2010

GPU Integration into a Software Defined Radio Framework

Joel Gregory Millage
Iowa State University

GPU integration into a software defined radio framework

by

Joel Gregory Millage

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Joseph Zambreno, Major Professor
Zhang Zhao
Manimaran Govindarasu

Iowa State University

Ames, Iowa

2010

DEDICATION

This thesis is dedicated to my fiance, Britnee, for her understanding and encouragement. Also to my family for their constant encouragement throughout all my years of schooling.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

Thanks to my advisor Joseph Zambreno for giving me input and ideas when I was stumped. I also want to thank my co-workers at Rockwell Collins for teaching me C++, CORBA and debugging over my years of working their which gave me the ability to jump into the work of this thesis with ease.

ABSTRACT

Software Defined Radio (SDR) was brought about by moving processing done on specific hardware components to reconfigurable software. Hardware components like General Purpose Processors (GPPs), Digital Signal Processors (DSPs) and Field Programmable Gate Arrays (FPGAs) are used to make the software and hardware processing of the radio more portable and as efficient as possible. Graphics Processing Units (GPUs) designed years ago for video rendering, are now finding new uses in research. The parallel architecture provided by the GPU gives developers the ability to speed up the performance of computationally intense programs. An open source tool for SDR, Open Source Software Communications Architecture (SCA) Implementation: Embedded (OSSIE), is a free waveform development environment for any developer who wants to experiment with SDR. In this work, OSSIE is integrated with a GPU computing framework to show how performance improvement can be gained from GPU parallelization. GPU research performed with SDR encompasses improving SDR simulations to implementing specific wireless protocols. In this thesis, we are aiming to show performance improvement within an SCA architected SDR implementation. The software components within OSSIE gained significant performance increases with little software changes due to the natural parallelism of the GPU, using Compute Unified Device Architecture (CUDA), Nvidia's GPU programming API. Using sample data sizes for the I and Q channel inputs, performance improvements were seen in as little as 512 samples when using the GPU optimized version of OSSIE. As the sample size increased, the CUDA performance improved as well. Porting OSSIE components onto the CUDA architecture showed that improved performance can be seen in SDR related software through the use of GPU technology.

CHAPTER 1. Introduction

Software Defined Radio (SDR) is a radio that can be transformed through changing of software and firmware [4]. Typically the software is changed within a GPP, DSP or FPGA. This requires any analog signals received over the air (OTA) to be converted to a digital signal for the processors to utilize the data[4]. With the ability to change software the radio becomes more dynamic and can easily be modified to operate at different frequencies and with multiple waveforms. SDR is an ever growing market in the defense industry for the United States and across the world. This is due to the fact that it is necessary for militaries to communicate with air, ground, and maritime units simultaneously during battle. This communication is done on multiple frequency bands with voice, navigation, and other types of data. This is not a new problem as radios have been used to communicate for decades in both the commercial and military sectors, but old radios used only simple software controllers and could only do a specific protocol on a particular frequency. As technology evolved, SDR was established and opened a new door for how radios of the future could be implemented. SDR has been used by the military for many years and over the last decade has grown in the open source sector with the immersion of GNU Radio. The GNU Radio supplies a completely open software and hardware solution to SDR [4] The Universal Software Radio Peripheral (USRP) was created as a Hardware platform (HW) for GNU Radio and is used on other open source SDR projects as well. The immersion of open source SDR projects has allowed non-industry SDR developers to take shape, especially in the area of research, to accommodate the implementation of new protocols and technologies.

A growing architecture in the defense area for SDR is Software Communication Architecture (SCA) which is a detailed architecture specification defining how a radio operates within

all of the software components. This includes the waveform as well as how the low-level components and services are designed at the C++ class level. Given SCA's wide use in the defense industry it was bound to find its way into the open source sector and this was accomplished by Open Source SCA Implementation: Embedded (OSSIE). OSSIE was developed as a SCA based open source SDR tool that can be used to build SCA waveforms and SCA components of the waveform using Eclipse as the development environment. Although OSSIE has the ability to build new components, the tool itself comes installed with a handful that are used in a typical SDR waveform. Functional components like FM and AM demodulation, encoding/decoding algorithms, a USRP controller, and signal amplifiers are all included on installation as well as the ability to communicate with PC sound cards. These components are written in C++ and run on the CPU of the PC it is installed on.

In embedded SDR, Field Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs) are commonly used devices for offloading signal processing from the General Purpose Processor (GPP) to improve radio performance. The problem with using DSPs and FPGA's is they can have their limitations depending upon use. The GPU becomes a very different type of component to use due to its inherent parallel nature. Taking the FPGA and DSP speed up a step further, GPUs will be used to determine if off loading the GPP onto the PC's GPU would improve performance of a similar manner or to a greater degree. NVIDIA developed the Compute Unified Device Architecture (CUDA) API for nearly all of the GPUs they have made over the last few years, giving any developer the ability to program their GPU in a relatively easy manner by installing the CUDA SDK. The real power of using the GPU is its ability to parallelize operations with ease by creating threads blocks with concurrent threads. This allows normal C loops to be transformed into having one thread take each loop iteration opposed to one thread doing each of them sequentially.

Porting OSSIE components of various complexity showed performance gains over the GPP on all components. Using C++ and CUDA timing measures, the performance of the functions were measured to compare the execution time of the components' implementation. The timing data recorded did not contain time to configure and setup the Common Object Request

Broker Architecture (CORBA) portions of the component or any other component specific configuration separate of the raw computation time of the procedure call. The setup of the CUDA functions was done through the constructor or an initializer of the class, therefore, only the minimal computation had to be done for component processing on the portion of data. Various computation runs were used with different sample sizes to give a wide spectrum of results. Given the parallel architecture of the GPU as the size of data samples increased the performance improved much quicker than the CPU implementation counterpart.

The thesis is composed as follows, Chapter 2 contains an overview of SDR as well as SDR and SCA combined including CORBA. Chapter 3 covers the GPU and GPU optimizations. Chapter 4 is the proposed approach of the paper. Chapter 5 is comprised of related work of SDR, GPU and SDR + GPU. The experimental setup is seen in chapter 6 followed by the experimental results in chapter 7. Chapter 8 concludes the paper and gives potential topics of future work.

CHAPTER 2. SDR Basics

SDR was a term coined back in the 1990's by Joe Mitola in his article where he discussed Software Radio Architecture [5], however DoD contractors had begun making software based radios back in the 1970s and 1980s. SDR relies heavily on multiple disciplines of engineering to accomplish a common goal, whereas "old fashioned" radios relied heavily on electrical engineers with some simple software processing. New radios rely on both hardware and software expertise to design an architecture that can be reused and reconfigured in multiple ways and on multiple platforms. This is especially important as the complexity of the radios grow and costs continue to rise as schedules continue to shrink.

The real key to SDR is to move as much of the hardware processing that would have been done normally in electrical components such as mixers, signal amplifiers, modulation and demodulation into software or firmware. This flexibility allows radios to operate in different ways, in new configurations, and with different features that do not require new hardware each time but only require distinct changes in software or firmware. This makes the radio more dynamic by having a common hardware architecture which can be used for various radios. The software inside can then be modified on the GPP or DSP to accommodate any new operation. This requires smarter design and reuse by the software engineer to create software that is highly object-oriented, therefore, as new hardware platforms are created the software can easily be reused and modified if necessary. Certain software of the radio will always have to be modified for new hardware. For example low level drivers and boot-up software, or possibly integration with different operating systems will always cause changes. Higher level components like waveform software should be abstracted enough to easily be moved from hardware platform to hardware platform with relative ease. Most HW changes for the WF portion will be a non-

issue since the change would generally be implemented on a similar processor and would only affect the Board Support Package (BSP), which would contain hardware specific software.

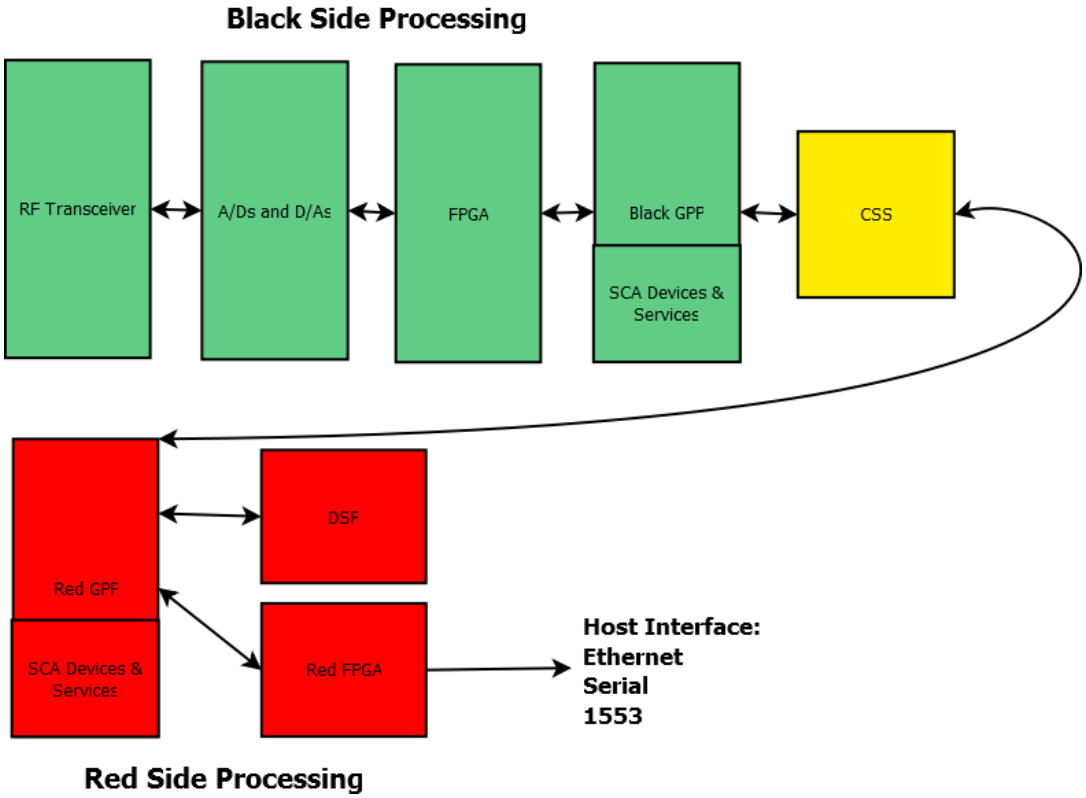

Figure 2.1 Example SDR Architecture [1]

It is also important for hardware engineers to be aware of portability in their design since new projects can change the radio hardware architecture. Radios may need to be reduced in size in order to fit into a smaller form factor or shrunk for other various reasons. It is also important to separate the pieces of the radios into different cards since most radios use similar components such as red side, black side, and a Cryptographic SubSystem (CSS). Certain cards will most likely be used more than others, but making the cards into a modular design with generic interconnects allows them to be used in a more "plug and play concept". Figure 2, adapted from [1], shows a possible SDR architecture that could easily be used in a SDR. The figure specifically shows the RF go into the ADCs and DACs followed by FPGA processing. The black GPP then receives data from the FPGA and passes it to the CSS. The red side processing would be even less complex since it would only contain a red side GPP then a

possible FPGA for handling of an ethernet, serial or 1553 bus. A DSP could also be used off the GPP for audio processing if necessary. The figure shows one instance of the RF RX chain but multiple could exist if the radio has multiple channels.

It can be hard to know where the line is drawn between HW and SW in a SDR, but since there is no general rule, it comes down to the application being implemented and how the designers choose to implement it. SW can also be a vague term used in a SDR because many radios do not simply use SW like C++ and C, but also have FPGAs which use languages like Verilog and VHDL to program firmware for the radio. This is usually done to offload processing from the GPP or DSP which can be done more efficiently on a FPGA device. A general example would be a DoD WF, called Link-16, which uses the reed-solomon algorithm on the signal to encode it before it is transmitted on the network and decodes the message upon reception [6]. Therefore this could be implemented on the GPP, DSP or FPGA. Depending on the system architecture each can have their advantages which will depend on timing, size or speed of devices or where it can be implemented. Also computation time lost to transfer the data between devices can be costly and needs to be considered. Some WFs have very tight timing requirements and wasted computation cycles must be kept to a minimum.

SDRs built under contract for the DoD have to implement a CSS that encrypts and decrypts data that is sent and received over the network [7]. This is necessary since most radios that are fielded can transmit Secret and Top Secret data and to do so are required to implement a CSS. A CSS device is generally an Application-Specific Integrated Circuit (ASIC) or FPGA and is used on multiple SDRs within the same company [7]. They implement multiple encryption and decryption algorithms that are commonly used on a SDR platform or a given SDR WF. Most of these CSS devices have to go through some type of National Security Administration (NSA) certification before they can be fielded which is very timely and expensive. Given the cost to have a certified CSS, it needs to be developed correctly the first time and not have to be redone for multiple radios and multiple programs. The CSS is generally used in SDR to pass data between CPUs; one side is deemed the red side which has the data unencrypted and takes the data in over a common bus like serial or ethernet from the host. The other side, the black

side, has encrypted data that has been encrypted by the CSS; which then passes the data to the modem to go out of the radio over RF [1].

