Beamforming Adaptive Arrays with Graphics Processing Units

Michael Romer

Abstract—**Beamforming is a signal processing technique by which an array of receivers sensitive to signals from all directions can be processed to form one larger more sensitive receiver that can identify which direction signals originate. Conventional beamforming methods can allow signals from noisy interferers to mask signals of interest if these interferers lie close to those directions to which the beamformer is sensitive. Adaptive beamforming (ABF) attempts to overcome this by minimizing the beamformer's output subject to certain constraints. At its core ABF is an optimization problem, and a robust ABF procedure that consistently provides an optimal solution is computationally expensive. Nevertheless, ABF is of particular interest to the US Navy, where personnel trained to analyze acoustic data from sonar receivers can locate and track quiet targets of interest, e.g. submarines, that may be masked by sources of both ambient and directional noise in the ocean.**

ABF can be implemented to operate concurrently on sets of frequency-independent data, thus making ABF well-suited for parallel processing. Additionally, due to ABF's high density of arithmetic operations, it is a suitable candidate for implementation on modern graphics processing units (GPUs). GPUs have been designed to quickly perform many concurrent arithmetic operations on large amounts of data. Furthermore, as of early 2007 they have reached a point at which they are not only capable of performing general-purpose tasks completely unrelated to graphics but also can be programmed to do such tasks far more easily and more naturally than has previously been possible.

I show a method for parallelizing an existing serial ABF algorithm on an NVIDIA Geforce 8800 GTX, one of the first GPUs to use a generalized stream processor-based architecture. I take a single program, multiple data (SPMD) approach where the same software kernel executes over multiple blocks of frequency-independent data in parallel. Further parallelism is exploited by subdividing each block into smaller subsets, independent of the direction the array is steered, and operating on these concurrently. Although initial results indicate that the GPU-based beamformer yields lower throughput than its serial counterpart, a number of possible optimizations are discussed that can allow the GPU implementation to match, if not exceed, the serial implementation.

I. Review of Conventional and Adaptive Beamforming

This section serves to give to the reader unfamiliar with beamforming the information necessary to understand its fundamentals and why it poses itself as a computationally complex problem. Those who are familiar with beamforming may choose to proceed directly to Section II on page 4.

As a matter of convention, bold faced lowercase and uppercase letters represent vector and matrix quantities, respectively. $\mathbf{A}^T$ and $\mathbf{A}^H$ denote the transpose and conjugate transpose of $\mathbf{A}$, respectively, and, unless specified otherwise, all vectors are assumed to be row vectors. Additionally, the conjugate of a complex number z is given by z^*.

For the purposes of discussion, it is assumed that all applications of beamforming are within the context of underwater acoustics, where the signals of interest are the results of sound waves propagating through the ocean. However, it should be noted that these techniques are not necessarily restricted to acoustic signals and in general may be applied to any type of propagating signal [1].

A. Overview

A beamformer is a system that uses a collection of L omnidirectional acoustic sensors, referred to as an *array*, in order to localize specific signals within the array's environment. Each sensor samples the acoustic signals in its environment at discrete intervals of time. As described in [1], it is common to model the output of sensor l, denoted by $x_l(t)$, in response to a signal $s(t)$ consisting of a single frequency component as

$$x_l(t) = a_l(\theta)s(t), \tag{1}$$

where $a_l(\theta)$ is a precomputed, complex constant of proportionality based upon some assumption of how $s(t)$ travels through the ocean when originating from a direction of arrival (DOA) θ and arriving at sensor l. It should be noted that $a_l(\theta)$ is also dependent upon the signal being considered.

The output of the entire array $\mathbf{x}(t)$ in the presence of a single signal originating from DOA θ is simply the outputs from each of the sensors given by

$$\mathbf{x}(t) = [x_1(t), \ldots, x_L(t)]^T = \mathbf{a}(\theta)s(t). \tag{2}$$

Here $\mathbf{a}(\theta) = [a_1(\theta), \ldots, a_L(\theta)]^T$ is a unit length vector called the *steering vector* or *replica vector* of the signal. Collectively the steering vector represents how the signal is recorded by the individual sensors as it approaches the array from a given DOA. When a signal approaching from a specific DOA is considered, the array is said to be *steered* in that direction, which is sometimes called the array's *steering direction* or *look direction*. If there are M signals in the environment, then the output of the array is simply the superposition of each signal $s_m(t)$:

$$\mathbf{x}(t) = \sum_{m=1}^{M} \mathbf{a}(\theta_m)s_m(t) = \mathbf{A}(\theta)\mathbf{s}(t), \tag{3}$$

where $\mathbf{A}(\theta) = [\mathbf{a}(\theta_1), \ldots, \mathbf{a}(\theta_M)]$ and $\mathbf{s}(t) = [s_1(t), \ldots, s_M(t)]^T$. When in the presence of environmental noise $\mathbf{n}(t)$, the final output of the array is modeled as:

$$\mathbf{x}(t) = \mathbf{A}(\theta)\mathbf{s}(t) + \mathbf{n}(t) \tag{4}$$

The beamformer receives the output of the array, weights each of the component sensor outputs by a corresponding complex scalar w_l, and then sums all of the weighted components together. Thus, the output or *response* of the beamformer $y(t)$ is the following linear combination of each of the sensor outputs:

$$y(t) = \sum_{l=1}^{L} w_l^* x_l(t). \tag{5}$$

Note that it is by convention that each sensor output is multiplied by the complex conjugate of the corresponding weight [2]. By letting $\mathbf{w} = [w_1, \ldots, w_L]$, (5) can be written more succinctly in vector notation as:

$$y(t) = \mathbf{w}^H \mathbf{x}(t). \tag{6}$$

In beamforming the individual signals and their DOAs are not known *a priori*. Furthermore, these signals typically occur in the presence of noise from both ambient and directional sources in the ocean. Ambient noise is simply the noise inherent in the environment that appears to arrive from all directions. Directional noise sources, on the other hand, emit discrete sounds and can be anything from merchant ships and warships to whales and other marine life that inhabit the environment being monitored. In US Naval applications, it is common to identify those signals that are of interest, e.g. submarines, and to have the beamformer compute weights that give the best DOA estimate for each signal.

Finding the optimal set of weights generally consists of assuming that the signal of interest approaches the array from a number of different directions and steering the array in each of these directions by computing a corresponding weight vector. The output power of the beamformer $P_{BF}(\mathbf{w})$ is then measured using each computed weight vector as follows:

$$\begin{aligned} P_{BF}(\mathbf{w}) &= |y(t)|^2 = |\mathbf{w}^H \mathbf{x}(t)|^2 \\ &= \mathbf{w}^H \mathbf{x}(t) \mathbf{x}^H(t) \mathbf{w} \\ &= \mathbf{w}^H \mathbf{R} \mathbf{w}. \end{aligned} \tag{7}$$

Here $\mathbf{R}$ is typically termed the *cross spectral matrix* (CSM). The weights that produce the maximal output power correspond to the steering direction that is the best estimate for the signal's DOA. Note that because it is typical to consider many different signals, each of which could possibly come from a number of directions, weights must be computed and evaluated on a per frequency, per direction basis in order to properly find the optimal set.

Different beamforming algorithms apply different sets of constraints and optimality conditions in order to compute the optimal weight vector. An extensive overview of some of the different types of beamformers is given in [2], but for the purposes of this project only two broad categories of beamformers are considered: *conventional beamformers* and *adaptive beamformers*.

B. Conventional Beamforming

Conventional beamforming (CBF) represents a classification of beamforming algorithms whose computed weight vectors do not depend on the output of the array. Instead conventional beamformers use what is known as a delay and sum technique, where each sensor output is delayed in order to bring all of the outputs in phase with each other. The delayed sensor outputs are summed together, and the result is normalized, thus allowing each sensor to maximally contribute to the output of the beamformer.

