UNIVERSITY OF HUDDERSFIELD

GPU Cluster for Acceleration of Scientific and Engineering Applications in the Context of Higher Education

Author:
Matthew NEWALL

Supervisor:
Dr Violeta HOLMES

A thesis submitted in fulfilment of the requirements for the degree of Masters By Research

High Performance Computing
School Of Computing and Engineering

February 2015

Copyright Statement

Abstract

Many fields of research now rely on High Performance Computing (HPC) systems which can process ever larger datasets, with increasing accuracy and speed. Many universities now provide a HPC service. Following the trend over the past few years of the worlds fastest supercomputers being accelerated using Graphical Processing Units (GPUs), there is a growing interest in the use of GPUs in Higher Education Institutions. The characteristics of GPUs make them excellently suited to any task exhibiting a high level of data parallelism. Recent developments in GPU technologies have focused on improving performance and integration in HPC, and for processing data other than display graphics.

To investigate the benefits such a system could have to the University of Huddersfield, a small GPU cluster has been deployed. The intention behind this thesis is to detail the deployment of the system and to demonstrate, through case studies, the required effort a potential user could expect in order to take advantage of it.

As a result of this work it can be demonstrated that even a modest GPU cluster can be of benefit to the University. The cluster is helping our researchers to analyse complex data using visualisation, and accelerating data processing.

Acknowledgements

I would like first and foremost to thank my supervisor Violeta Holmes for her invaluable advice and support both day and night. Without her this project would not have been possible. I would also like to thank Paul Lunn from Birmingham City University for providing his software for the second case study, and for his contributions to publications based on this work. Thanks to Graeme Greaves from the Electron Microscopy Materials Analysis group at Huddersfield for providing data, And to Hussam Muhamedsaleh for allowing use of his software in the final case study. Finally i would like to thank my collegues Ibad Kureshi, Stephen Bonner and John Brennan for their technical and logistical assistance.

Contents

List of Figures

List of Tables

Abbreviations

GPU	**G**raphics **P**rocessing **U**nit
HPC	**H**igh **P**erformance **C**omputing
AD	**A**ctive **D**irectory
QGG	**Q**ueens**G**ate **G**rid
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
MPI	**M**essage **P**assing **I**nterface
API	**A**pplication **P**rogramming **I**nterface
CFD	**C**omputational **F**luid **D**ynamics

Chapter 1

Introduction and Background

High Performance computing systems are now well established as an essential tool for research and analysis in Higher Education. A High Performance computing system is one which uses multiple computers to carry out one task, through parallel processing. Many Universities have some kind of HPC provision. At the University of Huddersfield a number of HPC systems have been deployed for the use of researchers and students (Kureshi, 2010). These systems are used regularly for tasks such as large CFD simulations, running MPI software, image rendering and more. However, the fastest computer HPC systems now include Graphic Processing Units (GPU) (Top500, 2013a). This research is attempting to answer the following questions

- Why do we need an alternative to CPU systems for scientific processing
- Why are GPUs so suited to this task
- What issues would be faced in deploying a GPU system.
- How would such a system support research in a Higher Education institution such as the University of Huddersfield.

Aside from the university of Huddersfield, many HE institutions have some kind of HPC system. However, while GPUs are seeing widespread adoption in research and scientific institutions, they still remain comparatively rare in HE.

1.1 Aims and Objectives

The overall goal of this project was to investigate the viability of a GPU cluster to compliment the HPC provision available at the University of Huddersfield. The key objectives for success were:

- Investigate the application of GPU technologies in HPC systems.
- Evaluate existing GPU systems in the context of Higher Education to support scientific research.
- Deploy a GPU cluster within a HE institutional grid.
- Investigate different platforms and programming environments for the purpose of highly parallel task and data processing and visualisation.
- Evaluate the suitability of the GPU cluster using representative case studies.
- Propose possible future developments of the system to accelerate HE scientific research.

The final outcome of this project was considered to be a functional GPU cluster, successfully integrated as part of the HPC provision at the university, justified through a number of case studies.

1.2 Methodology

In order investigate how best to achieve the project aims and objectives, a comprehensive literature review was conducted, A GPU cluster was deployed and integrated into the campus grid, and case studies were carried out to evaluate its usefulness in supporting the visualisation and processing of data.

The result of this study is presented in this thesis and is organised as outlined below:

- Chapter 2: Investigate existing HPC sytems, the use of GPUs in such systems and studies on existing GPU clusters, Cluster Middlewares, GPU programming models and software.

- Chapter 3: Design and deployment of a GPU cluster is described.
- Chapter 4: Outlines a selection of appropriate software, platforms, and programming environments for visualisation and processing of data, based on the literature.
- Chapter 5,6, and 7: Presents the results of testing the usefulness and suitability of the cluster through three representative case studies.
- Chapter 8 and 9: Evaluate the outcomes of the project and propose future developments.

Chapter 2

Literature Review

2.1 GPUs as general purpose processors

There have been numerous efforts up to this point to use GPUs as general purpose processors. An important concept to bear in mind when establishing the suitability of an algorithm or program for processing on a GPU is arithmetic intensity, which as stated by Harris (2005), is the ratio of computation to bandwidth. This is significant as computational speed currently increases more rapidly than communication speed, so memory access latency will negate potential computational speedup. Problems with a high arithmetic intensity are those which exhibit a high level of data parallelism, and have minimal dependency between data points.

Fung and Mann (2005) present an API, OpenVIDIA, which allows a user to leverage GPUs to perform computer vision and Image processing tasks. The API uses openGL to interact with the GPUs. In an image processing or filtering operation, the filters are written as shaders in Cg (described in the paper as ”fragment programs”), ”to apply these fragment programs to input images, the input images are initialized as textures and then mapped to quadrilaterals.” (Fung and Mann, 2005, chap. 3.1). The architecture of the GPU allows each pixel to be processed in parallel providing a significant speed boost. The same paper describes an implementation of a computer vision algorithm which uses the full pipeline of the GPU, a Hough Transform. An edge detection is performed on the GPU as a filter operation. The coordinates of these edge pixels are then fed back to the GPU as an array of vertices, these vertices are further processed

on the GPU through its hardware projection matrix, which has been programmed to perform the Hough Transform (Fung and Mann, 2005, chap. 3.2).

Efforts have also been made to abstract the process of GPU acceleration to high level operations. Tarditi et al. (2006) describe Accelerator; ”a system that uses data parallelism to program GPUs”. Accelerator allows programmers to accelerate suitable parts of their code using GPU accelerated functions, without exposing any aspect of the GPU. Unlike the traditional approach, which would involve compiling the data-parallel parts of the program as shader programs in native code, Acellerator compiles GPU parts of the code on the fly at runtime. Using this software Tarditi et al. (2006) were able to demonstrate an acceleration of up to 18 times over native C code running on a CPU. While it does not match the performance of hand written GPU code, its simplicity of use makes it an important step towards accessible GPU acceleration.

As the use of GPUs as accelerators became increasingly common, GPU manufacturers started to change the design of their units accordingly. Fig. 2.1 Shows the architecture of the Nvidia GeForce GPU; it is a design highly specialised to graphics with hardware designed to fulfil specific parts of the graphics pipeline, as was the case most GPUs up to this point. As can be seen in Fung and Mann (2005, chap. 3.2), and Tarditi et al. (2006, chap. 3.1), programming GPUs required determining which part of the graphics pipeline was best suited to the problem, and could require vastly different approaches depending on which part of the GPU was targeted. To use the full pipeline often meant numerous copies to and from host memory as it was not possible to utilise the internal transports on the GPU.

This changed with the Nvidia GeForce 8800, which was the first card based on the propriety compute unified device architecture (CUDA). As can be seen in Fig. 2.2, CUDA devices replace the hardware pipeline with high numbers of ’stream processors’. As seen in Fig. 2.3, most of the pipeline is now implemented on the stream processors. This makes CUDA GPUs much more easily programmable than was previously possible.

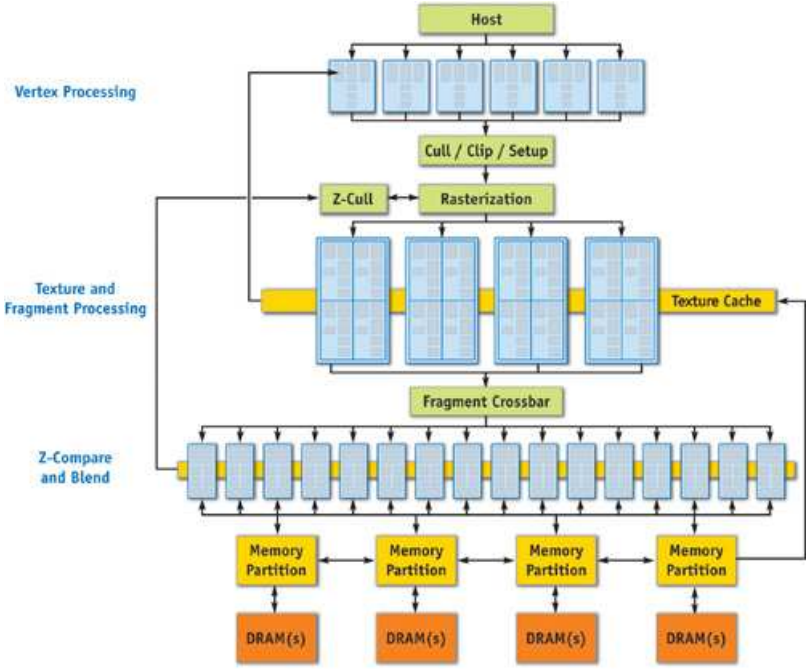


FIGURE 2.1: Architecture of the Nvidia GeForce 6800 GPU, showing pipelined design with separate units for Vertex processing, Rasterization and Blending (Nvidia, 2006)

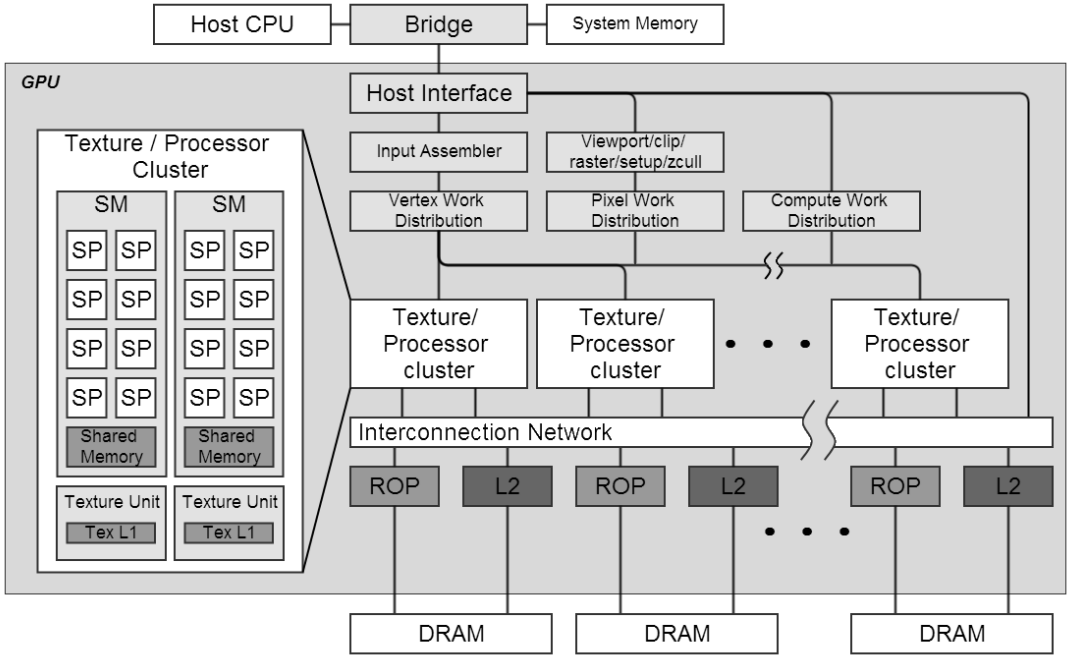


FIGURE 2.2: Architecture of the Nvidia TESLA GPU, showing unified design with general purpose streaming processors (Lindholm et al., 2008)

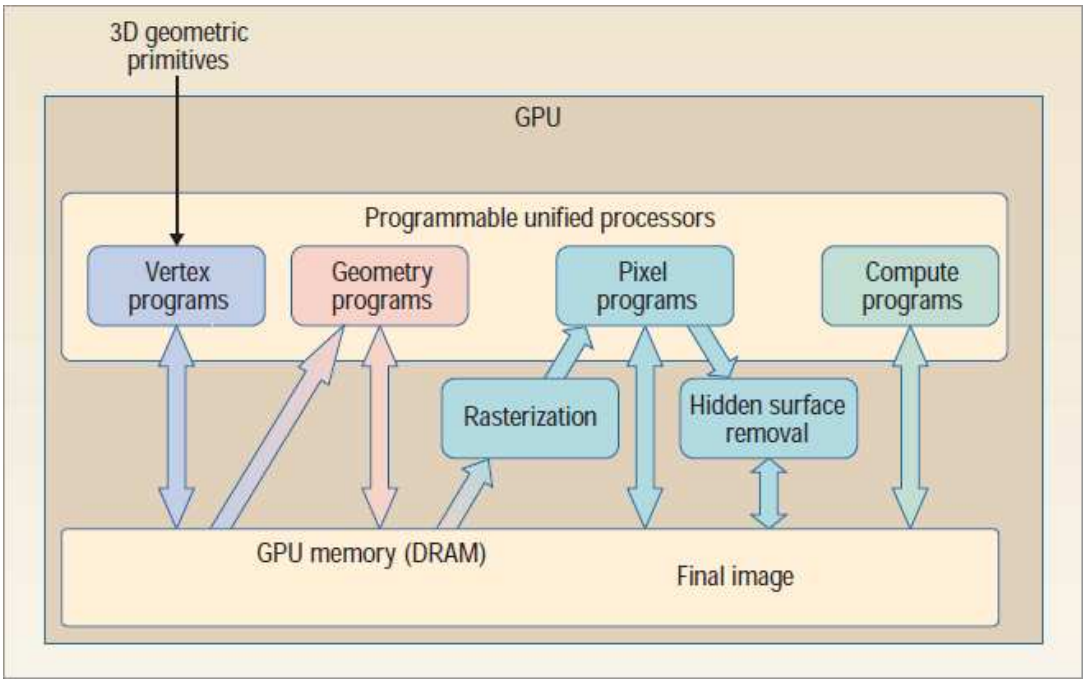


FIGURE 2.3: A representation of the graphics pipeline as implemented on CUDA GPUs (Luebke and Humphreys, 2007)

2.1.1 GPU Programming Frameworks

2.1.1.1 Nvidia CUDA

As well as the CUDA platform itself, Nvidia actively develop a programming model for its GeForce, Quadro and Tesla processors. It is highly scalable and will run on an arbitrary number of processors without the need to recompile. This is required because of the vast and varying number of processor cores in modern GPUs (Nvidia, 2008). As observed by Garland et al. (2008), the CUDA framework allows a programmer to use the massively multi-threaded nature of GPUs for a wide range of highly parallel problems. While this was possible previously, using graphics APIs, CUDA relieves the programmer of the requirement of intimate knowledge of the target hardware. Additionally, it removes the daunting task of having to negotiate a complicated and unfamiliar language, as was previously the case when targeting different parts of the graphics pipeline. All of this allows the programmer to instead ”focus on the important issues of parallelism - how to craft efficient parallel algorithms” (Garland et al., 2008, chap. 2).

CUDA programs consist of two parts;

- The host program - This can be one or multiple threads, running on the CPU

- Parallel kernels - CUDA kernels are scalar sequential programs which execute on the GPU. These sequential programs operate on a grid of thread blocks, and must be able to be execute independently.

As an example, consider the loop in Listing. 1.1 The code iterates over a grid of data and performs the function at each point one at a time. The parallel version in listing. 1.2 performs the same function, but rather than looping through, each point is processed essentially simultaneously. (Garland et al., 2008)

```
void serial_function (int n, float a, float *x, float *y)
{
    for (int i = 0; i<n; i++)
        y[i] = a*x[i] + y[i];
}
//perform on 1M elements
serial_function(4096*256, 2.0, x, y);
```

LISTING 2.1: A standard C function

```
void gpu_function (int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i<n) y[i] = a*x[i] + y[i];
}
//perform on 1M elements
gpu_function<<4096, 256>>(n, 2.0, x, y);
```

LISTING 2.2: The same function as might be written for execution on a CUDA supported GPUNvidia (2013a)

2.1.1.2 openCL

OpenCL is a parallel programming standard, with notable contributors such as Apple, ARM, AMD, Samsung and NVidia. It allows programs to take advantage of a very diverse array of processing devices such as GPUs CPUs DSPs and FPGAs. The standard is open source and provides mechanisms for hardware vendors to add access to hardware specific features (Nvidia, 2013d). Listing 2.3 shows and example of the previously shown GPU function as expressed in openCL.

```
float gpu_function (float a, float *x, float *y)
{
    return y = a*x + y;
}

kernel void function(global write_only y, global a, global x)
{
    int i=get_global_id(0);
    y[i]=gpu_function(2.0, x[i], y[i]);
}
```

LISTING 2.3: An example of a functiuon as might be written for OpenCL

2.2 Obstacles to using GPUs for scientific calculations

There are a number of caveats which must be kept in mind when using GPUs for calculations for which they were not designed. Graphics processing is highly tolerant to small errors so GPUs have not traditionally included error checking and correcting (ECC) memory systems. Scientific calculations, on the other hand, are highly sensitive to error. In response to this uncertainty, Haque and Pande (2010) have investigated the implications of using GPUs with non-ECC memory in non graphics calculations.By running a memory test program on more than 50,000 GPUs from the Folding@Home project, it was demonstrated that there is a statistically significant probability of transient errors occurring in GPU memory.

However, in response to the requirements of the scientific research community, GPU manufacturers have developed architectures specifically designed for general purpose processing.

Nvidia Fermi is one such architecture, developed from the start to be well suited for GPGPU purposes. It is described in detail in its whitepaper (P.Glaskowsky, 2009). Fermi finally introduces ECC memory protection for DRAM, as well as shared memories, L1 and L2 cache, and register files. Another notable improvement is double precision floating point performance, which was less important for graphics processing (which typically consists of single precision, 32 bit calculations), but very important for scientific calculations.

Even with ECC RAM there may be other precautions required when considering floating point accuracy on GPUs. While Nvidia GPUs with compute capability 2.0 and higher (Tesla M series and onwards) are compliant with the same IEEE 754 standard as CPUs for double precision floating point operations, Whitehead and Fit-florea (2011) suggest that there may be further precautions required to ensure accuracy when using Nvidia GPUs. There are fundamental differences in rounding modes on GPUs when compared to x86. as described in Whitehead and Fit-florea (2011, chap. 4.7) ”rounding modes are encoded within each floating poin instruction instead of dynamically using a floating point control word”. Unlike CPUs, There is no mechanism to indicate overflowed or underflowed calculations, or calculations with inexact logic. This implications of this are discussed later in the paper; while it is possible to produce the best floating point result for simple math operations, problems arise with more complex functions. While the same input will yeild the same results in an individual IEEE 754 compliant operation, differences in the potential sequencing of operations in GPUs, compared to CPUs, may mean differences in numerical results, even in newer hardware.

The authors offer some recommendations to ensure accuracy and performance; taking advantage of CUDA library functions (which have been written with these caveats in mind), careful comparison of results, and knowledge of the capabilities of the specific GPU being used.

2.3 Scientific applications of current GPU technology

GPUs have found use in a wide variety of fields. Cardenas-Montes et al. (2014) Describe their efforts at performing an accurate calculation of cosmic shear using GPUs. Cosmic shear describes the distortion of the observed shapes of distant galaxies as their light passes through gravitational potential, a phenomenon known as gravitational lensing. According to Cardenas-Montes et al. (2014), measuring cosmic shear allows the mass distribution causing the distortion to be derived. This ultimately allows the measurement of the accelerated expansion of the universe.

This calculation is highly computationally intensive, fitting with $O(N^2)$, and it is suggested that all previous attempts used a simplified approach to allow the calculation to be completed in a reasonable timeframe, at the cost of precision.

The code is written for an Nvidia Tesla c1070 card, which is a Fermi architecture card. When compared to the same calculation performed on the CPU, the new GPU code represents a 68-fold speed increase. While this shows that GPUs are very powerful if you have the right problem, it also highlights the importance of well considered memory management when constructing GPU programs. The authors demonstrated that the best performance could only be achieved after tuning copy patterns, chunk size, and cache utilization.

Anthopoulos et al. (2013) offer an improved GPU acellerated cell-list approach. Cell-lists are important in the field of molecular dynamics as they are used to show atomic interactions within a given radius. Again, not only is this problem highly computationally expensive, but it is not trivial to parallelise.

The authors benchmarked their improved algorithm extensively both for speed, an accuracy compared to the same algorthm when run on a CPU. Total runtime is measured as well as each discrete part of the process, this allows more detailed anaysis of where the best performace gains are, and which areas perhaps require further optimisation. To test accuracy it was deemed an unfair test to directly compare CPU and GPU results due to the differences in rounding methods on both as mentioned earlier. Instead, double precision results from the CPU code were cast to single then back to double precision to remove the need to presume rounding error in non-GPU code. Accuracy in this case is measured as deviation from the CPU results, and is shown to be accurate to four significant figures.

2.4 HPC Systems which use GPUs

A look at the top500 supercomputer list shows suggests that GPUs are now widely used in high performance computing. Two of the machines in the top 10 use GPU accelerators, Titan and Emerald, both use Nvidia K20 devices. Titan currently holds the number 2 spot. In addition, GPU clusters are seeing increasing use in academic institutions. Emerald, currently one of the largest GPU clusters in Europe (e-infrastructure South, 2013), is shared between STFC and a number of universities.

2.4.1 TITAN - Oak Ridge National Laboratory

Titan has a peak performance of 17.59 PetaFlops, and was once Number one in the Top500 supercomputer sites (Top500, 2013b), as of November 2013 it is still number 2. It has 18,658 nodes, standard ones ”each with a 16 core AMD opteron 6274 processor and an Nvidia tesla K20 GPU” (ORNL, 2011b) as well as over 700 Terra-bytes of memory. Titan demonstrates well the space and power efficiency of GPUs, as despite occupying the same space as its predecessor, Jaguar, and only using marginally more power, Titan outperforms Jaguar by a factor of 10 (1.75 PFlops for Jaguar as compared to 17.59 for Titan) owing to the fact that it has a much higher ratio of GPUs to CPUs.