As the complexities of SDR become more apparent, it can be seen that SDR begs many areas of expertise from several disciplines. Given the complexity of the software and firmware, the hardware portion of the SDR may appear to have the least amount of complexity. This could be true depending on the complexity of the WF protocol that is desired to be implemented. For example, the Link-16 waveform protocol operates on the 969 to 1206 MHz frequency band. However, this band has notches inside at two frequency portions that the radio cannot transmit on [6]. This is a very difficult task for the HW, RF (Radio Frequency) and Electromagnetic Capability (EMC) teams to ensure that the radio will not transmit on those frequency bands. Other hardware challenges are met in SDR but mostly vary upon the WF protocol that is being implemented. The end goal of software defined radio is to have a radio that is completely configurable and contains as much software and hardware reuse as possible. Paying attention to all these details should reduce budget and shorten schedule.

SDR is a growing field in government and commercial markets and it drives a considerable amount of research and development. One growing R&D effort is the cognitive radio effort of SDR. This takes SDR steps further than normal SDR implementations. Instead of the user being required to switch SDR WFs in a given radio (usually requiring power cycling or possible software reconfiguration). The radio can listen on multiple frequency bands and be able to sense what frequency spectrums are currently being used [8]. Depending on how the cognitive radio is setup and which WFs it contains, the radio can join these networks in real time without requiring new software loads or power cycles. All the software can then be on the radio simultaneously and the user can pick which network to join based on what is available at the time in OTA traffic [8]. This new extension to SDR brings larger challenges into the software and hardware realms and is still in the R&D phase. This will be a growing field over the next decade as companies try to field working prototypes that implement this functionality.

2.1 SDR with SCA

The majority of modern radios now being developed under DoD contracts use SCA as the basis for the radio architecture. SCA was developed under Joint Program Executive Office (JPEO) of the Joint Tactical Radio System (JTRS) as an architecture to assist in the development of designing software defined radios [2]. The SCA framework was designed to increase the portability and reuse-ability of the software in the radio, therefore reducing cost and development time of future radios. JTRS also uses the SCA architecture to allow DoD waveforms to be developed at one company where they then can be inserted into the waveform library held by JTRS. The waveform is then designed with generic SCA interfaces so that any approved company can take the WF software from the database and implement it on a SCA based framework. [2] defines the standard of SCA 2.2.2 which defines specific parts of the Core Framework (CF), Device Manager, Resource Factory, Application Factory, etc. The standard specifically discusses how each class should operate and what functions the class should have. Radios can become SCA certified which requires having an outside company examine the software and documentation of the radio and verifying that the radio is indeed following the SCA standard. It is also common for radios to be partially SCA compliant when only parts of the software are following the SCA standard. Having a radio which is SCA compliant or certified can be a selling point for a radio depending on the customer.

SCA brings many benefits when using it as a SDR architecture, especially if a company has more than just a handful of WFs on various radios. Using a common architecture across all the radios allows broad reuse in a variety of platforms also allowing new WFs to be written directly on top of the existing framework. The challenge then comes if particular WFs want certain features and have necessary components that others may not. These then need to be designed in a way that can easily plug in and out of the existing framework. SCA aids this by using Extensible Markup Language (XML) as a means to connect various components. The SCA Core Framework (CF) parses the XML for initial parameters of components and connection strings for CORBA connections, which will be discussed in more detail in the following section. This enables various components to coincide with each other. The XML can control whether

or not a given component is used, and if it is, what parameters it will be initialized to use. This is also useful in components which may be used across multiple WFs but need to be configured or implemented differently.

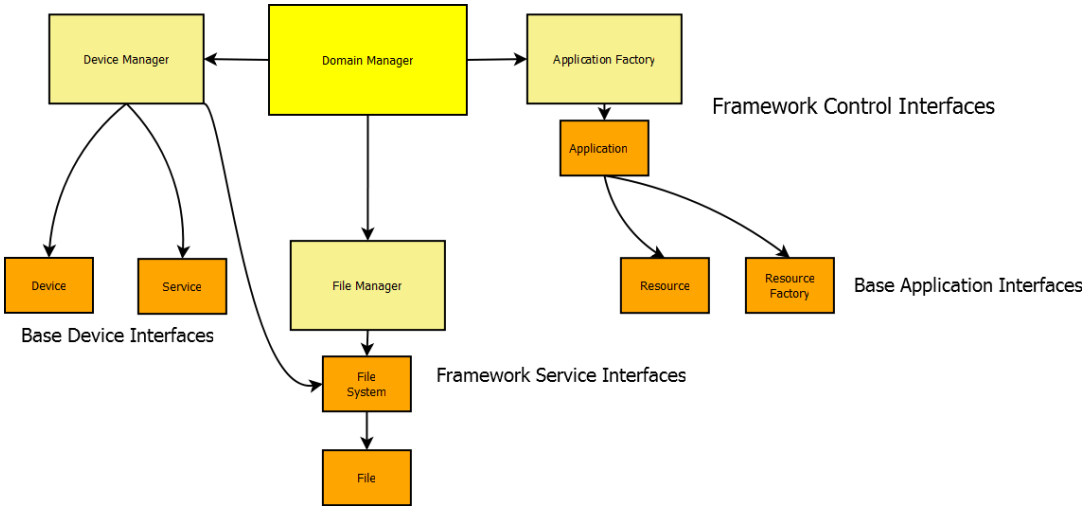

Figure 2.2 SCA CF Architecture Overview [2]

Though the CF generally contains devices of various use, devices such as driver abstractions are typically what they are used for. There are also services used across multiple radios. These services include fault management (how does the radio report things that have gone incorrectly), system control (how are modes and states of the radio at a system level defined, how do they transition etc.) as well as EMC protection (what does the radio do if it is operating outside its correct frequency spectrum). These types of services aid in the radio development and they can be modified and adapted easily for each specific implementation. In figure 2.1, adapted from [2], shows the general architecture of how the CF is designed. The domain manager acts as the controller for all the devices and services. The device manager then manages the services and devices individually. A file manager is used to manage the components and can also manage various data that needs to be written and saved during real time radio operation. The application factory than launches and starts the WF and its necessary components as well as starting their threads upon all components being started and initialized.

In figure 2.1, adapted from [2], shows a complete overview of how the CF interfaces with the CORBA portion of the SCA architecture and how it sits on top of the OS. The waveform

software, sits on top of all of those components and has to use CORBA to interact with any portion of the CF including services and devices. The waveform must always go through the BSP to communicate with any OS calls just as services and devices would have to.

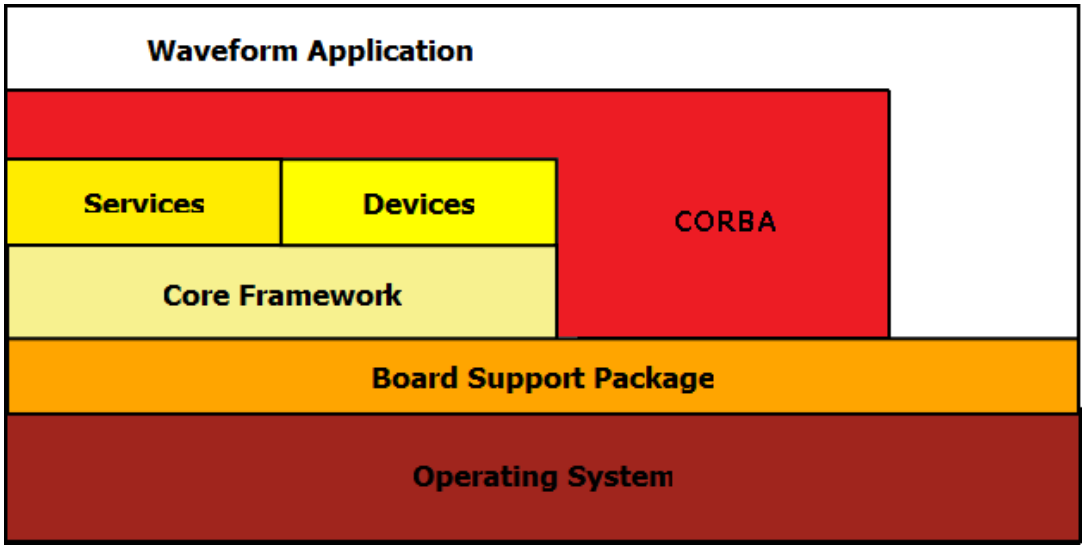

Figure 2.3 Overall Radio SW Architecture with SCA [2]

2.1.1 CORBA

One of the more prevalent portions of the SCA standard, which is the cause of latency and other concerns, is the requirement that all Computer Science Configuration Items (CSCIs) need to use CORBA to communicate between one another. CORBA is a distributed protocol standardized by Object Management Group (OMG) that abstracts function calls between CSCIs using Interface Definition Language (IDL) [9]. IDL is a generic language that looks and feels similar to C/C++, and is generic in the sense that any Object Request Broker (ORB) can use the IDL file that was written. It then can generate its specific client and server portion of the code for the developer to use in their programs. IDL ran through an ORB generates files in the language that is allowed or configured for by the ORB. These files are then used by the developer to create uses (client side implementation) and provides (server side implementation) ports which act as the interfaces to the CSCI. Provides ports are on the server side of the function call and is where the function is implemented. The user of the provides port is

identified by the uses port, the provides port is called when the user wants to use the specified function implemented in the provides port to either initialize the CSCI, pass data into it or get data back from the provides port. The CF is used to connect the ports upon start up of the terminal. The XML specified for each component in a SCA radio defines CORBA Distributed Computing Environment (DCE) strings that are used to uniquely identify each component and connect them up in the manner specified by the XML. XML for the radio is easiest to create in a XML generation tool designed for SCA components. The tool allows a model to be made of the entire software architecture of the radio and then creates XML based on the connections and parameters for all the components. This is done for every connection in the entire system and then auto generates the XML necessary to make the correct connections. This alleviates wasted time and effort of trying to write all the XML by hand.

Using the IDL files as interfaces in CORBA adds abstraction to the components so the interface between CSCIs can remain relatively consistent. Having this separation allows functionality internally to change without affecting how other components interact with it. This makes it easy to create dynamic functional components such as devices or services which plug into the CF as stated in the previous section. Using a common IDL interface allows a service or device to be used on multiple software programs with multiple WFs. This gives the framework more of a plug and play mentality or at least allows the interfaces to be predefined. This way if a new WF is developed that needs an existing service but new internal functionality, the interface is already defined and can be coded against immediately while the internal workings of the components are designed and implemented.

Though CORBA does have advantages in its ability to abstract components, CORBA can also be considered a downfall to the SCA architecture. CORBA licenses can be expensive and cut into the cost of the terminal or profits incurred from the radio. CORBA can be hard to initially integrate and implement into a terminal. This is especially true if the radio is being ported from an older radio that was not designed in such a modular way. It can also eat away at processing time in the radio which, depending on the implemented WF, can be very costly. The time needed to make a CORBA call depends upon the embedded device it is implemented

on and ORB being used, but if the radio has tight timing requirements it can have a large impact on system performance. This is especially important if redesigning an older software radio architecture which was not designed around CORBA and now wishes to be SCA compliant.

Seeing how CORBA fits into the SCA standard and SDR waveforms as a whole, the software complexity of the radio can be seen in its entirety. OSSIE encompasses all of these features into one software application and makes the development seamless. Since OSSIE handles developing the SCA interfaces around CORBA for the developer, modifying the operation of the components computation is all the needs to be changed when looking to optimize the components execution. This allows the implementation of the components to be done on another device, such as the GPU, without affecting the operation of the rest of the system.

CHAPTER 3. GPU Architecture and GPU computing

The GPU is a graphics processing unit that is used to run displays on computers and embedded devices which renders all aspects of the graphical user interface. Due to the requirements on graphics cards for high resolution video games and movies, both processing power in GPUs and the parallelism inside have increased drastically over the years. As the parallelism has increased in GPUs, CPUs have focused on efficient execution of single threads. Due to this development the GPU has found more popularity in research and heavily computational components because of the GPUs ability to parallelize pieces of software. As processors continue to have more cores, there still doesn't exist a API for wide use of these cores like the GPUs API.

In the fall of 2006, NVIDIA released an API for its GPU processors titled CUDA; this started initially with support for a handful of GPU processors [10]. As time has advanced so has the CUDA API support which gives nearly all GPUs developed by NVIDIA the ability to use the CUDA API. In the winter of 2007, AMD released its stream computing SDK that included Brook+ which was a AMD optimized version of Brook [11]. Brook is a extension of the C-language for parallel programming created at Stanford university, BrookGPU being the GPU variate [12]. The stream computing SDK also known as AMD Stream, has grown like CUDA and now allows for support of all AMD GPUs.

