __shared__ blocky[BLOCK_SIZEY];
4
5         if(xBlock+threadIdx.x < row && yBlock+threadIdx.y < col) {
6                in = xIndex + yIndex * row;
7                b_index = threadIdx.x + (BLOCK_SIZEX) * threadIdx.y;
8                blockx[b_index] = in_data[in].x;
9                blocky[b_index] = in_data[in].y;
10               out = yIndex * col + xIndex;
11               }
12        __syncthreads();
```

```
13
14        if(xBlock+threadIdx.x < row && yBlock+threadIdx.y < col) {
15               in_data[out].x = (blockx[b_index]*scale);
16               in_data[out].y = 0;
17               }
18        __syncthreads();
```

Listing 5.2: Accessing GPU Shared Memory

CUBLAS provided by NVIDIA is an implementation of the Basic Linear Algebra Subroutines (BLAS) on CUDA[32]. The basic model of operation is to create matrix and vector objects in GPU global memory space, call a sequence of CUBLAS functions, and transfer the results back to the host. The *cublasScnrm2* function available in CUBLAS computes the Euclidean norm of a single-precision vector. This was used in the phase recovery implementation to calculate the root mean square value needed to compute the error at each iteration. The data needed to compute the error is already present on the device at this point. CUBLAS operates on data already present on the device, and so using CUBLAS routines eliminates the need for another memory transfer in every iteration.

The CUDA Visual Profiler is a profiler available with CUDA 1.0 and higher. A detailed description of the counters is available in the documentation[34]. The profiler provides summary plots based on the counter selected. As an example, figure 5.3 indicates the percentage of time taken by the kernel functions for 2D phase recovery with input size 256×256. It can be seen that the FFT computations constitute the bulk of computation time - this is expected, since there are four DFT operations performed every iteration. One of the most useful features of the profiler is the detection of uncoalesced global memory loads and stores. Uncoalesced loads and stores are undesirable because bandwidth is wasted in reading memory locations

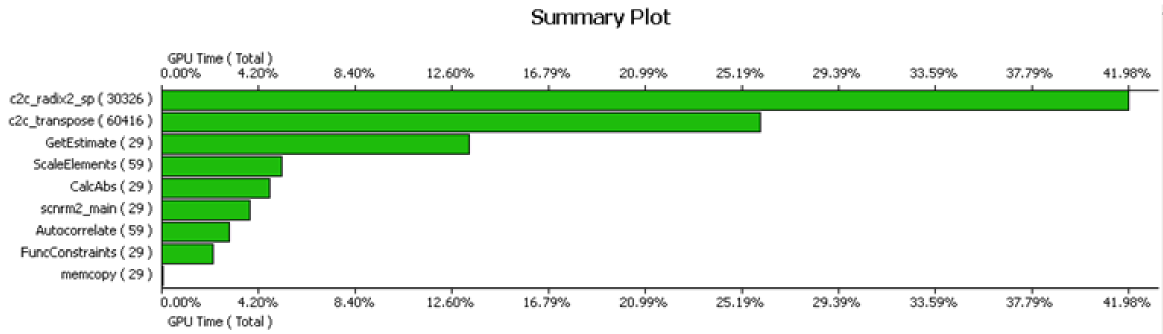

Figure 5.3: CUDA Visual Profiler Summary Plot for 2D Phase Recovery Size: 256×256

whose data is not being used. A description of this problem and solutions is in [29, pg.52-59]. Table 5.2 shows the profiler output for 2D Phase Recovery for edge length 256 and block length 8.

Table 5.2: Profiler Output Indicating Uncoalesced Global Memory Loads and Stores for 2D Phase Recovery

Method	#Calls	GPU μsec	%GPU Time	gld_ uncoalesced	gst_ uncoalesced
c2c_radix4_sp	740	42626.2	34.39	0	0
c2c_transpose	740	31300.4	25.25	0	0
getEstimate	92	22795.4	18.39	0	6.03E+06
scnrm2_main	92	10166.1	8.2	1.51E+06	1336
scaleElements	185	4671.26	3.76	0	0
autocorrelate	185	4565.28	3.68	0	0
funcConstraints	92	4345.22	3.5	0	0
calcAbs	92	3112.86	2.51	0	0
memcopy	93	337.952	0.27		

It can be seen that the getEstimate function has a large number of uncoalesced stores to global memory. In the implementation, this corresponds to the calculation of g'. This function takes as input an array whose real and complex parts are interleaved and stored in contiguous memory locations. Loading data into shared memory from the global memory space is coalesced because both real and complex parts are read. After computation, g' has only real values. Reading back only the real values from the array would cause alternate memory locations to be skipped and so the subsequent stores to global memory become uncoalesced. This problem was resolved by having the kernel return a real valued array half the size of the original input. In doing this, contiguous memory locations are read thereby eliminating uncoalesced stores to global memory.

Summary of GPU Implementation While the individual stages in the phase recovery algorithm are data-parallel, the sequence of operations must remain unchanged because of data dependence between stages. This means that if there is a DFT call between two data-parallel operations (such as calculating the autocorrelation), a separate kernel call is needed once before and one after the DFT operation. The two kernel calls cannot be merged because the DFT operation occurring in between alters the data. Since an FFT call cannot be made from the device, this entails transfer of control back and forth between the host and device. An efficient way to implement phase recovery on the GPU would be to have the operations preceeding and following the DFT computations merged into the DFT which obviates the initiation of separate kernel calls. While this is not possible with the CUFFT library, we identify an alternative implementation to overcome this problem in the last section of this thesis.