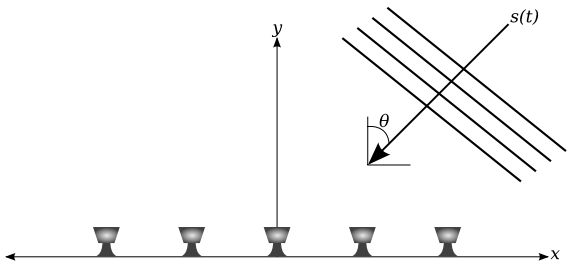

Figure 1. A sinusoidal plane wave $s(t)$ approaching a uniform linear array (ULA) from a DOA θ. Note that the wavefront arrives at each sensor at slightly different times, resulting in the sensor outputs becoming out of phase with each other.

To illustrate, consider an array of sensors arranged in a line spaced equidistantly apart. Such an array is commonly known as a *uniform linear array* (ULA). Figure 1 shows a top-down view of a sinusoidal wave $s(t)$ approaching the ULA from a DOA θ. Note that the wavefront will arrive at each sensor at different times, thus the output of each sensor will be out of phase with all other sensor outputs.

In the absence of any weighting, the beamformer would sum each of the out-of-phase sinusoids together, resulting in constructive and destructive interference that produces an output that differs from the original signal. However, if the output of each sensor is weighted such that all sensor outputs are brought into phase with each other, then the beamformer output will match the original signal observed and total output power will be maximized.

It can be shown that an optimal set of power maximizing weights $\mathbf{w}_{CBF}$ can be found that depends solely on the steering vector $\mathbf{a}(\theta)$ [1]:

$$\mathbf{w}_{CBF} = \frac{\mathbf{a}(\theta)}{\sqrt{\mathbf{a}^H(\theta)\mathbf{a}(\theta)}}. \tag{8}$$

When these weights are applied to (6), they effectively add a time delay to each of the sensor outputs, thus bringing them into phase with each other, and normalize the beamformer's response. The response of the beamformer in the DOA of the observed signal is thus maximized, and as a result of this time delay, a spatial filtering occurs that either attenuates or completely nullifies signals approaching from other DOAs.

The normalized response of a conventional beamformer when a given set of CBF weights is applied can be plotted as a function of DOA as shown in Figure 2. These *beam patterns* are comprised of a *main lobe* that coincides with the DOA of the signal of interest and realizes maximum response, as well as a number of *sidelobes* that only realize partial response. A number of *nulls* are also generated where the beamformer effectively has no response.

The advantage of CBF is that the weights can be computed

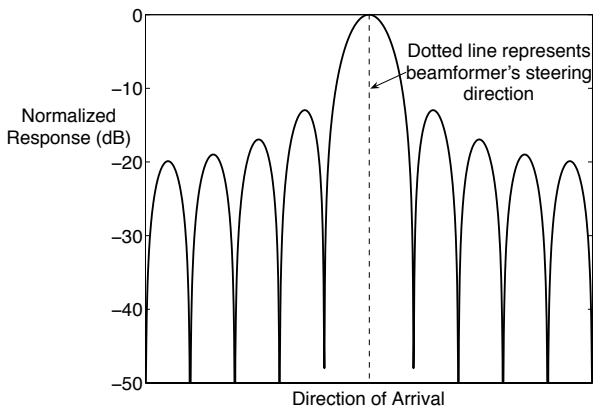

Figure 2. A beam pattern is comprised of a number of lobes. The lobe with the greatest response, called the *main lobe*, is aligned along the steering direction of the array and typically corresponds to the best estimate for the signal's DOA. Additional *sidelobes* lie along those directions to which the array is less sensitive, and the nulls between any two lobes represent those directions to which the array has effectively no response.

relatively quickly from the steering vectors alone. Furthermore, since the weights do not depend on the sensor data, they can be computed in advance and applied as many times as necessary. However, because the steering vectors are precomputed on a per frequency, per direction basis, the beam pattern for a given set of steering vectors and sensor data set is fixed. As a result, if a noisy interferer lies just off the DOA of the targeted signal, the beamformer will indiscriminately be sensitive to both the target and the interferer. In naval applications, targets such as submarines tend to emit low level signals, and the presence of noisy interferers in the environment more often than not results in the interfering signal dominating the target signal in the beamformer's output.

C. Adaptive Beamforming

Adaptive beamforming (ABF) attempt to overcome the problems associated with CBF by computing weights that reduce the beamformer's response to loud interferers while preserving response in the steering direction. Adaptive techniques typically correlate the output of the array with the weights computed in order to derive the optimal set of weights [3]. Many ABF algorithms exist, a number of which are discussed in detail in [4]-[8] and whose individual performances and merits are evaluated and compared in [9]-[13].

The beamformer selected for parallelization belongs to a class known as *linearly-constrained adaptive beamformers*. These beamformers compute weights such that the total output power is minimized while imposing some linearly constraint on the beamformer's response. As suggested in [14], the constraint imposed is to maintain unity response in the steering direction of the array; that is, signals arriving along the steering direction are neither amplified nor attenuated. Thus these beamformers are typically termed *minimum variance, distortionless response* (MVDR) beamformers whose optimality condition is expressed as

$$\min_{\mathbf{w}} \mathbf{w}^H \mathbf{R} \mathbf{w} \text{ subject to } \mathbf{w}^H \mathbf{a}(\theta) = 1. \tag{9}$$

Using Lagrange multipliers (9) can be solved to yield the following optimal ABF weights [15], [16]:

$$\mathbf{w}_{ABF} = \frac{\mathbf{R}^{-1}\mathbf{a}(\theta)}{\mathbf{a}^H(\theta)\mathbf{R}^{-1}\mathbf{a}(\theta)}. \tag{10}$$

Note that the calculation for the ABF weights consists of significantly more operations than that of the CBF weights given in (8). In general, for ABF the cross-spectral matrix $\mathbf{R}$ must be computed and inverted, which is commonly performed by estimating $\mathbf{R}$ from the sensor data and employing a singular value decomposition (SVD) in order to compute its pseudoinverse [41]. In practice, the SVD is favored over other means of matrix decomposition for a number of reasons, namely:

- The approximation for the cross-spectral matrix $\mathbf{R}$ is not guaranteed to be well-conditioned, thus procedures such as the SVD are highly favored for their numerical robustness.
- Certain variations of MVDR beamformers, such as Dominant Mode Rejection (DMR), adapt against only the D loudest interferers in the environment, where D is a parameter set prior to beamforming and the interferers correspond to the D largest eigenvalues of the cross-spectral matrix $\mathbf{R}$. The SVD has the advantage over other methods of solving for eigenvalues that the eigenvalues of $\mathbf{R}$ can be calculated directly from the singular values of the sensor data $\mathbf{x}(t)$ (recall that $\mathbf{R} = \mathbf{x}(t)\mathbf{x}^H(t)$, where $\mathbf{x}(t)$ is the sensor data, and the squares of the non-zero singular values of $\mathbf{x}(t)$ are the non-zero eigenvalues of $\mathbf{R}$).

For an mxn matrix the SVD has a known runtime complexity of $O(min(mn^2, nm^2))$, and since these calculations must be performed on a per frequency, per direction basis, ABF weight derivation is considered a computationally expensive operation.

By imposing a distortionless response constraint in the steering direction, the beamformer is forced to steer nulls in the directions of loud interferers not along the steering direction in order to minimize output power. However, (10) assumes the steering direction coincides with the DOA of the signal of interest, which is frequently not the case in practice. Errors in the steering vectors known as *mismatch* can actually cause the adaptive beamformer to steer a null in the direction of the desired signal, a problem known as *squinting*.