This thesis focuses on CUDA, which is the simple C-like API used to program parallel applications on the GPU. This allows C developers to make the transition between C and CUDA somewhat easier. Figure 3, adapted from [3], shows how the Nvidia GPU can be programmed using CUDA C, Open Computing Language (OpenCL), DirectCompute or CUDA Fortran. The CUDA architecture allows the user to create thread blocks which contain multiple threads

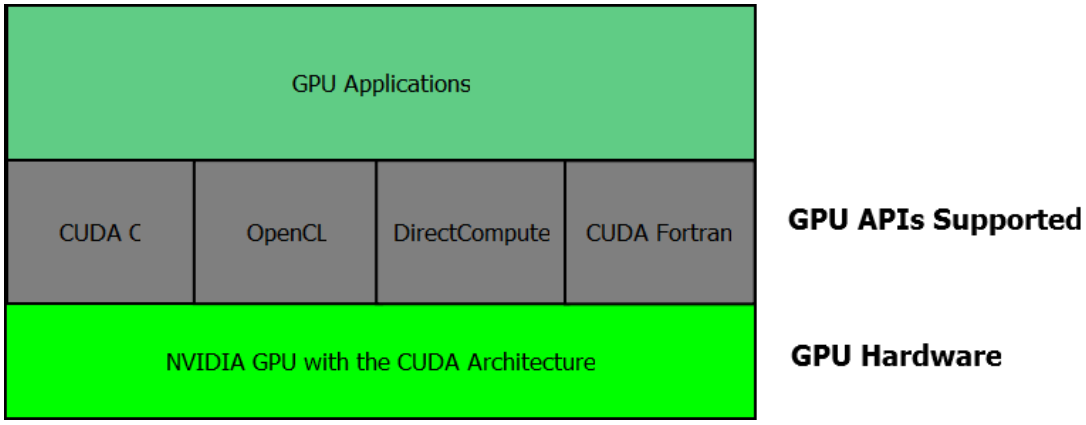

Figure 3.1 Nvidia GPU Overview [3]

within a block. The user specifies these settings prior to invoking the CUDA application from C or C++. Each block can have 512 threads and multiple blocks can be created to run on a given GPU core. The ability to have this large number of threads running in parallel, instead of sequentially, allows for certain computations to run significantly faster. The max number of threads in a thread block is 512 and when the given block is running on the GPU all the threads are allowed to run concurrently. Figure 3, extracted from [3], gives an example of how thread blocks can be mapped onto various GPU cores. The thread blocks, when ran, are swapped in-and-out of the GPU similarly to how an Operating System (OS) swaps processes in and out of a CPU. In later sections the various scheduling methods for the GPU blocks will be examined.

Though this thesis only uses the thread parallelization and memory management portions of CUDA. The CUDA API has many built-in functions for handling surface references, texture references, and multiple dimension data sets. CUDA also provides interoperability with other GPU interfaces like OpenGL, Direct3D 9, Direct3D 10 and Direct3D 11. In [13], the CUDA reference manual, all of the available CUDA-based functions for the most recent edition of CUDA are specified. This is very helpful to a developer who wants to know all the possible options when designing a CUDA based application.

One of the larger challenges in GPU programming is debugging the application being developed. NVIDIA supplies a CUDA debugger with its SDK, but to utilize it the GPU being used

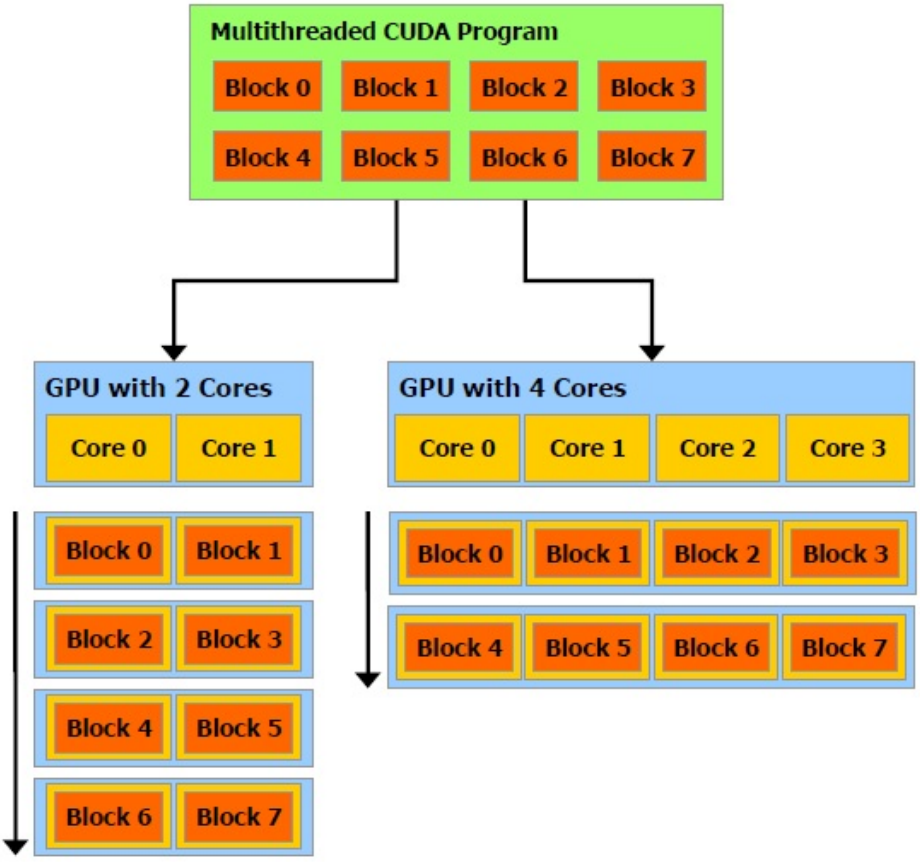

Figure 3.2 CUDA thread blocks on GPU cores [3]

to display the monitor needs to be different than the GPU used to run the CUDA debugger. Therefore, debugging requires having multiple graphics cards or a single NVIDIA GPU and an integrated graphics card on the computer's motherboard. All CUDA functions return error codes which are generally descriptive to what the problem is and once a CUDA application is configured initially it becomes easy to use the framework on other programs. The use of error codes doesn't require multiple GPU devices so it can be easily used by any developer.

CHAPTER 4. Proposed Approach

The basis of this work revolves around OSSIE, which is an open source SCA 2.2.2 implementation developed at Virgina Tech. OSSIE is a tool developed in C++ as a plug-in into Eclipse that can be used to create SCA based WFs and SCA based components. The Eclipse plug-in allows the developer to create the CORBA uses and provides ports defining how components will expose its external interface. The Eclipse builder uses these interface definitions to create the XML for the component that will be used when it is launched by the device manager. It also creates C++ code skeletons for launching the component and initializing its inter-workings. In summary, OSSIE encompasses all the components discussed in previous sections needed to make SCA functional, but incorporates them into one application. Ordinarily developers have to buy a license for the ORB and then create all the XML by hand or purchase a tool to create the XML by diagramming the system. Done either way, the XML has to be designed to connect the SCA components together and then to connect the WF. Using OSSIE abstracts away the inter-workings of SCA and makes it transparent to the user.

When the component is created a skeleton process_data function is constructed in the class of the given component. This is where the desired implementation of the component is inserted. After the components are constructed, all that needs to be passed to it are the correct XML strings. Then, based on the CORBA connections, the component gathers data, performs the given computation, and passes on the data. The connections that define what components pass the data in and where the data goes out is defined by how the CORBA connections are done within the WF Builder of the Eclipse plug-in.

When OSSIE generates the component, it creates three XML files used in the connections. The .prf file is the properties descriptor file which would contain any component specific proper-

ties that need to be set upon start-up. The .scd XML file is the Software Component Descriptor (SCD) that is used to describe what kind of interfaces and ports the component will be used to communicate on. Finally, the .spd XML file is a Software Package Descriptor (SPD) which defines the different implementations for a given component that are available, since various components can be cross-compiled for different target platforms [14].

The approach of this work is to take already created components in the OSSIE tool and modify the process_data function. Rather than doing the computation in C++ as it currently is implemented, the implementation will be changed to be completed on the GPU. The data has to be copied into GPU memory space by using CUDA device pointers for each input attribute. The pointers are then used as inputs for the given CUDA function. Once the CUDA function is finished the data is then copied back out in the GPP memory space. Given the data size and computation time, the time lost calling into CUDA and copying the data back and forth between the GPP and GPU is made up by the computation time that is gained using the GPUs parallelization. The CUDA 3.1 API, which is the most recent revision at the time of this research, was used throughout the implementation.

For the existing C++ software to compile and link with CUDA properly, the components are modified to contain a .cu file which contains the implementation of the CUDA software. The contents of the main.cpp file of each component is moved into the .cu containing the same implementation as it did before. This is done since it is easier to link with CDUA due to the fact that it is difficult to determine all the necessary libraries needed to link using g++ for all the CUDA symbols to be correctly found. The CUDA function is also created within this .cu file, which is defined as an extern, so that it can be called from the process_data portion of the component. The majority of the components in OSSIE are generally `for loops` over the I and Q channels of the data passed to it or while loops which continuously grab data from a component, process the data, pass it on and then repeat. Within these loops the data is then processed in a specific way to the desired result of the component. Using the CUDA architecture, the length of the data can be divided into blocks of threads defined by the data length to allow the number of blocks to expand or contract proportionally to the size of the data. Having

this dynamic architecture in each component allows the component to vary depending on the given set of data and still use CUDA's processing power to its full potential.

The initial call into CUDA is defined as a global function using the `__global__` CUDA directive. This is essentially the "main" in the CUDA portion of this function and is always used to enter computation in the CUDA software. Every call to various CUDA functions from then on is identified by using the `__device__` directive in CUDA. The majority of the OSSIE components which could potentially be ported are generally quite simple in complexity and only require a global function. The functions that are more complex and tied closely to the C++ object oriented mentality are harder to port to CUDA since CUDA is based more on C and classes have to be decomposed back to a "C like" mentality. This can be especially challenging given the number of signal helper classes that some of the components use. Only modifying the components internal workings and none of its external interfaces allow it to be used as it typically would in the OSSIE waveform builder. The OSSIE waveform launcher applications like nodeBooter and ALF are still able to be used to launch, visualize and debug the waveform applications. The only difference being that the actual implementation of the component now runs on the GPU and not the GPP. As for the OSSIE tools themselves, they are unaware of where the computation of the components are residing since they are concerned only about connecting the interfaces correctly.

The WF builder for OSSIE uses "drag and drop" to select components the developer desires to use as well as how they wish to connect up the components. Once the new GPU components are created and installed into the OSSIE build tree, the OSSIE builder will detect the components in the GUI interface and these will show up as additional components to select from. Running the components from nodeBooter and ALF is helpful to see if the data going in and out is the same as before in the non-GPU implementation. However, running in these tools makes it difficult to tell if anything is going wrong in the component itself. It is much easier to stub out the main of the component and send a piece of canned data into the component without any of the CORBA type connections. This lets the developer of the component verify it is operating correctly at an individual level prior to integrating with other components.

Once the CUDA portion of the software is verified against the C++, which is easily completed by running both components and comparing outputs, the component is then considered to be working well enough to be used with other components interchangeably.

CHAPTER 5. Related Work

Research on SDR has become a popular topic as more open source software projects produce efficient and viable testing environments for comparing WFs or testing new features. This is even more substantial as SDR WFs become more complex and the ability to test a new feature without HW is necessary to ensure proper performance and implementation. The GPU is also a common topic in researching the speed up of algorithms that can be parallelized and run with better performance than typical GPP architecture environments, especially in the simulation realm where complicated simulations can take extensive run times. Both of these topics pose interesting difficulties and challenges in themselves as well as research topics and opportunities. There are several existing papers on both of these topics with a broad spectrum of uses in both SDR and GPU as well as some containing the combination of the two.

5.1 SDR Related Work

The popularity of SDR has been ever growing over the last decade with advancement in technology and with software's ability to add increased complexity. The open source community has created a complete SDR implementation titled GNU Radio. GNU Radio is an open source framework for building, testing and configuring software defined radio waveforms. GNU Radio is separated into functional C++ components that are "connected" using Python. Being architected this way allows the C++ blocks to be abstracted from one another and connected in any way possible. GNU Radio can run on any computer with a sound card to do simple audio decoding but to do a full SDR test environment there needs to be HW to go with the SW. The hardware created to coincide with the software is the USRP. The USRP is an open source SDR hardware platform that supports transmit (TX) and receive (RX) connections as

well as external power amplifiers [15]. This hardware was designed for GNU Radio but is used on many open source projects [16]. The additional benefit of the USRP is that all the FPGAs on the board have open source firmware that can be downloaded and modified as well as open source Printed Circuit Board (PCB) layouts.

GNU Radio is an open source SDR that has a powerful test environment, but does not implement any of the SCA framework like OSSIE. OSSIE, unlike GNU radio, has created a functioning and detailed GUI (Graphical User Interface) plug-in to Eclipse which can be used for WF development. The eclipse plug-in can be used to create WF components from scratch or to connect default and user created components. OSSIE works by using the XML that is created from the eclipse GUI WF builder to launch the SCA CF and connects to any WF XML the developer chooses. The other added benefit is that the USRP, that is designed and used on GNU Radio, is also used and fully functional under OSSIE. When a component in the WF is used it is assigned to a device and can therefore be assigned to the USRP to be run on HW for testing opposed to being run strictly for simulation.

5.2 GPU Acceleration Related Work

GPU computing is a common technique used by researchers to speed up specific algorithms and computations since GPUs are generally used for graphic centric applications which require math intensive calculations done at very high rates. This extensive computation need causes GPUs to be designed in a highly parallel way to be able to make the most out of the computations done every second. Having this architecture at hand for researchers becomes very useful in increasing algorithms which can be rewritten in a parallel manner that can take advantage of the parallel architecture. CUDA is a language similar to C that is used to write applications for NVIDIA GPUs as stated earlier. Though the CUDA GPU Acceleration section discusses extensive research done in CUDA, the Other GPU Acceleration Techniques section shows research done with GPUs done with GPU APIs. OpenGL is a API that has been around for a significant amount of time and is widely used for graphics programming. OpenGL isn't designed as a parallel programming language as CUDA is, since OpenGL is designed around

being a API for graphics based applications. DirectX is another widely used graphics API developed by Microsoft, Direct3D being the GPU API widely used (DirectX being a collection of APIs for windows). High Level Shader Language (HLSL) is a shading language developed by Microsoft that is made for use with Direct3D. HLSL is a collection of various shaders that can be used to for graphics programming. Like OpenGL, Direct3D/HLSL is primarily a graphics programming language not a interface for general purpose graphics programming. Brook which was discussed briefly before, is a C-like programming language that focuses on parallel programming. Brook is similar to CUDA in that it is a programming API that allows for thread and data parallelism and was not designed for graphics processing like OpenGL and Direct3D.