5.4 Limitations

While the FFT libraries used in the two implementations discussed so far are highly tuned, the fact remains that phase recovery performance is still limited by that of the library. Figure 5.3 which indicates the breakdown of computation time on the GPU shows that FFTs take up $\approx 70\%$ of computation time. This means that even with the best possible DFT performance, the GPU phase recovery performance can be only 3.33 times faster than obtained in the current implementation.

The phase recovery algorithm consists of a number of point-wise operations (such as calculating the autocorrelation) interspersed with DFT operations. A straightforward optimization of these operations by fusing loops is not possible because of the data dependence introduced by the DFT operations occurring in between. A potential optimization is to fuse these point-wise computations into the DFTs, but inter-procedural optimizations such as this require manipulation at the algorithmic level. The libraries used so far have been treated as black-boxes, and hence this not possible with the current implementations.

In the following chapter, we present a solution to these shortcomings - an implementation of the phase recovery algorithm that does use inter-procedural optimizations. We present DFT benchmark results and provide a comparison of the performance of the phase recovery algorithm with the implementations discussed so far.

6. SPIRAL Implementation

This chapter provides an overview of the architecture of SPIRAL and benchmarks comparing SPIRAL FFT performance with that of Intel MKL. We describe the implementation of the phase recovery algorithm on SPIRAL, and present the performance of the sequential implementation of phase recovery. The complete phase recovery process can be formulated on SPIRAL, allowing the FFTs to be fused with the pointwise computations that are interspersed with the FFTs. The Operator Language (OL)[14] system that is built on the SPIRAL framework provides the infrastructure for the outer loop of phase recovery to be expressed in a mathematical form that can be directly translated into code without the need for an external calling program. This thesis, however, generates code for one iteration of phase recovery, and does not make use of the OL framework. We provide an estimate of SPIRAL's performance with vectorization and parallelization, and show speedup factors of 1.5 times for 2D, and 1.88 times for 3D phase recovery over the MKL implementations.

6.1 SPIRAL Overview

SPIRAL[35] is a program generator that generates platform-optimized code for a large set of linear signal processing transforms. SPIRAL combines a heuristic feedback-driven mechanism with information about the target platform's microarchitecture to generate highly tuned code for a user-specified transform type and size. Figure 6.1 shows the architecture of SPIRAL. A detailed explanation about each block, and more, is available in [35]. Here, we briefly go over the fundamental framework underlying SPIRAL and its applicability to the problem of microstructure reconstruction.

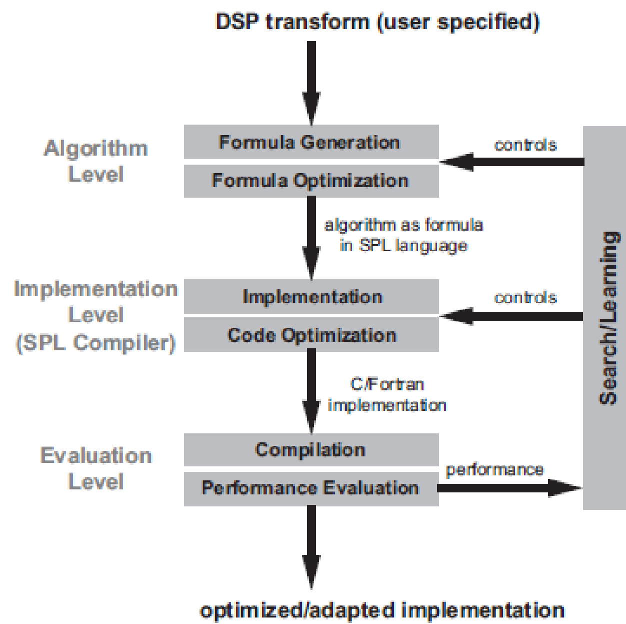

Figure 6.1: The Architecture of SPIRAL *Source:* [35]

6.1.1 SPL and Ruletrees

SPIRAL exploits the domain-specific mathematical structure of signal processing transforms to first derive an algorithmic representation of the transform called a *formula*. A formula is a symbolic representation of the transform algorithm in the *SPL* language. SPL, which stands for Signal Processing Language, is a language specially designed for symbolic computation that makes use of a small set of symbols and constructs to express fast algorithms for signal processing transforms as products of sparse matrices[35, pg. 7-13]. SPL, which is a key component of SPIRAL where all recursions and formulas are expressed efficiently in mathematical form, is the link between the "high level" mathematics of the transforms, and the "low level" code implementations.

SPIRAL generates algorithms for a given transform by applying a set of *breakdown rules*. Breakdown rules specify how to compute a transform by recursively applying another transform (of the same or different type) of a smaller size. The well-known Cooley-Tukey algorithm is an example of this method, where the Discrete Fourier Transform(DFT) is expressed as a matrix factorization involving DFTs of smaller sizes[8, 25]. As an example, the matrix factorization of a 4-point DFT denoted by DFT_4 is shown here

$$y = (DFT_2 \otimes I_2) \cdot T_2^4 \cdot (I_2 \otimes DFT_2) \cdot L_2^4 \cdot x \tag{6.1}$$

where y is the 4×1 output vector and x is the 4×1 input vector. In general, an N-point DFT matrix, represented by DFT_N is factorized as

$$DFT_{RS} = (DFT_R \otimes I_S) \cdot T_S^N \cdot (I_R \otimes DFT_S) \cdot L_R^N \tag{6.2}$$

where $\otimes$ represents the Kronecker product, T is the diagonal twiddle factor matrix and L is the permutation matrix.