Different methods for dealing with mismatch have been proposed in the literature, including [7] and [8]. The beamformer selected for parallelization uses the latter approach, which imposes an additional *white noise gain constraint* (WNGC) on the optimization in (9). This approach attempts to control the adaptivity of the beamformer by introducing a penalty for ABF weights that generate large sidelobes. The additional constraint modifies the weight computation in (10) to be:

$$\mathbf{w}_{ABF} = \frac{(\mathbf{R} + \varepsilon\mathbf{I})^{-1}\mathbf{a}(\theta)}{\mathbf{a}^H(\theta)(\mathbf{R} + \varepsilon\mathbf{I})^{-1}\mathbf{a}(\theta)}. \tag{11}$$

Here $\mathbf{I}$ is the identity matrix, and ε is a parameter that controls the adaptivity of the beamformer. If $\varepsilon = 0$, then (11) becomes the ABF weight equation given in (10), and as $\varepsilon \to \infty$,

the weights computed tend toward the CBF weights in (8). Ideally the beamformer computes a value for ε that satisfies both constraints, thus yielding a high amount of adaptivity while avoiding squinting; however, determining a suitable value for ε requires an iterative search such as Newton's method, adding further to the complexity of deriving optimal ABF weights [10].

II. Previous Means of Hardware Acceleration

In order to cope with ABF's computational complexity and the large data sets generated by acoustic arrays, it used to be the case that specialized signal processing hardware had to be employed to realize any sort of practical performance. While this approach is still the only viable option for certain applications, the advancements in technologies such as multicore processors, multiprocessor systems, and field programmable gate arrays have made these technologies attractive alternatives for lower-budget and smaller-scale operations.

The following is an survey of these technologies, their advantages and disadvantages, and comparisons of their relative performance in beamforming related applications, leading to a justification for the use of graphics processing units as a means of hardware acceleration.

A. Application Specific Integrated Circuits

Prior to 1990, military-related sonar applications predominantly used custom-made, dedicated hardware to perform much of the computationally intensive signal processing procedures necessary for these systems [17]. These pieces of hardware, also known as application specific integrated circuits (ASICs), are designed specifically for their intended use, thus they are capable of yielding higher performance than most other alternatives. In ABF applications, ASICs have been used to realize speeds as much as 30-50 GFLOPS [18], [19]. Also, due to their special-purpose nature, they can be built on a smaller and more compact scale than general-purpose hardware thus yielding better performance in terms of power consumption. It has been observed in ABF scenarios that ASICs are able to have as much as an order of magnitude less power consumption than field programmable gate arrays [20].

Despite their superior performance capabilities, ASICs are prohibitively expensive for many small- to medium-scale operations. This is due in part to the high non-recurring fixed costs spent in producing a suitable ASIC [21]. Typically a team of engineers is needed in order to create a design for the desired circuit and perform extensive hardware verification to ensure that it will perform according to its specifications, all of which adds substantially the overall cost of producing an ASIC. Additionally, these devices provide no means of reprogrammability, thus limiting their potential for reuse in varying situations.

B. Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) provide a compromise between ASICs and general-purpose processors. Modern FPGAs possess some of the features that make ASICs attractive such as increased speed and lower power consumption over general-purpose processors while making it possible to reconfigure their internal logic structure on the fly by uploading user software to the device [22]. These devices have been successfully programmed to execute core ABF functionality with observed speeds of 20 GFLOPS [20], [23].

Despite their advantages, FPGAs are still not as efficient as ASICs in terms of size, power, and raw performance [24]. Additionally, since it is common to write FPGA code in a hardware description language such as Verilog, software development times can be significantly longer than for a conventional general-purpose system [21].

C. General-Purpose Processor Systems

For many years the common trend in the general-purpose processor market has been to scale up the raw clock speed of a single processor, but, as it has been pointed out in [25], in recent years this method has reached a brick wall of sorts. This is the result of the culmination of three separate problems that have manifested as technology scaling has increased over time:

1) Single core processors cannot increase in transistor density and still be made power efficient.
2) As processor clock speed increases over time, latency incurred by accessing DRAM increases proportionally.
3) Beyond techniques such as branch prediction and out-of-order execution, there are diminishing returns for trying to exploit any further instruction level parallelism.

It has been argued that because of these problems it is all the more important that general-purpose systems move away from a uniprocessor approach and head towards making a multicore/multiprocessor paradigm the norm.

As has been pointed before, the problem of computing optimal weights for beamforming is one that must be done on a per frequency, per direction basis. This naturally lends itself well to parallelization. Many different means of implementing beamforming on parallel systems such as clusters and networks of distributed computers have been explored in [3] and [26]-[29] and have succeeded in gaining increased performance over their serial counterparts; however, the performance of these systems still does not match that of those systems based on either ASICs or FPGAs.

Another disadvantage is that in certain military applications where physical space is a premium and power requirements are stringent, e.g on an actual ship, these multiprocessor systems can be impractical or simply infeasible due to their higher power consumption and the larger physical size of their necessary peripheral hardware. Nevertheless, advances in general-purpose systems, their significantly lower costs compared to ASICs, and the relative ease of developing software for them compared to FPGAs have led defense contract sponsors to invest more heavily into these commercial off-the-shelf (COTS) systems within the past decade [30].

D. Graphics Processing Units

The use of graphics processing units (GPUs) for general-purpose computations began as an area of research in the early

	NVIDIA Geforce 8800 GT	Intel Xeon 5160
Number of Processing Elements	112	2
Clock Speed per Element (GHz)	1.5	3.0
Memory Interface	256 bit	64 bit
Memory Clock Speed (MHz)	900	1333
Peak Computation Performance (GFLOPS)	336	48
Peak Memory Bandwidth (GB/sec)	57.6	21.3
Max Power Consumption (watts)	110	80
Cost	$200	$879
FLOPS per Dollar	1.68	0.05
FLOPS per watt	3.05	0.6

Table I
COMPARISON OF NVIDIA GEFORCE 8800 GT GPU TO INTEL XEON 5160 CPU.

2000s when graphics cards with programmable units called *shaders* were first introduced [31], [32]. GPUs have since garnered the attention of many researchers who have tried to use them to accelerate different problems such as linear algebra solvers, traffic simulations, and data mining [33], [34].

What makes the GPU a compelling platform to explore is its potential computational power. Over the years in response to the growing demand for more realistic graphics in the interactive gaming market, commercial GPUs have transformed into highly data parallel devices with high-bandwidth memory controllers. This has ultimately resulted in a parallel coprocessor capable of outperforming general-purpose CPUs in terms of both FLOPS and memory bandwidth [33], [35]. Table I summarizes the differences between NVIDIA's Geforce 8800 GT graphics card and an Intel Xeon 5160 dual-core server class CPU, each based on NVIDIA's G92 graphics architecture and Intel's Core microarchitecture, respectively. Hardware specifications were obtained and performance metrics derived from product specifications given by each respective manufacturer [42], [43], and costs given are based on current retail prices as of April 2008.

Here a processing element refers to a core in the CPU or a shader in the GPU. The shaders, or stream processors as they are called within the context of NVIDIA's G80 and G92 graphics architectures, are purportedly capable of issuing one floating point multiply and add (MAD) instruction and one floating point multiply (MUL) instruction and retiring their results each clock cycle, yielding 3 floating point operations per cycle or 504 GFLOPS [37]. However, since the conditions necessary for consistent issuing of the second MUL instruction are not known, it is a more reasonable to assume that on average only the MAD instruction is issued, thus resulting in the 336 GFLOPS of peak throughput.

As can be seen from Table I, GPUs potentially offer superior performance both in terms of FLOPS per dollar and FLOPS per watt consumed. Furthermore, they are also capable of offering more performance per unit volume. A Dell PowerEdge M600 blade server stocked with dual Intel Xeon E5450 3.0 GHz quad-core processors offers 192 GFLOPS of peak performance, whereas an NVIDIA Tesla Server stocked with four GPUs offers a peak compute power of approximately 2,000 GFLOPS [38]. Both solutions fit in a 1U form factor, thus the GPU has the capability of providing much more computational power than general-purpose systems in the same amount of space.