Researchers working with ORNL (2011a) put extensive consideration, in the years before Titan was completed, into appropriate software which would be able to take full advantage of Titans then unique configuration. A selection of 6 codes was made, all from different scientific disciplines. These codes were then adapted to allow them to use the system to its full capability. These efforts highlight the importance of an awareness of the capabilities of specific hardware within a GPU cluster. Poor planning and lack of consideration of memory capacity and throughput, particularly in a GPU, can have a drastic effect on the performance of software.

Some of the problems Titan is used to solve include: Nanoscale materials analysis, to aid development of new magnets to improve designs of motors and generators; detailed modelling of the combustion of fossil fuels with the aim of improving the efficiency of internal combustion engines, and to reduce their impact on the environment; and detailed atmospheric modelling, to further climate change research.

2.4.2 Emerald

Emerald has a capacity of 114 TerraFlops and includes 372 Nvidia Tesla processors. Each node has 2 6-core Intel Xeon X5650 processors and either 3 or 8 Tesla M2090s (e-infrastructure South, 2013). The system is used by the STFC and universities of e-Infrastructure South consortium (Oxford University, 2013). Emerald serves as a good indicator that there is demand for GPU accelerated clusters within Higher Education.

The literature shows that there is high demand and utilisation of GPUs in a wide range of scientific disciplines. Some valuable information regarding designing GPU clusters, developing GPU codes, testing and benchmarking is available and will be put to use in guiding our own efforts.

Chapter 3

Vega GPU Cluster

Investigation into existing GPU systems currently being used to accelerate scientific computation has demonstrated a significant speedup of simulations, modelling and visualisation. In order to evaluate the benefit such a system could have for research at the university of Huddersfield it was necessary to deploy a small GPU cluster. This chapter details the assembly, installation and testing of this system. The cluster is integrated into the University campus network and is accessible internally by students and staff. The Microsoft Windows HPC server software was used to deploy the GPU cluster.

3.1 The Microsoft Windows HPC Platform

Microsoft provides tools to allow deploying and managing a cluster using Microsoft Windows Server. Microsoft (2014c) detail the deployment of a complete HPC system using Microsoft tools.

GPU graphics and compute drivers are widely available for Windows systems (Nvidia, 2013c). Windows HPC cluster manager provides a mechanism to include drivers with operating system images, to simplify the deployment process. In addition, Microsoft also provides a number of tools to aid in testing and benchmarking a windows HPC cluster (Micorsoft, 2014). The Message Passing Interface (MPI) is a standard for cross process communication designed for parallel compute systems. The MPI system included with Windows HPC deployments, MSMPI, is Microsofts implimentation of the

MPI-2 standard (Microsoft, 2014a). It is compatible with MPICH2, which makes porting MPI code from other platforms, such as Linux, relatively simple.

3.2 Cluster Layout

The following hardware was available for the deployment:

Item	Description
Netgear ProSafe GSM7224	24 port Gigabit network switch
Dell Poweredge R410	1x Quad core octo-thread Xeon E 5630 CPU running at 2.53 GHz, 32 GB RAM.
Dell Poweredge C6100	4 node chassis currently with 2 nodes installed. each with the following spec: 2x Quad core octo-thread Xeon E5620 CPUs, running at 2.4GHz, 24 GB RAM, and approx 400GB local storage.
Dell Poweredge C410x PCIe Expansion Chassis + 1 interface card	This chassis can host up to 16 GPUs and is connected to a compute node using a PCIe interface card (Dell, 2014)
2x Nvidia Tesla M1020 GPUs	These are FERMI architecture cards with 448 processor cores running at 1.15 GHz, with 3GB of ECC GDDR5 RAM with a clock speed of 1.546 (Nvidia, 2013e)

TABLE 3.1: Hardware available for VEGA

As the Poweredge C6100 is a four node chassis, it was selected to act as compute node. There is only a single PCIe interface card available so initially there is only a single compute node, with the option of adding more in the future. The Poweredge R410 acts as head node.

In line with Microsoft suggestions, the head node has 2 NICs installed (Microsoft, 2014c). One is connected to the University network, the other is connected to a Gigabit switch, which is used to communicate internally with the compute nodes. This topology is illustrated in Table. 3.1.

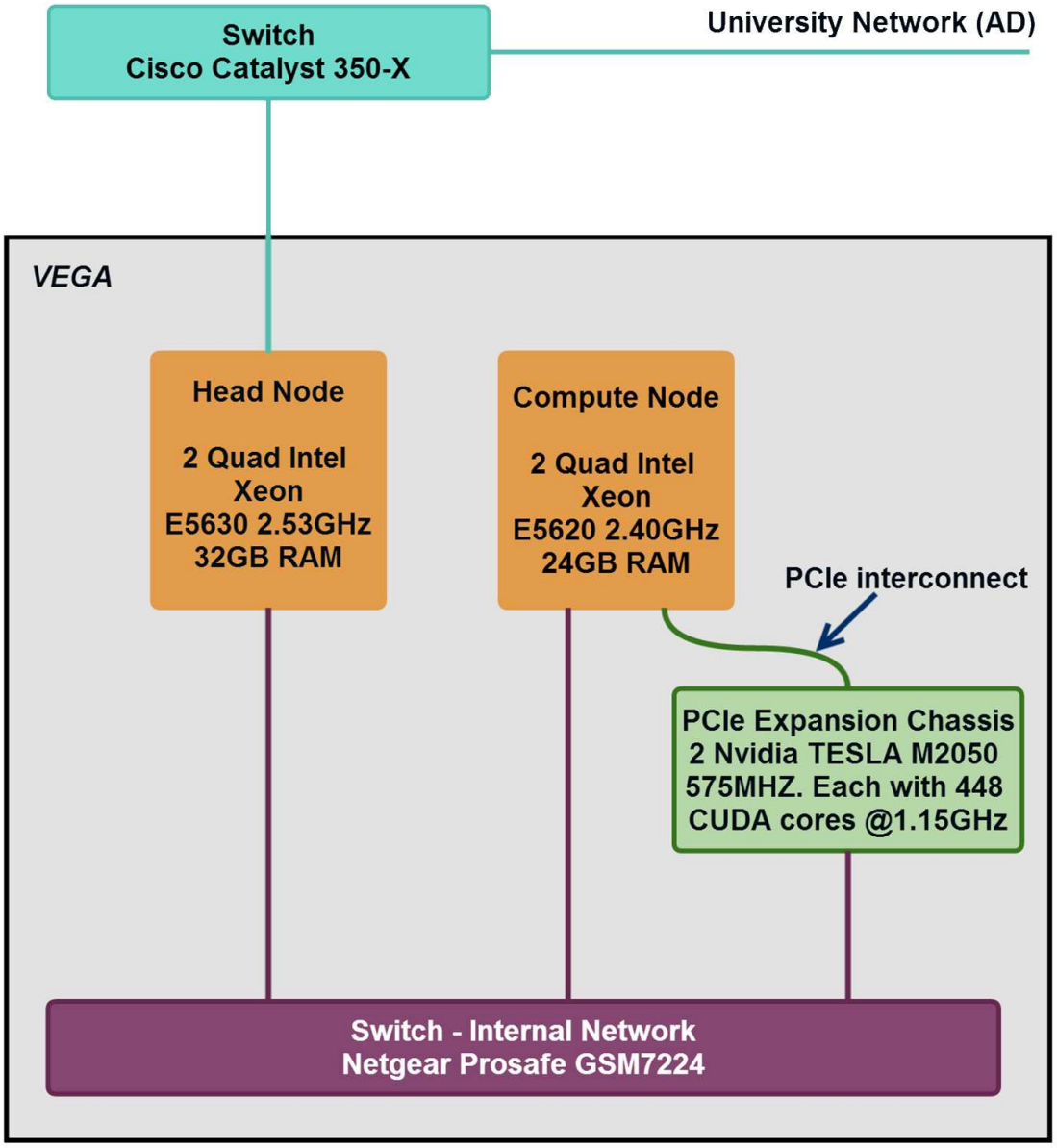


FIGURE 3.1: Current VEGA hardware layout and network topology.

3.3 Deployment

Initially, all hardware was tested to ensure it was viable. According to the deployment procedure outlined by Microsoft (2014c), was to install Windows Server 2008r2, which is a standard procedure. Once installed, the Head node is connected to the university network and then added to Active Directory by specifying the domain name. At this point the following steps are performed to deploy the GPU cluster:

- Microsoft HPC Pack is installed on the head node.

- The head node is attached to the private Gigabit switch.
- Compute nodes are deployed by first connecting them to the Private network then booting to PXE. The Windows HPC software will then load an operating system and required middleware over the network. This differs from the procedure offered by Microsoft (2014c), which suggests attaching nodes which have already been configured with windows server and HPC pack. As the nodes did not already have Windows installed, it was deemed much simpler to have the head node push out preconfigured system images to the compute node.
- Finally, drivers for the TESLA cards are installed on the compute node directly (Nvidia, 2013c).

A summary of the deployment steps required to deploy the GPU cluster, VEGA, can be seen in Fig. 3.2.

3.4 Testing

The Windows HPC cluster manager includes a number of diagnostic tests to verify cluster deployment. All relevant tests when run on VEGA passed without any problems. The standard measure of performance of HPC systems, and the test used to measure systems for inclusion in the top500, is the LINPACK benchmark. The benchmark consists of ”a dense system of linear equations”(Jack Dongarra and Stewart, 2013), and the ability of a system to process these is used as a measure of peak performance. The parallel implementation of LINPACK is known as High Performance LINPACK (HPL). Microsoft provides a self-tuning HPL utility, Lizard, to benchmark Windows clusters, the results and final tuned parameters of this can be seen in Fig. 3.3. This however does not include the extra performance provided by the GPUs, figures from Nvidia state that peak theoretical performance of each M2050 is 1.03 TFLOPS (Nvidia, 2013e).

More in depth tests were carried out to assess MPI and CUDA Functionality. A simple program, the source for which can be seen in Listing 3.4, was written to test MPI functionality. As seen in Fig. 3.5 MPI is fully functional on the cluster. After installing the CUDA toolkit (Nvidia, 2013b) a number of CUDA accelerated test programs are available. Fig. 3.6 shows a number of these running successfully.

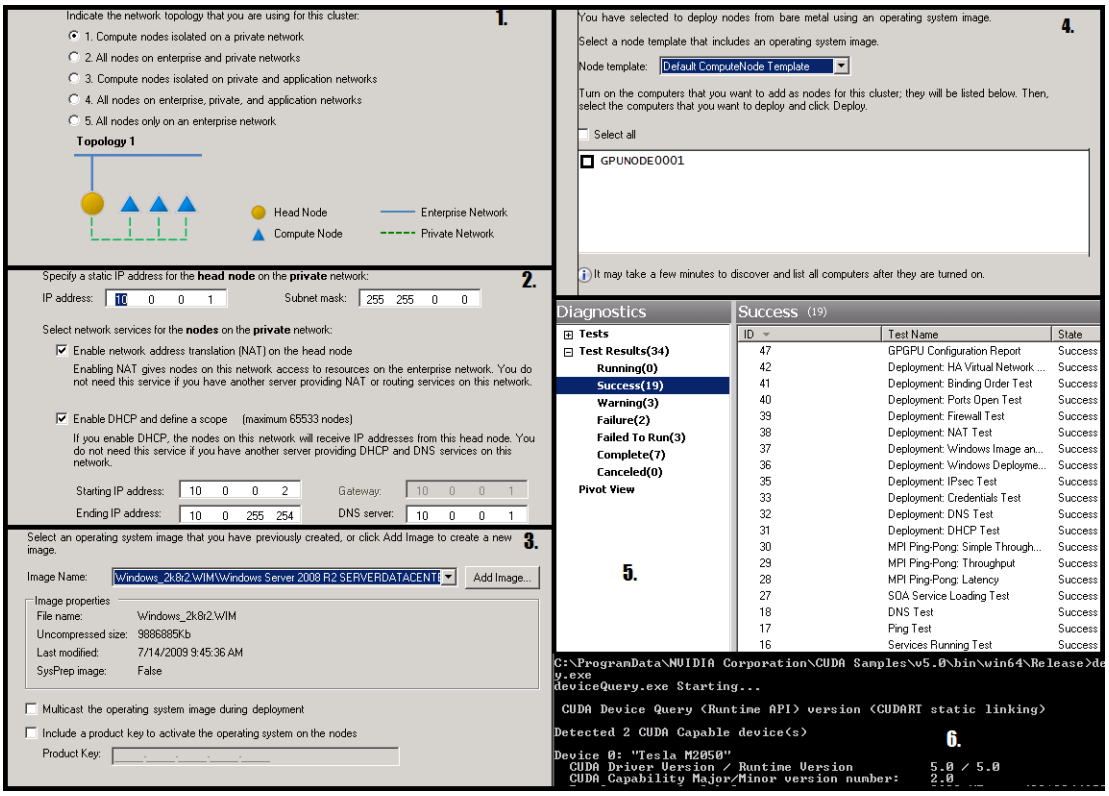
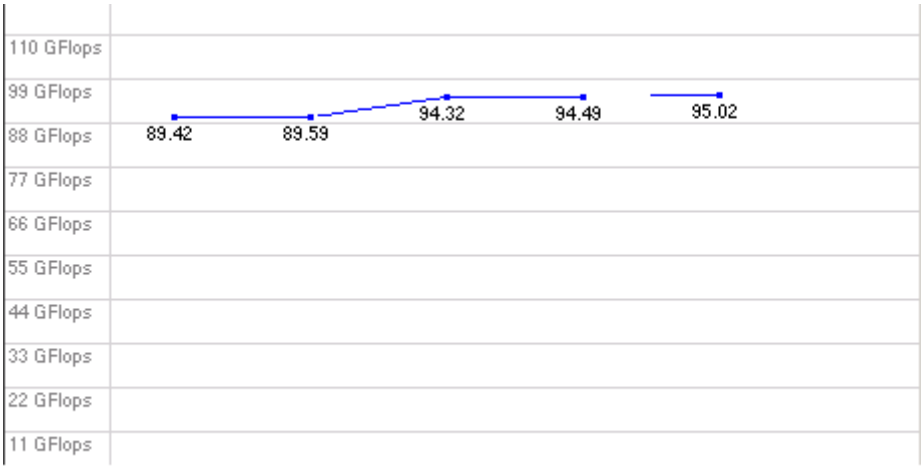


FIGURE 3.2: Key steps when deploying VEGA:
1. Selecting Network Topology
2. Configuring NAT and DHCP to allow compute nodes to access outside network
3. Creating a system image for compute nodes
4. Finding bare metal machines to deploy as compute nodes
5. Checking relevant diagnostic tests pass
6. Checking CUDA functionality

Windows was chosen as the operating system to allow familiarity with users, and easy integration with the existing active directory network. In addition, running Windows allows the option of adding the cluster to the backburner render system in use at the university.

However, CUDA is also available for Linux. Were it required, necessary drivers are available to allow the same functionality, using a cluster middleware such as OSCAR or Warewulf (ORNL, 2005), (LBL, 2014).

Although cluster integrity was confirmed using test codes and benchmarking utilities, it was necessary to examine programming models and environments to allow users to develop new parallel code and parallelise existing serial codes.



Parameter	N	NB	PMAP	P	Q	Threshold
Final Value	44992	304	0	2	12	16
Parameter	PFACT	NBMIN	NDIV	RFACT	BCast	Depth
Final Value	0	4	8	2	1	1
Parameter	SWAP	L1	U	Equilibration	Alignment	
Final Value	2	0	0	0	16	
CPU only performance = 95.02 GIGAFLOPS						

FIGURE 3.3: Tuning HPL parameters using Lizard

```
#include "stdafx.h"
#include <iostream>
#include "mpi.h"
#include<Winsock2.h>

#pragma comment(lib, "Ws2_32.lib")

using namespace std;

int main(int argc, char* argv[])
{
 int  nTasks, rank;

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&nTasks);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);

 char hname[128] = "";

 WSADATA wsaData;

 WSAStartup(MAKEWORD(2, 2), &wsaData);

 gethostname(hname, sizeof(hname));

 WSACleanup();

 printf ("Number of threads = %d, My rank = %d\n, My Host = %s", nTasks, rank, hname)↩
    ;

 MPI_Finalize();
 return 0;
}
```

LISTING 3.4: Simple MPI test program, based loosely on the example from Mattotorang (2009)

```
Number of threads = 24, My rank = 10
, My Host = GPUNODE0001
Number of threads = 24, My rank = 13
, My Host = GPUNODE0001
Number of threads = 24, My rank = 4
, My Host = GPUNODE0001
Number of threads = 24, My rank = 3
, My Host = GPUNODE0001
Number of threads = 24, My rank = 16
, My Host = VEGA
Number of threads = 24, My rank = 0
, My Host = GPUNODE0001
Number of threads = 24, My rank = 11
, My Host = GPUNODE0001
Number of threads = 24, My rank = 23
, My Host = VEGA
Number of threads = 24, My rank = 2
, My Host = GPUNODE0001
Number of threads = 24, My rank = 7
, My Host = GPUNODE0001
Number of threads = 24, My rank = 14
, My Host = GPUNODE0001
Number of threads = 24, My rank = 6
, My Host = GPUNODE0001
Number of threads = 24, My rank = 15
, My Host = GPUNODE0001
Number of threads = 24, My rank = 5
, My Host = GPUNODE0001
Number of threads = 24, My rank = 9
, My Host = GPUNODE0001
Number of threads = 24, My rank = 1
, My Host = GPUNODE0001
Number of threads = 24, My rank = 21
, My Host = VEGA
Number of threads = 24, My rank = 12
, My Host = GPUNODE0001
Number of threads = 24, My rank = 8
, My Host = GPUNODE0001
Number of threads = 24, My rank = 22
, My Host = VEGA
Number of threads = 24, My rank = 20
, My Host = VEGA
Number of threads = 24, My rank = 19
, My Host = VEGA
Number of threads = 24, My rank = 18
, My Host = VEGA
Number of threads = 24, My rank = 17
, My Host = VEGA
```

FIGURE 3.5: Output from MPI test program when run on all cores

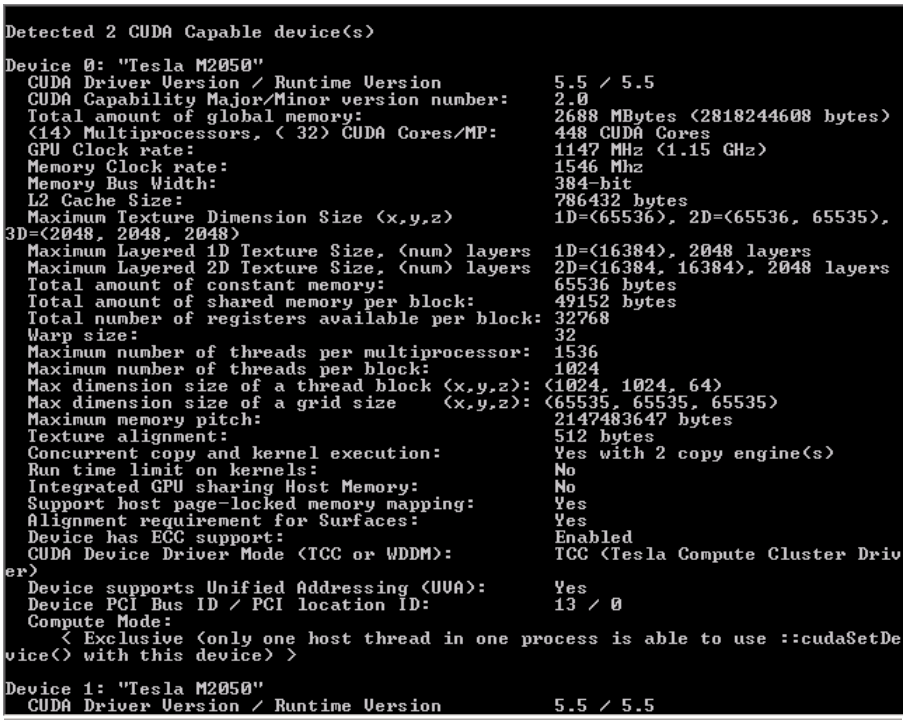


FIGURE 3.6: GPU test programs included with CUDA toolkit (Nvidia, 2013a), from top to bottom: Device Query, Testing Aligned Types, Matrix Multiplication

Chapter 4

Platforms and Programming Environments for Scientific Data Visualisation and Processing

There is a need to develop parallel programming environments to aid software development for parallel computer architectures. However, other than those already outlined in chapter 2, there are still few mature parallel programming environments. Perhaps more important is the absence of tools and IDEs that allow for simple debugging and testing of parallel software with the ease and reliability of those established for traditional serial programs. This is particularly true for software designed to make use of accelerators. While debugging and development tools do exist for accelerators such as GPUs, they are platform specific (for example Nvidia Nsight (Nvidia, 2014)) meaning that developing parallel software for heterogeneous systems remains a comparatively convoluted process.

This is an active area of research and calls for publications in this area regularly appear with significant funding interest from research councils, as in 2020 (2013). This chapter covers the environments selected for use on VEGA, based on the findings in Chapter 2, with particular focus on systems that support visualisation of scientific data, and multi-node parallelisation with GPU acceleration.

4.1 Visualisation

Visualisation of scientific data is not a novel concept. Matlab is used extensively to plot 2D and 3D images. Similarly, CFD and MD packages such as ANSYS (INC, 2014) and DL_POLY (STFC, 2014) provide inbuilt software for visualisation of processed data. When evaluating the available visualisation software for deployment on VEGA, VisIt, a parallel visualisation tool, was chosen for its focus on parallel processing using MPI, as well as GPU accelerated rendering. In addition, it boasts compatibility with a vast range of CAD, CFD, MD, graphing and mathematics software (LLNL, 2013). VisIt is highly flaxible in the data it can display, thanks to software functions which allow data to be structured appropriately programatically, in C or FORTRAN.

4.2 Multi-Node parallelism

An integral part of any HPC system is a mechanism to allow programs to utilise multiple CPU cores which do not necessarily occupy the same host node. MPI has proven itself over the past couple of decades in this regard and has become a standard used in the vast majority of HPC deployments. MPI supports a wide selection of programming languages such as C++ and and Java (Forum', 2014). Some key concepts to consider when using MPI; Communicators, Point to Point operations, and Broadcast and Collection. Communicators are used to group MPI processes related to the same task. Most MPI operations must specify a communicator, this ensures that the operation only effects those processes which are in the same communicator (for example MPI_COMM_WORLD). Point to Point operations, such as MPI_Send and MPI_Recv, are used to send data from one rank to another in a one to one relationship. Broadcast and Collection operations such as MPI_Bcast and MPI_Scatter are one to many or many to one operations, and are useful for dividing data among processes, and then collecting after processing.