SPIRAL applies these factorizations, or breakdown rules recursively to yield numerous potential implementations. SPIRAL creates tree data structures called *ruletrees* to hold this information. The DFT being factorized is called the *nonterminal*. The internal nodes of a ruletree represent a nonterminal of a size smaller than that of the input, and the leaf nodes represent the base cases that terminate the recursion[35, pg. 9-12]. The ruletree for an 8-point DFT DFT_8 factorized into DFT_{42} is as shown in Listing 6.1. Here the nonterminal DFT_8 is factorized into a nonterminal DFT_4 and DFT_2 which is the base case. The DFT_4 is then factorized into two DFT_2 base cases.

```
1 DFT_CT( DFT(8, 1),
2   DFT_Base( DFT(2, 1) ),
3   DFT_CT( DFT(4, 1),
4     DFT_Base( DFT(2, 1) ),
5     DFT_Base( DFT(2, 1) ) ) )
```

Listing 6.1: SPIRAL Generated Ruletree of an 8-point DFT

The corresponding SPL generated for this is

$$(F_2 \otimes I_4) \cdot T_4^8 \cdot (I_2 \otimes ((F_2 \otimes I_2) \cdot T_2^4 \cdot (I_2 \otimes F_2) \cdot L_2^4)) \cdot L_4^8 \tag{6.3}$$

The Sigma SPL (denoted by Σ SPL) component of SPIRAL rewrites the transforms in a ruletree to create iterative, non-overlapping sums by the application of a number of index mapping and simplification rules. The loop fusing and Σ SPL components merge the recursions in a ruletree to improve data locality and reuse by avoiding multiple passes through the data[39, 15]. These constructs are then translated into an intermediate stage C-like code as described in [40] by means of a code generation engine[35, pg. 13-17]. Figure 6.2 shows the phases in SPL compilation. Standard code generation backends then generate the optimized standard C/Fortran code. An SPL formula and the corresponding implementations undergo several levels of optimizations before the actual C code is generated. A detailed description can be found in [35, pg. 13-17] and [39, pg. 53-61].

6.1.2 Vectorization and Parallelization

Vectorization: With the introduction of short vector SIMD (Single Instruction Multiple Data) extensions to their instruction set architectures, many microprocessor vendors are offering a way of introducing fine-grain parallelism in an existing datapath. The most prominent examples are AMD's 3D Now! and Intel's MMX and SSE

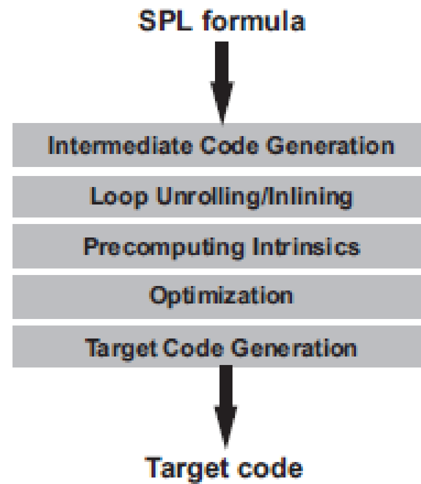

Figure 6.2: The SPL Compiler *Source:* [35]

series. While vectorization indicates a theoretical 4-fold speedup for single precision, and 2-fold speedup for double precision data, developing applications with these extensions requires the use of assembly-level intrinsics, and these intrinsics differ from vendor to vendor. Moreover, realizing maximal speedup is nontrivial because careful attention is must be paid to data access patterns as well.

SPIRAL generates vector code by first generating fully vectorized SPL formulas. An algorithm can be fully vectorized if it can be written as a product of formulas of the form $A \otimes I_v$, where v represents the vectorization width, and special class of permutations that can be performed on vector registers. SPIRAL rewrites SPL formulas in this form and makes use of a vector backend to generate C code with SSEx intrinsics. A detailed description of the mathematical formulations for vectorization in general is in [22, 17]. The SPL rewriting required for this is described in [39, pg. 75-85].

Parallelization: SPIRAL provides SMP support in a way similar to the vectorization approach. A rewriting system manipulates the structure of the transform in a way that eliminates false sharing and achieves load balancing. The mathematical formulations and implementation are described in detail in [16] and [39, pg. 85-96].

6.2 SPIRAL FFT Benchmarks

In this section, we present a comparison of the performance of the SPIRAL generated FFTs to Intel MKL. Figure 6.3 shows SPIRAL sequential, vectorized and vectorized and parallelized performance for 2D data. At data size 2^{16}, the threaded and vectorized performance is roughly 10 times faster than the sequential code. In figure 6.4, which shows a similar comparison with 3D data, it can be seen that again, there is a 10-fold increase in performance at data size 2^{15}. Figures 6.5 and 6.6 provide a comparison of the DFT performance of SPIRAL and Intel MKL. While the 2D performance of MKL and SPIRAL are comparable as indicated by figure 6.5, the 3D data shows an increase of approximately 48% at the same data size. These comparisons provide valuable insights about the performance of phase recovery on SPIRAL, and will be discussed in greater detail while providing an estimate of parallelized and vectorized SPIRAL performance for phase recovery.