In addition to their raw computational power, GPUs based on NVIDIA's G80 and subsequent architectures also share some of the same advantages that general-purpose processors have over ASICs and FPGAs, namely their lower cost and the relative ease of developing software for the hardware. Prior to the G80, programming for the GPU required one to cast the problem being solved in terms of a graphics-related problem. Data sets would be treated as either texture or vertex data, small programs, sometimes called *kernels*, written in a graphics-centric shading language would execute over the "graphics" data on the GPU, and the results would be written or "rendered" to another texture as output. Nowadays, with the use of NVIDIA's *Compute Unified Device Architecture* (CUDA), developers can write GPU kernels using simply C with a minimal set of extensions that declare those portions of the code that execute on the GPU and those portions that execute on the CPU. This allows developers to write software for the GPU in a manner that is more natural since it is not necessary to reinterpret the problem at hand in terms of graphics.

Despite their strong advantages, GPUs do pose their own problems. In general the GPU programming model is inconsistent across various GPUs. For instance, NVIDIA GPUs based on architectures older than G80 cannot make use of CUDA, and developing software for these GPUs requires that the kernels are written in some sort of shading language. Furthermore, if portability across GPUs is important, it is necessary that the GPU kernels are written using a graphics API in conjunction with some shading language, e.g. OpenGL with GLSL.

Another area of concern for many researchers in the scientific computing field has been compliance to the IEEE 754 specification for floating point numbers. While most GPUs conform specification for the *storage* of floating point numbers, the *arithmetic* performed is not fully compliant. In the case of G80-based GPUs, a number of deviations from the IEEE 754 specification are given in [39], some of which include differences in the algorithm used for the division of two floating point numbers, the rounding modes used, and the treatment of denormalized numbers.

In spite of these disadvantages, the potential computational power of a GPU warrants further exploration. Their highly parallel design and their strong performance in quickly executing floating point operations make GPUs a particularly suitable target hardware for the implementation of an adaptive beamformer. Furthermore, due to their relative strengths versus

other means of high performance computing, GPUs provide a solid compromise in terms of throughput, power consumption, size, and price that makes them an attractive means of hardware acceleration for small- to medium-scale applications.

III. Introduction to the NVIDIA Geforce 8800 GTX and CUDA

A. Hardware Organization

For this project, I use an NVIDIA Geforce 8800 GTX graphics card, which is based on the G80 graphics architecture developed by NVIDIA and implemented on graphics cards released as early as the beginning of 2007. Table II lists many of the hardware specifications of the Geforce 8800 GTX.

	NVIDIA Geforce 8800 GTX
Number of Stream Processors	128
Clock Speed per Stream Processor (GHz)	1.35
Memory Interface	384 bit
Memory Clock Speed (MHz)	900
Peak Computation Performance (GFLOPS)	345
Peak Memory Bandwidth (GB/sec)	86.4
Max Power Consumption (watts)	280

Table II
NVIDIA Geforce 8800 GTX Hardware Specifications

The G80 architecture marks a significant departure from previous graphics architectures and introduces a number of changes, namely the support of what is known as the *unified shader model*. Rather than following the traditional GPU pipeline of having different processor groups for each stage of rendering, e.g. vertex processing and pixel fragment shading, the unified shader model uses a collection of generalized scalar stream processors that can be used for any of the operations executed in the traditional pipeline. This allows for potentially better utilization of the graphics hardware because the stream processors can be allocated for various tasks on an as-needed basis by the GPU. More information on the G80 architecture and its unified shader architecture can be found in [37].

Figure 3 shows the hardware layout of the Geforce 8800 GTX. This GPU specifically consists of 128 stream processors divided into 16 groups of eight stream processors each. Each group is termed a *multiprocessor*, and each multiprocessor's set of eight stream processors executes simultaneously in a SIMD fashion. Each multiprocessor also consists of a small 16 KB portion of on-chip memory called the *parallel data cache* which is divided into 16 equally sized banks. A multiprocessor's parallel data cache is only accessible by those stream processors located on the multiprocessor and implements the shared memory space among the processors, thus the parallel data cache is sometimes also simply referred to as *shared memory*. Each multiprocessor also consists of on-chip read-only constant and texture caches, which are designed to speed up transactions to and from constant memory and texture memory, respectively. In addition to the different types of on-chip memory, the Geforce 8800 GTX has a large amount of DRAM not located on any of the multiprocessors and is globally accessible by any stream processor, hence its name *global memory*.

One of the main advantages of putting the shared memory in such close proximity to the individual stream processors is that it allows for fast memory transactions to and from the shared memory space. It takes, on average, four cycles for a stream processor to issue either a read or write instruction to memory, and assuming no bank conflicts, accessing shared memory takes no longer than accessing a register. On the contrary, accessing data residing in global memory takes an additional 400-600 cycles [39].

It is also important to note that the GPU's global memory is *not* cached. It is therefore imperative to stage as much data within the shared memory in order to minimize latency incurred from accessing global memory.

B. NVIDIA's Compute Unified Device Architecture

NVIDIA's Compute Unified Device Architecture, or more commonly referred to as CUDA, makes it possible for developers to access the graphics hardware for general-purpose computations. CUDA treats the GPU as a *compute device* that can execute a large number of threads in parallel. Figure 4 illustrates the organization of threads executed on the GPU. Software called kernels are written for the device in such a manner that each kernel makes use of a large number of threads when executed. In CUDA, all threads used by a single kernel are organized into a single compute *grid*, with the grid being divided into a number of *thread blocks*. Each thread block consists of an equal number of threads, and individual thread blocks are issued to run on a single physical multiprocessor on the GPU. As a consequence, threads in different thread blocks cannot communicate or synchronize with each other.

CUDA kernels are written in either C or C++ and make use of the CUDA Device and/or Driver APIs. The kernels make use of a small number of extensions added to C/C++ that specify whether the code is to run on the GPU or the CPU. In CUDA, the GPU and CPU are referred to as the device and host, respectively, with functions designated for the device being declared with the either the `__global__` or `__device__` type qualifier. Host functions may optionally specify the`__host__` type qualifier, but if no type qualifier is given, it is automatically assumed to be compiled for the host. Table III lists all available type qualifiers along with relevant information for each.

Once a CUDA kernel has been implemented, it is compiled using NVIDIA's `nvcc` compiler driver [44]. The driver will perform the preprocessing necessary to separate host code from device code and send each to the appropriate compiler. Host code is compiled using the host platform's compiler tools, and device code is sent to a specialized compiler provided