5.2.1 CUDA GPU Acceleration

Many papers have been published on the use of CUDA for application acceleration. In [17] CUDA is used with MATrix LABoratory (MATLAB) to do 2D wavelet-based medical image compression. Using CUDA with MATLAB for 2D image compression they were able to see large potential in the parallel architecture of the GPU along with future potential for 3D and 4D images. Virtual machines are used more and more in companies and educational environments to allow many users to log into one machine without affecting each-other. Virtual CUDA (vCUDA) was created to accelerate High Performance Computing (HPC) and give the user access to hardware acceleration [18]. Using vCUDA they were able to access graphics hardware from the virtual machine and were able to get nearly identical performance as the non-virtual machine environment.

Image registration is an important topic in the medical area. A preferred tool for solving linear image registration problems is the FMRIB's Linear Image Registration Tool (FLIRT) [19]. A major drawback of this tool, however, is its time to run which makes it perfect for a CUDA optimization task. Using CUDA they were able to obtain a 10x speed up by optimizing the algorithm for a parallel architecture. Given that CUDA is a graphics based API, image topics are very common; algorithm and computation performance improvements are often desired and CUDA can be a great environment for that type of optimization. One time-consuming

equation set is the advective-diffusion equation (A-DE) calculation. This equation is used as a description of pollutant behavior in the atmosphere [20]. These computations are done using numerical integration and could require a long period of time. In the paper the researchers were able to parallelize much of the equations and see large results in using a highly paralleled architecture like CUDA.

In [21], a weather climate and prediction software piece is modified onto CUDA and run on multiple NVIDIA GPUs achieving up to eight times faster performance then the GPP. A 3D imaging algorithm called Fast Iterative Reconstruction Software for Tomography (FIRST) is used for forward and backward image reconstruction [22]. Using CUDA and the GPU with varying NVIDIA chip-sets, the parallel GPU threads showed a speed up of up to thirty-five times. A just-noticeable-distortion (JND) based video carving algorithm for real time re-targeting is used to display video at different resolutions on different displays [23]. The researchers were able to prove its high parallelization ability as well as its performance improvements using the CUDA GPU architecture. [24] used a CUDA/OpenCL GPU programming environment for work on dense linear algebra equations that involve solving multiple equations for multiple unknowns. They did not obtain a certain speed up so much as prove that using a CPU + GPU in tandem can gain performance improvement, however, they were able to show it does scale well in providing increased performance when adding-in more processors. [25] implemented a Fine-Grained parallel Immune Algorithm (FGIA) which is used for solving complicated optimization problems. Using CUDA on a GPU, depending on the given problem, they were able to see speed up of two to eleven times over a CPU.

In [26] a Finite Element Navier-Stokes solver, they originally had an algorithm implemented in OpenGL and ported it to CUDA. Using the accelerated algorithm on CUDA they were able to see a speed up of up to twelve times. Zernike moments are transcendental digital image descriptors used in biomedical image processing that take-up extensive computation time. Using CUDA [27] was able to achieve speed up of 300 to 500 times faster then the CPU algorithm. A real-time forest simulation for a flight simulator shown in [28] looked to see the speed up given from using a CUDA-related GPU for performance speed up. Using CUDA a significant

speed up was seen over a Pentium 4 and Core 2 Duo GPP and they were able to achieve near real-time results. [29] used CUDA to gain a speed up on the Particle Swarm - Pattern search optimization (PS2) algorithm proving that the algorithm can be parallelized on a GPU. [30] used CUDA to introduce a dense deformable registration for automatic segmentation of detailed brain structures. Using a CUDA based GPU, segmentation took eight minutes opposed to an excess of two hours on a typical GPP based run. Finally, [31] used CUDA to parallelize the EFIE (electric field integral equation) for solving two-dimensional conducting body problems. In their matrix-fill equations, speed up was 150 to 260 times better, whereas the conjugate gradient solver only gained 1.25 to 20 times.

5.2.2 Other GPU Acceleration Techniques

CUDA isn't the only development environment for doing GPU computing, other shading languages are used that operate differently then CUDA. [32] used OpenGL to optimize a Finite-Difference Time-Domain (FDTD) algorithm. The algorithm is a two dimensional electromagnetic scattering problem that can be very slow at simulating on GPPs. In [33] OpenGL was also used as a graphics programming interface for real time 3D imaging. Using a GPU, more features were able to be tracked on a GPU than a CPU, which gave accurate and detailed images. The High Level Shader Language (HLSL) and DirectX9.0c were used in [34]. The paper implemented Fast Fourier Transforms (FFT) and Convolution for image filtering on a GPU and saw an average speed up of 7.4 over a normal 32bit CPU. In [35], Brook was used, which acts as an extension of C for programing on GPU environments. They were able to speed up Method of Moments (MoM) calculations for electromagnetic scattering by over 100 times by using Brook on a high performance NVIDIA GPU. [36] used OpenGL to accelerate Time-Domain Finite Element Methods(TD-FEM) simulations. Increasing the number of nodes in their simulation they were able to achieve sizable performance of up to seven times greater than a CPU implementation.

5.3 SDR + GPU Related Work

In the previous sections, examples have been given of various SDR frameworks for designing and testing SDR, generally in an open source software realm. Examples have also been shown that demonstrate the ability to use GPU to optimize and improve performance of various applications, generally ones that are computationally intensive and require extensive GPP computation time. These sections aim to combine the two into one and show examples of SDR and GPU integrated together, which directly relates to what this paper aims to demonstrate.

In [37] a design change is proposed to change the implementation of the SDR modem from a DSP or FPGA to a GPU. Their design is meant to be generic and apply to various SDR applications but is applied directly to a Worldwide Interoperability for Microwave Access (WIMAX) implementation. [37] decided to use a GPU implementation over a GPP implementation even though modern CPUs with dual or quad core processors have enough power to run an SDR modem. Since the GPU is already tasked with multiple processes like other parts of the application, drivers, OS etc offloading processing of the CPU onto other components can be beneficial. Using the GPU lets the designer take advantage of the parallelism of the GPU architecture. Like many other GPU implementations, this one uses the NVIDIA CUDA SDK for its GPU API. The algorithm executed on the GPU and also on a Texas Instruments (TI) DSP was a viterbi decoder. Viewing just throughput of the decoder on each, the GPU could handle 181.6Mb/s where the DSP could only do 2.07Mb/s showing a substantial improvement. As for their actual implementation performance, the GPU out-performs the DSP across the board performing the same tasks in just a fraction of the time. The authors point out that as SDR evolves the high cost and low overall processing performance of the DSP and the FPGA creates the bottleneck of the radio, and running on a GPU helps alleviate this problem.

[38] shows a framework on a desktop PC of using a GPU to implement a FM receiver based on a desktop PC. Using the GPU and other components the authors' goal is to create an architecture which is parallel and easily scalable. Portions of the new architecture were run on the GPU using the CUDA API. They were able to see large performance increases from their previous design due to the GPU's parallelism. Their future goal is to port more of the receiver

onto the GPU to extend their performance improvement even farther.

In [39] a 802.11n implementation is taken that had been executed on a GPP and a portion of the protocol is run on a 128 core GPU to compare performance against the GPP. The writers of the paper implemented the PHY algorithms on the GPU as well as investigated using the GPU as a networking device to see how the performance of each aspect would be affected. The authors chose CUDA as the programming API to implement the Multiple-Input and Multiple-Output Orthogonal Frequency-Division Multiplexing (MIMO-OFDM) portion of the 802.11n PHY algorithm. Using the CDUA API, the CUDA implementation was able to outperform the CPU eight to nine times over. Using the GPU as a networking device they effectively integrated the GPU implementation of MIMO-OFDM with an external network interface and web browser. The authors were able to run the simulator in user-mode only which reduced some of the potential performance gain. This is true because much of the CUDA functionality is lost when run in user-mode since it allows function calls that are not able to be made when running completely on the device.

As briefly seen in previous papers, MIMO is a technique that can be used to increase the throughput of a wireless network by using multiple antennas for both the transmitter and receiver [40]. [40] realized that MIMO requires computationally intensive detectors and must be flexible enough to be used in a variety of applications. Most MIMO designs are done on ASICs that have an average throughput of 73Mbps with a signal-to-noise ration of roughly 20 dB [40]. They determined the most computationally difficult parts of the MIMO receiver are the detector and channel decoder. Given this fact, they decided to implement the MIMO detector on the GPU using the CUDA API. They then tested their CUDA detector with various modulations schemes of 4, 16, 64 and 256 Quadrature Amplitude Modulation (QAM) using an average execution time of over a thousand runs. They then compared their results to various ASCIs, Application-Specific Instruction-Set Processor (ASIP), and FPGAs. Generally the ASIC could gain higher throughput with fewer hardware resources but could not scale well to an increasing number of antennas and different modulation types. Depending on the implementation, the slight loss in throughput may be a wash in the scheme of things if extensibility is a more

important trait.

Low-Desity Parity-Check (LDPC) error correcting codes were created in the 1960's but until recently were never used in communications due to the high computation performance requirements [41]. LDPC decoders can be done in Very-Large-Scale Integration (VLSI) but the costs of such an implementation can be quite significant. In [41] they propose a methodology for LDPC using the CUDA API to allow simultaneous multi-word decoding. WiMax, which is also coined 4G in the cell phone market, is an application that uses LDPC decoders implemented in an ASIC format. The authors used a min-sum algorithm to do their comparisons between existing costly ASIC designs and their proposed CUDA design. Running simulation examples on the GPU showed 100Mbps and up throughput which competes well with many of the ASIC implementations on the market today. It is also mentioned that GPUs on the market at the time that the paper was written have twice as many cores as the GPU they used for testing and simulation which they predict would yield even better results than has already been seen. The main drawback they saw on the GPU implementation is when having so many concurrent processes running and accessing memory at the same time, the memory access time becomes the bottleneck of the system.

CHAPTER 6. Experimental Setup

Porting OSSIE components from C++ to CUDA requires setup processing prior to the actual CUDA procedure call, which minus the input parameters to the function, were nearly identical between all the components ported. In each component their is a process_data function which is typically used in the OSSIE component for general computation. This function is modified to call into the CUDA portion of the software. Any data that is passed into CUDA has to be done through pointers; this requires pointers to be declared in the process_data function for any data that needs to be copied into GPU memory space. The CUDA setup code needs to initialize the memory for the CUDA device and then pass the CUDA pointers into the CUDA based function call. Any input parameters needing to be passed into the function must have memory allocated for it in both the GPP memory space and the GPU memory space.

The process_data function uses the PortTypes::ShortSequence class which use for creating a CORBA-based stream of data contained in 16bit words. This class has various overloaded functions to make it easier to put data in and get data out, very similar to a C++ vector or dequeue. The get_buffer function in PortTypes::ShortSequence is used to get the pointer to the desired data; this is needed to use the `memcpy` function to get data into or out of a CORBA sequence. Using the get_buffer() function allows the developer to pass the pointer into the CUDA function alleviating unnecessary memory copies of the data. This also allows data coming out of the function to be operated on with the same pointer that will be used to transfer data to the next connected component.

The CUDA function called from the process_data function needs to be declared inside of a .cu file. For the function to be seen by other object files and for the executable to link, the function being called needs to be declared as extern. This way the linker knows where the

function resides that it is trying to call since they functions are not contained within the same object file. Below is an example of how the setup code could be done, an explanation will follow below:

```
extern "C"
void runCude(short* inputPtr, short* outputPtr)
{
CUmodule cuModule;
CUfunction someFunc;
CUdeviceptr inputDevPtr;
CUdeviceptr outputDevPtr;
unsigned int timer  = 0;
float timerval;
cutCreateTimer(&timer);

//Inititalize GPU Device
cuInit(0);

CUdevice device;
//Create object of device
cuDeviceGet(&device,0);

CUcontext pctx;
//get context based on device, using auto scheduling
cuCtxCreate(&pctx, CU_CTX_SCHED_AUTO, device);

//Load the cubin file into module
cuModuleLoad(&cuModule, "/home/pathToCubinFile/something.cubin");
```

```
//Get a specfic function out of the cubin file
cuModuleGetFunction(&someFunc, cuModule, "functionName");

//Allocate memory for device pointers
cuMemAlloc(&inputDevPtr, someNumberOfBytes);
cuMemAlloc(&outputDevPtr, someNumberOfBytes);

//Setup size of first function parameter
cuParamSeti(someFunc, 0, inputDevPtr);
cuParamSeti(someFunc,8, outputDevPtr); //Always use offset of 8 for pointers

//Set end length
cuParamSetSize(someFunc,16);

//Copy any data in host pointers into device pointers
cuMemcpyHtoD(inputDevPtr,  inputPtr, sameNumberOfBytesAsAllocated);
cuMemcpyHtoD(outputDevPtr,  outputPtr, sameNumberOfBytesAsAllocated);

//Determine thread block layout
cuFuncSetBlockShape(someFunc, xSize, ySize, zSize);

//Start Timer
cutStartTimer(timer);

//Launch the function with a specified grid of thread blocks
cuLaunchGrid(someFunc, xSize, ySize);

//this blocks device until all prior tasks are completed
```

```
cuCtxSynchronize();

//stop timer
timerval = cutGetTimerValue(timer);
printf("Done in %f (ms) \n", timerval);

//Copy any output or changed data back to the host
cuMemcpyDtoH(inputPtr, inputDevPtr, sameNumberOfBytesAsAllocated);
cuMemcpyDtoH(outputPtr, outputDevPtr, sameNumberOfBytesAsAllocated);

//Free any Device poiters and destroy the context
cuMemFree(inputDevPtr);
cuMemFree(outputDevPtr);
cuCtxDestroy(pctx);
}
```

The example above is code that could be used to setup a CUDA function which has two input parameters. The function, no matter how it will behave, is setup the same way initially. The GPU device object is created by the cuInit() function. This initializes the GPU device to be used by the host software. A CUdevice object is created and passed into cuDeviceGet. This call passes the parameter by reference in order to populate the CUdevice object with the available CUDA device. To tell CUDA how to operate the task scheduler, cuCtxCreate needs to be called. The cuCtxCreate function takes in three inputs. First, is a CUcontext object which is passed in by reference. Second, is a parameter which identifies what type of scheduling will be used in CUDA to arbitrate thread blocks on the GPU. Finally, a third object is passed which is the CUdevice object initialized by cuDeviceGet. Figure 6, adapted from [3], shows how threads in a GPU block form an x by y grid of threads. The threads then create multiple blocks which all contain the exact same threads.