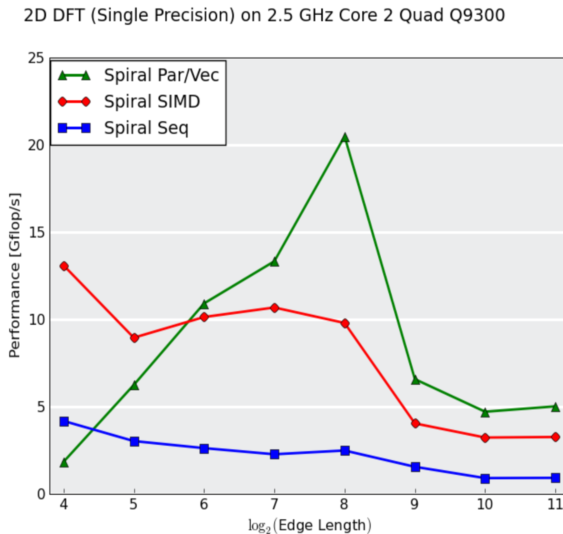

Figure 6.3: 2D FFT on SPIRAL

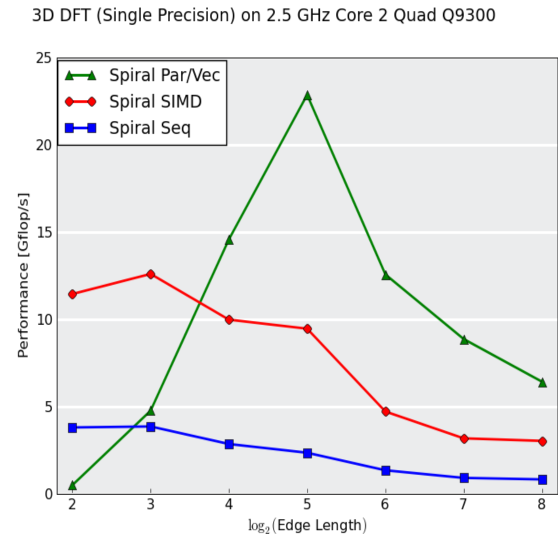

Figure 6.4: 3D FFT on SPIRAL

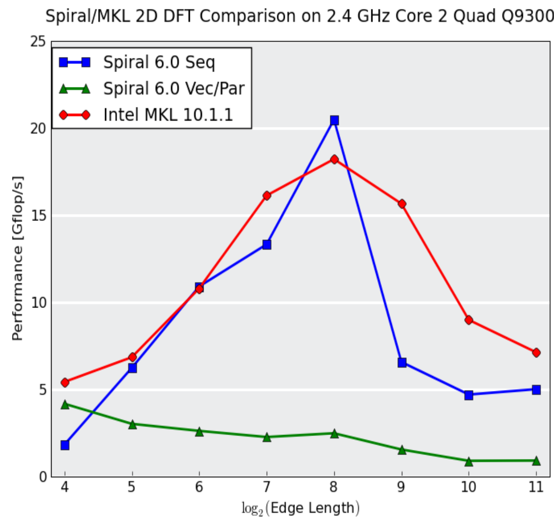

Figure 6.5: 2D FFT Comparison with SPIRAL and Intel MKL

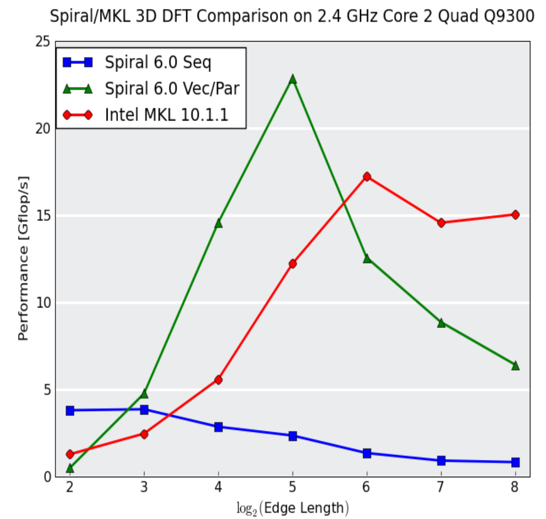

Figure 6.6: 3D FFT Comparison with SPIRAL and Intel MKL

6.3 Microstructure Reconstruction on SPIRAL

To implement phase recovery on SPIRAL, three basic stages were identified within one iteration of phase recovery. These are

- Computing the estimate and applying function constraints
- Finding the error for the computed estimate
- Computing the modified input for the next iteration

The steps to be performed in SPIRAL within each of these stages are as follows:

- Add a new *nonterminal* that accepts the dimension and size of the input as arguments
- Create a *breakdown rule* record for the nonterminal. The breakdown rule indicates how the nonterminal must be decomposed into smaller nonterminals

- Add diagonal objects that perform the intermediate operations
- Create Σ SPL rules for these diagonals so they are commuted with the adjacent gather and scatter matrices and propagated into the innermost loop
- Create a new *codegen* method for each diagonal describing the C-specific constructs that must be created by the code generation engine

The code generated by these three stages computes the error and modified g indicated in 2.16 *only for one iteration*, but code for the complete process can be generated with OL. Figure 6.7 shows the structure of the SPIRAL implementation. Appendix

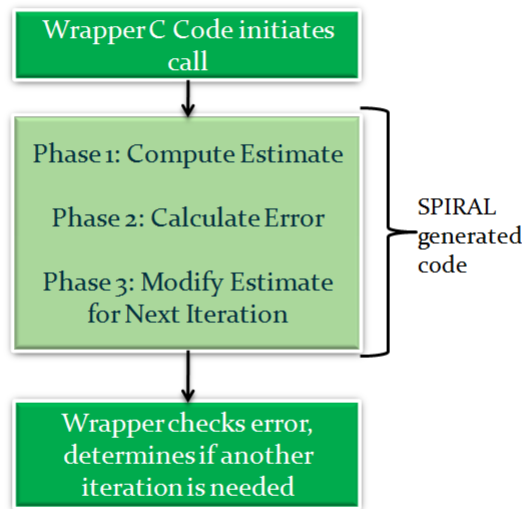

Figure 6.7: Phase Recovery on Spiral

A lists the diagonals and rules added to SPIRAL for phase recovery. An external C program generates the random data and calls the SPIRAL-generated code. Checking the error value to determine whether the stopping criterion has been satisfied is also done outside of SPIRAL. The following section describes each of the three stages in

phase recovery.