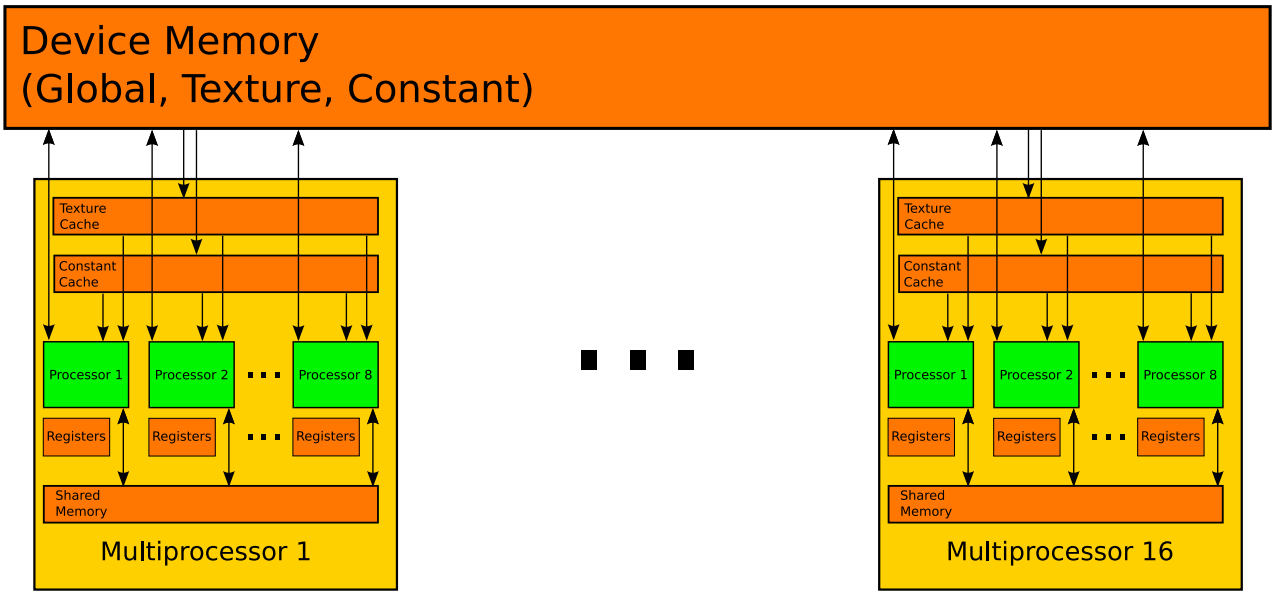

Figure 3. Hardware organization of the Geforce 8800 GTX. The GPU consists of a large global DRAM and 16 multiprocessors. Each multiprocessor has eight scalar stream processors that can access any of the different memory caches available on the multiprocessor, including a bank of registers per processor, a shared parallel data cache, a texture cache for texture memory, and a constant cache for constant memory. Note that there is no caching for data stored in the global memory.

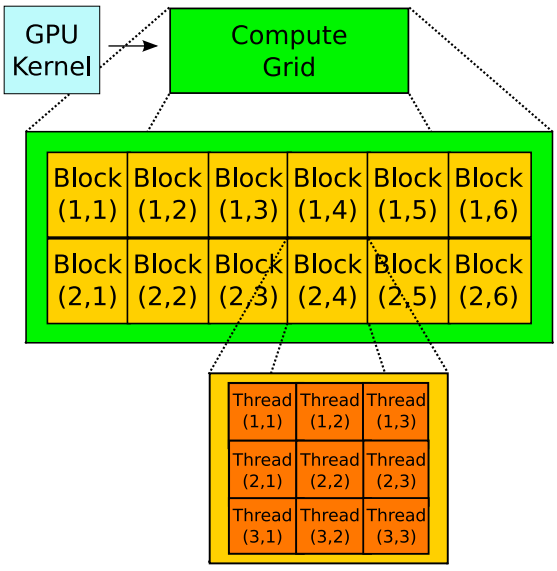

Figure 4. A single GPU kernel is executed as a batch of threads. The threads are organized as a single compute grid, which is divided into a number of thread blocks indexed by either a 1- or 2-dimensional thread block ID. Each thread block contains the same number of threads, each of which are accessed by either a 1-, 2-, or 3-dimensional thread ID.

by NVIDIA. This compiler will form a binary image that is typically embedded with the final compiled binary and is loaded by the CUDA runtime driver for execution on the GPU.

Further information on the CUDA software model and the extensions and APIs provided is given in [39].

Type Qualifier	Callable By	Executed On
`__global__`	Host	Device
`__device__`	Device	Device
`__host__`	Host	Host

Table III
SUMMARY OF TYPE QUALIFIERS FOR CUDA FUNCTIONS

IV. IMPLEMENTING ADAPTIVE BEAMFORMING ON THE GRAPHICS PROCESSING UNIT

Following is a discussion of the ABF algorithm being parallelized, as well as the details of the final system implemented.

A. Implementation Overview

Prior to the execution of the beamformer, the input sensor data is transformed via a Fast Fourier Transform (FFT) in order to divide the input sensor data into its frequency components. The basic ABF algorithm then divides the input data into frequency-independent blocks known as CSMs. In this implementation, the CSM structures represent only an approximation of the true cross spectral matrices used in deriving ABF weights. Here each CSM also corresponds to a frequency for which the beamformer wants to derive weights and effectively "listen" in on.

The implemented system actually supports both CBF and ABF. If a particular CSM only requires CBF weights, then the system will bypass the ABF weight derivation procedure, derive the CBF weights from the input steering vectors, and apply them to the input sensor data. However, if ABF weights are required, then the following steps must be performed for each CSM on which ABF is performed:

1) Compute the singular value decomposition (SVD) on the matrix containing the input sensor data in order to obtain the matrix's singular vectors and singular values.
2) Transform the singular values of the data matrix into the eigenvalues of the cross spectral matrix.
3) Use the eigenvalues and singular vectors to construct the pseudoinverse of the true cross spectral matrix.
4) Calculate an initial set of ABF weights for each direction the beamformer will be steered for this CSM using the passed in initial guess for ε, the input steering vectors, and the inverted matrix computed in the previous step. Also calculate the gain of the white noise each set of weights allows to pass through to the beamformer output.
5) Check to see if the associated white noise gain computed in the previous step falls within the beamformer's white noise gain constraint for each steering direction. For those steering directions that satisfy their white noise gain constraint, the associated weights are written to an output buffer. For those that did not meet their constraint, set the current value for ε as the new upper or lower bound for ε, depending on whether the white noise gain respectively fell above or below its constraint.
6) Perform a bisection search for ε by using the midpoint between the new upper and lower bounds for ε. Repeat steps 4-6 using this midpoint as the new guess for ε until either all steering directions have satisfied their white noise gain constraint or the maximum number of iterations for the bisection search has occurred. If a subset of the steering directions has not yet met its white noise gain constraints, then the last set of weights for each steering direction in that subset is taken to be that direction's final ABF weights.
7) Apply the final ABF weights to the input sensor data in order to compute the beamformer's final output.

Since CSMs represent the fundamental units of work in this algorithm, the system naturally parallelizes across them. Furthermore, as noted in Step 4 above, individual steering directions also represent independent units of work within a CSM. The system I implement attempts to take advantage of both of these course- and fine-grained levels of parallelism within the ABF algorithm. Further details on how this parallelism is exploited by the GPU will be given later, but first a brief discussion on an initial GPU implementation that failed is given. In particular, the reasons for failure provide the motivation and rationale for the current software architecture employed.

B. 100% Pure GPU - A Failed Approach

By design, the initial implementation consisted of a pure GPU approach in which the kernel compiled for the Geforce 8800 GTX would execute all steps listed in Section IV-A on the previous page. While conceptually sound, this approach was never realized due to the inability to compile an SVD for the GPU. It should be noted that during the time of implementation there existed no linear algebra libraries for CUDA. As a result, I hand-wrote much of the linear algebra code necessary for the ABF algorithm, much of which consisted of matrix-matrix multiplication, matrix-vector multiplication, and vector-based operations. However, in the interest of time, instead of attempting to hand-write an SVD, I had ported the LAPACK routine CGESVD and all its dependencies, located in the University of Tennessee's Netlib repository [45], to the GPU using CUDA.

Attempts to compile the SVD code resulted in complete failure. While the code itself had been demonstrated to run correctly on the CPU, any attempts to simply *compile* the source code for the GPU using `nvcc` would result in the consumption of all system resources. Specifically, all physical and virtual memory on the system would be exhausted, resulting in either the compiler process being killed by the operating system or the system becoming completely unresponsive. The compiler had been tested on systems with RAM ranging from 2 GB to 16 GB with no discernible difference in the end result.

Initially it was believed that a cycle might have been present in the SVD's call graph. Such a situation would pose a serious problem since CUDA kernels do not maintain a stack for any of the executing threads and thus cannot contain recursive code. By default, calls to device functions within the kernel are inlined, and thus, barring no such bugs in the CUDA compiler, the compiler could potentially inline cycles in the call graph indefinitely. This proved to be irrelevant, however, after generating a call graph for CGESVD and noticing no cycles existed.