There are a number of distributions which comply with the MPI standard. MPICH is the (MPICH, 2014) is perhaps one of the most widely used. Most HPC clusters will offer a choice of MPI version. Some other options are Open MPI and MSMPI. MSMPI is an MPI version distributed by Microsoft. It is fully MPICH compatible and is tightly

integrated with the Windows HPC job scheduler, the scheduler used in VEGA, allowing simple submission of MPI tasks to the cluster. This close integration makes MSMPI particularly suited for Multi-node parallel software when Windows HPC is used (Microsoft, 2014b).

Options other than MPI include OpenMP which allows ”multi platform shared memory parallel programming” (Board, 2013). This offers some flexibility when developing parallel software. The programming experience is very similar to that of using MPI, with the added benefit of shared memory. Parallel Virtual Machine (PVM) is an approach which aggregates discrete systems which do not necessarily need to have the same hardware or operating system. PVM allows a heterogenous cluster of machines to appear as a single virtual machine (ORNL, 2009).

4.3 GPU Programming Framework

The two main GPU programming frameworks remain Nvidia CUDA and OpenCL, although more are gaining traction, such as C++ AMP, in development at Microsoft. A number of studies, such as those carried out by (Karimi et al., 2010), and (Fang et al., 2011) have shown that the in many cases, CUDA is able to outperform OpenCL when both are given the same task. This, coupled with the Nvidia hardware available in VEGA and existing familiarity with Nvidia tools, are all factors contributing to the choice to use CUDA for GPU processing. Some important things to bear in mind when developing CUDA software is effective memory management, as well as device management. When using GPUs to process data, the most efficient use of the device is if all cores are occupied. This requires that the host program can stream enough data. This does not necessarily mean filling GPU memory, as depending on the task being executed on each GPU core, a smaller data stream may be enough make full use of the hardware. Minimising copy operations will also help towards efficient use of the hardware. For example if a process loop (eg a matrix multiplication) only uses a half of the available CUDA cores, then organising into batches, so that two data ranges are processed simultaneously, will reduce time spent copying to and from the device. Also important to consider is the fact that devices can only be controlled by a single host thread at a time,

so programs which use multiple GPUs must take care to check which GPU is being requested to avoid crashing.

The challenge of parallel programming is a problem that is yet to be addressed fully. The approach taken as part of this research is outlined in the following section in the form of three case studies. The first two of these case studies were published in the proceedings of the Science and Information conference 2014 (Newall et al., 2014). In addition, the final two studies were published as a chapter in in the Springer series book, Studies in Computational intelligence. These publications can be seen in appendices D and E respectively.

Chapter 5

Case Study 1: Visualisation of large datasets

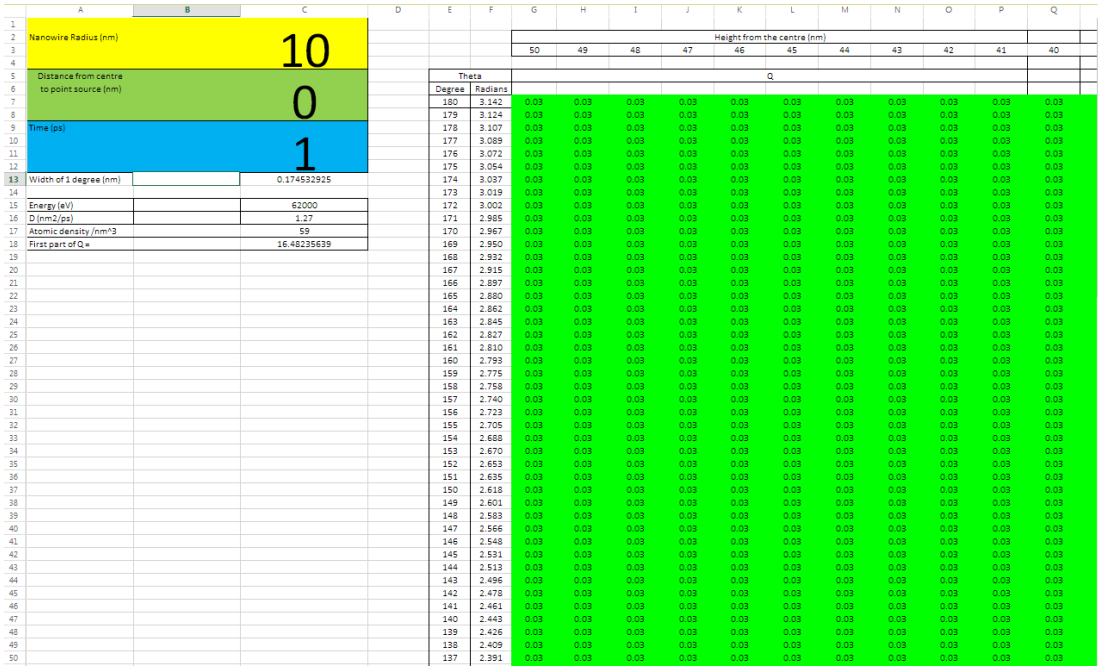
As models and datasets become larger and more complex, new systems are required to enable useful analysis. To support the need of researchers at the University, it was deemed necessary to deploy a visualisation tool which could work with large or complex datasets.

To this end, and existing software, VisIt, was obtained and installed on VEGA. VisIt is a software developed by the Lawrence Livermore National Laboratory (LLNL), described as ”Parralel, Interactive, Visualisation” (LLNL, 2013). It is specifically designed to utilise HPC resources to enable interactive simulation of terrascale datasets, and supports GPU accelerated rendering.

5.1 Electron Microscopy Data

The Electron Microscope Materials Analysis group (EMMA) at the university, is tasked with ”the interaction of energetic particles with matter” (EMMA, 2014). The research carried out in this area generates large volumes of data, both through experimentation and simulation. One simulation in particular was of interest as the visualisation methods being employed at the time, a large colour coded spreadsheet, were not deemed adequate. The data in question is an array of cells, the value of each representing the

energy of a point on a cylinder, and the values change over time. The spreadsheet that was in use was unwieldy and cannot all be seen at once as seen in Fig 5.1. This made proper analysis difficult, ideally the data would be shown as points on an actual cylinder as a 3D contour. For these reasons the data was deemed a good candidate for case study.



Nanowire Radius (nm)		10
Distance from centre to point source (nm)		0
Time (ps)		1
Width of 1 degree (nm)		0.174532925
Energy (eV)		62000
D (nm2/ps)		1.27
Atomic density /nm^3		59
First part of Q =		16.48235639

		Height from the centre (nm)										
		50	49	48	47	46	45	44	43	42	41	40
Theta		Q										
Degree	Radians											
180	3.142	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
179	3.124	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
178	3.107	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
177	3.089	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
176	3.072	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
175	3.054	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
174	3.037	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
173	3.019	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
172	3.002	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
171	2.985	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
170	2.967	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
169	2.950	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
168	2.932	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
167	2.915	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
166	2.897	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
165	2.880	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
164	2.862	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
163	2.845	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
162	2.827	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
161	2.810	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
160	2.793	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
159	2.775	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
158	2.758	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
157	2.740	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
156	2.723	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
155	2.705	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
154	2.688	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
153	2.670	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
152	2.653	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
151	2.635	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
150	2.618	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
149	2.601	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
148	2.583	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
147	2.566	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
146	2.548	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
145	2.531	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
144	2.513	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
143	2.496	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
142	2.478	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
141	2.461	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
140	2.443	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
139	2.426	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
138	2.409	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
137	2.391	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03

FIGURE 5.1: Visualisation method before VisIt

5.2 VisIt

Visit displays data that is defined on 2D or 3D meshes, this presents a challenge; The electron microscopy data is simply a comma delimited list of values. The solution was to build a software tool to format this data, and save it to an appropriate file format. The SILO library was developed by LLNL fo use with VisIt in situations such as this, and allows production of data meshes programmatically (LLNL, 2010). Following the recommendations in the SILO users guide, the software was built uses standard SILO functions to save the data in 3 different formats:

- A 2D mesh where the data is represented by colour (similar to the function of the original spreadsheet)
- A 3D Mesh representing a cylinder (also colour coded)

- and finally a 3D mesh which deforms based on the value of each point over time (contour).

Fig. 5.2 shows a summary of important parts of the program, full listings can be seen in appendix 1.

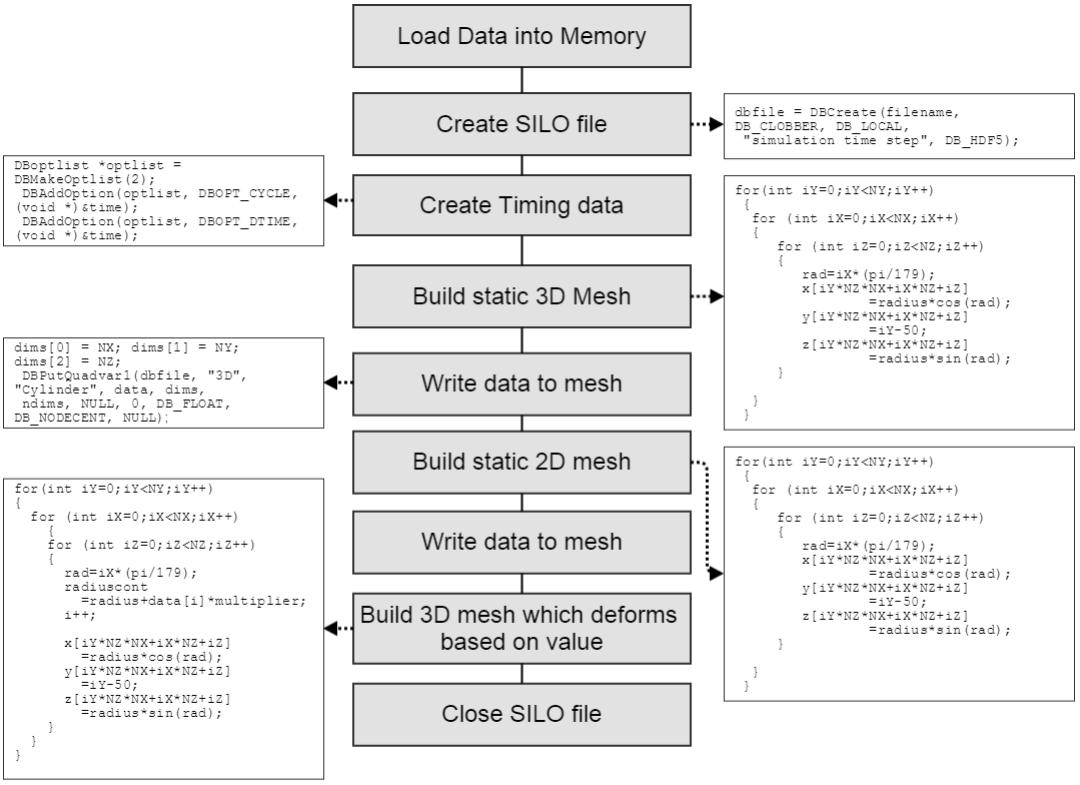


FIGURE 5.2: Program flow for the SILOWRITE code

5.3 Evaluation of Results

Fig. 5.3 Shows a few images of each visualisation method at different time steps. The final 3D contour model fulfils the original requirements of the project and allows real time interactive analysis of the simulation data. The main advantage of this model is that all the data for each time step can be seen at once. The animation can be easily saved to video.

Although VisIt is designed for visualisation scientific data, unless the data is already formatted to fit one of the file types that VisIt supports, it is necessary to pre-process the data from often a large repository to structured data suitable for visualisation. This

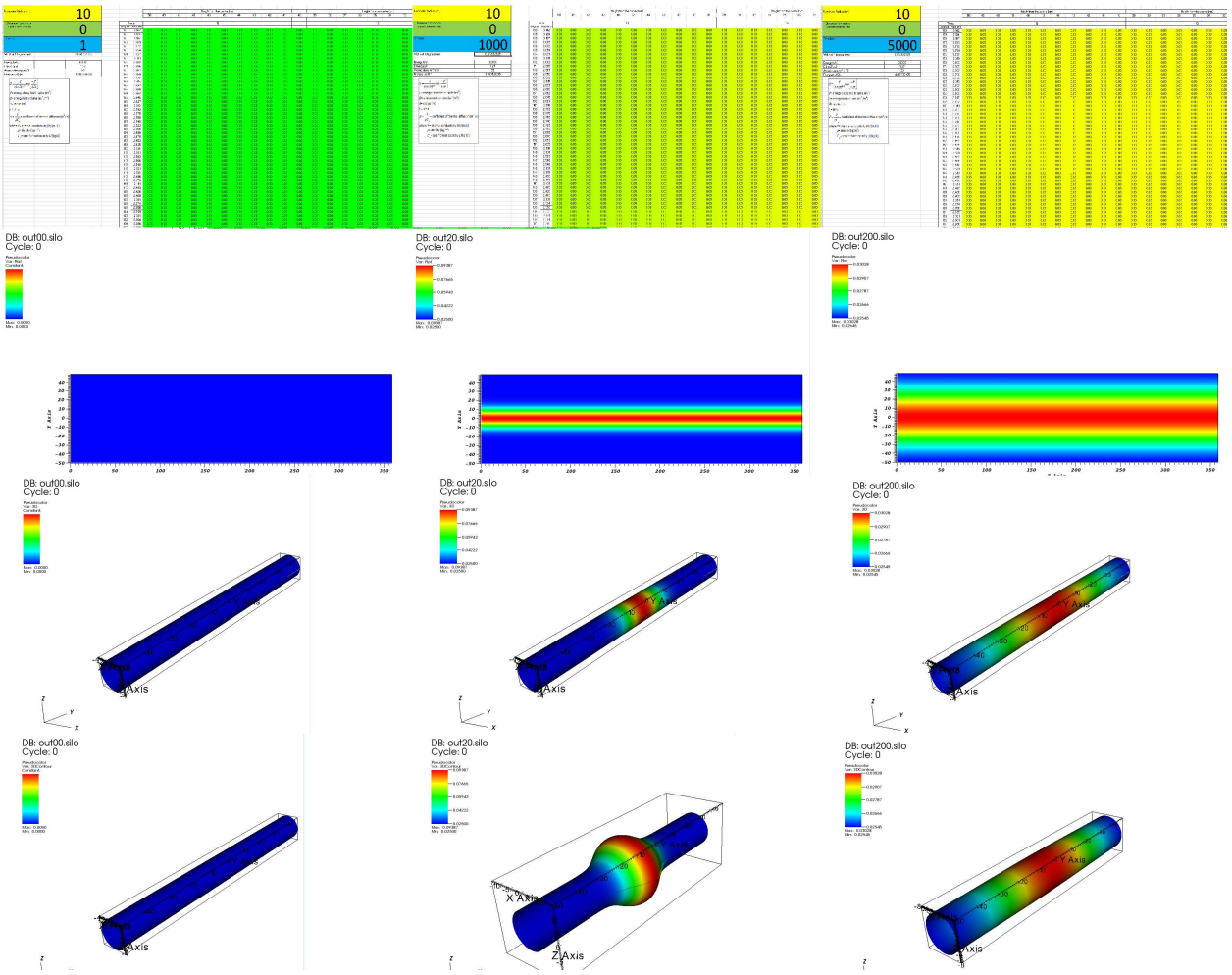


FIGURE 5.3: Comparison of Visualisation methods, from top to bottom: Original spreadsheet, 2D Mesh in VisIt, 3D mesh in VisIt, 3D contour mesh in VisIt

illustrates that visualisation of raw scientific data is not a straight forward process, and that the main effort is still the responsibility of a software engineer.

Chapter 6

Case Study 2: Accelerated Processing of Radio Telescope Data Using CUDA

The Serch for Extra Terrestrial Intelligence (SETI) is a group whos mission is to serch for non earth signals from space, in the hopes of locating intelligence. One of the main aspects of this endevour, is the analysis of vast amounts of radio telescope data. There have been numerous projects which have intended to accelerate this process such as processing the data into images to speed up human analysis of the data, even allowing enthusisasts at home to look through the data. But it is a time consuming task, even with these methods the data is being collected faster than it can be analysed so more efficient methods are required. To this end, the SonicSETI project has started sonifying the data. Sonification is a process whereby a stream of data is converted into sound. Studies have shown that sonified data can be reviewed by humans at a much greater rate than visual anaylsis, and patterns are much more descernable from background noise (Kramer, 1993). Perhaps the most prominent example of sonification is that of the Geiger counter, which uses audible clicks which increase in frequency as radiation levels increase. While the sonified data may be much easier to analyse, processing the data is a time consuming proesss in itself, so investigation was started into the potential benefits of GPU acceleration.

6.1 The SETIFFT software

In order to sonify the data, a piece of software was written in JAVA by Paul Lunn for the Sonic SETI project. Fig. 6.1 shows the program logic. In order to asses which parts of the software would best benefit from acceleration, different sections of the code were timed, these timings can be seen in Table 6.1. In each test, the software performs 244 FFTs using a 2GB file as input, measurements are averaged over 10 separate runs.

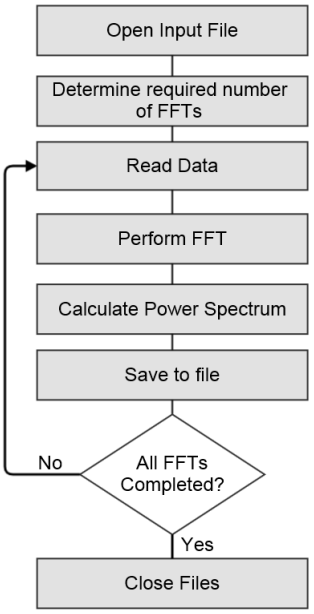


FIGURE 6.1: Program flow of the SETIFFT software

Action	Time Taken (Ms)
Read data	3713182 (1:01:53)
Perform FFT	4491666 (1:13:52)
Calculate Power Spectrum	477218 (0:07:57)
Save to File	1344062 (0:22:24)
Total run time	10026326 (2:47:06)

TABLE 6.1: Timings for original JAVA program

Timings showed that the program spent most of its time reading in data and performing the FFT. As each FFT takes over a minute, and the full datasets can require over 500 FFTs, the first step was to replace the serial FFT with a parallel, GPU accelerated one. The Nvidia CUFFT library fits this purpose, and there exist JAVA wrappers for its functions (CUFFT is C). Using this library, FFT precessing time was successfully reduced by over 30 times, as seen in Table 6.2

Action	Time Taken (Ms)
Read data	3801657 (01:03:22)
Perform FFT	916464 (00:15:16)
Calculate Power Spectrum	483007 (00:08:03)
Save to File	1404548 (0:23:25)
Total Run Time	5711437 (1:35:11)

TABLE 6.2: Timings for Java program with wrapped CUDA functions

In order to improve read and write time, and to make better use of CUDA functions, the program was rewitten in C++, first with a serial FFT as seen in table 6.3, then with CUFFT as seen in table 6.4.

Action	Time Taken (Ms)
Read data	29686 (0:00:27)
Perform FFT	1541246 (0:25:41)
Calculate Power Spectrum	56436 (0:00:56)
Save to File	29686 (0:00:30)
Total Run Time	1810008 (0:30:10)

TABLE 6.3: Timings for C++ with serial FFT

Action	Time Taken (Ms)
Read data	17344
Perform FFT	90720
Calculate Power Spectrum	70781
Save to File	17344
Total run time	277674

TABLE 6.4: Timings for C++ with CUFFT

6.1.1 Optimising CUDA performance

The original software, as well as initial CUDA versions, read enough data to perform a single FFT, process it, and write to file. However this is an inefficient use of GPU memory. Each element in the complex array used to process the data is 16 bytes in size, and FFT size is 8388608. Accounting for the output array, the 3GB on the Tesla cards can hold enough data to process 10 FFTs for each copy operation. Table 6.5 shows timings for memory copy and FFT execution time.

Batch size	Copy operations	Time (Ms)
Single FFT	488	90720 (0:01:31)
5 FFTs	98	36225 (0:00:36)
10 FFTs	50	27313 (0:00:27)

TABLE 6.5: Timings for C++ with CUFFT and MPI, running on multiple GPUs

Making these changes decreased FFT processing time for a 2GB file by 63.4 seconds, an improvement of over 69%.

6.1.2 Using MPI to increase performance

To utilise the second GPU in the system, and to allow the program to scale to even larger systems, the program was modified to implement MPI. Fig. 6.2 shows the program structure of the initial version. Files are read and written by the master process rank, and data is sent to and from the worker ranks using MPI.

This initial MPI version was successful in reducing actual processing time, but the large MPI data transfers added significant overhead, in excess of any performance gained in processing. The solution was to make use of the shared storage available on VEGA to remove the need for large MPI send/recieve operations. Fig. 6.3 shows the structure of the final version of the software. Rather than data being sent to each rank using MPI functions, each rank opens the relevant file directly. In this version MPI messages are only used to tell the workers which files to open and where to start and stop reading.

Table. 6.6 shows timings for the final code. Each rank processes a share of the data, and are timed simultaneously , hence two timings for each section of the program.

Action	Time Taken (Ms)(rank 0)	Time Taken (rank 1)
Read data	118043	145642
Perform FFT	12310	12570
Calculate Power Spectrum	40271	40304
Save to File	118043	145642
Total run time	249777	

TABLE 6.6: Timings for C++ with CUFFT and MPI, running on multiple GPUs

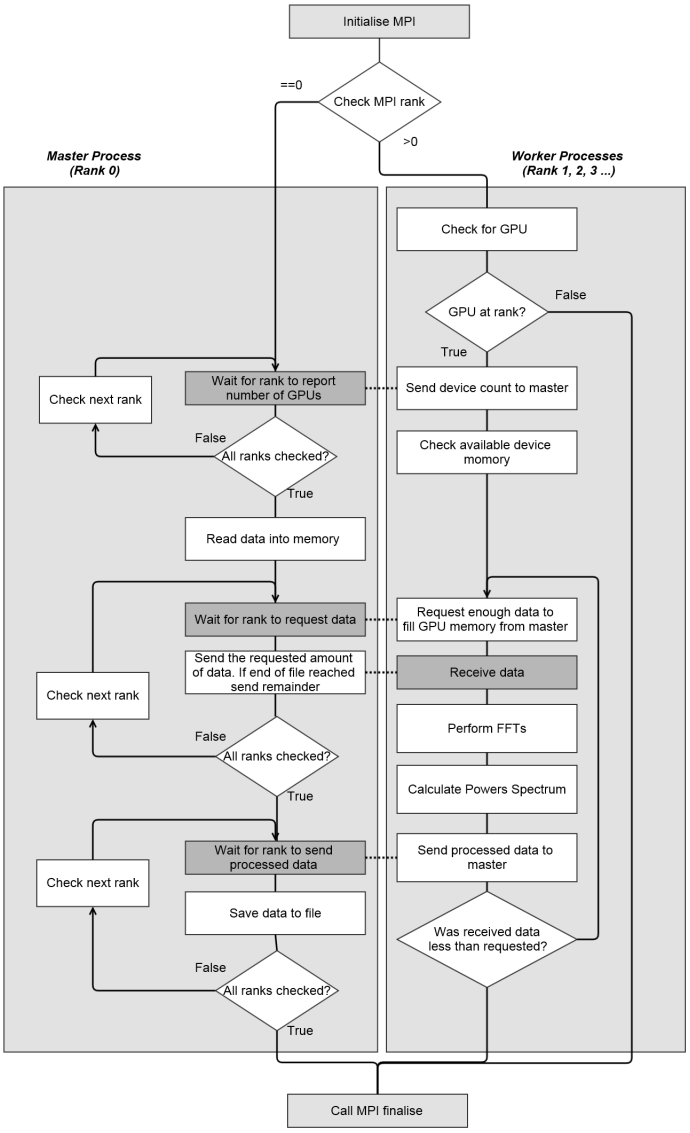


FIGURE 6.2: Program flow of the initial MPI program

6.2 Evaluation of Results

It can be seen that read/write time is increased in the MPI version compared to the single process. This is because each process is accessing the same file simultaneously. Table 6.7 shows times when the program is run with two files from the dataset. As the datasets are often larger than 8GB, they are split into 2GB files (SETI, 2013a), so this would not be outside the normal use of the software.