6.3.1 Computing the Estimate

The first step in phase recovery is to find an estimate using the known autocorrelation F and randomly generated data g, i.e.

$$G = DFT(g) \tag{6.4}$$

$$G' = \frac{G}{\sqrt{G_{re}^2 + G_{im}^2}} * \sqrt{F} \tag{6.5}$$

$$g' = IDFT(G') \tag{6.6}$$

$$\hat{g'} = Th(g') \tag{6.7}$$

The multiplication and division indicated in 6.5 are point-wise calculations, operating on the real and complex parts of G separately. The known autocorrelation F is a compile-time constant, and so can be represented as a symbolic list in SPIRAL. A new nonterminal PE, and a diagonal $PEDiag$ were added to compute 6.4 through 6.6. Equation 6.8 shows the breakdown rule for PE.

$$PE_Rule = Th * Scale * IDFT * PEDiag * DFT \tag{6.8}$$

The scaling operation scales down each element in the output by the total number of elements. This is needed because a forward DFT followed by an inverse DFT results each element of the input being scaled up by the total number of elements. The thresholding diagonal Th operating on PE then applies the function constraints that cap the value of g' to 1 in 6.7. Listing 6.2 shows the rule tree generated for the PE nonterminal for a 2D input size of 4×4. Here PE_Rule denotes the breakdown rule as indicated in line 1. Lines 2 and 12 indicate the chosen DFT breakdown rules[23].

Lines 2 through 11 represent the inverse DFT in the breakdown rule, and lines 12 through 20 represent the forward DFT.

```
1  PE_Rule( PE([ 4, 4 ]),
2    MDDFT_Dimless( MDDFT([ 4, 4 ], 15, false),
3      MDDFT_Base( MDDFT([ 2 ], 1, false),
4        DFT_Base( DFT(2, 1, false) ) ),
5      MDDFT_RowCol( MDDFT([ 2, 4 ], 7, false),
6        MDDFT_Base( MDDFT([ 2 ], 1, false),
7          DFT_Base( DFT(2, 1, false) ) ),
8        MDDFT_Base( MDDFT([ 4 ], 3, false),
9          DFT_CT( DFT(4, 3, false),
10           DFT_Base( DFT(2, 1, false) ),
11           DFT_Base( DFT(2, 1, false) ) ) ) ) ),
12   MDDFT_RowCol( MDDFT([ 4, 4 ], 1, false),
13     MDDFT_Base( MDDFT([ 4 ], 1, false),
14       DFT_CT( DFT(4, 1, false),
15         DFT_Base( DFT(2, 1, false) ),
16         DFT_Base( DFT(2, 1, false) ) ) ),
17     MDDFT_Base( MDDFT([ 4 ], 1, false),
18       DFT_CT( DFT(4, 1, false),
19         DFT_Base( DFT(2, 1, false) ),
20         DFT_Base( DFT(2, 1, false) ) ) ) ) )
```

Listing 6.2: RuleTree for 2D *PE* of size 4x4

Loop Merging and Σ SPL Rewriting: A diagonal SPL can be propagated into the adjacent iterative sum in one of two ways.

$$(\sum_{j=0}^{m-1} S_j F G_j) D \tag{6.9}$$

$$(\sum_{j=0}^{m-1} S_j F G_j D) \tag{6.10}$$

The first case is when the diagonal gets merged into the iterative sum to its left as shown in 6.9 where S_j is the scatter matrix and G_j represents the gather matrix. The

merged diagonal is as indicated in 6.10.

$$D(\sum_{j=0}^{m-1} S_j F G_j) \tag{6.11}$$

$$(\sum_{j=0}^{m-1} D S_j F G_j) \tag{6.12}$$

The second case is when the diagonal gets merged into the iterative sum to its right as shown in 6.11. The merged diagonal in this case as indicated in 6.12. New rules were created for both cases as per the index simplification and rewrite rules described in [39, p.30-35].

Code Generation: The $PEDiag$ object created here has so far been propagated through all of SPIRAL's formula generation and optimization stages, but no functionality has been attached to it yet. For all of the diagonals added to SPIRAL for phase recovery, this functionality is defined in the code generation stage. The pseudocode for a simple diagonal is as indicated in Listing 6.3

```
loop(i, end, assign(nth(y,i), diag(i) * nth(x,i)))
```

Listing 6.3: Code Generation of a Diagonal

where x is the symbolic representation of the input array, i is the loop index, end represents the upper bound of the loop, y is the symbolic representation of the output array and nth is SPIRAL's internal representation of array indices.

$PEDiag$ requires two additional functionalities that must be incorporated. Firstly, the known autocorrelation that is available in a symbolic list must be multiplied point-wise with the calculated DFT. Secondly, all elements in the computed DFT must be divided by the respective magnitude values. The pseudocode for $PEDiag$ is

as indicated in Listing 6.4.

```
loop(i, end,
        // Multiply with the known autocorrelation
        assign(temp1, diag(i) * nth(x,i)),
        // Compute magnitude of G
        assign(temp2, sqrt((nth(x,i) * nth(x,i)) + (nth(x, i+1)*nth(x, i+1)))),
        // Put the result into y
        assign(nth(y,i), nth(x,i) * temp1 / temp2))
```

Listing 6.4: Code Generation of $PEDiag$

6.3.2 Error Calculation

The next stage in phase recovery is the error calculation, the first step of which is to compute the autocorrelation of g

$$g_a = IDFT(DFT(g) * DFT^*(g)) \tag{6.13}$$

as indicated in 6.13. DFT^* is the complex conjugate of the computed DFT. The root mean square value of the point-wise difference between g_a and F is the required error.