Without the ability to compile the converted CGESVD routine for the GPU, the options became either hand-writing an SVD or developing a new software architecture to work around this problem by moving the SVD off the GPU and onto the CPU. In the interest of producing a functioning system, I opted for the latter, and what follows is a discussion of the approach taken.

C. A GPU-Centric Parallel Pipelined Approach

As alluded to previously, the current software architecture uses a hybrid approach where the CPU is primarily responsible for computing the SVD of the input data, while the GPU performs the remainder of the ABF weight derivation. A four stage pipelined architecture is implemented where each stage runs in parallel and operates on individual CSMs. Figure 5 on the following page shows a high level overview of the data flow for the entire pipeline.

The entire pipeline is implemented as a collection of worker threads maintained by one master thread. Each stage of the pipeline maintains an input queue holding pointers to CSMs, and the CSMs in this queue represent those that are to be processed by that stage. The queues themselves represent the shared state of the pipeline, and access to them is synchronized through the use of locks and condition variables. Each stage implements true transfer of ownership, that is, upon completely processing a CSM in the input queue, the stage will remove its reference from its queue and add it to its successor's queue. Thus, no two stages are capable of accessing the same CSM simultaneously.

The master thread typically represents the process's main thread of execution and is responsible for the creation of the

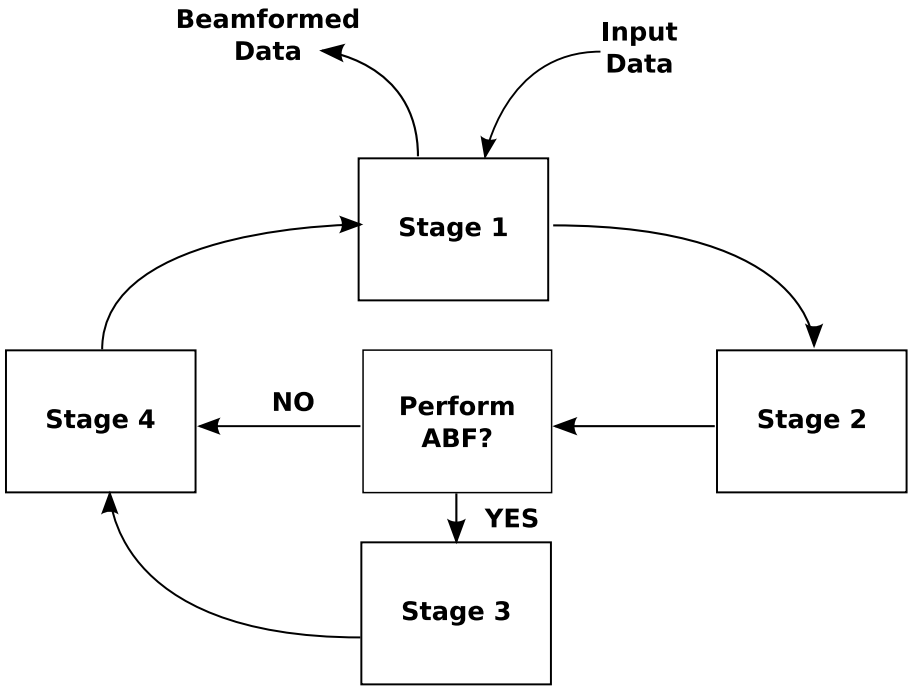

Figure 5. The parallel pipelined architecture. Input data are transformed into CSMs at Stage 1 and sent down the pipeline to Stage 2. Stage 2's worker threads will either compute the CBF weights for each CSM or compute the SVD of the input data as appropriate and then transfer the results to either Stage 3 (in the case of ABF) or to Stage 4 (in the case of CBF). Stage 3 computes the ABF weights for each CSM passed to it on the GPU, then passes the results to Stage 4. Worker threads in Stage 4 will perform final application of weights to the input data for each of the CSMs. Once all CSMs have been processed, Stage 4 signals to Stage 1 the completion of the pipeline, at which point Stage 1 will write out the final beamformed data.

pipeline and all its stages along with all worker threads that are used by the pipeline. By design, Stage 1 executes within the master thread, thus all worker threads created are distributed amongst Stages 2-4. Since Stage 3 represents the stage where the GPU computations are performed, it is allocated exactly one thread. Additional threads do not need to be allocated since contexts to the graphics hardware cannot be safely shared across threads. The remainder of the worker threads are allocated between Stage 2 and Stage 4 according to external parameters that the user can adjust in order to achieve optimal load balancing of the pipeline.

The primary role of Stage 1 is to read in the FFTed input data and divide it into the frequency-independent CSM structures. In addition to the FFTed acoustic data, additional data elements are associated with the CSM, including the steering vectors and any additional processing parameters supplied externally. As a new CSM is created, it is added immediately to Stage 2's input queue. Once all CSMs have been created from the input data, Stage 1 enters a suspended state where it will wait for a signal from Stage 4. Upon receiving this signal, Stage 1 will resume execution, where it will write all beamformed CSMs to the output data stream.

Upon receiving a CSM, one of Stage 2's worker threads will claim ownership of the CSM and determine whether CBF or ABF is to be performed on it. If CBF is requested, the thread will then derive CBF weights for the CSM based on the associated steering vectors. If ABF is requested, the thread performs an SVD over the CSM's input data and stores the resulting singular values and singular vectors with the CSM. The thread will then send the CSM to either Stage 3 or Stage 4, depending on whether the CSM will have ABF or CBF performed on it, respectively.

Stage 3 implements the kernel that is executed on the Geforce 8800 GTX. For performance considerations, this stage will wait until some number of CSMs are present in its input queue. Currently, this number is set to 32, which is two times the number of multiprocessors on the Geforce 8800 GTX. Once the number of CSMs equals the prescribed amount, those CSMs are transferred to the GPU, and the kernel is invoked for ABF weight derivation.

The GPU kernel partitions the grid into a number of thread blocks, as described in Section III-B on page 6. Specifically, the number of thread blocks is set to the number of CSMs transferred to the GPU. Since the GPU schedules a thread block to execute on a single multiprocessor, this effectively maps two CSMs per multiprocessor on the Geforce 8800 GTX. I exploit the finer-level of parallelism described in Section IV-A on page 7 by setting the number of threads per thread block to be the number of steering directions per CSM for which weights are derived; therefore, ultimately, a single GPU thread computes the weights for a single frequency-direction combination. Once all ABF weights have been derived on the GPU, the results are transferred back from the device to the host, and Stage 3's worker thread transfers the processed

CSMs, along with their ABF weights, to Stage 4.

Stage 4 performs the final application of the computed beamforming weights to the input data of each CSM. For each CSM in its input queue, one of Stage 4's worker threads simply takes ownership of one of the CSMs and multiplies the weight matrix derived in Stage 2 or Stage 3 with the CSM's input data matrix. The CSM and final beamformed data are then transferred back to Stage 1.

Once all CSMs have been processed, Stage 4 sends a signal to Stage 1. At this time Stage 1 resumes from its suspended state and writes all beamformed data and any additional information requested by the user, e.g. final weights derived, etc., to the output data stream. Once all beamformed data has been written, Stage 1 stops, and the master thread deallocates all resources used by the pipeline prior to exiting.

V. Testing and Results

The system used for testing consists of a desktop PC running a 64-bit version of Red Hat Enterprise Linux 4 Update 4 and containing two Intel Xeon 5140 2.33 GHz dual-core CPUs with 4 GB of RAM. The hardware specifications of the Xeon 5140 are similar to that of the Xeon 5160 shown in Table I on page 5. The Geforce 8800 GTX is also installed in this system.

The entire beamformer is primarily implemented in Matlab, and only the core ABF procedure has been rewritten to utilize the GPU. The code for this has been incorporated in a Matlab MEX object, which is Matlab's means of interoperating C/C++ code with Matlab code. In addition to a GPU-based kernel, a serial implementation written entirely in C has been written in order to provide a comparison between the GPU kernel and the Matlab reference implementation. The raw runtime performance of the beamformer was evaluated for a fixed input data set. The results are summarized in Figure 6.