When each process is able to work on a separate file, read/write time is similar to the time taken when working on a single file, even though double the amount of data is being processed. During normal operation, the only time multiple process ranks would

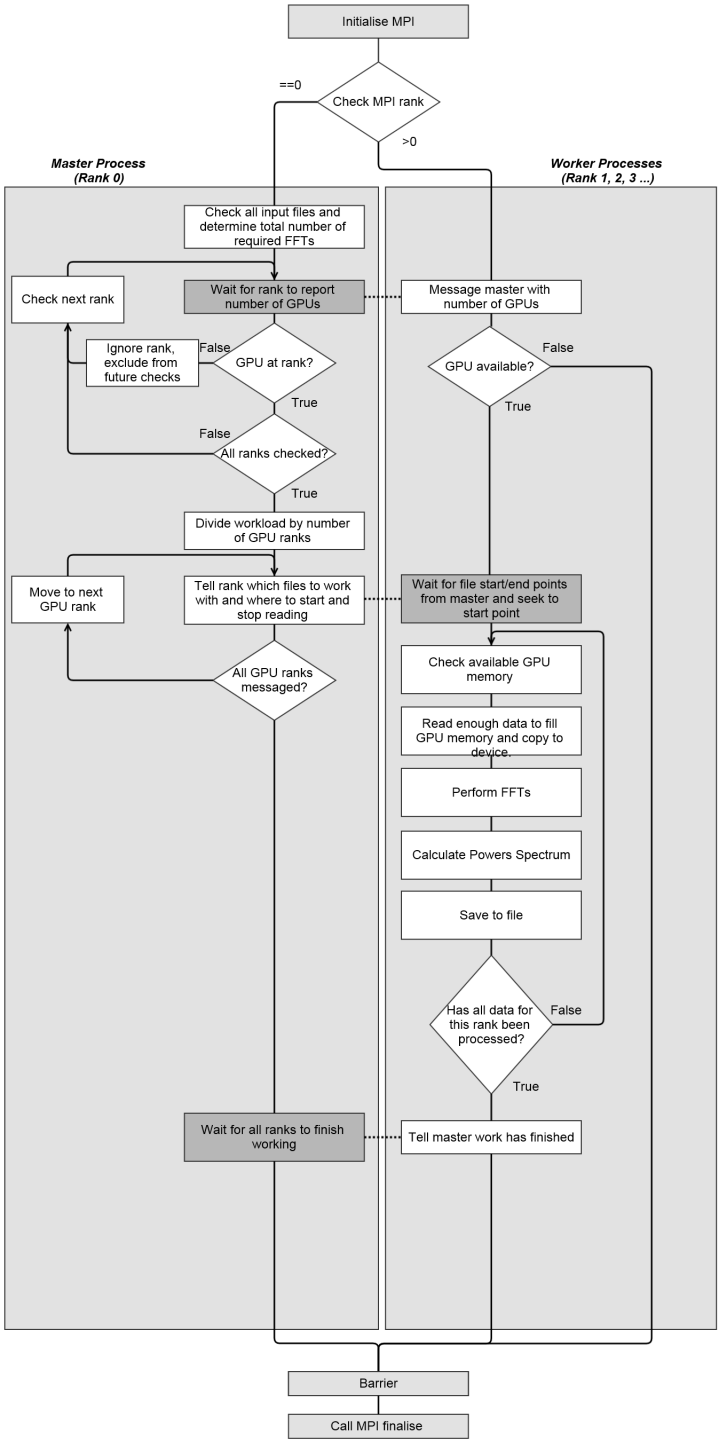


FIGURE 6.3: Program flow of the final MPI version

access the same file would be if there were an odd number of input files. Were the software to be developed further it may be worthwhile to factor for this by ensuring that the final file in an odd numbered set is only processed by one rank.

Files	Total FFTs	Read/Write time
1	244	145642
2	488	145816

TABLE 6.7: Read/write time for the MPI software when reading from different numbers of files

Figure 6.4 compares total running times of each significant version of the software, running with a full dataset 5GB in size (split into 2GB files). Through parrallelisation (using both CUDA and MPI) and optimisation it has been possible to significantly accelerate the process of sonifying the radio telescope data.

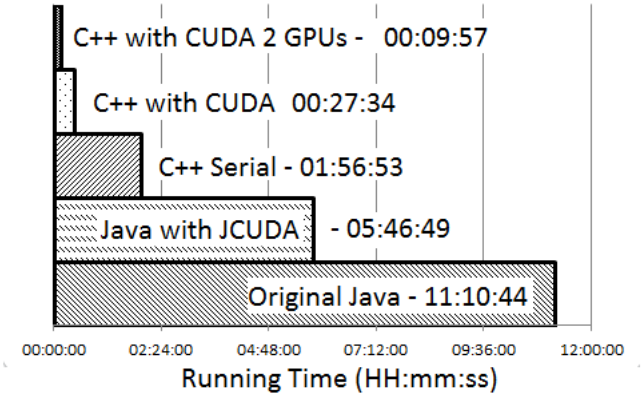


FIGURE 6.4: Total running times of each version

The primary implication of this is that the rate at which the data can be analysed is greatly increased, and the time between any potential discoveries is decreased. This case study is evidence that simply using more powerful hardware will not always bring proportional performance improvements without efficient and sustainably designed software.

Chapter 7

Case Study 3: CUDA Accelerated Analysis of Wavelength scanning Interferometry Data

Optical interferometry is a widely used surface metrology technique. Wavelength scanning interferometry developments have been made that allow the process to be immune to environmental noise using phase compensation. However this compensation as well as data analysis processes limit performance, and hamper efforts to inspect this data as the measurement takes place. The paper Muhamedsalih et al. (2012) details a method which uses CUDA to accelerate this process with a single GPU. However, while the results were promising, it wass still not possible to achieve the real time analysis that was desired.

7.1 The Interferometry Software

The original CUDA program loads a set of bitmap frames, and the noise cancellation is calibrated by loading a matrix which has been processed by MATLAB. After calibration the data is processed through an FFT algorithm, and all data is saved to disk. Fig. 7.1 shows a summary of the function of the program. As with the work carried out in the previous case study, GPU acceleration comes in the form of a parallel FFT algorithm, CUFFT, as well as a few other mathematical operations.

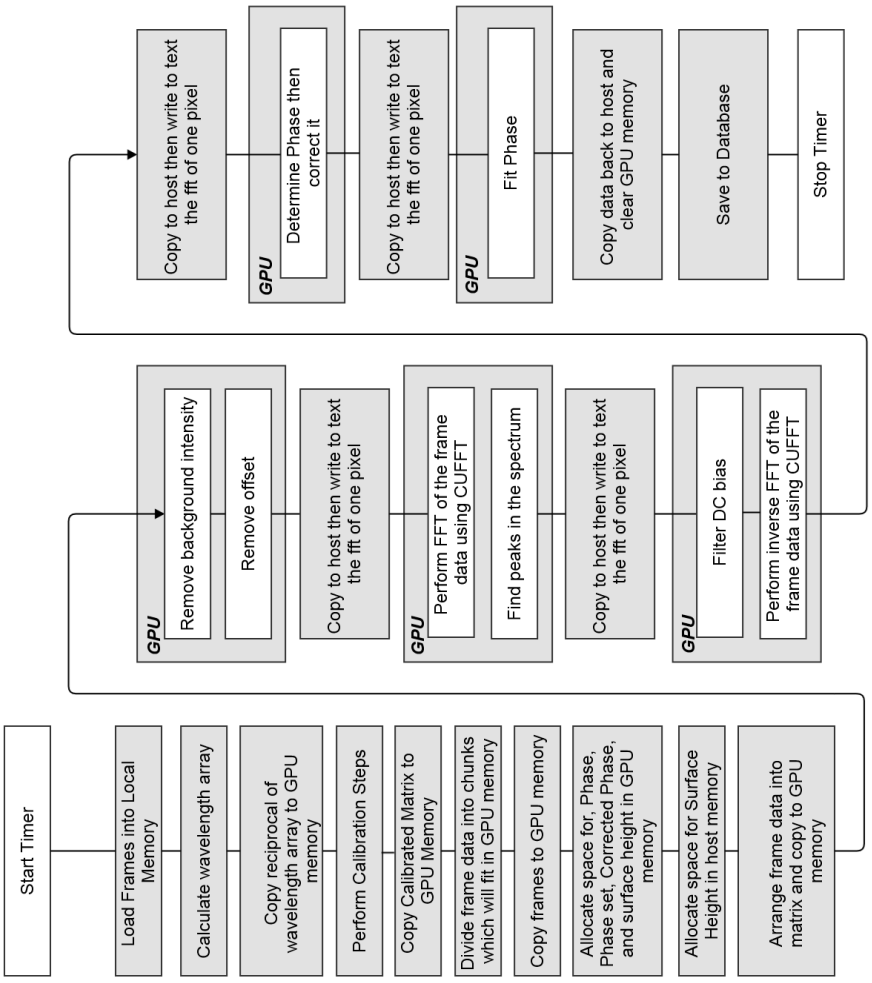


FIGURE 7.1: Program flow for the original CUDA code

7.2 Improving Performance Using MPI

Using the lessons learned in the previous study, the program was modified to allow it to use multiple GPUs. After including the necessary MPI libraries the program was modified in a way which results in a structure closely resembling the original program, but which is duplicated over multiple processes which can each claim its own GPU. The intended outcome of this modification is a doubling of the total throughput, to allow the processed data to analysed closer to real-time. The modified program structure is shown in Fig. 7.2.

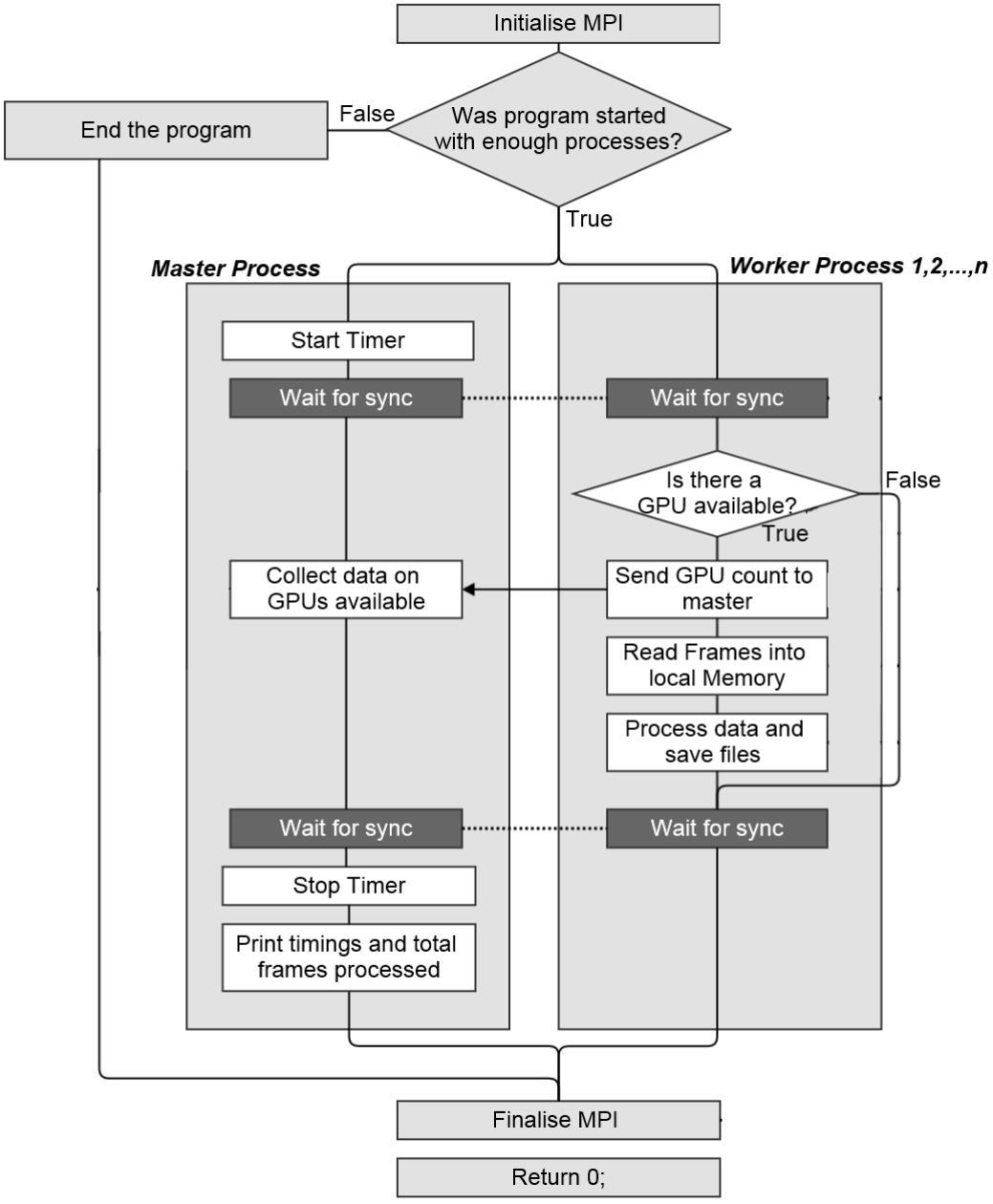


FIGURE 7.2: Program flow for the MPI version using multiple GPUs

7.3 Evaluation of Results

The graph in Fig. 7.3 shows total running times for both the original software and the MPI version. It is important to note that in this graph the MPI version is processing double the amount of data with only a small increase in running time. By dividing the running time by the number of frames processes we can calculate a representative per-frame processing time for each version, this is shown in 7.4.

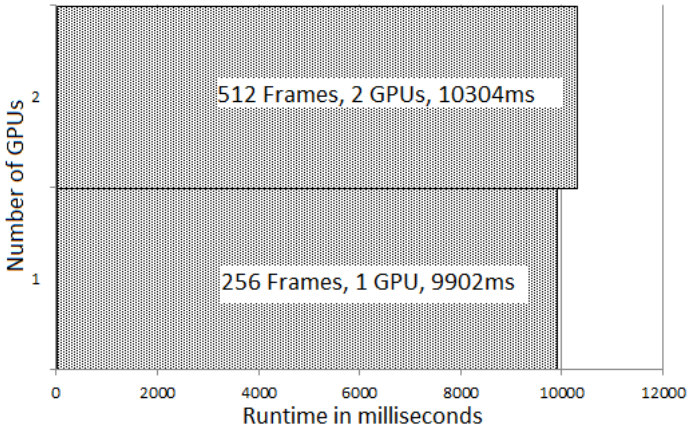


FIGURE 7.3: Total running times for each version

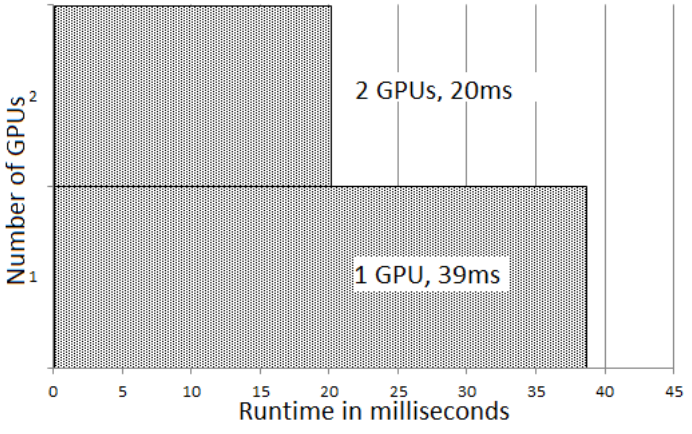


FIGURE 7.4: Representative per-frame processing time

Chapter 8

Conclusion

Existing GPU clusters were evaluated, with a focus on the Higher Education context and scientific research. This research allowed informed decisions regarding the deployment of our own system, and lead to the deployment of VEGA, a small GPU cluster integrated into to campus network forming part of the institutional grid. With this system in place it was possible to investigate and evaluate different platforms and programming environments to support the goal of highly parallel task and data processing. Using case studies the suitability of the GPU cluster was evaluated. The first case study showed that useful visualisation of seemingly abstract data could be achieved using the appropriate platforms, and enabled a level of analysis previously not possible for this data. By using the CUDA GPU programming model, as well as MPI, significant performance gains were possible, both for sonification of radio telescope data, and analysis of wavelength scanning interferometry data. While the initial objective was to use CUDA with Java and C++ to accelerate applications using multi-core GPU devices, It has been possible to achieve even greater performance gains though implementation of multi-system MPI code, to allow multiple GPUs to process data. Not only will the CUDA implimentations scale automatically to larger, faster GPUs, but MPI allows scaling to multi node GPU clusters.

The project has shown that even a modest investment in GPU systems would assist research in higher education institution, as with a comparatively small physical and energy footprint, data processing has been accelerated to levels comparable to a much larger HPC cluster.

It has become evident through the work carried out that the major obstacle in achieving potential speed up using GPU technology is lack of parallel platforms for creating, debugging and tenting parallel programs. To make effective use of a GPU cluster implies a greater effort on the part of the software author, compared to other HPC programming. Future investment in integrated and development environments is necessary to enable better utilisation of the existing GPU hardware. These are issues included as part of the EPSRC E-Infrastructure roadmap and Software and an infrastructure strategy (EPSRC, 2014a). There were a number of calls from the EPSCRC council related to Software of the Future which will attempt address these issues (EPSRC, 2014b).

Chapter 9

Further Work

The case studies detailed here have been designed to scale to much larger systems through using MPI. Running the software on larger systems will allow the efficiency and performance of the software to be assessed, giving opportunity to discover bottlenecks and develop more concrete methods for software design on GPU clusters.

This work shows the importance of efficient and sustainable software design. Timings showed that the performance gains from efficient design and language choices are just as important as targeting more powerful hardware. In order to make the most time and cost effective of hardware, evidence shows that investment in sustainable software infrastructures will be essential (Venters et al., 2014).

Appendix A

SILOWRITE tool

```
#include <silo.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>


using namespace std;

const float pi = 3.14159265359;


int main(int argc, char *argv[])
{
    int NX= 360;
    int NY= 100;
    int NZ= 1;
    int nnodes = NX*NY*NZ;
    if(argc<3)
    {
            cerr<< "USAGE:"<<argv[0]<<" 100 //(time in ns) 10.0 //radius"<<endl;
            system("pause");
            return -1;
    }

    stringstream tim, radi, mul;
    tim<<argv[1];
    radi<<argv[2];
    mul<<argv[3];
```

```
int maxtime;
tim>>maxtime;
float radius,multiplier;
radi>>radius;
mul>>multiplier;

float * x=new float[nnodes];
float * y=new float[nnodes];
float * z=new float[nnodes];

float *data=new float[nnodes];

for (int time=0; time<=maxtime; time++)
{
    /*Open input data*/
    ostringstream oss;
    oss << "data/" << time << "ns.dat";
    string fname = oss.str();
    string *str;
    str=new string[nnodes];
    float *indata;
    indata=new float[nnodes];

    ifstream file(fname);
    int i = 0;
    if(!file) //Always test the file open.
    {
                cout<<
                "Error opening input file,
                either incorrect path specified
                or insufficient data "<<endl;
                system("pause");
                return -1;
    }
    while(!file.eof())
    {
      getline(file,str[i],' ');
      if(i ==nnodes-1)
      {
           i=0;
           getline(file,str[i],' ');
      }
      i++;
    }
    for (int i=0; i<nnodes;i++)
    {
        indata[i]=atof(str[i].c_str());
    }
```

```
DBfile *dbfile = NULL;

/* Open the Silo file */

char filename[100];
sprintf (filename, "output/out%04d.silo", time);
dbfile = DBCreate(filename, DB_CLOBBER, DB_LOCAL,
        "simulation time step", DB_HDF5);
if(dbfile == NULL)
{
    fprintf(stderr, "Could not create Silo file!\n");
    return -1;
}

/*Copy in data*/

for (int i=0; i<nnodes;i++) data[i]=indata[i];

/*Add timing data for VisIt*/

DBoptlist *optlist = DBMakeOptlist(2);
DBAddOption(optlist, DBOPT_CYCLE, (void *)&time);
DBAddOption(optlist, DBOPT_DTIME, (void *)&time);

/*Create Geometry */


float rad=0;

/*3D Mesh*/

for(int iY=0;iY<NY;iY++)
{
    for (int iX=0;iX<NX;iX++)
    {
        for (int iZ=0;iZ<NZ;iZ++)
        {

            rad=iX*(pi/179);

            x[iY*NZ*NX+iX*NZ+iZ]=radius*cos(rad);
            y[iY*NZ*NX+iX*NZ+iZ]=iY-50;
            z[iY*NZ*NX+iX*NZ+iZ]=radius*sin(rad);
        }

    }
}
```

```
int dims[] = {NX, NY, NZ};
int ndims = 3;
float *coords[] = {(float*)x, (float*)y, (float*)z};
DBPutQuadmesh(dbfile, "Cylinder", NULL, coords, dims, ndims,
            DB_FLOAT, DB_NONCOLLINEAR, optlist);

/*Add data*/

dims[0] = NX; dims[1] = NY; dims[2] = NZ;
DBPutQuadvar1(dbfile, "3D", "Cylinder", data, dims,
ndims, NULL, 0, DB_FLOAT, DB_NODECENT, NULL);

/*2D Mesh*/
for(int iY=0;iY<NY;iY++)
{
    for (int iX=0;iX<NX;iX++)
    {
        for (int iZ=0;iZ<NZ;iZ++)
        {
            x[iY*NZ*NX+iX*NZ+iZ]=iX;
            y[iY*NZ*NX+iX*NZ+iZ]=iY-50;
        }

    }
}

int dims2[] = {NX, NY};
ndims = 2;
float *coords2[] = {(float*)x, (float*)y};
DBPutQuadmesh(dbfile, "FlatMesh", NULL, coords2, dims2, ndims,
            DB_FLOAT, DB_NONCOLLINEAR, optlist);


/*Add data*/

dims[0] = NX; dims[1] = NY;;
DBPutQuadvar1(dbfile, "Flat", "FlatMesh", data, dims,
ndims, NULL, 0, DB_FLOAT, DB_NODECENT, optlist);

/*3D countour mesh*/

float radiuscont;
i=0;
for(int iY=0;iY<NY;iY++)
{
    for (int iX=0;iX<NX;iX++)
    {
        for (int iZ=0;iZ<NZ;iZ++)
```

```
                {

                    rad=iX*(pi/179);
                    radiuscont=radius+data[i]*multiplier;
                    i++;

                    x[iY*NZ*NX+iX*NZ+iZ]=radius*cos(rad);
                    y[iY*NZ*NX+iX*NZ+iZ]=iY-50;
                    z[iY*NZ*NX+iX*NZ+iZ]=radius*sin(rad);
                }

            }
        }

        int dims3[] = {NX, NY, NZ};
        ndims = 3;
        float *coords3[] = {(float*)x, (float*)y, (float*)z};
        DBPutQuadmesh(dbfile, "Contour", NULL, coords3, dims3, ndims,
                    DB_FLOAT, DB_NONCOLLINEAR, optlist);

        /*Add data*/
        dims[0] = NX; dims[1] = NY; dims[2] = NZ;
        DBPutQuadvar1(dbfile, "3DContour", "Contour", data, dims,
        ndims, NULL, 0, DB_FLOAT, DB_NODECENT, optlist);

        /* Close the Silo file. */
        DBFreeOptlist(optlist);
        DBClose(dbfile);
    }
    delete[] x,y,z,data;
    return 0;
}
```

LISTING A.1: "main.cpp"

Appendix B

SETIFFT sonification code

Original program written by Paul Lunn for the SonicSETI project (SETI, 2013b). It was modified extensively and mostly rewritten to take advantage of multiple GPUs. MemoryMapped is a function written by Brumme (2014) and is used here to handle the large input files.