The scheduling of how the thread blocks are switched in and out of the GPU is generated

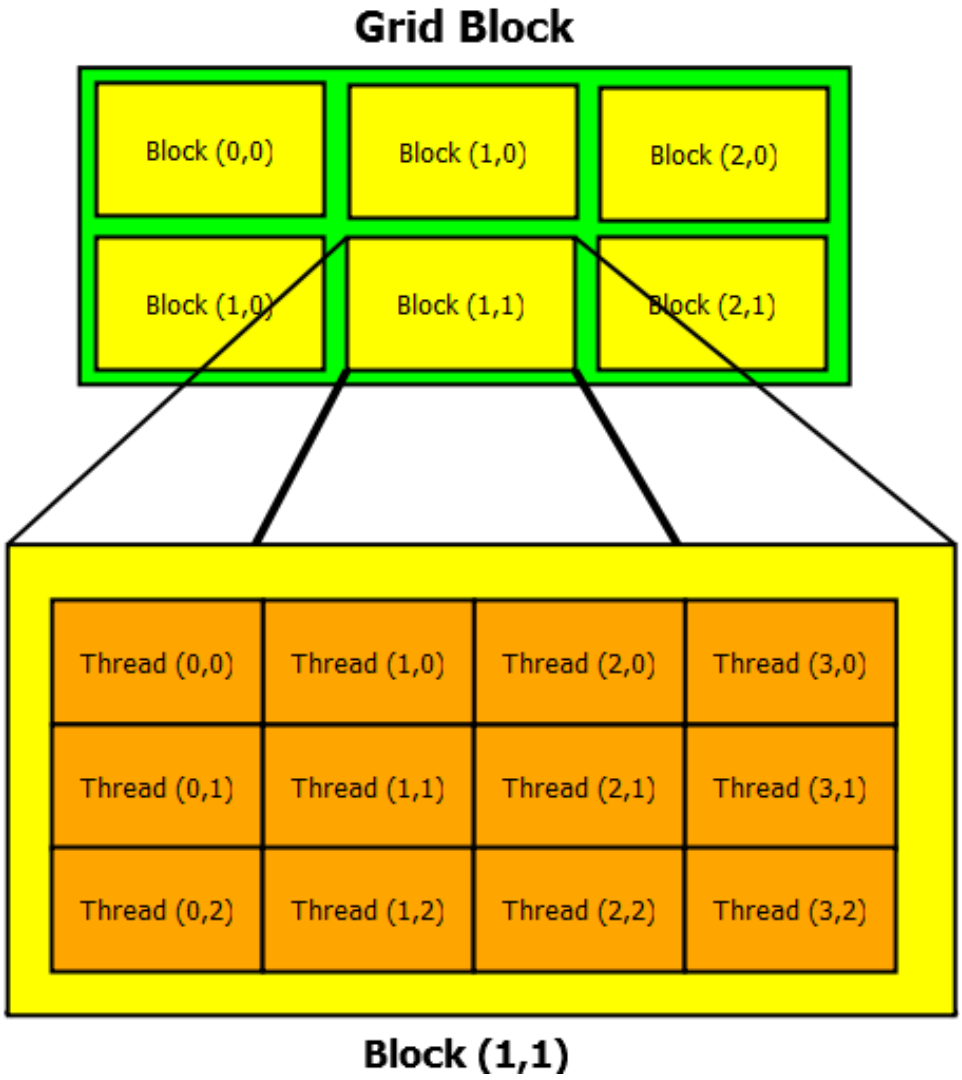

Figure 6.1 CUDA grid thread and thread blocks [3]

by the thread scheduler, a parameter set in the cuCtxCreate function. When defining the scheduling parameter, multiple options exist, one being CU_CTX_SCHED_AUTO which is the default value to the function. This uses a heuristic that is determined by the number of active CUDA contexts in the given process and the number of logical processors. If the number of processes is greater than the number of logical processors, CUDA will allow other OS threads to run when it is waiting for the GPU to finish its computation. If this doesn't occur, it will simply let the processor spin and wait until the computation is complete which can be a waste of CPU cycles depending upon the desired implementation [42]. If the CU_CTX_SCHED_SPIN

scheduler is used, it notifies CUDA that it should spin in its thread and wait until it gets results back from the GPU [42]. The CU_CTX_SCHED_YIELD scheduler instructs CUDA to pause in its thread while waiting for results from the GPU [42]. The CU_CTX_BLOCKING_SYNC scheduler tells CUDA to block the CPU thread on a semaphore while it is waiting for the GPU to finish its computation [42]. Using CU_CTX_BLOCKING_SYNC syncs the GPU device into the calling CPU thread. The third parameter to the cuCtxCreate is the device object that was returned from the cuDeviceGet function.

Once the GPU has been initialized and configured, the module can now be loaded. The function called to execute this is cuModuleLoad; it is passed the CUModule object by reference, which is populated by the function. The other parameter passed is the path to the .cubin file; this file is generated by the nvcc compiler. The .cubin file can be named and placed wherever the developer would like; at runtime when cuModuleLoad is called it reads this .cubin file which tells it what CUDA global functions are available to call. This file is created by passing the -cubin argument to the compiler opposed to -c which is normally passed to generate a .o file.

In order to abstract-out the desired function that is to be called by CUDA, the cuModuleGetFunction needs to be called. This process uses the CUModule object initialized by cuModuleLoad to locate the desired function defined in the .cubin file. To use any CUDA function, the cuModuleGetFunction has to be called individually for each function. This function takes in a CUfunction object, which is populated by the function as an object to be used by the developer for calling the specified function. Besides passing in the CuModule, a string is also passed that is the name of the CUDA function that is desired to be abstracted out of the cubin file. At his point a object for each function in the GPU is created that can be used to execute the respective functions.

Initializing and configuring the GPU device is the portion of the GPU configuration that should be common among all CUDA function calls. The portion where the GPU setup differs is setting up memory on the GPU device. For every input or output parameter used on the device, a CUdeviceptr object needs to be created which acts as a pointer to memory within the GPU. To allocate any GPU memory, cuMemAlloc needs to be called on each CUdeviceptr to

reserve the desired amount of memory, just as a C `malloc` or C++ `new` call would be used on the GPP. These functions take in the reference of the deviceptr as well as the desired number of bytes to allocate just as the C and C++ variants would.

One of the large differences between C and CUDA is that allocating memory for use on the device is not all the developer is required to do to call the function with the correct parameters. In CUDA the device needs to be instructed of what size the data is which will be passed into each function being used. To accomplish this, the cuParamSeti function is used; this takes in the function name for which the data is being declared for, the offset for the input, and the CUdeviceptr that will be its input. The offset passed into this function needs to correlate the order of parameters of the function being called. If done incorrectly, the data will go in and out of the wrong memory locations. For example, the first input would be cuParamSeti(function,0,input1) because it is the first parameter the offset zero. A note to make here is that for CUDA to align the data correctly, typically an offset of eight needs to be used for each input. Since CUDA defaults to 64bit pointers, a pointer of data less in size still needs to be defined in the cuParamSeti function as needing a 64bit offset. The CUdeviceptr is 64bits in width so if the offsets are not the desired data size (like 16bits for a short), the data becomes misaligned and the data read into and out of the function becomes incorrect. This does not appear to be documented in any of the CUDA reference guides but is the result of the CUDA implementation used in this paper. To complete the pointer alignment setup by cuParamSeti another function needs to be called. The cuParamSetSize function is used to finalize the set of parameters. The cuParamSetSize function defines the total length (in bytes) of all parameters being used in the function. To determine the total length of the parameters, the sum of the number of inputs and outputs used and their respective data sizes are required. For example, with 5 input parameters each offset 8, would require 40 to be passed, 8 past the last input parameter.

Since any data that was allocated on the devicePtrs is currently empty, any data held by the host that needs to be used in the CUDA function has to be copied into the devicePtrs. This allows any parameters that were inputs or inputs/outputs to have the data available that

is required by the GPU. For every parameter this is required for, the cuMemcpyHtoD function is called. This works like a standard `memcpy` used in C or C++.

To use the power of CUDA, multiple thread blocks need to be used to take advantage of the parallelism. To accomplish this, the parameters of the thread blocks which are doing the computation need to be set. This essentially defines how large the thread block can be in terms of x, y and z parameters (or a 3-dimensional block). A thread block can have a maximum of 512 threads [3] which is an important note to keep in mind when setting up the block size. When a C `for` loop is implemented in CUDA it is done so that each thread can do one loop iteration (ideally) so up to 512 can run concurrently. A given system can have any number of blocks, but those thread blocks are switched in and out by CUDA (similar to how a OS switches out processes). This is important to note because even if a developer has multiple thread blocks of 512 threads, no more then one thread block can be running at a time, essentially limiting 512 threads to running concurrently. Keeping this in mind when defining a thread block, it needs to be determined what the dimension of array the threads will be operating over. If it is only one, only threads for the x direction are needed, if it is 2-dimension, x and y would be required and etc. The cuFuncSetBlockShape function takes in the function that is going to be called, as well as x, y and z values which define the dimension to be used. As an example, a `for` loop over a 1-dimensional array 500 times could easily be done as 500,1,1 where 500 is the x value and 1 is the value of y and z.

Depending on the use of the developer, timing data may or may not need to be recorded. The function to record time is cutStartTimer; this is executed after cutCreateTimer is called on an `unsigned int` timer object. This call simply initializes the timer object to start counting from this point forward after being initially called. If the developer wishes to measure the time of execution for the CUDA function only, the function should be called prior to cuLaunchGrid and cuCtxSynchronize. Doing so would gather time of execution for only the execution of the function used and nothing else. The time granularity here is returned in milliseconds and this data can then be saved off or printed out for the user to view.

All of the functions stated up to this point are needed to setup and initialize the GPU

for running the function. To actually invoke the GPU kernel two functions need to be called, cuLaunchGrid and cuCtxSynchronize. The function cuLaunchGrid takes in the function that is going to be called, as well as the width and height of the thread blocks. In other words a 2-dimensional number that defines the number of thread blocks, recalling here that cuFuncSetBlockShape actually defined the 3-dimensional size of the thread blocks themselves. The other function is cuCtxSynchronize; this function is used to block until the device has completed executing all the previously requested tasks [42]. If the cuCtxCreate was called using the CU_CTX_BLOCKING_SYNC flag, then the CPU thread will continue to block until the GPU has finished all its computation. These previous two functions call into whatever function in the GPU that has been defined for use using the `__global__` directive. Once this function is completed by all the defined threads, control returns back to the host function.

Any data that is computed and created by the GPU function will be stored in GPU memory space upon completion of the function. To gather the data out of CUDA, the data has to be copied out of the CUdeviceptr and into the C or C++ pointers. To accomplish this the function cuMemcpyDtoH is used. This function works the same way as cuMemcpyHtoD but copies in the other direction. This then ensures that all the data has been copied out of CUDA memory and back into GPP memory space and can be returned for further GPP modification. To verify no memory leaks exist in the software being created, some memory cleanup should be done by the developer. For all CudevicePtrs the cuMemFree function should be used which acts like a C `free` call or a C++ `delete`. The cuCtxDestory function should also be called and passing it in the context that was previously created in cuCtxCreate. This closes the device for further use until a new context is created.

A general note on the CUDA setup is that each function throws various error codes upon success or failure, and the cuda.h file from the SDK specifies what each value means. A CUDA_SUCCESS, or 0, is returned for a successful function call. Various other error codes could be returned upon an error or failure case. Checking the return values of each function can be extremely helpful for debugging by simply checking return values for non-zero numbers. If performance data is desired by the developer or information on GPU usage is necessary, the

environment variable CUDA_PROFILE can be used. Defining the variable and setting it to "1" enables the profiling and setting it to "0" disables it. This outputs a file into the running directory where the CUDA software is executed and gives performance characteristics. The file shows the the execution time of any CUDA functions ran on the GPU as well as any of the C software ran before or after the call. With each function call time for CUDA it also shows how much of the GPU (in a percentage) was used to execute that function. This becomes quite useful since it is generally desired to have the GPU not fully utilized to 100% capacity. Seeing the runtime results can help the developer try different block and thread sizes to find the most optimized result.