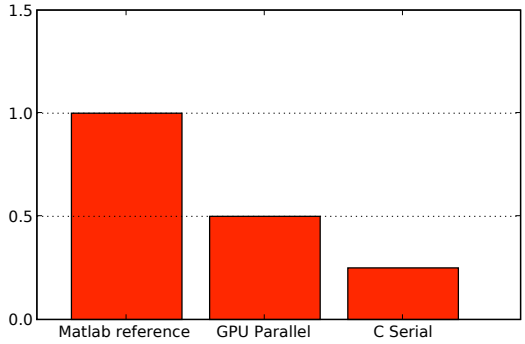

Figure 6. Relative runtime performance of the adaptive beamformer. All runtime performances are normalized and relative to the Matlab reference implementation. Here, lower is better.

As can be seen, the GPU kernel performs twice as fast as the Matlab reference implementation, but only at half the speed of the serial C implementation. This factor of two slowdown can be attributed to the following:

- The current GPU kernel makes no use of the multiprocessor's shared memory nor does it coalesce memory transactions to the GPU's global memory.
- There are currently no optimized linear algebra libraries for CUDA, whereas the CPU makes use of highly optimized BLAS and LAPACK routines for much of its linear algebra. This primarily includes matrix-matrix operations, matrix-vector operations, and most notably the SVD factorization routine.
- Currently, improper load balancing of the pipeline architecture creates stalls at various stages, most notably between Stage 2 and Stage 3.

The lack of use of shared memory has been deliberately done in order to simplify the initial implementation of the GPU kernel. Proper use of shared memory will require further examination of data access patterns in order to determine the best way to stream data through the GPU's parallel data cache. Nevertheless, it has been observed that by making proper use of shared memory and, more importantly, coalescing reads and writes from and to global memory, performance on the Geforce 8800 GTX can increase by an order of magnitude [40].

The lack of linear algebra libraries for GPU kernels written with CUDA also poses a problem. General-purpose systems have had the advantage of having libraries such as BLAS and LAPACK tweaked and optimized for various processor architectures over the past several decades. General-purpose computation on GPUs will, similarly, greatly benefit in terms of increased performance once more optimized libraries become available for use within device code.

Finally, the current pipelined architecture requires better load balancing between its stages, specifically between Stage 2 and Stage 3 and between Stage 3 and Stage 4. Stage 2 currently stalls as it waits for Stage 3 to remove CSMs out of its input queue, and Stage 4, similarly, waits for Stage 3 to transfer processed CSMs to Stage 4's input queue. These stalls can partially be alleviated by simply optimizing Stage 3's GPU kernel to utilize some of the suggestions already discussed.

VI. Future Work

From this point, work can progress in one of two different directions:

1) Further develop, profile, and optimize the existing pipelined architecture.
2) Create a functioning pure GPU kernel, thus obviating the need for the pipelined architecture.

In either case, devising ways to make better use of the GPU's shared memory presents itself as the most fruitful means of improving the kernel. As was pointed out in Section V, proper coalescing of memory transactions to and from shared memory can improve runtime performance by an order of magnitude.

However, there are caveats when optimizing the kernel's use of shared memory that one must consider. For example, while properly coalesced memory accesses give multiple GPU threads the capability to execute an instruction that will transfer either 32-, 64-, or 128-bit chunks of data at once, only those access patterns that handle 32 bits of data per transaction, per thread will realize fully the potential increase in speed [39]. Furthermore, due to the small amount of memory available in the parallel data cache on each multiprocessor, data sets for each CSM may or may not completely fit within shared

memory as beamforming parameters vary. Because no cache exists between the GPU's global and shared memory, it would also, in general, be beneficial to implement some caching mechanism that keeps frequently used data within shared memory as long as possible while streaming in and out less frequently used data.

In addition to making the best use of the GPU's shared memory, further work on the actual parallel pipelined architecture can yield further enhancements to the overall beamformer's performance. Due to length of time required for Stage 3 to derive ABF weights, it is unable to clear space within its input queue in a timely manner. This results in a stall between Stage 2 and Stage 3 as Stage 2 attempts to write its results to Stage 3's already-filled input queue. Upon failing to do so, Stage 2 enters a suspended state until Stage 3 signals that queue space has been made available. Again, improving the runtime performance of the GPU kernel through the use of shared memory can help minimize stalling and potentially increase the overall throughput of the beamformer. Upon optimizing Stage 3, further load balancing would be necessary as certain stages become more or less of bottlenecks in the overall system. In response to these varying dynamics, a pipeline with the ability to load balance itself at runtime in response to changing parameter and data sets would prove to be an interesting and possibly fruitful extension to the existing architecture in terms of further improving the runtime performance of the beamformer.

An alternative approach to improving on the existing architecture would to write a kernel that performed the entire beamforming procedure on the GPU. The obvious hurdle to this approach is the current lack of a functioning SVD on the GPU. However, certain algorithms such as one-sided Jacobi methods, which have been known to parallelize well, could be explored and implemented for the GPU. Once implemented, the entire parallel pipelined architecture could, in theory, be discarded because all the software components necessary for beamforming on the GPU would be available. In practice, though, since the CPU would otherwise be idle as it waits for the GPU to derive the ABF weights, some amount of pipelining would be desired in order to make better utilization of the existing hardware.

In such a case, it would seem to prove most reasonable to divide the CPU and GPU workloads based upon the type of beamforming being performed. Since the derivation of CBF weights is simply done by computing the norms of the steering vectors and applying these to the sensor data via a matrix multiplication, a pool of worker threads can easily perform these relatively simple operations on the CPU, thereby avoiding the penalty incurred by transferring data across the PCI-Express bus for such a relatively minute amount of work. The GPU could then continue to derive the ABF weights in parallel and apply them to their corresponding sensor data.

VII. Conclusions

I have demonstrated a software architecture that performs adaptive beamforming utilizing an NVIDIA Geforce 8800 GTX GPU using CUDA. While the initial results have shown that the GPU-based approach results in less throughput than a serial C implementation, factors for this apparent slowdown have been identified and possible means of optimization have been proposed and discussed. With further optimizations, I expect the GPU kernel to meet, if not exceed, the performance of the serial C implementation.

VIII. Acknowledgements

I would like to thank Dr. Bill Mark and Dr. Don Fussell for agreeing to be my faculty advisors for this project. Additionally I would like to thank the Applied Research Labs (ARL) for providing the funding for this project. Extra special thanks goes to my supervisor, Greg Thomsen, at ARL for the time and energy he's put into helping me make this project happen and all the guidance and knowledge he has imparted on me over the last few years.

References

[1] H. Krim and M. Viberg, "Two decades of array signal processing research," *IEEE Signal Processing Magazine*, vol. 13, no. 4, pp. 67-94, Jul. 1996.

[2] B.D. Van Veen and K.M. Buckley, "Beamforming: A versatile approach to spatial filtering," *IEEE ASSP Magazine*, vol. 5, no. 2, pp. 4-24, Apr. 1988.

[3] P. Sinha, "Parallel algorithms for robust broadband MVDR beamforming," *Journal of Computational Acoustics*, vol. 10, no. 1, pp. 69-96, Mar. 2002.

[4] C. Lee and J. Lee, "Eigenspace-based adaptive array beamforming with robust capabilities," *IEEE Trans. Antennas and Propagation*, vol. 45, no. 12, pp. 1711-1716, Dec. 1997.

[5] S.M. Kogon, "Robust adaptive beamforming for passive sonar using eigenvector/beam association and excision," in *Proc. Sensor Array and Multichannel Signal Processing Workshop*, 2002, pp. 33-37.