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <mpi.h>
#include <vector>
#include <sys/stat.h>
#include "cuComplex.h"

#include <Winsock2.h>
#pragma comment(lib, "Ws2_32.lib")

#include <cuda_runtime.h>
#include <cufft.h>
#include <cuda.h>
#include <helper_functions.h>
#include <helper_cuda.h>

#include "setifile.h"

//MPI TAGS
#define MSG 1
```

```
#define RANGE 2
#define MEM 3
#define DATA 4

#define PI 3.14159265358979323846




int myrank, totalranks;

MPI_Status Stat;


//------------------------------------------------------------
int checkErrors(int err)
{
    if (err>1) return 1;

    switch(err)
    {
        case 1: return 1;
        default: std::cout<<"Something's wrong!?!?"<<std::endl;break;
        case 0: std::cout<<"\n\nRequired:\n'-if <path>' specifies the path to the ↩
    input file(s). There can be any number of these, at least one is required\n\↩
    nOptions:\n-v Enables progress messages, may increase runtime\n-d Enables debug ↩
    mode, will definitely increase runtime\n-t Prints timing information\n-help ↩
    Shows these hints"<<std::endl; break;
        case -1: std::cout<<"Could not open input file!"<<std::endl;break;
        case -2: std::cout<<"Could not open output file!"<<std::endl;break;
        case -3: std::cout<<"Problem writing to output file!"<<std::endl;break;
        case -4: std::cout<<"Problem allocating GPU!"<<std::endl;break;
        case -5: std::cout<<"Problem reading from input file!"<<std::endl;break;
    }

    Sleep(2000);
    if (err <= 0) exit(1);
    return 0;
}
int concatinateFiles(const std::vector<std::string> &fileNames)//this bit is 100% not↩
     portable and is almost certainly poor form, but its faster than reading in the ↩
    files (Were already using windows.h anyway)
{ //joins multiple files with filename format *<integer>.<extension>
    size_t numFiles = fileNames.size();
    std::string command;
    std::stringstream ss;
    ss<<"copy ";
    for (int i=0; i<numFiles; i++)
```

```
    {
        ss<<fileNames[i]<<" ";
    }
    ss>>command;
    system(command.c_str());
    return 1;
}
bool checkExistance (const std::string& name)
{
    struct stat b;
    return (stat(name.c_str(), &b) == 0);
}
int cleanUp(const std::vector<std::string> &fileNames)
{
    size_t numFiles = fileNames.size();
    std::vector<std::string> concFiles;
    std::string concChk;
    int file=0;
    for (int i=0;i<=0;i=i)
    {
        concChk=fileNames[file]+std::to_string(i);
        if(checkExistance(concChk))
        {
            concFiles[i]=concChk;
            i++;
        }
        else
        {
            concatinateFiles(concFiles);
            file++;
        }
    }
    return 1;
}
int main(int argc, char **argv)
{

    int totalranks, myrank, master = 0;
    MPI_Status Stat;
    MPI_Request sreq = MPI_REQUEST_NULL;
    MPI_Request* rreq;

    int exit=0;

    //OPTIONS (set in command switches)=-=-=-=-=-=-=-=-=-=-=-=-
    int verbose=false;
    bool debug=false;
```

```
//-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &totalranks);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
rreq = new MPI_Request[totalranks];

int fftPower=23;
std::vector<std::string> fileNames;
int fileCount=0;

// parse options/use defaults
if(myrank==master) std::cout << argv[0]<<" ";

if (argc < 1&&myrank==master)
{
    return checkErrors(0);
}

MPI_Barrier(MPI_COMM_WORLD);
for (int i = 1; i < argc; i++)
{
    if(myrank==master) std::cout << argv[i]<<" ";
    if (std::string(argv[i]) == "-if")
    {
        (fileNames.push_back(argv[++i]));
        fileCount++;
    }
    else if (std::string(argv[i]) == "-v")
    {
        verbose=atoi(argv[++i]);
    }
    else if (std::string(argv[i]) == "-d")
    {
        debug=true;
    }
    else if (std::string(argv[i]) == "-help"&&myrank==master)
    {
        return checkErrors(0);
    }
    else if (myrank==master)
    {
         return checkErrors(0);
    }
}

if (fileCount==0)
{
```

```
        return checkErrors(0);
    }
    if(debug&&myrank==master)
    {
        std::cout << "Input once debugger is attatched... " << std::endl;
        std::cin.get();
    }


    MPI_Barrier(MPI_COMM_WORLD);


    //=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-


    int range[3] = {0,0,0}; //first file, first fft, total ffts


    bool data=true;
    char instruction = 0;

    int totalFFTs=0;
    int fftsPerNode=0;

    if (myrank == master)
    {
        bool* skiprank=new bool[totalranks];
        bool* GPUrank=new bool[totalranks];
        for(int i=0;i<totalranks;i++)
        {
            skiprank[i]=false;
            GPUrank[i]=false;
        }
        int gpus=0;

        for(int i=1;i<totalranks;i++)
        {
            rreq[i] = MPI_REQUEST_NULL;
            char testgpu=false;
            MPI_Recv(&testgpu,1,MPI_CHAR,i,MSG,MPI_COMM_WORLD,&Stat);
            if(testgpu) gpus++;
            else skiprank[i]=true;//ranks without gpus are skipped
        }
        //Divide data based on number of gpus
        int *totalFFTsInFile = new int [fileCount];
        for (int i=0;i<fileCount;i++)
        {
```

```
            checkErrors(totalFFTsInFile[i]=SETIFILE::getNumFFTs(fileNames[i],fftPower↩
    ));
            totalFFTs+=totalFFTsInFile[i];
        }

        fftsPerNode=totalFFTs/gpus;
        int rem=totalFFTs%gpus;
        int file=0;
        int sentTotal=0;
        int sentTotalInFile=0;
        for(int i=1;i<totalranks;i++)
        {
            if(!skiprank[i])
            {
                range[0]=file;
                range[2]=fftsPerNode+rem;
                sentTotal+=fftsPerNode+rem;
                sentTotalInFile+=fftsPerNode+rem;
                if (sentTotalInFile>=totalFFTsInFile[file])
                {
                    sentTotalInFile-=totalFFTsInFile[file];
                    file++;
                }

                rem=0;//if totalFFTs is not devided evenly, remainder goes to first ↩
    rank
                MPI_Send(range,3,MPI_INT,i,RANGE,MPI_COMM_WORLD);
                range[1]+=range[2];

            }
        }
        if(verbose>1) std::cout << "Master: Total FFTs: "<<totalFFTs<<". FFTs per ↩
    rank: "<<fftsPerNode <<std::endl;
        for(int i=1;i<totalranks;i++)
        {
            if(!skiprank[i])
            {
                MPI_Irecv(&instruction,1,MPI_CHAR,i,MSG,MPI_COMM_WORLD,&rreq[i]);
            }
        }
        while(data)//wait for any response from any rank
        {
            for(int i=1;i<totalranks;i++)
            {
                if(!skiprank[i])
                {
                    int flag=0;
                    MPI_Test(&rreq[i],&flag,&Stat);
```

```
                if(flag)//if there was an instruction do something, else move on
                {
                    switch(instruction)
                    {
                        case 3: data=false; break;
                        case 4: data=false; break;
                    }
                }
            }
        }
    }
}
else if (totalranks>=2)
//WORK=-=-=-=-=-=-=-=-=
{
    char testGPU=SETIFILE::getDeviceCount();
    MPI_Send(&testGPU,1,MPI_CHAR,master,MSG,MPI_COMM_WORLD);

    SYSTEMTIME st;
    GetSystemTime(&st);
    int startTime = ((((st.wHour*60)+st.wMinute)*60)+st.wSecond)*1000 + st.↩
wMilliseconds;

    if(testGPU!=0)
    {
        MPI_Recv(&range,3,MPI_INT,master,RANGE,MPI_COMM_WORLD,&Stat);

        int file = range[0];
        int first = range[1];
        int total = range [2];


        int rankFileCount=1;

        int templen = range[2]-range[1];

        while (templen>SETIFILE::getNumFFTs(fileNames[file],23))
        {
            templen-=SETIFILE::getNumFFTs(fileNames[file++],23);
            rankFileCount++;
        }


        file = range[0];
        int readtime=0, ffttime=0, powertime=0, writetime=0;
        for(int i=0; i<rankFileCount;i++)
        {
            std::string ofn = fileNames[file];
```

```
                ofn.resize(ofn.size()-3);
                ofn.append("setifile"+std::to_string(myrank));
                SETIFILE seti(fftPower,fileNames[file],ofn,myrank-1);

                if(total+first>seti.getNumFFTs()) total=seti.getNumFFTs();

                if(verbose>0)std::cout<<"Rank "<<myrank<<": Performing "<<total<<" FFTs from file "<<fileNames[file]<<std::endl;



                checkErrors(seti.doFFT(first,total));



                file++;
                total=range[2]-total;
                readtime+=seti.readtime;
                powertime+=seti.powerstime;
                writetime+=seti.writetime;
                seti.~SETIFILE();
            }
            if(verbose>0)std::cout<<"Rank "<<myrank<<"read time: "<<readtime<<"fft time: "<<fftime<<"calculate powers time: "<<powertime<<"write time: "<<readtime<<std::endl;
            instruction=3;
            GetSystemTime(&st);
            int taskTime = (((((st.wHour*60)+st.wMinute)*60)+st.wSecond)*1000 + st.wMilliseconds);
            taskTime = taskTime - startTime;
        std::cout<<"Total time: "<< taskTime<<std::endl;
            MPI_Send(&instruction,1,MPI_CHAR,master,MSG,MPI_COMM_WORLD);
        }
    }
    else
    {
        std::cout << "SETIFFTcu: Must be started with at least -n 2" << std::endl;
    }

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();
    return exit;
}
```

LISTING B.1: ”SETIFFTcu.cpp”

```
#pragma once

#include "MemoryMapped.h"
#include "cuComplex.h"
#include <cuda.h>


class SETIFILE
{
private:
    int fail;

    int N, N_2;

    std::string inName, outName;

    MemoryMapped inFile;
    FILE* outFile;

    size_t fileLength;

    int numFFTs;

    cuDoubleComplex *data;
    unsigned long long* power;

    //Process
    int read(int);
    int calculatePowers(int);
    int writePowers(int);

    size_t pos;

    int capacity;

    CUcontext cudaContext;
    CUdevice cudaDevice;

public:

    int readtime, ffttime, powerstime, writetime;
    int doFFT(int,int);

    //Get
    int getN();
    int getNumFFTs();
    size_t getFileLength();
```

```
    static int getNumFFTs(std::string,int);
    static size_t getFileLength(std::string);
    static int getDeviceCount();

    void printTime();
    std::string getFileName();

    int checkFail();

    SETIFILE(int,std::string,std::string, int);
    ~SETIFILE();
};
```

LISTING B.2: ”setifile.h”

```
#include "setifile.h"
#include "MemoryMapped.h"
#include "cuComplex.h"
#include <cuda_runtime.h>
#include <cufft.h>
#include <cuda.h>
#include <helper_functions.h>
#include <helper_cuda.h>
#include <iostream>

SETIFILE::SETIFILE(int fftPower, std::string fileName, std::string ↩
    destinationFileName, int device)
{
    readtime=0;
    ffttime=0;
    powerstime=0;
    writetime=0;

    fail = 1;

    N=(int)pow(2,fftPower);
    N_2 = N/2;

    inName = fileName;
    outName = destinationFileName;

    inFile.open (inName,MemoryMapped::WholeFile, MemoryMapped::SequentialScan);
    if (!inFile.isValid()) fail = -1;

    outFile = fopen(outName.c_str(), "wb");
    if (outFile==NULL) fail = -2;

    fileLength=inFile.size();

    numFFTs=(int)fileLength/N;

    inName = fileName;
    outName = destinationFileName;
    pos=0;
    capacity=0;
    checkCudaErrors(cuInit(0));
    CUresult result = cuDeviceGet(&cudaDevice,device);
    if (result == CUDA_SUCCESS)
    {
        checkCudaErrors(cuCtxCreate(&cudaContext, CU_CTX_SCHED_AUTO, cudaDevice));
    }
}
```

```
SETIFILE::~SETIFILE()
{
    cuCtxDestroy(0);
    DEVICE_RESET;
    inFile.close();
    fclose(outFile);
}

int SETIFILE::doFFT(int firstFFT, int totalFFTs)
{
    SYSTEMTIME fftst;
    GetSystemTime(&fftst);
    int fftstartTime = ((((fftst.wHour*60)+fftst.wMinute)*60)+fftst.wSecond)*1000 + ↩
    fftst.wMilliseconds;

    size_t fmem, tmem;

        int running=totalFFTs;
        checkCudaErrors(cuMemGetInfo(&fmem, &tmem));

        int size=sizeof(cuDoubleComplex);
        capacity=fmem/size;
        capacity/=N;
        capacity/=2; //CUFFT makes another array on the GPU while processing data, so↩
     we can only fill half of GPU memory
        if (capacity<=0)
        {
            fail= -4;
            return fail;
        }

        int num=capacity;
        while (running>0)
        {
            if (running<capacity)
            {
                num=running;
            }
            data=new cuDoubleComplex[num*N];

            fail = read(num);


    //Allocate device memory
            size_t mem_size=sizeof(cuDoubleComplex) * N*num;
            cuDoubleComplex *d_data;
            checkCudaErrors(cudaMalloc((void**)&d_data, mem_size));
            //Copy data to device
```

```
            checkCudaErrors(cudaMemcpy(d_data, data, mem_size, cudaMemcpyHostToDevice↩
    ));
            //CUFFT plan
            cufftHandle plan;

            checkCudaErrors(cufftPlan1d(&plan, N, CUFFT_Z2Z, num));
            checkCudaErrors(cufftExecZ2Z(plan, d_data ,d_data ,CUFFT_FORWARD));
            //Copy data back to host
            checkCudaErrors(cudaMemcpy(data, d_data, mem_size,cudaMemcpyDeviceToHost)↩
    );
            //Destroy CUFFT complex
            checkCudaErrors(cufftDestroy(plan));
            cudaFree(d_data);




            calculatePowers(num);

            running-=num;
            delete [] data;
        }
        GetSystemTime(&fftst);
        int ffttaskTime = (((((fftst.wHour*60)+fftst.wMinute)*60)+fftst.wSecond)*1000↩
     + fftst.wMilliseconds);
        ffttaskTime = ffttaskTime - fftstartTime;
        fftime+=ffttaskTime;

    return fail;
}
int SETIFILE::read(int num)
{
    SYSTEMTIME st;
    GetSystemTime(&st);
    int startTime = ((((st.wHour*60)+st.wMinute)*60)+st.wSecond)*1000 + st.↩
    wMilliseconds;

    int ret= -5;
    for(int i=0;i<num*N;i++)
    {
        char re=0;
        char im=0;

        if(pos<getFileLength()) re = inFile[pos++];
        else re=-1;
        if(pos<getFileLength()) im = inFile[pos++];
        else im=-1;
```

```
        data[i]=make_cuDoubleComplex(re,im);
        ret=1;
    }
    GetSystemTime(&st);
    int taskTime = (((((st.wHour*60)+st.wMinute)*60)+st.wSecond)*1000 + st.↩
    wMilliseconds);
    taskTime = taskTime - startTime;
    readtime+=taskTime;
    ffttime-=taskTime;

    return ret;
}
int SETIFILE::calculatePowers(int num)
{

    SYSTEMTIME st;
    GetSystemTime(&st);
    int startTime = ((((st.wHour*60)+st.wMinute)*60)+st.wSecond)*1000 + st.↩
    wMilliseconds;
    power = new unsigned long long[num*N_2];
    double *powtemp=new double[num*N_2];

    for (int a=0;a<num;a++)
    {
        for(int i = 0; i< N_2; i++)
        {
            powtemp[i] = (hypot(data[i].x,data[i].y)/(N_2));
            char ch[8],ch2[8];
            memcpy( ch, &powtemp[i], 8 );
            for (int c=0;c<8;c++) ch2[c]=ch[7-c];
            memcpy( &power[i], ch2, 8 );
        }
    }


    delete [] powtemp;
    GetSystemTime(&st);
    int taskTime = (((((st.wHour*60)+st.wMinute)*60)+st.wSecond)*1000 + st.↩
    wMilliseconds);
    powerstime += taskTime - startTime;
    ffttime-=taskTime;

    fail=writePowers(num);


    delete [] power;
    return 1;
}
```

```
int SETIFILE::writePowers(int num)
{
    SYSTEMTIME(st);
    GetSystemTime(&st);
    int startTime = ((((st.wHour*60)+st.wMinute)*60)+st.wSecond)*1000 + st.↩
    wMilliseconds;
    size_t ret = -3;
    if (outFile==NULL) ret = -2;
    else
    {
        ret=fwrite(power,sizeof(unsigned long long),num*N_2,outFile);
    }
    GetSystemTime(&st);
    int taskTime = (((((st.wHour*60)+st.wMinute)*60)+st.wSecond)*1000 + st.↩
    wMilliseconds);
    taskTime = taskTime - startTime;
    writetime+=taskTime;
    ffttime-=taskTime;

    return ret;
}

size_t SETIFILE::getFileLength() {return fileLength;}
std::string SETIFILE::getFileName() {return inName;}
int SETIFILE::getN() {return N;}
int SETIFILE::getNumFFTs() {return numFFTs;}
int SETIFILE::checkFail() {return fail;}

//Static Functions-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
int SETIFILE::getNumFFTs(std::string fname,int power)
{
    size_t len = getFileLength(fname);
    if (len<0) return -1;
    size_t num=len/pow(2,power);
    return num;
}
size_t SETIFILE::getFileLength(std::string fname)
{
    MemoryMapped infile;
    infile.open (fname,MemoryMapped::WholeFile, MemoryMapped::SequentialScan);
    if (!infile.isValid()) return -1;
    size_t length=infile.size();
    infile.close();
    return length;
}

int SETIFILE::getDeviceCount()
{
```

```
    int devCount;
    cudaGetDeviceCount(&devCount);
    return devCount;
}
```

LISTING B.3: ”setifile.cu”

The following file is an example of a batch script used to execute the MPI task.