$$\epsilon = \sqrt{\sum_{j=1}^{N} (g_a - F)^2} \tag{6.14}$$

where N is the total number of elements.

```
loop(i, end,
        // Compute g_a
        assign(nth(y,i), (nth(x,i) * nth(x,i)) + (nth(x, i+1)*nth(x, i+1))),
```

Listing 6.5: Code Generation of a $PMul$ Object

A new nonterminal AC (for autocorrelation), and diagonals $PMul$ and PD (indicating Point-wise Multiplication and Point-wise Difference respectively) were created. The SPL equivalent of the row vector is used to sum up the calculated point-wise differences. The resulting breakdown rule for AC is as follows -

$$AC_Rule = RowVec * PD * Scale * IDFT * PMul * DFT \tag{6.15}$$

All stages up until code generation remain the same as in $PEDiag$. The code generation method for $PMul$ is as indicated in Listing 6.5, and that for PD is as shown in Listing 6.6. The two diagonals cannot be implemented in the same method because of the inverse DFT operation that occurs after $PMul$. The code generation method for RowVec already exists in SPIRAL, and so only the sections needed for $PMul$ and PD were added.

```
loop(i, end,
        // Compute point-wise difference
        assign(nth(y,i), (diag(i)-nth(x,i) * diag(i)-nth(x,i)))
```

Listing 6.6: Code Generation of a PD Object

6.3.3 Computation of Modified Input

The computation of g for the next iteration is very similar to that of PE. The nonterminal created is called $PRes$. The estimate g' is obtained the same way as in the case of PE, but the output produced is adjusted with the β value, which is fixed at 1.1. β can also be accepted as a parameter which makes the design more flexible.

6.3.4 The Complete Implementation

A new nonterminal called PR (for Phase Recovey) was created with a breakdown rule PR_Rule as

$$PR_Rule = AC * PE \tag{6.16}$$

which in turn calls 6.8 and 6.15 and returns the error for the current iteration. The pseudo code for the phase recovery process is as indicated in Listing 6.7

```
for(i=0; i<MAX_ITER; i++) {
        // Generate random data g of size N

        // Get the estimate
        PE(g_prime, g);

        // Get error for current iteration
        AC(e, g_prime);

        // Check error to determine convergence

        // Get modified g for next iteration
        PRes(g, g_prime);
}
```

Listing 6.7: Phase Recovery using SPIRAL

The final result is available in g_prime. The formula constructed for phase recovery generates code for an input size that must be known at compile time. The implication is that in order to generate code for a range of input sizes, the SPIRAL formula must be executed separately for each input size. However, this is not a serious limitation as only a small range of sizes are typically used, and the code is generated only once for each platform.

6.4 Phase Recovery Benchmark Results on SPIRAL

SPIRAL code for phase recovery was generated using the formulas described in the previous section. This code was timed in the same way as described in section 5.1. A comparison of SPIRAL phase recovery performance with Intel MKL is shown in figures 6.8 and 6.9. It must be noted that the SPIRAL performance is for a sequential implementation, whereas the MKL implementation is both threaded and vectorized. The dashed lines in both plots indicate the estimated SPIRAL performance with vectorization and parallelization.

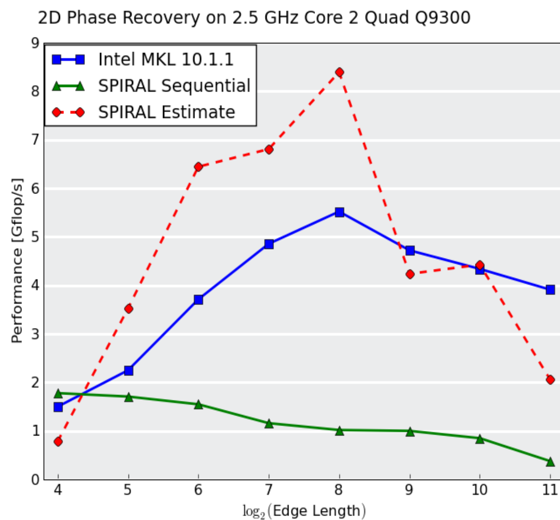

Figure 6.8: 2D Phase Recovery Comparison between Intel MKL and SPIRAL

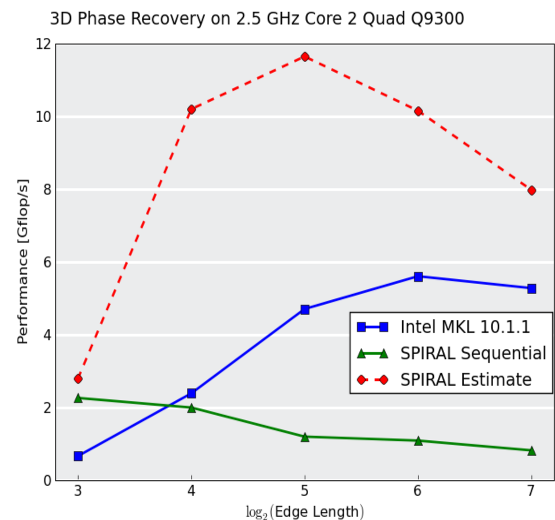

Figure 6.9: 3D Phase Recovery Comparison between Intel MKL and SPIRAL

Estimation of SPIRAL performance: Figure 6.8 shows the SPIRAL sequential phase recovery performance at data size 2^{16} to be 1.018 GFlops/s. The sequential DFT performance from figure 6.8 for the same data size is 2.49 GFlops/s, indicat-

Table 6.1: Estimated Performance of Vectorized and Parallelized SPIRAL Phase Recovery Implementation

	2D Performance in GFlops/s	3D Performance in GFlops/s
SPIRAL Seq DFT	2.49	2.356
SPIRAL Seq Phase Recovery	1.018	1.09
"Scale-Down Factor"	2.49/1.018 = 2.45	2.356/1.09 = 2.16
SPIRAL Par/Vec DFT	20.49	22.863
Estimated SPIRAL Phase Recovery	20.49/2.45 = 8.377	22.863/2.16 = 10.578
Intel MKL	5.52	5.608
Speedup	8.377/5.52 = **1.51**	10.578/5.608 = **1.88**

ing that phase recovery is roughly 2.45 times slower than the DFT. The estimated vectorized and threaded SPIRAL phase recovery performance is obtained by scaling down the corresponding DFT performance by the same amount as in the sequential case. The vectorized and threaded SPIRAL DFT benchmark is 20.49 GFlops/s. The corresponding phase recovery GFlops/s value is $20.49/2.45 \approx 8.377$ GFlops/s. Comparing this to the Intel MKL performance of 5.52 GFlops/s, the vectorized and parallelized SPIRAL implementation of phase recovery is estimated to be 8.377/5.52 $\approx$ **1.5** times faster. A similar analysis on the 3D data at size 2^{15} indicates a speedup of **1.88** over the MKL implementation. The calculations for both 2D and 3D data are summarized in Table 6.1. The estimated performance was calculated for all the other data sizes using the same approach.