[6] D.A. Abraham and N.L. Owsley, "Beamforming with dominant mode rejection," in *Proc. OCEANS '90*, 1990, pp. 470-475.

[7] P. Stoica, Z. Wang, and J. Li, "Robust Capon beamforming," in *Conf. Rec. Thirty-Sixth Asilomar Conf. Signals, Systems, and Computers*, 2002, pp. 876-880.

[8] H. Cox, R. Zeskind, M. Owen, "Robust adaptive beamforming," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 35, no. 10, pp. 1365-1376, Oct. 1987.

[9] B. Widrow and J. McCool, "A comparison of adaptive algorithms based on the methods of steepest descent and random search," *IEEE Trans. Antennas and Propagation*, vol. 24, no. 5, pp. 615-637, Sept. 1976.

[10] J. Ward, H. Cox, S.M. Kogon, "A comparison of robust adaptive beamforming algorithms," in *Conf. Rec. Thirty-Seventh Asilomar Conf. Signals, Systems, and Computers*, 2003, pp. 1340-1344.

[11] M.D. Zoltowski, "On the performance analysis of the MVDR beamformer in the presence of correlated interference," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 36, no. 6, pp. 945-947, Jun. 1988.

[12] V. Reddy, A. Paulraj, and T. Kailath, "Performance analysis of the optimum beamformer in the presence of correlated sources and its behavior under spatial smoothing," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 35, no. 7, pp. 927-936, Jul. 1987.

[13] T.R. Messerschmitt, "Evaluation of the dominant mode rejection beamformer," Applied Research Labs, Austin, TX, Tech. Rep. ARL-TR-96-2, Jan. 1996.

[14] P.E. Green Jr., E.J. Kelly Jr., M.J. Levin, "A comparison of seismic array processing methods," *Geophysical Journal International*, vol. 11, no. 1, pp. 67-84, Sept. 1966.

[15] O.L. Frost, III, "An algorithm for linearly constrained adaptive array processing," *Proc. of the IEEE*, vol. 60, no. 8, pp. 926-935, Aug. 1972.

[16] A.E. Bryson, Jr., and Y.C. Ho, *Applied Optimal Control*. Waltham, MA: Blaisdell, 1969.

[17] T. Curtis. (2004). *Sonar technology - past and current* [Online]. Available: http://www.curtistech.co.uk/papers.htm

[18] R.L. Walke, R.W.M. Smith, and G. Lightbody, "Architectures for adaptive weight calculation on ASIC and FPGA," in *Conf. Rec. Thirty-Third Asilomar Conf. Signals, Systems, and Computers*, 1999, pp. 1375-1380.

[19] G. Lightbody, R. Woods, J. McCanny, R. Walke, and D. Trainor, "Rapid design of a single chip adaptive beamformer," *IEEE Proc. on Signal Processing Systems*, pp. 285-294, Oct. 1998.

[20] R.L. Walke, R.W.M. Smith, and G. Lightbody, "20 GFLOPS QR processor on a Xilinx Virtex-E FPGA," in *Proc. of SPIE Advanced Signal Processing Algorithms, Architectures, and Implementations X,* 2000, pp. 300-310.

[21] K. Jaervinen. (October 2, 2007). Field Programmable Gate Arrays [Online]. Available: http://www.automationit.fi/file.php?id=787

[22] M.C. Smith, J.S. Vetter, and S.R. Alam, "Scientific computing beyond CPUs: FPGA implementations of common scientific kernels," Oak Ridge National Laboratory, Oak Ridge, TN, Tech. Rep. MAPLD2005/187, 2005.

[23] C. Dick, F. Harris, M. Pajic, and D. Vuletic (2007). Implementing a real-time beamformer on an FPGA platform. *Xcell Journal* [Online]. 60, pp. 36-40. Available: http://www.xilinx.com/publications/xcell_60/index.htm

[24] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *Proc. of 2006 ACM/SIGDA 14th Int. Symposium on Field Programmable Gate Arrays*, pp. 21-30.

[25] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and K.A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California at Berkeley, EECS Dept., Berkeley, CA, Tech. Rep. UCB/EECS-2006-183, Dec. 2006.

[26] K. Hwang and Z. Xu, "Scalable parallel computers for real-time signal processing," *IEEE Signal Processing Magazine*, vol. 13, no. 4, pp. 50-66, Jul. 1996.

[27] A.D. George, J. Garcia, K. Kim, and P. Sinha, "Distributed parallel processing techniques for adaptive sonar beamforming," *Journal of Computational Acoustics*, vol. 10, no. 1, pp. 1-23, Mar. 2002.

[28] A.D. George, J. Markwell, R. Fogarty, "Real-time sonar beamforming on high-performance distributed computers," *Parallel Computing*, vol. 26, no. 10, pp. 1231-1252, Aug. 2000.

[29] K. Kim, A.D. George, P. Sinha, "Experimental analysis of parallel beamforming algorithms on a cluster of personal computers," in *Proc. of Int. Conf. on Signal Processing, Applications, and Technology*, 2000.

[30] J. Kerstetter, S. Crock, and R.D. Hof. (2003, April). More bang for the byte. *Business Week* [Online]. pp. 29-30. Available: http://www.businessweek.com/magazine/content/03_14/b3827613.htm

[31] NVIDIA Corp. (Last accessed: 2008, April). *NVIDIA nfiniteFX engine: Programmable pixel shaders* [Online]. Available: http://matt.insidegamer.org/documents/pixelshaders.pdf

[32] NVIDIA Corp. (Last accessed: 2008, April). *NVIDIA nfiniteFX engine: Programmable vertex shaders* [Online]. Available: http://matt.insidegamer.org/documents/vertexshaders.pdf

[33] D. Luebke, M. Harris, J. Krueger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn, "GPGPU: General purpose computation on graphics hardware," in *ACM SIGGRAPH 2004 Course Notes*.

[34] S. Che, J. Meng, J.W. Sheaffer, K. Skadron, "A performance study of general purpose applications on graphics processors," presented at *First Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, Oct. 2007.

[35] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krueger, A.E. Lefohn, and T.J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80-113, Mar. 2007.

[36] D. Luebke, "General-purpose computation on graphics hardware," presented at *Supercomputing 2006,* Tampa Bay, FL.

[37] NVIDIA Corp. (2006, November). *Technical brief: NVIDIA GeForce 8800 GPU architecture overview* [Online]. Available: http://www.nvidia.com/object/IO_37100.html

[38] NVIDIA Corp. (2007, May). *NVIDIA Tesla: GPU computing technical brief* [Online]. Available: http://www.nvidia.com/docs/IO/43395/Compute_Tech_Brief-v1-0-0_final_Dec07.pdf

[39] NVIDIA Corp. (2007, November). *NVIDIA CUDA Compute Unified Device Architecture: Programming guide* [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf

[40] M. Harris, "High performance computing with CUDA: Optimizing CUDA," presented at *Supercomputing 2007*, Reno, NV.

[41] L.H. Sibul, "Application of singular value decomposition to adaptive beamforming," in *Conf. Rec. IEEE International Conf. Acoustics, Speech, and Signal Processing*, 1984, vol. 9, pp. 750-753.

[42] NVIDIA Corp. (Last accessed: 2008, April). *Geforce 8800*, product specification page [Online]. Available: http://www.nvidia.com/page/geforce_8800.html

[43] Intel Corp. (Last accessed: 2008, April). *Intel Xeon Processor 5000 Sequence*, product specification page [Online]. Available: http://www.intel.com/products/xeon5000/ specifications.htm?iid=products_xeon5000+tab_specs

[44] NVIDIA Corp. (2007, November). *The CUDA Compiler Driver NVCC* [Online]. Available: http://www.nvidia.com/object/io_119170069217.html

[45] Netlib Repository (Last accessed: 2008, April). *CLAPACK (f2c'ed version of LAPACK)* [Online]. Availabe: http://www.netlib.org/clapack/index.html