```
:: Switches:

:: Required:
:: '-if <path>' specifies the path to the input file(s). There can be any number of ↩
    these, at least one is required

:: Options:
:: -v Enables progress messages, may increase runtime
:: -d Enables debug mode, will definitely increase runtime
:: -h prints these hints, then quits


mpiexec -n 3 SETIFFTcu.exe -v 5 -d -if 1.dat -if 2.dat -if 3.dat

:: exit codes: 0 = Normal exit, all data processed, 1 = Fail, no data processed, 2 = ↩
    Fail, some data processed
```

LISTING B.4: ”SETIFFT with MPI.bat”

Appendix C

Wavelength Scanning Interferometry Software

Full code for case study 3, based on the original CUDA software described by Muhamedsalih et al. (2012). Functions and data operations taken directly from original code, the program structure was modified and MPI functions added.

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <cutil_inline.h>
#include <conio.h>
#include <string.h>
#include <sstream>
#include <helper_cuda.h>
#include <cuda.h>
#include <mpi.h>
#include <cuda_runtime.h>
#include <windows.h> //for handling bitmap files, not portable, remove for other ↩
    platforms

//MPI TAGS
#define MSG 1
#define RANGE 2
#define MEM 3
#define DATA 4
```

```
long lNumImage=256;//128//128//256//1024//512//64//256//300//128
long lNumPulses2=1024;
int LensX=5;
float *Kv;




//extern
int DeviceFFT(LPBYTE lpLinear,long FrameWidth, long FrameHeight,char* szFileName, ↩
    float* Kv, long lNumImage, long lNumPulses2, int LensX, int rank);
float* CalcuateWaveNo();


int main(int argc, char* argv[])
{

    int totalranks, myrank, master = 0;
    MPI_Status Stat;
    MPI_Request sreq = MPI_REQUEST_NULL;
    MPI_Request* rreq;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &totalranks);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    rreq = new MPI_Request[totalranks];
    if(myrank==master)
    {
        if (totalranks<2)
        {
            std::cout<<"Program must be started with more than one rank"<<std::endl;
            MPI_Finalize();
            return 0;
        }
        std::cout<<"Enter to start"<<std::endl;
        std::cin.get();
    }
    MPI_Barrier(MPI_COMM_WORLD);
    if(myrank==master)//all timing done in master thread
    {
        SYSTEMTIME st;
        GetSystemTime(&st);
        int startTime = ((((st.wHour*60)+st.wMinute)*60)+st.wSecond)*1000 + st.↩
    wMilliseconds;
        bool*   gpuRanks=new bool[totalranks];
        int numGpus =0;
        MPI_Barrier(MPI_COMM_WORLD);
```

```
        for(int i=1;i<totalranks;i++)
        {
            char result=0;

            MPI_Recv(&result,1,MPI_CHAR,i,MSG,MPI_COMM_WORLD,&Stat);

            if(result)
            {
                gpuRanks[i]=true;
                numGpus++;
            }
            else gpuRanks[i]=false;
        }

        MPI_Barrier(MPI_COMM_WORLD);

        GetSystemTime(&st);
        int taskTime = (((((st.wHour*60)+st.wMinute)*60)+st.wSecond)*1000 + st.↩
    wMilliseconds);
        taskTime = taskTime - startTime;

        std::cout<< lNumImage*numGpus <<" Frames processed in: "<< taskTime <<" ↩
    milliseconds"<< std::endl;
    }

    else //work
    {
        MPI_Barrier(MPI_COMM_WORLD);
        LPBYTE      lpLinear;
        long        FrameWidth=640;
        long        FrameHeight=480;
        char szFileName[128];
        sprintf(szFileName,"%d",myrank);
        strcat(szFileName,"\\out.sdf");
        std::stringstream ss;

        bool    gpuRank = false;


        /**
        (1) Upload you data here (The 256 frames) and generate a pointer (lpLinear) ↩
    to indicate the location
        (2) Generate a path to save the SDF File szFileName
        **/


        FILE* bmp;
        DWORD size = lNumImage*FrameHeight*FrameWidth;
```

```
    BYTE* pixdata = new BYTE[size];
    BITMAPFILEHEADER bmpheader;
    std::string fn;
    for (int i = 0; i < lNumImage; i++)
    {
        ss.clear();
        ss << "data\\" << i+1000 << ".BMP";
        ss >> fn;
        bmp = fopen(fn.c_str(),"rb");
        if (bmp==NULL) std::cout << "File: " << fn << " could not be opened" <<↩
std::endl;
        fread(&bmpheader,sizeof(BITMAPFILEHEADER),1,bmp);//read bitmap header ↩
data

        if(bmpheader.bfType!=19778)//check file is valid BMP
        {
            std::cout << "File: " << fn << " is not valid BMP format" << std::↩
endl;
        }
        fseek(bmp, bmpheader.bfOffBits, SEEK_SET);//seek past header bits and to ↩
actual image data

        fread(&pixdata[i*size],sizeof(BYTE),size,bmp);//read image data
        fclose(bmp);
    }
    lpLinear=pixdata;
    int device = myrank-1;

    checkCudaErrors(cuInit(0));

    CUcontext cudaContext;
    CUdevice cudaDevice;
    CUresult result = cuDeviceGet(&cudaDevice,device);
    if (result == CUDA_SUCCESS)
    {
        result = cuCtxCreate(&cudaContext, CU_CTX_SCHED_AUTO, cudaDevice);
        gpuRank=true;
    }

    MPI_Send(&gpuRank,1,MPI_CHAR,master,MSG,MPI_COMM_WORLD);

    if(gpuRank==true)
    {
        Kv=CalcuateWaveNo();
        if(DeviceFFT(lpLinear,FrameWidth, FrameHeight,szFileName,Kv,lNumImage,↩
lNumPulses2,LensX,myrank)==-1) std::cout <<"Insufficient Memory on GPU"<<std::↩
endl;
```

```
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}


float* CalcuateWaveNo()
{
    int FrameStep=1; //This value will be used to jump one or three frames in case ↩
    captured frames are 128 or 64 respectively
    int WavelengthCounter=0;
    int i=0;

    float *KvFit=(float*)malloc(sizeof(float)*lNumImage); //Allocate a matrix for ↩
    KvFit
    float *Kv=(float*)malloc(sizeof(float)*lNumImage); // 1/lambda

    //------Loading the (kv=1/wavelength) matrix-----------------
    FILE * pCalMat; // calibration matrix
    pCalMat=fopen("KvFit256.bin","rb"); // Change the Path
    if (pCalMat!=NULL)
    {
    fread(KvFit,sizeof(float),lNumImage,pCalMat);
    fclose(pCalMat);
    }
    //--------------------END-----------------------------------


    if(lNumImage==128){
        FrameStep=2;
    }
    else if(lNumImage==64){
        FrameStep=4;
    }
    else{
        FrameStep=1;
    }

    for(i=0;i<lNumImage;i++)
    {
        //Kv[i]=1/Lambda[WavelengthCounter];
        Kv[i]=KvFit[WavelengthCounter]*0.902;//0.902 is a calibration factor
        WavelengthCounter=WavelengthCounter+FrameStep;
    }

    Kv[0]=1/683.42; // measured value with spectrometer// used
```

```
    Kv[lNumImage-1]=1/590.516;// measured value with spectrometer


    free(KvFit);
    return Kv;

}
```

LISTING C.1: ”main.cu”

```
#include <cutil_inline.h>
#include <cutil.h>
#include <math.h>
#include <FFT.h>
#include <FFT_kernel.cu>
#include <helper_cuda.h>
#include <cuda.h>
#include <cuda_runtime.h>

long BuffElement=0;
long        DataElement;
long        ElementWritten;
int i,j;
int         FrameCounter;
FILE * IntensityValueFile;
char Hussbuffer [50]; // delete it
FILE * pPixelFile;

//char MonitorFileName[128];

void WriteIntoText(Complex* HostMontrMatrx,long FrameWidth,long FrameHeight,long ↩
    lNumImage,char* MonitorFileName);
void WriteFloatIntoText(float* HostFloatMatrx,long FrameWidth,long FrameHeight,long ↩
    lNumImage,char* MonitorFileName);
int DeviceFFT(LPBYTE lpLinear,long FrameWidth,long FrameHeight,char*  szFileName,↩
    float* Kv, long lNumImage,long lNumPulses2, int LensX, int rank);

int DeviceFFT(LPBYTE lpLinear,long FrameWidth,long FrameHeight,char*  szFileName,↩
    float* Kv, long lNumImage, long lNumPulses2, int LensX,int rank)
{
    CUresult result;
    char fileName[128];
    //————————Transfer the reciprocal of wavelength arrary (Kv) to GPU Global memory ↩
    ————
    float *Kv_d;

    size_t mem_size = lNumImage*sizeof(float);

    cudaMalloc((void**)&Kv_d,mem_size); // see page 48 the bottom note for cudaMalloc↩
     function

    cudaMemcpy(Kv_d,Kv,mem_size,cudaMemcpyHostToDevice);//Copy wavelength values
    //————————————————————

    /*———The following code statments are calibration steps
      ———The clibrated matrix has been processed by Matlab and saved in D drive———↩
    */
```

```
//————————————One Frame alibration————————————————
FILE * pCalOneFrame; // calibration one Frame
//—— Allocate memory space for the background Inensity array in CPU
float *OneFrameIntHost=(float*)malloc(sizeof(float)*FrameHeight*FrameWidth);//on ↩
the host side
sprintf(fileName,"%d",rank);
strcat(fileName,"\\OneFrameIntensity.bin");
pCalOneFrame=fopen(fileName,"rb");
if (pCalOneFrame!=NULL)
{
fread(OneFrameIntHost,sizeof(float),FrameHeight*FrameWidth,pCalOneFrame);
fclose(pCalOneFrame);
}
float *OneFrameInt_d;// discontinuous phase matrix
mem_size=sizeof(float)*FrameHeight*FrameWidth;
cudaMalloc((void**)&OneFrameInt_d,mem_size);
cudaMemcpy(OneFrameInt_d,OneFrameIntHost,mem_size,cudaMemcpyHostToDevice);
//————————————END————————————————


//———Single Pixel Calibration at different wavelengths———
FILE * pCalMat; // calibration one pixel (200,200)
float *CalMatHost=(float*)malloc(sizeof(float)*lNumImage);//on the host side

sprintf(fileName,"%d",rank);
if (LensX==2)
{
    if (lNumImage==256)
    {
    strcat(fileName,"\\RemoveBackground_256.bin");
    }
    else if (lNumImage==128)
    {
    strcat(fileName,"\\RemoveBackground_128.bin");
    }
    else if (lNumImage==64)
    {
    strcat(fileName,"\\RemoveBackground_64.bin");
    }

    }
else if (LensX==5)
{
    if (lNumImage==256)
    {
    strcat(fileName,"\\RemoveBackground_256.bin");
    }
```

```
        else if (lNumImage==128)
        {
        strcat(fileName,"\\RemoveBackground_128.bin");
        }
        else if (lNumImage==64)
        {
        strcat(fileName,"\\RemoveBackground_264.bin");
        }
    }
    else if (LensX==10)
    {
        if(lNumImage==256)
        {
        strcat(fileName,"\\RemoveBackground_256.bin");
        }
        else if (lNumImage==128)
        {
        strcat(fileName,"\\RemoveBackground_128.bin");
        }
        else if(lNumImage==64)
        {
        strcat(fileName,"\\RemoveBackground_64.bin");
        }
    }
    pCalMat=fopen(fileName,"rb");
    if (pCalMat!=NULL)
    {
    fread(CalMatHost,sizeof(float),lNumImage,pCalMat);
    fclose(pCalMat);
    }

    float *CalMat_d;// discontinuous phase matrix
    cudaMalloc((void**)&CalMat_d,sizeof(float)*lNumImage);
    cudaMemcpy(CalMat_d,CalMatHost,sizeof(float)*lNumImage,cudaMemcpyHostToDevice);

    //----------------END of Calibration----------------------

    //---Copy the BUFFER data to allocate memory BufData
    mem_size=lNumImage*FrameWidth*FrameHeight;
    byte *BufData=new byte[mem_size];
    memcpy(BufData,lpLinear,mem_size);

    //--------------------------------------------------------

    //-----Divide the Captured Data into 4 parts because gof memory limitation↩
    ----------
    int ProcessStep=4;
    int HeightDivider=FrameHeight/ProcessStep;
```

```
int CorrectionFactor=HeightDivider/BLOCK_SIZE;
HeightDivider=CorrectionFactor*BLOCK_SIZE;
int RemainPixels=FrameHeight-HeightDivider*ProcessStep;
//-----------------------------------------------------------------



unsigned int size_StepHeight_d=HeightDivider*FrameWidth;  //
unsigned int mem_size_StepHeight_d=sizeof(float)*size_StepHeight_d;
float *HostFinalStepHeight=(float*)malloc(sizeof(float)*HeightDivider*FrameWidth*↩
 ProcessStep);


for(int ProcessCounter=0;ProcessCounter<ProcessStep;ProcessCounter++)
{
//----Allocate a memory space for Data in the GPU
Complex *data_d;
unsigned int size_data_d=lNumImage*HeightDivider*FrameWidth;  // 1D_array_matrix
unsigned int mem_size_data_d=sizeof(Complex)* size_data_d;
cudaMalloc((void**)&data_d,mem_size_data_d);
//----

//----Allocate a memory space for Phase in the GPU
float *phase_discont_d;// discontinuous phase matrix
unsigned int size_phase_discont=lNumImage*HeightDivider*FrameWidth;  // 1↩
D_array_matrix
unsigned int mem_size_phase_discont=sizeof(float)* size_phase_discont;
cudaMalloc((void**)&phase_discont_d,mem_size_phase_discont);
//----

//----Allocate a memory space for phase set and corrected phase in the GPU
float *phase_set_d;// discontinuous phase matrix
cudaMalloc((void**)&phase_set_d,mem_size_phase_discont);
float *phase_cont_d; // discontinuous phase matrix
cudaMalloc((void**)&phase_cont_d,mem_size_phase_discont);
//----

//----Allocate a memory space for Suface Height in the GPU
float *StepHeight_d;
unsigned int size_StepHeight_d=HeightDivider*FrameWidth;  //
unsigned int mem_size_StepHeight_d=sizeof(float)*size_StepHeight_d;
cudaMalloc((void**)&StepHeight_d,mem_size_StepHeight_d);
//----

//----Allocate a memory space for Suface Height in the Host
float *HostStepHeight=(float*)malloc(mem_size_StepHeight_d);//on the host side
//----
```

```
/*************************************************
/   Replace the Buffer information into Data matrix
*************************************************/
unsigned int size_Data=lNumImage*FrameWidth*HeightDivider;//FrameHeight*↩
FrameWidth;
unsigned int mem_size_Data=sizeof(Complex)*size_Data;
Complex *Data=(Complex*)malloc(mem_size_Data);// the data that is going to be ↩
processed
DataElement=0;

for (FrameCounter=0; FrameCounter<lNumImage;FrameCounter++)
{
    BuffElement=0;
    for(i=0; i<HeightDivider; i++){
        for(j=0; j<FrameWidth; j++){
            DataElement=(i*FrameWidth*lNumImage)+(j*lNumImage)+FrameCounter;    ↩
//var_height(j)=row number  & var_width(i)= col number
            Data[DataElement].x=(float)(BufData[BuffElement+((ProcessCounter*↩
HeightDivider)*FrameWidth)+FrameCounter*(FrameHeight*FrameWidth)]);
            BuffElement=BuffElement+1;
        }
    }
}
//————End of Data Arrangement————————————



//————————————Start Slop phase Analysis using GPU————————

//——  (1) Transfer arrangement frames into GPU
cudaMemcpy(data_d,Data,mem_size_data_d,cudaMemcpyHostToDevice);
//——

//——  (2) Thread Generation
// This condition has set because the multiplied matrix is square and can divide ↩
over tile width
dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);   //specify the dimensions of each BLOCK in↩
 terms of number of thread
dim3 dimGrid(FrameWidth/BLOCK_SIZE,HeightDivider/BLOCK_SIZE);   //specify the ↩
dimensions of each GRID in terms of number of thread
//——

//——   (3) Remove the Background Intensity
CalibrationIntensityPattern<<<dimGrid,dimBlock>>>(data_d,CalMat_d,OneFrameInt_d,↩
ProcessCounter,FrameWidth,HeightDivider,lNumImage);

//——   (3.a) Remove the offset
```

```
RemoveOffset<<<dimGrid,dimBlock>>>(data_d,ProcessCounter,FrameWidth,HeightDivider↩
,lNumImage);

//——   (Non) Write text file the fft of one pixel (1)
Complex *HostMontrMatrx=(Complex*)malloc(mem_size_data_d);
cudaMemcpy(HostMontrMatrx,data_d,mem_size_data_d,cudaMemcpyDeviceToHost);
sprintf(fileName,"%d",rank);
strcat(fileName,"\\CalibratedSignal.text");
WriteIntoText(HostMontrMatrx,FrameWidth,FrameHeight,lNumImage,fileName);
//————————————————————————————

//——   (4) Apply FFT
cufftHandle plan;
cufftPlan1d(&plan, lNumImage, CUFFT_C2C, FrameWidth * HeightDivider);
cufftExecC2C(plan, (cufftComplex *)data_d, (cufftComplex *)data_d, CUFFT_FORWARD)↩
;

//——   (5) Find the peak of spectrum to identify the filtering cutting edge
fftFindPeaks<<<dimGrid,dimBlock>>>(data_d,HeightDivider,FrameWidth,lNumImage);

//——   (Non) Write text file the fft of one pixel (2)
cudaMemcpy(HostMontrMatrx,data_d,mem_size_data_d,cudaMemcpyDeviceToHost);
sprintf(fileName,"%d",rank);
strcat(fileName,"\\FFTPowerDensity.text");
WriteIntoText(HostMontrMatrx,FrameWidth,FrameHeight,lNumImage,fileName);
//————————————————————————————


//——   (6) Filter the DC bias, the conjugate of the phase and noises
filtering_data_d<<<dimGrid,dimBlock>>>(data_d,FrameWidth,HeightDivider,lNumImage)↩
;

//——   (7) Apply inverse FFT
cufftExecC2C(plan, (cufftComplex *)data_d, (cufftComplex *)data_d, CUFFT_INVERSE)↩
;
cufftDestroy(plan);// important step; otherwise you can not repeat the propgram


//——   (Non) Write text file the fft of one pixel (3)
cudaMemcpy(HostMontrMatrx,data_d,mem_size_data_d,cudaMemcpyDeviceToHost);
sprintf(fileName,"%d",rank);
strcat(fileName,"\\iFFTPowerDensity.text");
WriteIntoText(HostMontrMatrx,FrameWidth,FrameHeight,lNumImage,fileName);
//————————————————————————————

//——   (8) Determine the phase
Determine_phase_discont<<<dimGrid,dimBlock>>>(data_d,phase_discont_d,FrameWidth,↩
HeightDivider,lNumImage);
```

```
//——   (9) Compute the 2 pi phase distrubution
Implement_phase_set<<<dimGrid,dimBlock>>>(phase_discont_d,phase_set_d,FrameWidth,↩
HeightDivider,lNumImage);

//——   (10) Correct the phase by adding 9 to 8
Calculate_phase_cont<<<dimGrid,dimBlock>>>(phase_cont_d,phase_discont_d,↩
phase_set_d,FrameWidth,HeightDivider,lNumImage);// adding the discontinuous ↩
phase to the phase set function

//——   (Non) Write text file the fft of one pixel (4)
float *HostFloatMatrx=(float*)malloc(mem_size_phase_discont);
cudaMemcpy(HostFloatMatrx,phase_cont_d, mem_size_phase_discont,↩
cudaMemcpyDeviceToHost);
sprintf(fileName,"%d",rank);
strcat(fileName,"\\Phase.text");
WriteFloatIntoText(HostFloatMatrx,FrameWidth,FrameHeight,lNumImage,fileName);
//————————————————————————————

//——   (11) Fit the phase using least square approach
LeastSquareFitting<<<dimGrid,dimBlock>>>(phase_cont_d,FrameWidth,HeightDivider,↩
lNumImage);

//——   (Non) Write text file the fft of one pixel (5)
cudaMemcpy(HostFloatMatrx,phase_cont_d, mem_size_phase_discont,↩
cudaMemcpyDeviceToHost);
sprintf(fileName,"%d",rank);
strcat(fileName,"\\FittedPhase.text");
WriteFloatIntoText(HostFloatMatrx,FrameWidth,FrameHeight,lNumImage,fileName);
//————————————————————————————

//——   (12) Find the Sufrace Height
Find_StepHeight<<<dimGrid,dimBlock>>>(StepHeight_d,phase_cont_d,FrameWidth,↩
HeightDivider,Kv_d,lNumImage);

//——   (13) upload the result back to CPU
cudaMemcpy(HostStepHeight,StepHeight_d,mem_size_StepHeight_d,↩
cudaMemcpyDeviceToHost);//FrameWidth*HeightDivider*sizeof(float)

//——   (14) Place the result in a successive manner in the surface height data ↩
that going to written as SDF
memcpy(HostFinalStepHeight+(ProcessCounter*FrameWidth*HeightDivider),↩
HostStepHeight,mem_size_StepHeight_d);

//——Delete the arrays——
cudaFree(data_d);
cudaFree(phase_discont_d);
cudaFree(phase_set_d);
```

```
    cudaFree(phase_cont_d);
    cudaFree(StepHeight_d);
    free(HostStepHeight);
    free(HostMontrMatrx);
    free(Data);
    free(HostFloatMatrx);


}

    //—— (15) Write the Complete Sufrace Height Data as SDF file

        //—— (15.a)    Defines the entry point for the console application.
        char SDF_ver_num[9]="bBCR-1.0";      // This step must be wriiten first
        char SDF_ManufacID[11]="THP-FORM  ";
        char SDF_CreateDate[13]="040520121442";
        char SDF_ModDate[13]="020620120931";
        unsigned int SDF_NumPoints=FrameWidth;
        unsigned int SDF_NumProfiles=HeightDivider*ProcessStep;//+RemainPixels
        double SDF_Xscale=3.5300E-006;
        double SDF_Yscale=3.5300E-006;
        double SDF_Zscale=1.00E-009;
        double SDF_Zresolution=-1;
        char SDF_Compression=0;//NULL;
        char SDF_DataType=3; // 3 for float 6 for long 5 for short
        char SDF_CheckSum=0;//NULL;

        //—— (15.b)        Set the laterial scale according to the magnification ↩
    scale
        if (LensX==5)
        {   SDF_Xscale=1.1905E-006;
            SDF_Yscale=1.1905E-006;
        }
        else if (LensX==10)
        {   SDF_Xscale=0.6098E-006;
            SDF_Yscale=0.6098E-006;

        }

        else if (LensX==50)
        {   SDF_Xscale=0.1185E-006;
            SDF_Yscale=0.1185E-006;
        }
        else
        {   SDF_Xscale=2.9412E-006; // The 2X magnification is the default one
            SDF_Yscale=2.9412E-006;

        }
```

```
        FILE * SurfaceFile;
        SurfaceFile = fopen (szFileName,"wb");

        //—— (15.c) write the information onto SDF file
        fwrite(SDF_ver_num,1,8,SurfaceFile);
        fwrite(SDF_ManufacID,1,10,SurfaceFile);
        fwrite(SDF_CreateDate,1,12,SurfaceFile);
        fwrite(SDF_ModDate,1,12,SurfaceFile);

        fwrite(&SDF_NumPoints,1,2,SurfaceFile);
        fwrite(&SDF_NumProfiles,1,2,SurfaceFile);

        fwrite(&SDF_Xscale,1,8,SurfaceFile);
        fwrite(&SDF_Yscale,1,8,SurfaceFile);
        fwrite(&SDF_Zscale,1,8,SurfaceFile);

        fwrite(&SDF_Zresolution,1,8,SurfaceFile);
        fwrite(&SDF_Compression,1,1,SurfaceFile);
        fwrite(&SDF_DataType,1,1,SurfaceFile);
        fwrite(&SDF_CheckSum,1,1,SurfaceFile);

        ElementWritten=fwrite(HostFinalStepHeight,sizeof(float),HeightDivider*↩
    FrameWidth*ProcessStep,SurfaceFile);//SizeHostStepHeight

        fclose (SurfaceFile);

    //—— (16) Free the remain arrays
        free(HostFinalStepHeight);
        free(Kv);
        cudaFree(Kv_d);
        free(BufData);
        free(CalMatHost);
        cudaFree(CalMat_d);
        cudaFree(OneFrameInt_d);
        free(OneFrameIntHost);


        return 1;
}



void WriteIntoText(Complex* HostMontrMatrx,long FrameWidth,long FrameHeight,long ↩
    lNumImage,char* MonitorFileName)
{
```

```
    pPixelFile = fopen (MonitorFileName,"w");
    if (pPixelFile!=NULL)
    {

        //172 number of rows
        //172 number of columns
    for(i=(0*lNumImage*FrameWidth)+(0*lNumImage);i<(0*lNumImage*FrameWidth)+(0*↩
    lNumImage+(lNumImage));i++)

    //for(i=(110*FrameWidth*107+200*107);i<(110*FrameWidth*107+200*107+(107-1));i++)
            {
            //sprintf (Hussbuffer, "%d\n",((int)FinalhostResultConv[i]));
            sprintf (Hussbuffer, "%f\n",((float)HostMontrMatrx[i].x));

            fputs (Hussbuffer,pPixelFile);
            }
    fclose (pPixelFile);
    }

}

void WriteFloatIntoText(float* HostFloatMatrx,long FrameWidth,long FrameHeight,long ↩
    lNumImage,char* MonitorFileName)
{

    pPixelFile = fopen (MonitorFileName,"w");
    if (pPixelFile!=NULL)
    {

        //172 number of rows
        //172 number of columns
    for(i=(0*lNumImage*FrameWidth)+(0*lNumImage);i<(0*lNumImage*FrameWidth)+(0*↩
    lNumImage+(lNumImage));i++)

    //for(i=(110*FrameWidth*107+200*107);i<(110*FrameWidth*107+200*107+(107-1));i++)
            {
            //sprintf (Hussbuffer, "%d\n",((int)FinalhostResultConv[i]));
            sprintf (Hussbuffer, "%f\n",((float)HostFloatMatrx[i]));

            fputs (Hussbuffer,pPixelFile);
            }
    fclose (pPixelFile);
    }

}
```

LISTING C.2: "Device_functions.cu"

Appendix D

GPU Cluster for Accelerated processing and Visualisation of Scientific Data

Paper based on the research described in this thesis, published in the proceedings of the Science and Information Conference 2014

GPU Cluster for Accelerated Processing and Visualisation of Scientific and Engineering Data

Matthew Newall
School of Computing and Engineering
University of Huddersfield
Huddersfield, UK HD1 3DH
Email:m.newall@hud.ac.uk

Violeta Holmes
School of Computing and Engineering
University of Huddersfield
Huddersfield, UK HD1 3DH
Email: v.holmes@hud.ac.uk

Paul Lunn
School of Digital Media Technology
Birmingham City University
Birmingham, UK
Email: Paul.Lunn@bcu.ac.uk

***Abstract*—The ability to process, visualise, and work with large volumes of data in a way that is fast, meaningful, and accurate is an essential part of many fields of scientific research today. The success of video game industry has resulted in ongoing developments in the complexity of Graphical Processing Units (GPU), as well as rapidly falling cost per core. Their characteristics make them excellently suited to any task exhibiting a high level of data parallelism. Recent development of GPU architectures is aimed at HPC systems and applications.**

In this paper we are presenting our experience in designing and deploying a small dedicated GPU based cluster for processing and visualising data generated by engineering and scientific application. This GPU cluster is helping our researchers to analyse complex data using visualisation, and to accelerate large data processing. We have shown that our GPU cluster solution can achieve five to ten times speed up compared to the CPU system. As a result of our work we can demonstrate that even a small GPU cluster can benefit Higher Education institutions.