7. Conclusions and Future Work

Conclusions In this thesis, we have investigated a variety of hardware platforms and software libraries in order to achieve an efficient, scalable implementation of the phase recovery algorithm. Difficulties with the current implementation were identified and addressed in successive phases, at which performance was evaluated and compared. We have designed high-performance implementations on two separate platforms - a multi-core CPU and a GPU accelerator. The SPIRAL implementation addresses the limitations of these implementations by identifying a mathematical form of the phase recovery algorithm which allows for inter-procedural optimizations.

The DFTs constitute approximately 70% of computation time in the GPU implementation. As described in Section 5.4, we can expect at most, a 3.33-fold speedup even if the FFT time approaches zero. This is mainly because the library is being treated as a black-box, and the point-wise operations are computed in separate function calls. With the SPIRAL implementation, however, the point-wise operations can be hidden in the DFT computations, and so, the phase recovery performance can be approximated by the FFT performance. The potential speedup moving from a sequential implementation to one that is fully vectorized and parallelized is 10 fold. While the initial projections do not achieve this speedup, a significant performance improvement was observed over the other platforms discussed in this thesis.

Future Research The SPIRAL code generated in this thesis is sequential. The first step towards further optimization is a SPIRAL implementation of the algorithm with vectorization and parallelization so that the projected speedup is realized. With a fully vectorized and threaded SPIRAL implementation, additional optimizations

that can completely hide the point-wise operations amid the DFTs are possible, so that the phase recovery performance is approximately that of the FFT performance.

The SPIRAL code generated in this thesis was for one iteration of phase recovery. With Operator Language (OL)[14], a framework that is built on SPIRAL, it is possible to incorporate while-loop constructs so that code is generated for the complete phase recovery process, rather than for just one iteration. This will permit pipelining operations within phase recovery.

The SPIRAL framework can be extended to generate code for the GPU. Interprocedural optimizations between the DFTs and point-wise operations are not possible in the current GPU implementation which uses the CUFFT library. While the modules for such an extension are not completely operational on SPIRAL, some inital experiments were performed in extending SPIRAL for the GPU.

Bibliography

[1] FFTW Home Page. `http://www.fftw.org/`.

[2] GPGPU - General Purpose Computation on Graphics Hardware. `http://gpgpu.org/`.

[3] SPRIAL: Software/Hardware Generation for DSP Algorithms. `http://www.spiral.net/`.

[4] *Numerical Recipes: The Art of Scientific Computing*, pages 504–521. Cambridge University Press, second edition, 1992.

[5] OpenCL Overview. `http://www.khronos.org/opencl/`, February 2009.

[6] *Intel Math Kernel Library Documentation*, August 2008. Version 007 `http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/`.

[7] Heinz H. Bauschke, Patrick L. Combettes, and Russell D. Luke. Phase retrieval, error reduction algorithm, and fienup variants: a view from convex optimization. *J. Opt. Soc. Amer. A*, 19:1334–1345, 2002.

[8] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[9] John Nickolls (NVIDIA Corporation). GPU Parallel Computing Architecture and CUDA Programming Model. In *Session on Multi-Core and Parallelism I, Hot Chips 19, A Symposium on High Performance Chips*, Stanford University, CA, August 19-21, 2007.

[10] CUDA Zone - The Resource for CUDA Developers. NVIDIA Corporation. `http://www.nvidia.com/object/cuda_home.html`.

[11] J. R. Fienup. Phase retrieval algorithms: a comparison. In *Applied Optics, Issue 15*, volume 21, pages 2758–2769, 1982.

[12] J. R. Fienup. Phase retrieval using boundary conditions. *J. Opt. Soc. Am. A*, 3(2):284–288, 1986.

[13] J. R. Fienup and C. C. Wackerman. Phase-retrieval stagnation problems and solutions. *J. Opt. Soc. Am. A*, 3(11):1897–1907, 1986.