Up to this point, setup for the CUDA device and how to gather debug and performance data has been stated in detail. This only defines initial function operation; actual design and programming of the computation of the CUDA function needs to be done as well. Each component will implement a different set of functionality but since most of the components loop over a 1-dimensional I and Q channel, the overall for loop setup remains the same. In C++ or C, a `for` loop is usually defined using the syntax of `for(...)`. The loop will then run a number of loop iterations over some dimensional array, where the CPU processes each one individually and consecutively, assuming no specific compiler optimizations. On CUDA, since up to 512 threads can run concurrently at the disposal of the developer, it is no longer necessary to do the looping sequentially. CUDA has three built-in objects to index threads within blocks and to determine which block is used. This is done by using blockIdx, blockDim, and threadIdx. Each of these is a three dimensional object which has attributes of x, y and z. When blockIdx and blockDim are used, they are multiplied by each other for a specified dimension to give a specific block's x component. The threadIdx object, also in the same dimension, is then added onto the product of the previous two to give a value of the current thread that is executing on that block. For example, if a 2-dimensional loop is needed, a variable, "i" , could be created and initialized so: `i = blockIdx.x * blockDim.x + threadIdx.x`, and then "j" could be initialized as: `j = blockIdx.y * blockDim.y + threadIdx.y`. The variables "i" and "j" would then be used to index the arrays just as they would have done previously within the

`for` loop. This could be extended into the z dimension as well, to account for a 3-dimensional thread block. The array indexes still need to be in an if statement to check against the size of the data that needs to be computed. This is simply done by passing the length into the CUDA function as an input, especially since the length will already need to be known to copy the correct amount of data in and out of the GPU memory. 6, used from [3], shows how the serial host SW (the C code) executes between CUDA calls. The figure, 6, demonstrates how the SW can move in and out of the GPU computation.

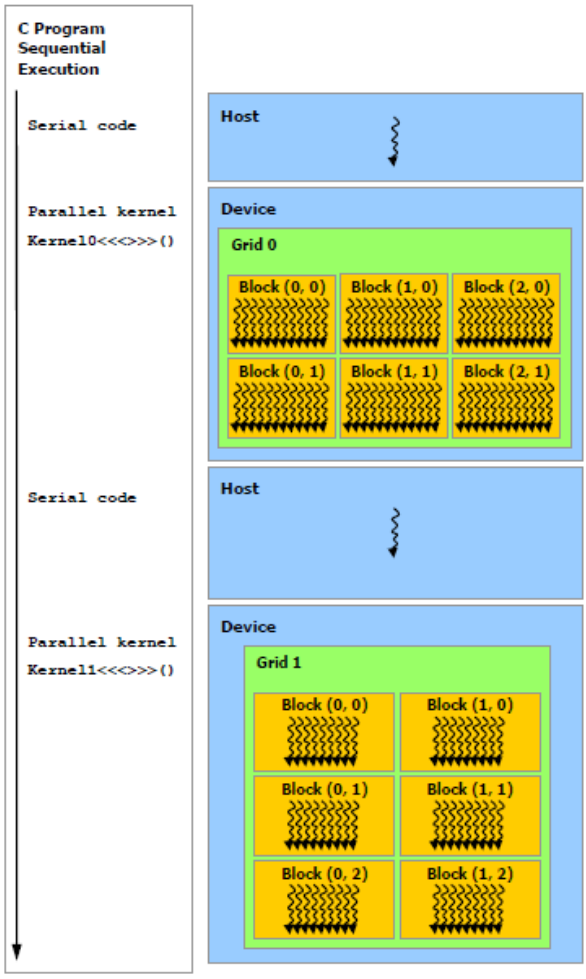

Figure 6.2 CUDA program execution between CUDA and C [3]

OSSIE is setup to automatically build the necessary configuration and make files for each component. This works well for the standard configuration of the components, but this requires

some changing to have the make files and configure files work correctly for CUDA. The main things that have to be changed within the build are that nvcc (which is the CUDA compiler) should be used in place of the normal g++ compiler. The nvcc compiler needs to be used for all CUDA files, as well as linking the executable, whereas the g++ compiler is only used for C and C++ file compilation. It is possible to use g++ for the linker, but this requires finding all the necessary libraries which becomes hard to identify. In its current implementation, the make file will not correctly identify when a CUDA file needs to be rebuilt and relinked. A simple way to get around this is to have the CUDA file built every time a C or C++ file is touched. This way, anytime a C or C++ file is changed, all CUDA files are rebuilt, with the C/C++ file and then the executable is re-linked. The CUDA object file, as well as the CUDA library (`cuda.a`) and the cutil library (cutil_x86_64 on a 64bit operating system), all must be added to the linker list when the application is linked. The CUDA library is necessary so that links from the standard CUDA operations that are needed for CUDA function calls are included. The cutil library is required for the timer function and any other CUDA utilities functions. For each CUDA file, the nvcc compiler must to be invoked twice, once to generate the -cubin file which is then used to load the function which is called from C or C++. The second time is to actually build the object file that is used in the linker. Other parameters need to be passed for the application to link correctly: `-Xcompiler -D__builtin_stdarg_start=__builtin_va_start` . This seems to be necessary due to CUDA not being able to correctly figure out where the main function is located in CUDA 3.1 when it is located in a .cu file. Adding the specified flag to the compiler arguments seemed to alleviate the problem for use in this research. The above is all that is necessary to change the make files to build and link with the CUDA compiler.

Generally, if performance and timing data is being gathered in CUDA, it is desired that similar data is recorded in the C and C++ so the two can be compared. To acquire C++ performance, significant less work is required; the process_data portion of each component is removed and put into a standalone main.cpp file to be made into a standalone executable. Creating a standalone application makes it easier to get only the performance data desired and not have to run the entire WF in OSSIE to gather the results. The main function only creates

classes as the normal component would: start the C++ timer function, run the `for loop` over the data, stop the timer, and display the results. The displayed results are also in the same time granularity to make the comparison of data easier.

Each component below is optimized in the same way for two reasons of reasons. One, it makes it easy to compare between performance in components since the same optimization is done for each. Second, it is the simplest way to optimize a looping based component. The original code inside of the process_data function appeared as shown below:

```
void process_data(void *data)
{
        PortTypes::ShortSequence I_out, Q_out;
        while(1)
        {
               PortTypes::ShortSequence *I_in(NULL), *Q_in(NULL);
               channel->dataIn->getData(I_in, Q_in); //Gets RX data;
               I_out.length(I_in->length());
               Q_out.length(Q_in->length());
               for(int i = 0; i < I_in->length(); ++i)
               {
               //Do what ever processing is necessary
               //Setting I_out and Q_out as the result values
               }
               channel->dataOut->pushPacked(I_out,Q_out);
         }
}
```

Each component below would then have different code inside of that `for loop`. The code here is only modified slightly. All of the code inside that `for loop` is removed and put into an external CUDA function call and replaces the code in the `for loop` as shown below:

```
unsigned short* length = (unsigned short*)malloc(2);
```

```
*length = I_in->length();
runCuda(I_in->getBuffer(), Q_in->getBuffer(),
                        I_out.getBuffer(), Q_out.getBuffer(), length);
free(length);
```

Note here that the getBuffer attribute of the PortTypes::ShortSequence returns a pointer to the objects help within the sequence. Since CUDA does not understand this data type, the length needs to be passed in separately. The `for loop` that was removed needs to be put somewhere, that is in the new CUDA function called from runCuda, or in the example shown above the "someFunc" function, this function, derived from the `for loop` becomes:

```
extern "C"
__global__ void someFunc(short * inputPtr, short * outputPtr, short * length)
{
      int i = blockIdx.x * blockDim.x + threadIdx.x;
      if(i < *length)
      {
            //Do the same processing here as before
       }
}
```

Doing this allows each thread in a thread block currently running on the GPU to take a given index i, determined by the blockIdx, blockDim and threadIdx and compute that one iteration of code. Multiple dimension arrays can be done using the y and z portions of those objects, but all the components shown are one dimension arrays.

In 6, a flow diagram is shown of how the flow of execution from the host side SW, to CUDA setup, CUDA execution, destroying the CUDA interface, and then returning to host side execution. Each of the components in the following subsections contains this flow of execution.

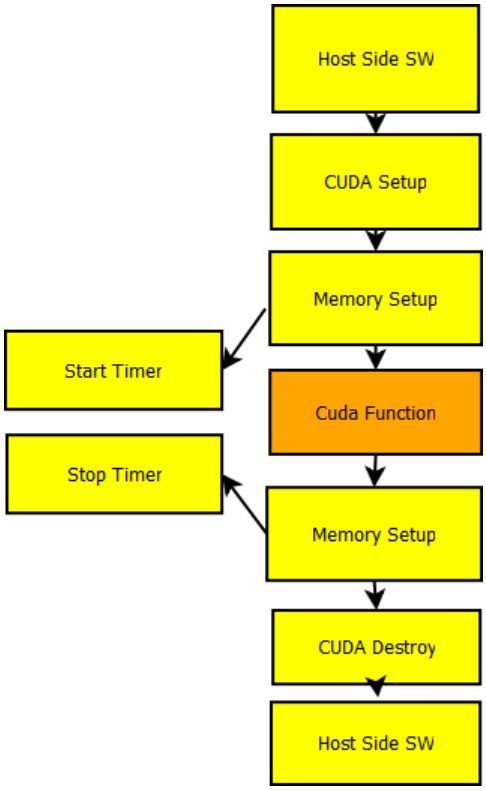

Figure 6.3 Example execution of OSSIE Components

6.1 AM Demodulation Component

The AM demodulation component, is a component used to demodulate an AM signal received over the air. AM (Amplitude Modulation) is a signal which is amplitude (linearly) modulated. A modulated signal is just simply spectral shifting the signal to gain certain advantages, i.e. making the signal easier to receive, etc [43]. AM demodulation then is simply extracting out the data inside of the carrier signal. This is accomplished quite easily, as AM demodulation is one of the easier modulation methods.

Once the component is setup from the above section, it is quite similar to the general C++ function that it was derived from. Using the thread and block indexes described above, the `for loop` is turned into a one-dimensional thread block and thread id index. The AM Demodulation is relatively simple in itself. For each element the I channel is taken and multiplied by half of the I channel, the same is done for the Q channel element and then they are added together. The output of this is then square rooted and set to the I channel output for that

element. Next, the I output value is checked against a max value, which is initially zero, and each time the I output channel is greater then the old max value, the max value is updated to what the current I output value is. Next the Q output is set to be what the I output is. This is repeated using the blockIdx, blockDim and the threadIdx of the x dimension only. Each loop is checked against the max length desired to ensure only necessary data is operated on. The length is passed in from the C++ code based on the length given from the CORBA uses port from the previous component.

6.2 FM Demodulation Component

The FM demodulation component, once again, is a component used to demodulate a modulated signal. However, this time instead of amplitude modulation, it is done using Frequency Modulation (FM). FM signals carry information on the carrier wave by varying its frequency, which is different from AM since in AM the frequency is always constant but the amplitude varies [43].

This component, as well, resembles its C++ version very closely; same as before using the thread block and thread id index, the sequential `for loop` is able to be transposed into a parallel operation. The FM component works as so; the I and Q input signals are passed into the function. For each element in the I and Q channel the atan2 is computed based on two inputs. The first input is done by using the current I channel multiplied by the previous I channel added to the current Q channel minus the previous Q channel. The second component is the previous I channel multiplied by the current Q channel then subtracted from the current I channel multiplied by the previous Q channel. This is all multiplied by 5000 to acquire the current I channel output. It is important to note that for the very first I channel element that the I out value is just zero since the algorithm implemented here is designed around the ability to use past elements. The Q output channel is then just set to be what the I channel was. The remainder of the component operation is the same; the passed in length is used to determine what thread in the GPU handles which component index.

6.3 Amplifier Component

The amplifier component is a component simply used to amplify the I and Q channel signals coming into it and pass the amplified signals out of it. This amplified value is passed into the component using CORBA but the value is defined in its XML in the .prf file, which is the properties file as stated previously. This component is setup like the others, turning the `for loop` into a parallel operation by using the thread blocks and thread indexes. The I and Q channels are once again passed in with their given lengths. This length value is used to define the number of operations, or threads needed, to complete the amplification. Each iteration just multiplies the I and Q channels by the desired signal gain, then proceeds to set the I and Q output values to the new computed signal value. These these values are next pushed onto the next CORBA component.

6.4 Decimator Component

The decimator component is used to compress the signal by picking and removing samples of the signal, also referred to as down-sampling. This can only be done for discrete time signals (digital signals), and it is generally accomplished by low pass filtering the signal and then down-sampling [43].

The decimator component works exactly as described above and the data is taken in via CORBA as an I and Q channel. The value used by the decimator to determine the coefficient to decimate by is passed in like the amplifier component, through the .prf XML file. Both the I and Q channel elements, as well as their lengths, are passed into the function. The I and Q channels, element by element, are passed into the Finite Impulse Response (FIR) filter.