Keywords*—*GPU, CUDA, GPU Cluster, Visualisation

I. Introduction

Conventional High-Performance Computing (HPC) resources, used for processing large number data or tasks, rely on parallel/serial processing using cluster computing facilities from Commodity Off The Shelf (COTS) hardware, dedicated HPC servers and recently GPU accelerated systems. GPU architectures are increasingly used to accelerate scientific application processing. In the last few years, Tesla GPUs and CUDA have had a great impact on the HPC research.

Based on the latest trend of GPU clusters in the top500 list, it is evident that GPU computing can provide supercomputing power not only for institutions with large budgets but also affordable HPC power to the universities and research institutions.

Graphics rendering is fundamentally a data parallel problem. Typically rendering is a single process thread which executes for each pixel to be rendered[1]. As each pixel can be processed independently from the rest, parallel processing was a logical outcome. This has lead to the introduction of dedicated graphics processors which contain multiple identical processors (modern cards now have thousands) which perform the same process on a constant stream of data. The data-parallel architecture of GPUs makes them ideally suited for many scientific computing problems and they are finding increased use in this area[2] [3]. The use of GPUs in scientific programming also has the added benefit that advances in GPU technology are driven by the lucrative video games industry, ensuring that processing power increases every year by much more than increases seen in CPUs and other hardware[4], [5], [6].

Using GPUs for general purpose programming has in the past been a complicated and time consuming task, requiring extensive knowledge of the specific hardware being used. To aid this effort, programming models and platforms have been developed which make creating GPU code simpler and makes the resulting applications more portable. Currently the most prevalent of these platforms are Nvidia CUDA (Compute Unified Device Architecture) which enables the production of code to run on CUDA supported hardware[7], and OpenCL which supports a wide range of hardware allowing the use of heterogeneous systems consisting of CPUs GPUs DSPs and FPGAs[8].

To explore the integration of GPU based clusters within our university campus grid and its use in accelerating the processing and visualisation of data generated by the researchers at the University of Huddersfield UK, we have deployed a small GPU based cluster, named VEGA. The cluster is fully integrated into the University network which enables job submission to this cluster by researchers across the university. The system uses Nvidia TESLA M2050 accelerators. This is a model designed specifically for High Performance Computing and per card includes 3GB of GDDR RAM per 448 processor cores running at 1.15 GHz, and connected via an x16 PCIe interface[9]. An overview of the architecture of this card can be seen in Fig. 1. As CUDA is optimised for Nvidia hardware, this paper will focus on that platform.

As well as general purpose processing, GPUs in a HPC system can still be used for traditional rendering. An example of software which allows this is the VisIt Parallel Visualisation software. The software allows visualisation and graphical analysis of massive data sets in real time. When running in parallel the software will use GPUs available to each node to accelerate graphics rendering [11].

The remainder of the paper is organised as follows. Section II details our efforts in designing and deploying a GPU cluster for visualization and accelerated processing of scientific data. An overview of GPU programming models are outlined in section III. Detailed design and evaluation of two case studies are presented in Sections IV and V. Section VI contains the

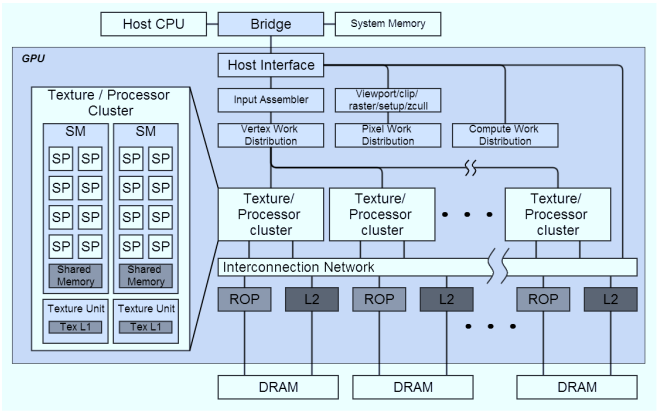


Fig. 1. Detail of the TESLA graphics and computing GPU architecture. Terminology: SM: streaming multiprocessor; SP: streaming processor; Tex: texture, ROP: raster operation processor[10].

summary of the work completed in this project.

II. Deployment of the GPU Cluster

There were a number of steps involved in deploying our GPU cluster:

- Windows Server 2008 r2 and the Microsoft HPC Pack were installed on the head node.
- The head node is attached to the University Active Directory (AD) network, as well as to a private gigabit switch.
- Compute nodes, which are powerful multicore systems with GPU's attached, are deployed by first connecting them to the Private network then booting to PXE. The Windows HPC software will then load an operating system and required middleware over the network.
- Finally, drivers for the TESLA cards are installed on the compute node directly.

Key steps when deploying and testing are:

- Selecting Network Topology
- Configuring NAT and DHCP to allow compute nodes to access outside network
- Creating a system image for compute nodes
- Finding bare metal machines to deploy as compute nodes
- Checking relevant diagnostic tests pass
- Checking CUDA functionality.

This can be seen in detail in Fig. 2

For our deployment the following underlying hardware was utilised:

- The Head Node - 1* Quad Core Intel Xeon E5630 2.53Ghz, 32GB RAM, Windows Server 2012 R2

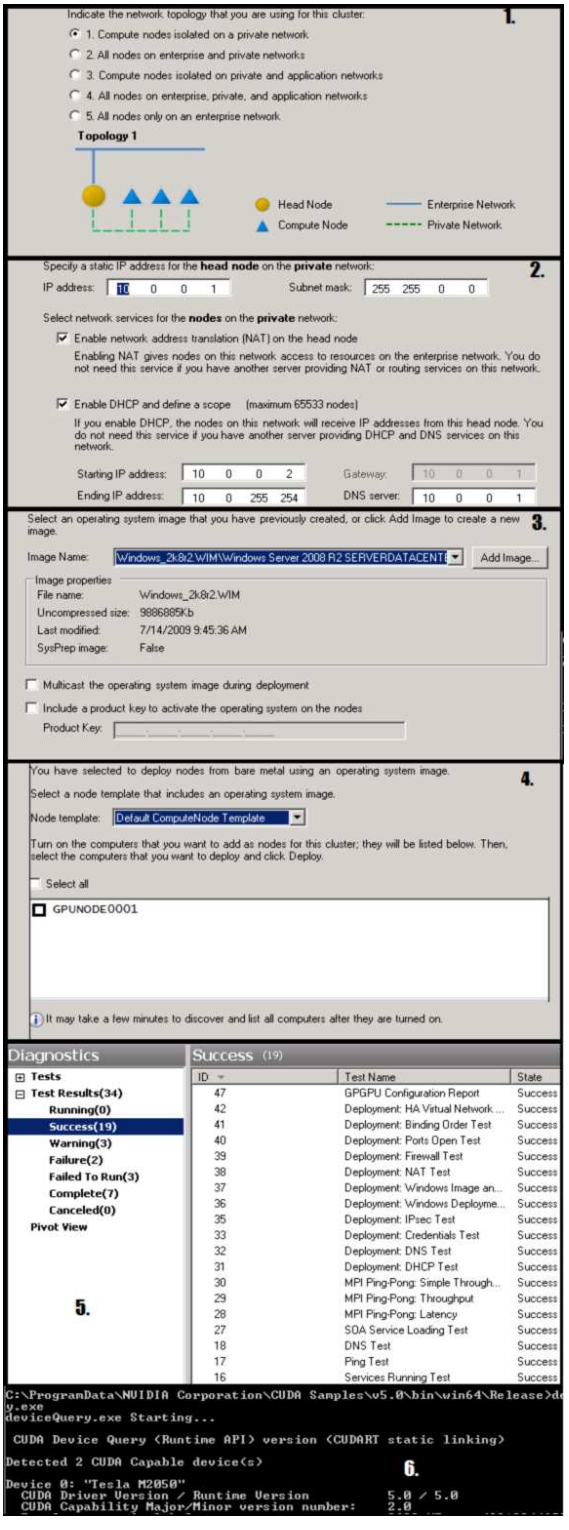


Fig. 2. Key steps when deploying VEGA.

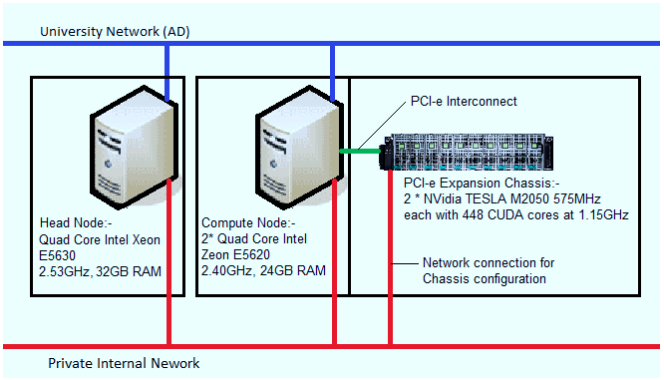


Fig. 3. Current VEGA hardware layout and network topology.

- The Compute Nodes - 2* Quad Core Intel Xeon E5620 2.40Ghz, 24GB RAM, 2 * NVidia TESLA M2050, Windows Server 2012 R2

The head node and compute node communicate via a Gigabit private switch. Job submission can be done directly on the head node itself, or from any machine on the University Active Directory network. The hardware and network layout for VEGA can be seen in Fig. 3.

This GPU cluster has formed a test-bed for exploring visualisation and acceleration of scientific data.

III. GPU PROGRAMMING MODELS

There currently exist two main models for developing software to run on GPUs: OpenCL and CUDA. Both models use the concept of kernels to contain parts of program structure which interact with compute devices, but differ in hardware support and scope.

OpenCL is a parallel programming standard, with notable contributors such as Apple, ARM, AMD, Samsung and NVidia. It allows programs to take advantage of a very diverse array of processing devices such as GPUs CPUs DSPs and FPGAs. The standard is open source and provides mechanisms for hardware vendors to add mechanisms for access to hardware specific features[8].

```
void serial_function (int n, float a, float *x,
    float *y)
{
  for (int i = 0; i<n; i++)
    y[i] = a*x[i] + y[i];
}
//perform on 1M elements
serial_function(4096*256, 2.0, x, y);
```

Listing 1. A standard C function

CUDA is developed by Nvidia for its GeForce, Quadro and Tesla processors. It is highly scalable and will run on an arbitrary number of processors without the need to recompile. This is required because of the vast and varying number of processor cores in modern GPUs[12]. As CUDA functions are called from standard C or C++ it makes GPU programming much more accessible than has previously been possible. An example of the required effort to produce CUDA compatible code can be seen in listings 1 and 2. The CUDA programming model was used in our case study to accelerate processing of radio astronomy data produced by SETI

```
void gpu_function (int n, float a, float *x, float *
    y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i<n) y[i] = a*x[i] + y[i];
}
//perform on 1M elements
gpu_function<<4096, 256>>(n, 2.0, x, y);
```

Listing 2. The same function as might be written for execution on a CUDA supported GPU[13]

IV. VISUALISATION WITH VISIT ON THE VEGA GPU CLUSTER

At the University, there are a number of resources which routinely generate large volumes of data, such as X-Ray tomography and electron microscopy, and so a system able to display this data in a useful way could be be beneficial. VisIt is an existing tool which is capable of displaying interactive 2D and 3D models of terascale simulation datasets and is able to use GPUs to accelerate rendering VisIts user interface can be seen in detail in Fig. 4.

To demonstrate the ease with which data can be visualised with this software, a simulation of surface energy of a cylinder over time was provided by a researcher from the University of Huddersfield Electron Microscopy and Materials Analysis group (EMMA). Initially, the data was generated and analysed in a large spreadsheet, which was impossible to see all at once. To allow useful visualisation, the simulation was taken into Matlab and a software tool was created to take the data from the spreadsheet and prepare it in a format compatible with VisIt. Key parts of the Matlab script and code preparing the data can be seen in Listings 3 and 4.

```
for time=0:maxtime
    q=(eV/((4*pi*D*time)^1.5))/aD;

    for y=0:360
        for x=0:100
            height=50-x;
            rad=deg(y+1)*(pi/180);

            val(x+1, y+1)=(1/40)+(q*exp(-((sqrt(
                height^2+(radius*sin(rad))^2+(radius
                *cos(rad)-ctop)^2))^2)/(4*D*time)));

        end
    end

    filename = ['data\' num2str(time) 'ns.dat'];
    dlmwrite(filename, val, ' ');

end
```

Listing 3. A sample of the Matlab script used to generate data

By running VisIt on the VEGA GPU cluster, rather than on a standard machine, real time visualisations can use larger or more complex models. In cases where the model is too large to view in real time, GPU acceleration reduces the rendering time needed to create video or images.

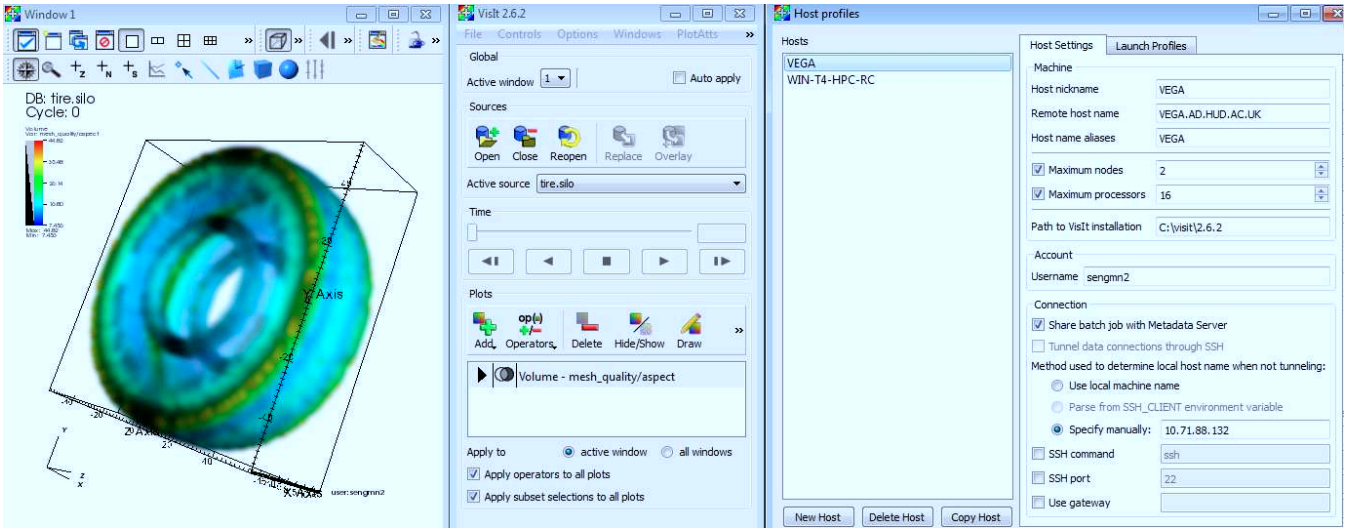


Fig. 4. VisIt user interface showing, from left to right: Viewer window, Plot selection and time controls, Host profile window, which allows VisIt to use a remote compute engine eg. on VEGA.

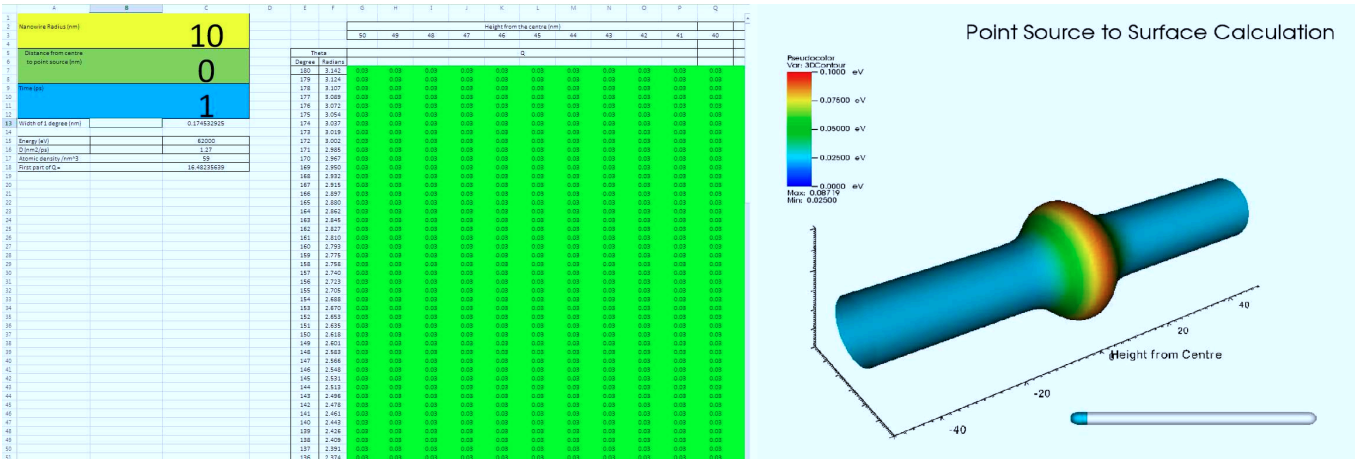


Fig. 5. Comparison of Visualisation Methods - Left is the original large spreadsheet and Right is the 3 dimensional representation generated by VisIt from the same data.

A. *Evaluation of results*

The results of this can be seen in Fig. 5. On the left is the data as it was being viewed previously, on the right is the data presented by VisIt. The generated model is fully 3D and can be manipulated in real-time.

V. Acceleration of Scientific and Engineering Data Processing

The Search for Extra-terrestrial Intelligence (SETI) explores radio astronomy data to discover evidence of technology based signals generated by civilizations outside of our own solar system. The data is explored with signal processing techniques or image based techniques, such as SETILive, where images of this data are observed by the public who try to detect patterns in this data. SonicSETI is a project where radio astronomy data produced by SETI[14] is converted into sound (or sonified) so that the public can listen to this data to detect anomalous sounds. Sonification is a process where data is transformed to sound[15].

A problem presents itself however as the processing of this data is time consuming in itself, taking almost 12 hours to process an 8GB set of data. The solution to this problem is to use GPU accelerated FFT libraries, such as the one provided by NVidia[16].

The original software, written in JAVA, reads data from a file then determines how many FFTs to perform, before processing the data and saving to a new file. The time taken to process each data set was deemed unacceptable, at around 12 hours per 8GB dataset. The first effort towards acceleration was to replace the FFT function with calls to a CUDA accelerated FFT function, CUFFT. In the JAVA code this was done via JCUDA, a java wrapper for various cuda functions, demonstrating that GPU acceleration is accessible from a variety of languages.

To further increase acceleration, it was deemed necessary to rewrite the software in C++, in order to have more complete access to various CUDA functions.

```
/*Build 3D Mesh*/

for(int iY=0;iY<NY;iY++)
{
  for (int iX=0;iX<NX;iX++)
  {
    for (int iZ=0;iZ<NZ;iZ++)
    {

      rad=iX*(pi/179);

      x[iY*NZ*NX+iX*NZ+iZ]=radius*cos(rad);
      y[iY*NZ*NX+iX*NZ+iZ]=iY-50;
      z[iY*NZ*NX+iX*NZ+iZ]=radius*sin(rad);
    }

  }
}

int dims[] = {NX, NY, NZ};
int ndims = 3;

float *coords[] = {(float*)x, (float*)y, (float*)z};

AddMesh(dbfile, "Cylinder", NULL, coords, dims,
    ndims, DB_FLOAT, DB_NONCOLLINEAR, optlist);

/*Add data to the mesh*/

dims[0] = NX; dims[1] = NY; dims[2] = NZ;

AddVariable(file, "3D", "Cylinder", data, dims,
ndims, NULL, 0, DB_FLOAT, DB_NODECENT, NULL);
```

Listing 4. A sample of the code preparing the data for use in VisIt

A. Evaluation of results

The graph in Fig. 6 compares the performance of the software, in Java, Java modified to use JCUDA, C++, and C++ with CUDA. Running regular FFT code compared to the GPU accelerated CUFFT library.