[14] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC)*, 2009.

[15] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal loop merging for signal transforms. In *Programming Languages Design and Implementation (PLDI)*, pages 315–326, 2005.

[16] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. FFT program generation for shared memory: SMP and multicore. In *Supercomputing (SC)*, 2006.

[17] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. A rewriting system for the vectorization of signal transforms. In *High Performance Computing for Computational Science (VECPAR)*, volume 4395 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2006.

[18] M. Frigo and S.G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, Feb. 2005.

[19] David T. Fullwood, Stephen R. Niezgoda, and Surya R. Kalidindi. Microstructure reconstructions from 2-point statistics using phase-recovery algorithms. *Acta Materialia*, 56(5):942 – 948, 2008.

[20] R. W. Gerchberg and W. 0. Saxton. A practical algorithm for the determination of phase from image and diffraction plane pictures. *Optik*, 35:237, 1972.

[21] Wen-Mei Hwu, David Kirk, Shane Ryoo, John A. Stratton, and Kuangwei Hwang. Performance of Non-Graphics Applications on the GeForce 8800 and the CUDA Parallel-Programming Environment. In *Session on Multi-Core and Parallelism I, Hot Chips 19, A Symposium on High Performance Chips*, Stanford University, CA, August 19-21, 2007.

[22] J. R. Johnson, R. W. Johnson, D. Rodriquez, and R. Tolimieri. A methodology for designing, modifying, and implementing fourier transform algorithms on various architectures. *Circuits Syst. Signal Process.*, 9(4):449–500, 1990.

[23] Jeremy Johnson and Xu Xu. A recursive implementation of the dimensionless FFT. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2003.

[24] Erik Lindholm and Stuart Oberman (NVIDIA Corporation). NVIDIA GeForce 8800 GPU. In *Session on Multi-Core and Parallelism I, Hot Chips 19, A Symposium on High Performance Chips*, Stanford University, CA, August 19-21, 2007.

[25] C. Van Loan. Computational Framework of the Fast Fourier Transform, 1992.

[26] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. GPGPU: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, New York, NY, USA, 2004. ACM.

[27] NVIDIA Corporation. *NVIDIA CUDA Software Development Kit (CUDA SDK) Documentation*. Version 2.1 `http://developer.download.nvidia.com/compute/cuda/2_1/SDK/CUDA_SDK_release_notes_windows.txt`.

[28] NVIDIA Corporation. *NVIDIA CUDA Visual Profiler 1.0*. Version 2.1 `http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/CUDA_Release_Notes_2.1_windows.txt`.

[29] NVIDIA Corporation. *CUDA 2.0 Programming Guide*, July 2008. Version 2.0 `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`.

[30] NVIDIA Corporation. *CUDA 2.0 Quickstart Guide*, AUGUST 2008. Version DU04165001_v01 `http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/CUDA_2_Quickstart_Guide.pdf`.

[31] NVIDIA Corporation. *CUDA 2.0 Reference Manual*, June 2008. Version 2.0 `http://developer.download.nvidia.com/compute/cuda/2_0/docs/CudaReferenceManual_2.0.pdf`.

[32] NVIDIA Corporation. *Documentation for CUDA BLAS (CUBLAS) Library*, March 2008. Version PG00000002_V2.0 `http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/CUDA_2_Quickstart_Guide.pdf`.

[33] NVIDIA Corporation. *Documentation for CUDA FFT (CUFFT) Library*, April 2008. Version: PG00000003_V2.0 `http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/CUDA_2_Quickstart_Guide.pdf`.

[34] NVIDIA Corporation. *NVIDIA CUDA Visual Profiler 1.0*, May 2008. `http://developer.download.nvidia.com/compute/cuda/2_1/cudaprof/cudaprof.html`.

[35] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[36] W. T. Rhodes, J. R. Fienup, and B. E. A. Saleh. Transformations in optical signal processing. In *SPIE - The International Society for Optical Engineering*, volume 373, page 234, 1984.

[37] Joseph Rosenblatt. Phase retrieval using boundary conditions. *Communications in Mathematical Physics*, 3(3):317–343, 1984.

[38] L. Taylor. The phase retrieval problem. *Antennas and Propagation, IEEE Transactions on*, 29(2):386–391, Mar 1981.

[39] Yevgen Voronenko. *Library Generation for Linear Transforms.* PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, 2008.

[40] Jianxin Xiong, Jeremy Johnson, Robert W. Johnson, and David Padua. SPL: A language and compiler for DSP algorithms. In *Programming Languages Design and Implementation (PLDI)*, pages 298–308, 2001.

APPENDIX A: Hardware and Software Specifications

Hardware Specifications

CPU	Intel® Core™ 2 Quad Q9300
Operating Frequency	2.5 GHz
L1 Cache	32KiB Data + 32KiB Instruction
L2 Cache	6MiB Unified
Architecture	Intel® 64 Technology

GPU	NVIDIA® GeForce™ 9800 GX2
Processor Cores	256 (128 per GPU)
Processor Clock	1.5 GHz
Memory Interface	512-bit
Memory Bandwidth	128 GB/s (64 per GPU)

Software Specifications

Following are the release/build versions of the libraries used in this thesis.

Intel® MKL 10.1.1.019
Intel® C++ Compiler 11.0.081
Intel® VTune Performance Analyzer 6.1 for Linux
FFTW 3.2.1
MATLAB 7.6.0.324 (R2008a)
NVIDIA CUDA 2.1
NVIDIA CUDA Visual Profiler 1.1