The FIR filter is simply a discrete time filter where the filter coefficients determine what the filters impulse response is [43]. The coefficients for the FIR filter are specified by the decimator value multiplied by 8 and then added by one. The component then calls DesignRRCFilter which creates a root raised-cosine filter based on the desired samples per symbol, the delay, and roll off factor (this is between 0 and 1). Next, this populates a fourth parameter which ends up being the coefficients needed for the decimator. For the decimator, the roll off factor

(or beta) is always equal to 0.5, the delay is always 2, and the samples per symbol is 2 times the decimating factor. This is initialized prior to any data being accepted by the component. Each FIR filter output gives the response of that specific I or Q element; which is accomplished by taking each I and Q input separately and multiplying each element passed in by each coefficient and summing this number up and passing it as the output. This is set to the I and Q respective output parameters, which are then passed out of the GPU and onto the next component. The decimator is optimized as the other components by transforming the `for loop` into data that can be computed on individually by each thread.

6.5 Interpolator Component

The Interpolator component is used to construct new samples of data within an already existing signal data range. Interpolation is used to expand the signal. For example, in the case where certain data points in a given range were missing. The XML .prf file is used to configure the interpolator component just as the other components did. The interpolator factor, the shape of the filter, symbol delay, and the bandwidth factor can all be set and passed in through XML. The interpolator works by filtering the I and Q elements that are already obtained by utilizing user defined coefficients, but for all the new data points, a value of zero is input into the filter to generate the predicted output signal elements.

The interpolator component works the same as the decimator, by grabbing data, filtering on it, and then passing the data on. The filter is initialized in a similar way by using the XML parameters to configure the FIR filter coefficients by calling the DesignRRCFilter just as the decimator did. The difference is this time none of the parameters are defaulted but instead every single parameter to the filter is passed in through the XML. This component, however, processes the data differently though it loops over the entire length of I and Q sample as the decimator did. It now computes a FIR filter value, and then uses the symbol delay to insert a value of 0 into the FIR filter to generate output samples for each element of length of symbol delay. Once this is completed, the data is then passed onto the next component. This also utilizes the same mentality of modifying the `for loop` to be processed by parallel threads.

CHAPTER 7. Experimental Results

The data gathered from running the components in C++ and CUDA was obtained using the timer set up in chapter six. The C++ timer uses built in structures to the language that obtain time from the operating system. The Linux function gettimeofday populates the timeval struct by obtaining time since the previous Epoch. The function gettimeofday is called before and after the desired function being measured, just like the CUDA timer. Each call stores the data in the timeval struct, and then the tv_usec element of the struct is subtracted from each to return the time in terms of nanoseconds.

To make the tests all equal, the same timer functions were used among all the CUDA applications and all the C++ applications, but the thread blocks also needed to be equal as well. This was done by creating 500 threads in each thread block. The number of blocks was determined by taking the number of elements passed in, divided by 500 (number of threads) and adding 1 to get enough thread blocks to do the entire computation. Doing this showed that the GPU was using 66.7% utilization for each run. This was viewed using the CUDA_PROFILE variable stated previously. This is important for multiple reasons, one in that each test run is the same, and two, so that the GPU is not overloaded. If the GPU is running at 100% utilization it can cause it to run in a non-optimal manner.

Before digging into the performance results, it is important to note that when using the CUDA_PROFILE flag to gather performance results of timing and utilization of the GPU operation causes the run time to be skewed. Generally having the profiling on would cause the results to be 20% worse. To alleviate this, runs were completed separately to gather the utilization data from the timing data when allowed while the CUDA SW was running. Gathering this data separately allowed the least amount of overhead to exist.

Table 7.1 CPU Performance in milliseconds in power of 2 sample increments

Components	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
AM Demodulation	0.024	0.041	0.081	0.158	0.308	0.603
FM Demodulation	0.019	0.033	0.069	0.135	0.261	0.516
Amplifier (Gain = 10)	0.006	0.011	0.028	0.058	0.112	0.223
Decimator (Factor = 10)	0.072	0.141	0.293	0.572	1.136	2.266
Interpolator (Factor = 10)	2.771	5.54	10.482	20.14	36.877	71.471

Table 7.2 GPU Performance in milliseconds in power of 2 sample increments

Components	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
AM Demodulation	0.039	0.041	0.047	0.066	0.083	0.131
FM Demodulation	0.044	0.045	0.051	0.076	0.104	0.177
Amplifier (Gain = 10)	0.038	0.043	0.052	0.077	0.112	0.183
Decimator (Factor = 10)	0.353	0.359	0.404	0.460	0.826	1.37
Interpolator (Factor = 10)	1.152	1.734	3.322	6.419	14.873	24.87

Table 7.1 contains all of the performance data obtained in running the components on the CPU. The sample data was inserted into the component starting at 512 and doubling up to 16384. 512 was used as the starting sample size since most of the demo WFs in OSSIE used 512 as a sample size. The AM and FM Demodulation components had no specific parameters that needed to be incorporated like the amplifier, decimator and interpolator components did. For the three components that did, a value of 10 was used for the gain in the amplifier and the factor in the decimator and the interpolator. These same values were used when acquiring the GPU performance results as well. The setup CPU performance data was run on a Intel Core 2 Duo E6750 at 2.66Ghz with 4GB of RAM. The GPU was run on a GeForce 8800 GTS with 512Mhz shader clock, 320MB of video memory and 96 CUDA cores to be used.

To comment briefly on the data itself, in all the components, no matter the complexity, the time of execution increased linearly doubling as the sample size was doubled. This is to be expected given the sequential nature of the CPU. The CPU can only execute one thread at a time so any additional data cannot be sped up by running multiple threads like they can be in the GPU. It is also worth noting that the interpolator results are somewhat deceiving because of its long execution time, but this is due to the fact that for each input sample, 10

Table 7.3 GPU Setup in Milliseconds

Components	Device Setup	Mem Setup (Prior)	Mem Setup (After)	Destroy
AM Demodulation	23.95	0.264	0.116	15.303
FM Demodulation	25.26	0.254	0.12	12.91
Amplifier	22.48	0.269	0.111	13.24
Decimator	23.81	0.298	0.112	12.809
Interpolator	23.32	0.313	1.45	14.18

extra samples were created (due to its factor being 10). This is part of the reason it has such a longer execution time opposed to the decimator.

Now looking at the GPU data in table 7.2, on first look the data is defiantly on the same order of magnitude as the data seen in table 7.1 but as the data set size increases, the same trend does not form. This time the execution time does not scale linearly as it did with the CPU data. The GPU execution time certainly increases as the data set size rises, but at a significantly slower rate. It is worth noting that on the 512 data set and the 1024 data set, the GPU has a longer execution time than the CPU. That is due to the cost of making the CUDA function call, since once the cost of the CUDA call itself can be reclaimed based on faster performance of the execution of the desired processing, the cost is less apparent. This is alluded to in the 2048 sample size case, but is seen more in the larger data sets. Depending on the given component, the GPU can see speed up of 4.6 times in the AM Demodulation case, but only 1.65 in the Decimator case. This has to do with the sort of computation the component is doing and how well the GPU can perform at doing it. Depending on what size data sets are used the GPU can really shine performance wise, but given the sample size used, the speed up could be very minimal.

In table 7.3, the table shows the execution cost to setup the GPU device, copy data into GPU device pointers, copy data out of GPU device pointers back into CPU memory, and then to destroy the GPU device. As can be seen immediately, the cost of creating the GPU device and destroying the device is quite high; on average 23ms to create and 13ms to destroy. This cost is the same for all CUDA programs no matter the data size or complexity. The small variance in time between the runs is due to the difference in components. Though this cost is

high, it can just be s accomplished in the constructor and destructor portion of the C++ class. This allows the cost of the call not to affect the performance of the component in actual operation, only in construction. Generally startup time does not matter, unless there are timing requirements on how long a radio has to come up and tear down.

The memory setup before and after are the pieces of setup that have to be done for each component. This entails allocating memory for each input parameter, setting up the size of the input parameters to be passed, and memory copying any data from the CPU memory that needs to be in the GPU memory. Since nearly all the components had the same amount of input parameters, except the decimator and interpolator, the performance was nearly the same. The decimator and interpolator had three extra input parameters, but as can be seen, the cost of these extra parameters is minimal. The memory setup after involved copying any of the data the GPU computed back into the CPU memory space. Since all the components just computed an updated I and Q channel, the times are nearly all the same, except for the interpolator component. This is due to the fact that the interpolator created 10 extra samples for every input sample, dividing the 1.45ms by 10 gives .145ms which is right on par with where the rest of the components were at.

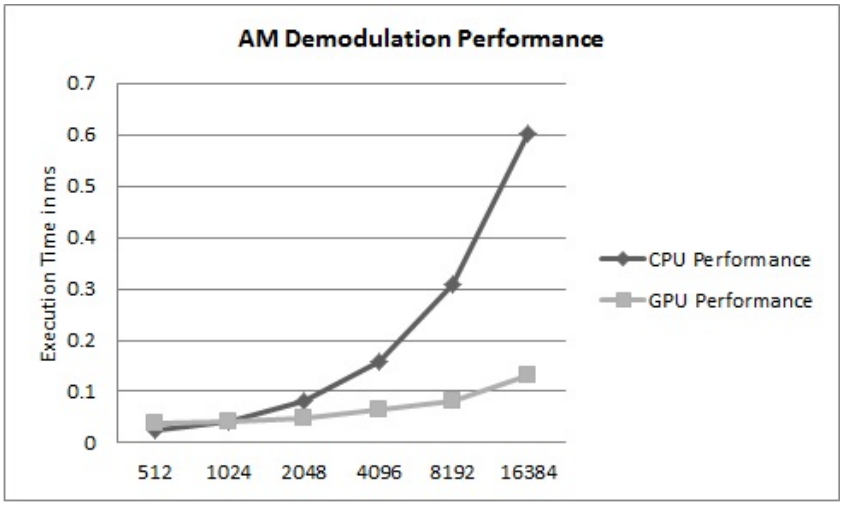

Figure 7.1 Plot of GPU vs CPU of AM Demodulation Performance

Figure 7 is a plot of the execution time of the GPU performance versus the performance of the CPU for the AM demodulation component. The GPU is able improve over the GPP on

this component as early as 1024 samples, and then increase at a very moderate compared to the CPU performance.

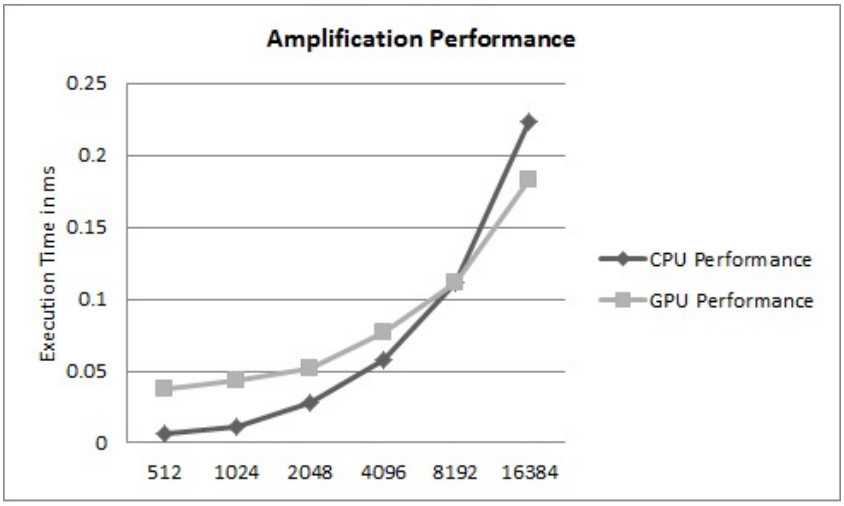

Figure 7.2 Plot of GPU vs CPU of Amplifier Performance

Figure 7 is a plot of the execution time of the GPU performance versus the performance of the CPU for the amplifier component. The GPU on this component took a lot longer to improve over the GPP, all the way up to the 8192 samples, the worst of any the components. This was most likely due to the simple nature of the component, since is only multiplied the I and Q channel by a constant; the cost of the CUDA call was very high then in relation to the computation performed.

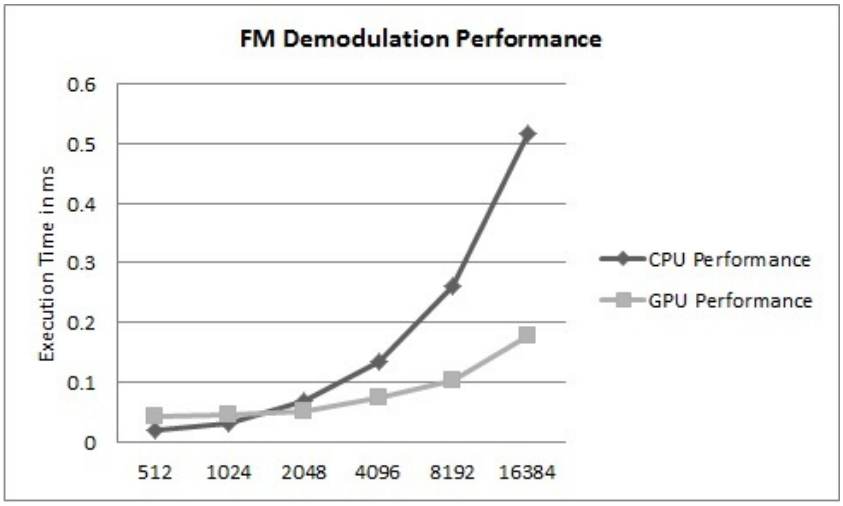

Figure 7.3 Plot of GPU vs CPU of FM Demodulation Performance

Figure 7 is a plot of the execution time of the GPU performance versus the performance of the CPU for the FM demodulation component. This component, similar to the AM demodulation component, has the GPU gaining the computational edge around the 1024 sample mark. From this point on the CPU showed increasingly bad performances in all sample sizes.

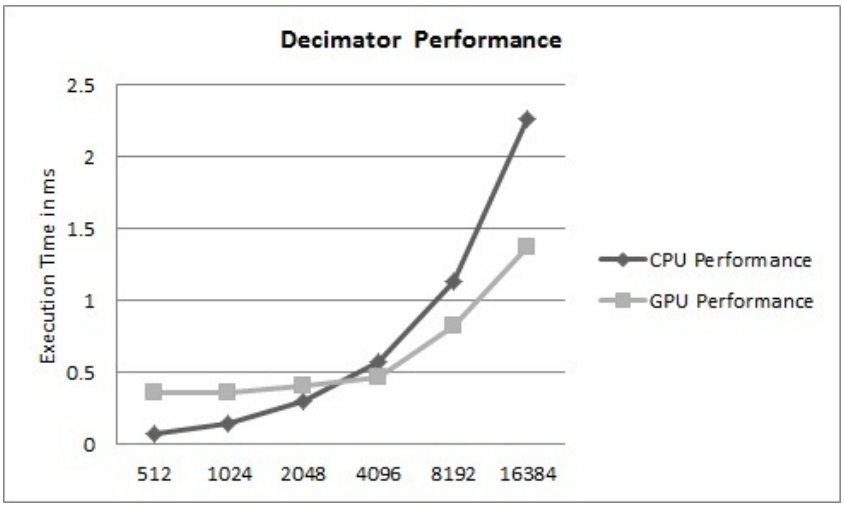

Figure 7.4 Plot of GPU vs CPU of Decimator Performance

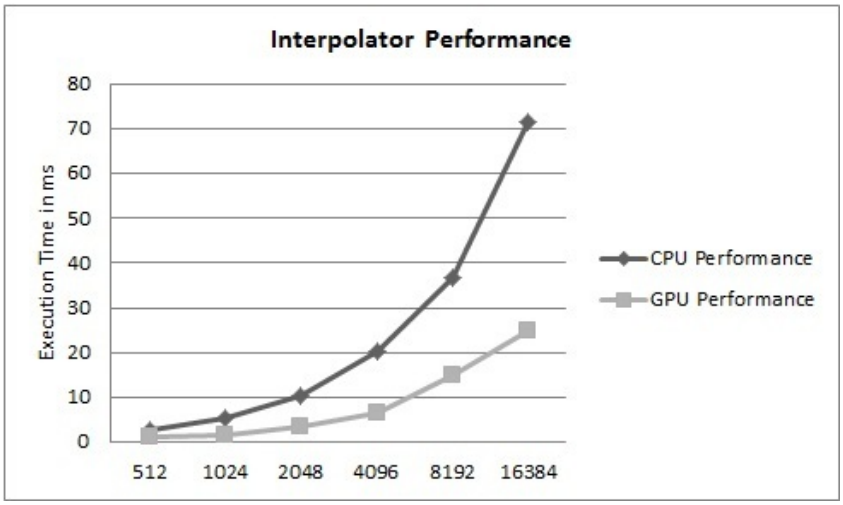

Figure 7.5 Plot of GPU vs CPU of Interpolator Performance

Figure 7 is a plot of the execution time of the GPU performance versus the performance of the CPU for the decimation component. The GPU is able to improve over the CPU approximately 4096 samples, but the GPU performance follows the CPU performance at a consistent

rate and does not show as high of a speed up as with other components.

Figure 7 is a plot of the execution time of the GPU performance versus the performance of the CPU for the interpolator component. This is the one component where the GPU performs better then the CPU in all the sample sets, and as the sample size increases the GPU performance only gets better.

All of the performance data gathered between the five components tested demonstrate that depending on the computational task of the component, the sample size at which the GPU outperforms the CPU can vary. However, there is always a point at which the GPU will outperform the CPU. This becomes more visible as the sample size grows larger in size, the performance difference becomes more and more apparent.

CHAPTER 8. Conclusion and Future Work

Throughout this thesis we have shown how OSSIE can be modified to run SDR components on the GPU, opposed of its normal implementation on a GPP. The GPU has shown performance improvements over the GPP of various magnitudes from the amplitude component where it took a significantly large amount of samples to see the performance gain, to the interpolator component where the improvement was seen almost immediately. This paper only looked at a small set of components and specific data set within those components. In this data set though, the original expectations were met. As expected, the CPU performed worse as the sample sizes were increased. The GPU showed in various component contexts that even in the simplest of computations, if it can be parallelized, the GPU can improve the performance. The only question remains is how many samples does it take before the GPU outperforms. The more computationally intensive or more parallelizable it is, the faster those results can be obtained. The main down side of using the GPU as a gateway to gaining performance improvement, is that designing an embedded SDR which has a Desktop PC connection is not realistic. Fortunately, given the drastic rise in multimedia applications on cell phones, an increasing number of embedded devices now have GPUs that contain significant processing power. Due to this, multiple manufactures are creating powerful embedded GPUs that give the developers the ability to use GPUs on embedded systems to boost performance in non-graphics based applications.

This thesis used components in OSSIE that were easily decoupled to make the porting to the GPU less complex. Future work in this area could be seen in taking the more complicated components that involve multiple intertwined classes to be decoupled and re-written in a C-like architecture for CUDA to take advantage of its parallelized architecture. Given the knowledge

that has been gained from this paper, porting new, more complex components would be simpler. The initial CUDA hurdles which have already been broken since the main complexity is getting data in and out of the component. In the beginning of this paper, the way to create new components in OSSIE was discussed. The components that have been ported in this paper have been done in a consistent way that would make it even easier to do so in a newly created component since OSSIE will create the rest of the necessary SCA portions of the component. This will be of added benefit since the process_data portion can be created initially in a manner that is designed to be used in a parallel way opposed to the typical sequential type of programming used in C and C++.

Though the OSSIE tool contains many components which directly apply to SDR, it is missing the more advanced components that are commonly used in more complete commercial SDR implementations. Components that could fall into this category would be applications like encoders and decoders. Examples are reed-solomon and viterbi as well as encryption and decryption algorithms like Advanced Encryption Standard (AES) and Data Encryption Standard (DES). These types of components are commonly used in SDR to improve security to protect data from intruders listening into a live network. The encoder and decoder algorithms are used for data parity to ensure that no data is lost between the communication of terminals. Given the nature of these algorithms, they can be quite computationally intensive and would fit well in a GPU parallel architecture. The GPU design would greatly improve upon sequential performance that would be seen in a C++ implementation. These components would provide great added value to OSSIE given the typical use of SDR and would make OSSIE more dynamic in its ability to adapt to different types of SDR WFs.

As was brought up earlier, OSSIE creates three XML files for each component created, one of them being the .spd file. The method used generally in this thesis was to copy the component that already existed, append the GPU term on the end of the existing name in all classes, and then modify the make files and XML. Instead, it may be possible to write the .spd file in such a way that depending on the XML settings specified, the GPU or GPP version of the component could be used. Using XML as the configuration management, switching between the GPU and

GPP versions of the component becomes minute and does not require a recompile. There is great potential for various future SDR work which can be done with this OSSIE and GPU set of ideas as well as SDR and GPU in general. As other concepts are further explored, the GPU will continue to grow in popularity because of the improvements that can been seen in all software complexity aspects.

Bibliography

[1] L. Pucker and G. Holt, "Extending the SCA core framework inside the modem architecture of a software defined radio," *IEEE Communications Magazine*, vol. 52, pp. S21–S25, 2004.

[2] JTRS, *Software Communications Architecture Specification*, JTRS Std., Rev. 2.2.2, May 2006.

[3] NVIDIA, *NVIDIA CUDA C Programming Guide*, May 2010.

[4] B. Brannon, "Software defined radio," *Analog Devices, Inc.*

[5] J. Mitola, "The software radio architecture," *IEEE*, 1995.

[6] C. Kopp, "Network centric warfare fundamentals," *DefenseToday Magazine*.

[7] M. R. Turner, "Software defined radio solutions experience making JTRS work, from the SCA, to waveforms, to secure radios," *SDR Technical Conference and Product Exposition*, 2005.

[8] B. Le, F. A. Rodriguez, Q. Chen, B. P. Li, F. Ge, M. ElNaina, T. W. Rondeau, and C. W. Bostian, "A public safety cognitive radio node," *SDR Forum Technical Conference*, 2007.

[9] OMG, *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1*, Object Management Group Std., Rev. 3.1, January 2008.

[10] K. Dove, "Nvidia unveils CUDA - the GPU computing revolution begins," November 2006.

[11] AMD, "Brook+," AMD, Tech. Rep., 2007.

[12] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *SIGGRAPH*, 2004.

[13] NVIDIA, *NVIDIA CUDA reference Manual*, 3rd ed., June 2010.

[14] M. Carrick, "Logical representation of FPGAs and FPGA circuits within SCA," Master's thesis, Virginia Polytechnic Insitute and Unversity, July 2009.

[15] E. Blossom. (2004, November) Exploring GNU radio. http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html.

[16] Unknown. (2006) GNU Radio passive radar project. http://comsec.com/gnuradio-project-2006.html.

[17] I. imek and R. Rakesh, "GPU acceleration of 2D-DWT image compression in MATLAB with CUDA," *Second UKSIM European Symposium on Computer Modeling and Simulation*, 2008.

[18] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU accelerated high performance computing in virtual machines," *IEEE*, 2009.

[19] S. Chen, ling Qin, Y. Xie, W.-M. Pang, and P.-A. Heng, "CUDA-based acceleration and algorithm refinement for volume image registration," *BioMedical Information Engineering*, 2009.

[20] V. Simek, R. Dvorak, F. Zboril, and V. Drabek, "GPU accelerated solver of time-dependent air pollutant transport equations," *Digital System Design, Architectures, Methods and Tools, Euromicro Conference*, 2009.

[21] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction," *NSF*, 2008.

[22] J. L. Herraiz, S. Espaa, S. Garca, R. Cabido, A. S. Montemayor, M. Desco, J. J. Vaquero, and J. M. Udias, "GPU acceleration of a fully 3D iterative reconstruction software for PET using CUDA," *IEEE Nuclear Science Symposium Conference Record*, 2009.

[23] C.-K. Chiang, S.-F. Wang, Y.-L. Chen, and S.-H. Lai, "Fast JND-based video carving with GPU acceleration for real-time video retargeting," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, 2009.

[24] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," *IEEE*, 2010.

[25] L. L. Jianming Li, Lihua Zhang, "A parallel immune algorithm based on fine-grained model with GPU-acceleration," *IEEE*, 2009.

[26] D. Goddeke, S. H. Buijssen, H. Wobker, and S. Turek, "GPU acceleration of an unmodified parallel finite element navierstokes solver," *High Performance Computing and Simulation International Conference*, 2009.

[27] M. Ujaldon, "GPU acceleration of zernike moments for large-scale images," *IEEE International Symposium*, 2009.

[28] J.-M. Laferte, G. Daussin, J. Flifla, and P. Haigron, "Real-time forest simulation for a flight simulator using a GPU."

[29] W. Zhu and J. Curry, "Particle swarm with graphics hardware acceleration and local pattern search on bound constrained problems," *Swarm Intelligence Symposium, SIS 2009*, 2009.

[30] X. Han, L. S. Hibbard, and V. Willcut, "GPU-accelerated, gradient-free MI deformable registration for atlas-based MR brain image segmentation," *IEEE*, 2009.

[31] T. Killian, D. L. Faircloth, and S. M. Rao, "Acceleration of TM cylinder EFIE with CUDA," *IEEE*, 2009.

[32] S. E. Krakiwsky, L. E. Turner, and M. M. Okoniewski, "Graphics processor unit (GPU) acceleration of finite-difference time-domain (FDTD) algorithm," *IEEE*, 2004.

[33] J. M. Ready and C. N. Taylor, "GPU acceleration of real-time feature based algorithms," *IEEE*, 2007.

[34] O. Fialka and M. Cadik, "FFT and convolution performance in image filtering on GPU," *IEEE*, 2006.

[35] S. Peng and Z. Nie, "Acceleration of the method of moments calculations by using graphics processing units," *IEEE*, 2008.

[36] L. Kun, "Time-domain finite element method (TD-FEM) algorithm," *IEEE*, 2009.

[37] J. Kim, S. Hyeon, and S. Choi, "Implementation of an SDR system using graphics processing unit," *IEEE*, 2010.

[38] P. Szegvari and C. Hentschel, "Scalable software defined FM-radio receiver running on desktop computers," *IEEE*, 2009.

[39] A. Akapyev and V. Krylov, "Implementation of 802.11n on 128-core processor," *ISCA PDCCS 2008*.

[40] M. Wu, S. Gupta, Y. Sun, and J. R. Cavallaro, "A GPU implementation of a real-time MIMO detector," *IEEE Workshop on Signal Processing Systems*, p. 6, 2009.

[41] G. Falcao, V. Silva, and L. Sousa, "How GPUs can outperform ASICs for fast LDPC decoding," *ACM*, p. 10, 2009.

[42] NVIDIA. (2009, September) NVIDIA CUDA library documentation. http://www.owlnet.rice.edu/ comp422/resources/cuda/html/index.html.

[43] B. Lathi, *Linear Systems and Signals*, 2nd ed. Oxford University Press, 2005.