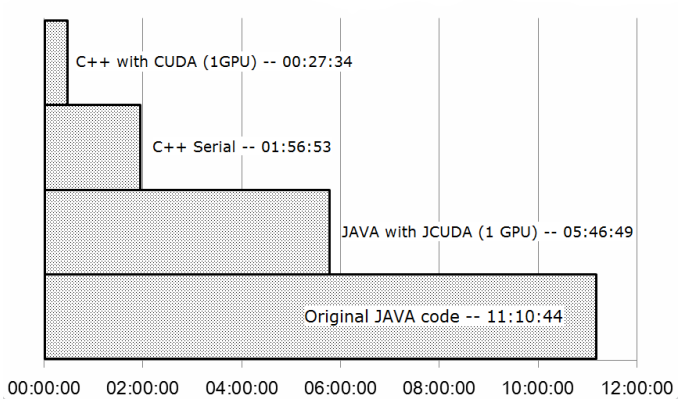


Fig. 6. Run time of each method

As there are 2 GPUs available the program was rewritten using MPI, allowing the data to be split between the two GPU's. Fig 7 shows the run time of the FFT part of each C++ method, this is the part which has been implemented on the GPU so gives the best indication of acceleration. While refactoring the code to take advantage of both GPUs the way in which data was copied to the GPU was changed to better

```
//Allocate device memory
mem_size=sizeof(cuDoubleComplex) * N;
cuDoubleComplex *d_cufftData;
checkCudaErrors(cudaMalloc((void**)&d_cufftData,
    mem_size));

//Copy data to device
checkCudaErrors(cudaMemcpy(d_cufftData, cufftData,
    mem_size, cudaMemcpyHostToDevice));

//Set FFT parameters and execute
cufftHandle plan;
checkCudaErrors(cufftPlan1d(&plan, N, CUFFT_Z2Z, 1))
    ;
printf("Starting FFT %d of %d \n", ffts, num_ffts);
checkCudaErrors(cufftExecZ2Z(plan, d_cufftData,
    d_cufftData ,CUFFT_FORWARD));

//Copy data back to host
checkCudaErrors(cudaMemcpy(cufftData, d_cufftData,
    mem_size,cudaMemcpyDeviceToHost));

//Destroy CUFFT complex
checkCudaErrors(cufftDestroy(plan));

cudaDeviceReset();
```

Listing 5. Using CUFFT to execute Forward FFT of Complex array 'fftData' on the GPU

utilise the memory on-board the device. Previously, enough data for a single fft was copied to the device before being executed and copied back. In the MPI version, enough data is sent to fill the GPU memory before executing a batch of ffts. This change reduced copy operations from 680 to 34.

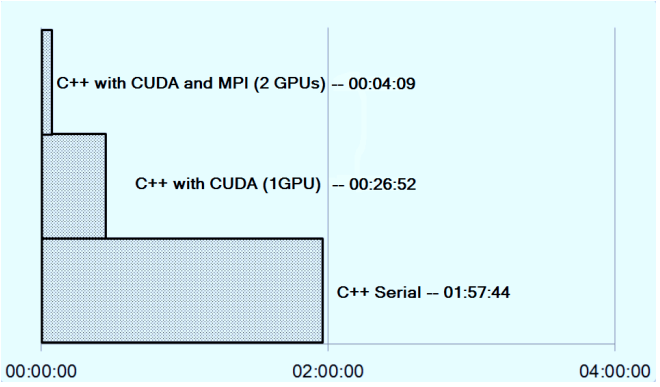


Fig. 7. Run time of the parallel/FFT part of each method

It was interesting to note that Java performance was poorer than C even without GPU acceleration, the result of slower disc access and the fact that JAVA uses big endian memory organization, so byte order has to be swapped before sending to GPU.

VI. Conclusion and Future Work

In this paper we presented a GPU cluster deployment and its utilisation in accelerating processing and visualisation of large volumes of data generated by engineering and scientific applications.

To demonstrate the GPU acceleration of data processing we have presented the results from two case studies illustrating the

improvements in processing time and 2D and 3D visualisation of data generated by an electron microscope, and Sonification of radio telescope data.

Some promising results are evident, and show that even a modest GPU cluster can bring large performance increases to suitable problems. Implementing MPI alongside CUDA allowed software to take advantage of the multiple GPUs available to VEGA to further improve performance. It is anticipated that further increases in performance could be achieved by fine tuning memory use and reducing required copy operations.

The GPU cluster deployment within the university campus grid is providing an easy access to this HPC resource for the researchers in engineering and scientific disciplines, and reducing the time required to process and visualise their scientific data. These results and experiences may be helpful as a methodology for other HE institutions who are considering GPU cluster support of visualisation and acceleration of data processing.

Further case studies for acceleration may be found within the university, such as Wavelength Scanning Interferometry code[17], and real-time processing of surface metrology data.

VII. Acknowledgement

We would like to acknowledge the use of QGG campus grid at the University of Huddersfield, UK and thank Graeme Greaves of the Electron Microscopy and Materials Analysis group, for providing the simulation used to demonstrate VisIt.

References

[1] J. Nickolls and W.J. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, March 2010.

[2] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *in Proceedings of the 12th international conference on Architectural*, pages 325–335, 2006.

[3] Zhe Fan, Feng Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 47–47, Nov 2004.

[4] Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it?, 2011.

[5] Mark Harris. Mapping computational concepts to gpus. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[6] Hiroyuki Takizawa and Hiroaki Kobayashi. Hierarchical parallel processing of large scale data clustering on a pc cluster with gpu co-processing. *J. Supercomput.*, 36(3):219–234, June 2006.

[7] Nvidia. What is cuda?, 2013.

[8] Nvidia. Opencl, 2013.

[9] Nvidia. Nvidia tesla m2050 specification, 2013.

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

[11] LLNL. About visit, 2013.

[12] Nvidia. Introduction to cuda, 2008.

[13] Nvidia. The cuda parallel computing platform, 2013.

[14] Nvidia. The allen telescope array, 2013.

[15] Terri Bonebright, Perry Cook, John Flowers, Nadine Miner, John Neuhoff, Robin Bargar, Stephen Barrass, Jonathan Berger, Grigori Evreinov, W Tecumseh Fitch, et al. Sonification report: Status of the field and research agenda.

[16] Nvidia. Nvidia cuda zone, 2013.

[17] Hussam Muhamedsalih, Xiang Jiang, and F. Gao. Accelerated surface measurement using wavelength scanning interferometer with compensation of environmental noise. *Procedia Engineering: 12th CIRP Conference on Computer Aided Tolerancing*, April 2012.

Appendix E

Delivering faster results through parallelisation and GPU acceleration

Book chapter on the research described in this thesis, published in the Springer Seried book: Studies in Computational Intelligence.

Delivering faster results through parallelisation and GPU acceleration

Matthew Newall, Violeta Holme, Colin Venters, and Paul Lunn

University of Huddersfield, High Performance Computing Research Group,
Queensgate, Huddersfield, HD1 3DH

Birmingham City University
Franchise Street, Birmingham B42 2SU

Abstract. The rate of scientific discovery depends on the speed at which accurate results and analysis can be obtained. The use of parallel co-processors such as Graphical Processing Units (GPUs) is becoming more and more important in meeting this demand as improvements in serial data processing speed become increasingly difficult to sustain. However, parallel data processing requires more complex programming compared to serial processing. Here we present our methods for parallelising two pieces of scientific software, leveraging multiple GPUs to achieve up to thirty times speed up.

Keywords: GPU, CUDA, GPU Cluster, Parallelisation

1 Introduction

Some of the strategic drivers for software development in computational science and engineering are outlined by EPSRC [1]. In particular, the focus "development of novel code, the development of new functionality for existing codes and the development and re-engineering of existing codes. Strategic drivers are: developing code for emerging hardware architectures; developing researchers with key software engineering skills and software sustainability" [2] is pertinent to code used in HPC. We consider this strategy one of the key drivers in the context of software sustainability [3], and an important challenge in the development of scientific and engineering software.

In our research we have focused on improving the efficiency and scalability of existing software. The examples here have been designed to address the challenges in processing large radio telescope data (SETI), and optical inferometry data used in surface measurements. The existing codes were re-engineered to support different GPU architecture, and enable scaling to larger GPU systems. In doing this we are addressing some 'software for the future' issues, taking into account the new hardware trends in GPUs deployment for HPC software.

Using GPUs in addition to more traditional High Performance Computing Resources to perform complex tasks or process large volumes of data has become increasingly common in supercomputing centres over the recent years. This trend

can be seen by looking at the Top500 (A ranking of the worlds top scoring supercomputing sites [4]) over the past few years.

3D graphics rendering typically executes a single instruction at a time for every pixel to be rendered, and calculations for a single pixel are independent from those for other pixels [5]. This has resulted in graphics processors becoming largely parallel devices with hundreds of stream cores on a single device, capable of performing an instruction on a constant stream of data at high speed. Driven by the lucrative video games industry, GPUs are not only outpacing CPUs in terms of the rate of technological improvement, but also have much lower cost and power demands per core [6]. Owing to their original intended use in graphics processing, a fundamentally data parallel problem, GPUs can provide a significant speed boost to tasks which exhibit high data parallelism. Many fields of scientific research use software that fits these criteria, and GPUs are seeing increased use in this area [7] [8] [9]. In response to this new GPU architetures have been designed specifically for general purpose processing, such as Nvidias TESLA series, shown in Fig. 1.

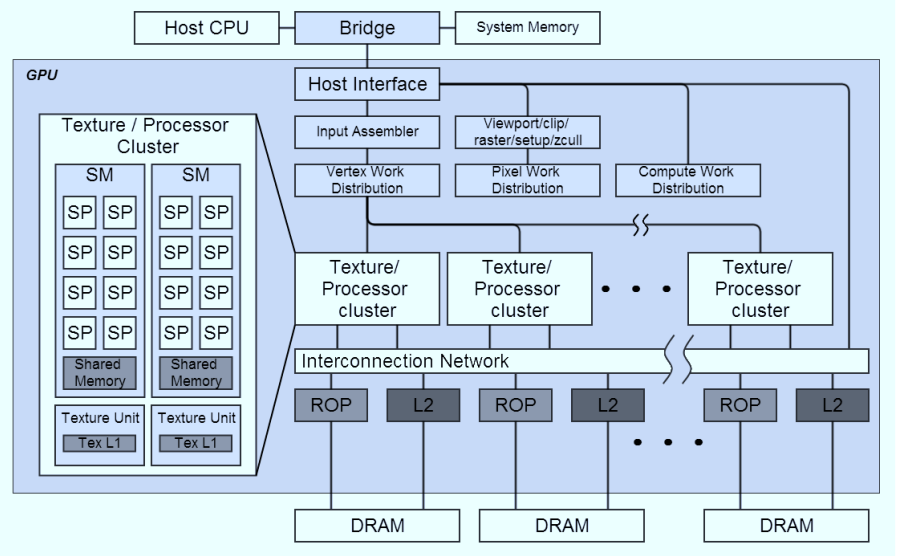


Fig. 1. Detail of the TESLA graphics and computing GPU architecture. Terminology: SM: streaming multiprocessor; SP: streaming processor; Tex: texture, ROP: raster operation processor[10].

To explore the potential for speed up in scientific applications, two existing software cases have been examined for sections appropriate for parallelisation. These examples were rewritten to allow them to execute on a GPU cluster, the deployment of which is detailed in [11].

2 GPU programming models

In order to make general purpose processing on GPUs more acessible, there have been numerous models and libraries developed. Currently, the most mature of these are OpenCL and CUDA. Both models use the concept of kernels to contain parts of program structure which interact with compute devices, but differ in hardware support and scope.

OpenCL is an open source parallel programming standard, with notable contributors such as Apple, ARM, AMD, Samsung and Nvidia. It allows programs to take advantage of a very diverse array of processing devices such as GPUs, CPUs, DSPs, and FPGAs. The standard provides mechanisms for hardware vendors to add mechanisms for access to hardware specific features, which serves to increase its flexibility[12].

CUDA is developed by Nvidia for its own series of GeForce, Quadro and Tesla processors. It is flexible in its scalability and will run on an arbitrary number of processors without the need to recompile. This relieves the programmer of the burden of requiring specific knowledge of the hardware, which today can have vastly different clock speeds, RAM and numbers of cores depending on the model [13]. As CUDA functions are called from standard C or C++ it makes GPU programming much more accessible than has previously been possible. An example of the required effort to produce CUDA compatible code can be seen in listings 1 and 2. The CUDA programming model was used in our case study to accelerate processing of radio astronomy data produced by SETI, as well as increasing the throughput of wavelength scanning interferometry data analysis.

3 Accelerated Processing of Radio Telescope data

The Search for Extra-terrestrial Intelligence (SETI) employs various methods in their attempt to discover evidence of technology based signals generated by civilizations outside of our own solar system. To this end vast amounts of radio telescope data must be analysed. The data is explored with signal processing techniques or image based techniques, such as SETILive, where images of this data are observed by the public who try to detect patterns in this data. Sonification is a process where data is transformed to sound[15]. SonicSETI is a project where radio astronomy data produced by SETI[16] is converted into sound (or sonified) so that the public can listen to this data to detect anomalous sounds.

However, processing this data is somewhat time consuming, taking almost 12 hours to process an 8GB set of data. The solution to this problem is to use GPU accelerated FFT libraries, such as the one provided by Nvidia[17].

The original software, written in JAVA, reads data from a file then determines how many FFTs to perform, before processing the data and saving to a new file. The time taken to process each data set was deemed unacceptable, at around 12 hours per 8GB dataset. The first effort towards acceleration was to replace the FFT function with calls to a CUDA accelerated FFT function, CUFFT. In the

Listing 1: A standard C function

```
void serial_function (int n, float a, float *x, float *y)
{
for (int i = 0; i<n; i++)
y[i] = a*x[i] + y[i];
}
//perform on 1M elements
serial_function(4096*256, 2.0, x, y);
```

Listing 2: The same function as might be written for execution on a CUDA supported GPU[14]

```
void gpu_function (int n, float a, float *x, float *y)
{
int i = blockIdx.x*blockDim.x + threadIdx.x;
if (i<n) y[i] = a*x[i] + y[i];
}
//perform on 1M elements
gpu_function<<4096, 256>>(n, 2.0, x, y);
```

JAVA code this was done via JCUDA, a java wrapper for various cuda functions, demonstrating that GPU acceleration is accessible from a variety of languages.

To further increase acceleration, it was deemed necessary to rewrite the software in C++, in order to have more complete access to various CUDA functions. Shown in Listing 3 is a section the final C++ CUDA code which shows the host to device memory copy and using CUFFT to perform FFT on the device.

Listing 3: Using CUFFT to execute Forward FFT of Complex array 'fftData' on the GPU

```
//Allocate device memory
mem_size=sizeof(cuDoubleComplex) * N;
cuDoubleComplex *d_cufftData;
checkCudaErrors(cudaMalloc((void**)&d_cufftData, mem_size));

//Copy data to device
checkCudaErrors(cudaMemcpy(d_cufftData, cufftData, mem_size,
cudaMemcpyHostToDevice));

//Set FFT parameters and execute
cufftHandle plan;
checkCudaErrors(cufftPlan1d(&plan, N, CUFFT_Z2Z, 1));
printf("Starting FFT %d of %d \n", ffts, num_ffts);
```

```
checkCudaErrors(cufftExecZ2Z(plan, d_cufftData ,d_cufftData
,CUFFT_FORWARD));

//Copy data back to host
checkCudaErrors(cudaMemcpy(cufftData, d_cufftData,
mem_size,cudaMemcpyDeviceToHost));

//Destroy CUFFT complex
checkCudaErrors(cufftDestroy(plan));

cudaDeviceReset();
```

3.1 Evaluation of results

The graph in Fig. 2 compares the performance of the software, in Java, Java modified to use JCUDA, C++, and C++ with CUDA. Running regular FFT code compared to the GPU accelerated CUFFT library.

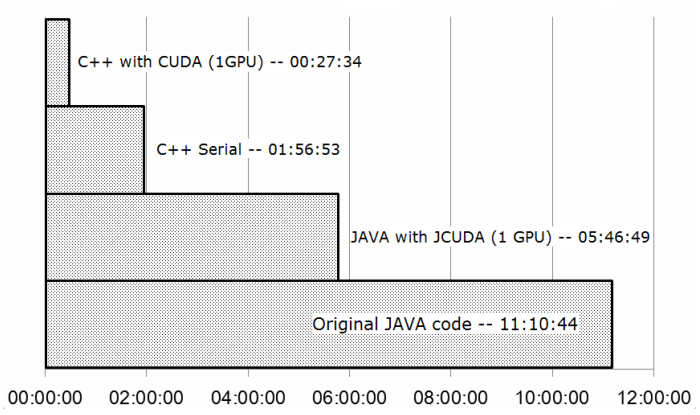


Fig. 2. Run time of each method

The program was rewritten using MPI, to allow it to take advantage of multiple GPUs. Fig 3 shows the run time of the FFT part of each C++ method; this is the part which has been implemented on the GPU so gives the best indication of acceleration. While restructuring the code to take advantage of both GPUs, the way in which data was copied to the GPU was changed to better utilise the memory on-board the device. Previously, enough data for a single FFT was copied to the device before being executed and copied back. In the MPI version, enough data is sent to fill the GPU memory before executing a batch of FFTs. This change reduced copy operations from 680 to 34.

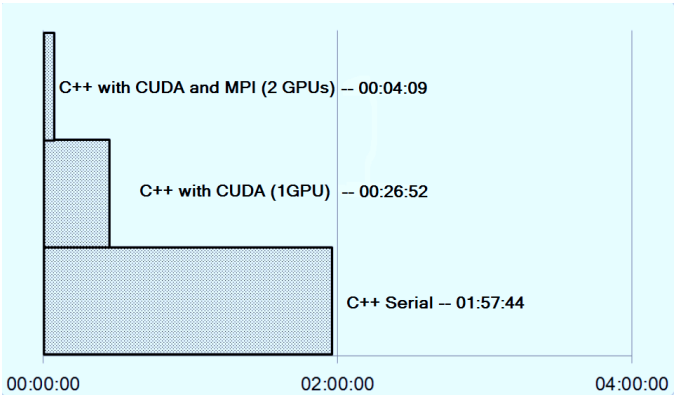


Fig. 3. Run time of the parallel/FFT part of each method

An interesting finding was that Java performance was poorer than C even without GPU acceleration. It was determined that this was the result of slower disk access and the fact that JAVA uses big endian memory organization, so byte order has to be swapped before sending to GPU.

As this approach uses MPI, it would be relatively simple to scale this to any number of GPUs, the only mitigating factor being that network overhead would increase for every additional node, eventually making the addition of more nodes impractical.

4 Accelerated surface measurement with environmental noise compensation

Optical interferometry is a widely used surface metrology technique. Wavelength scanning interferometry developments have been made that allow the process to be immune to environmental noise using phase compensation. However this compensation as well as data analysis processes limit performance, and hamper efforts to inspect this data as the measurement takes place. The paper [18] details a method which uses CUDA to accelerate this process with a single GPU. Using a Multi-GPU system such as VEGA [11] this process can be accelerated further to allow a greater number of frames to be processed without a significant increase in process time.

The original CUDA program loads a set of bitmap frames, and the noise cancellation is calibrated by loading a matrix which has been processed by MATLAB. After calibration the data is processed using Nvidias CUFFT GPU accelerated parallel FFT algorithm, and all data is saved to disk. By using an MPI based method to submit to 2 GPUs, two sets of frames can be processed in parallel effectively doubling throughput, or alternatively one set can be divided in two to reduce processing time and increase the efficiency of in-process analysis. As with the sonification study, the program is split into a master process and a

worker process - which must be able to run an arbitrary number of times, while the master co-ordinates. As there are 2 GPUs in our system we run 3 processes - one master and two workers. Fig. 4 shows the main function of the program, Fig. 5 describes the MPI program which allows the CUDA program to executed on multiple GPUs.

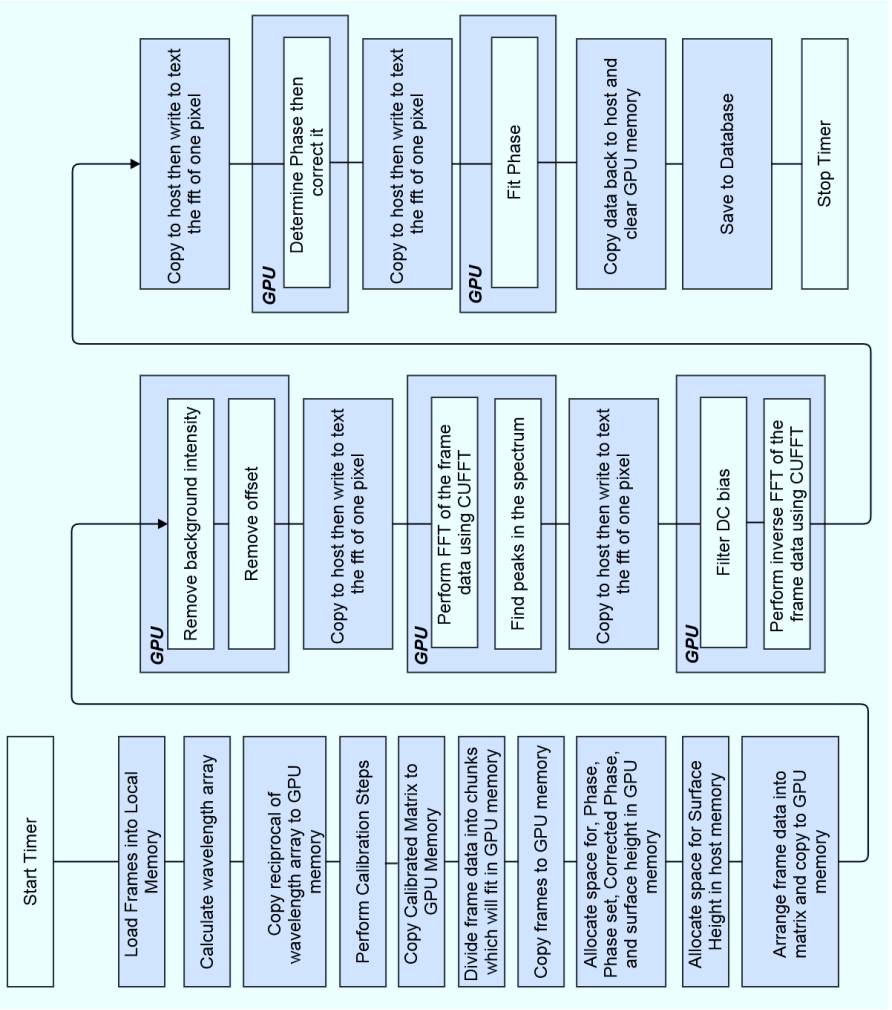


Fig. 4. Program flow for the original CUDA code

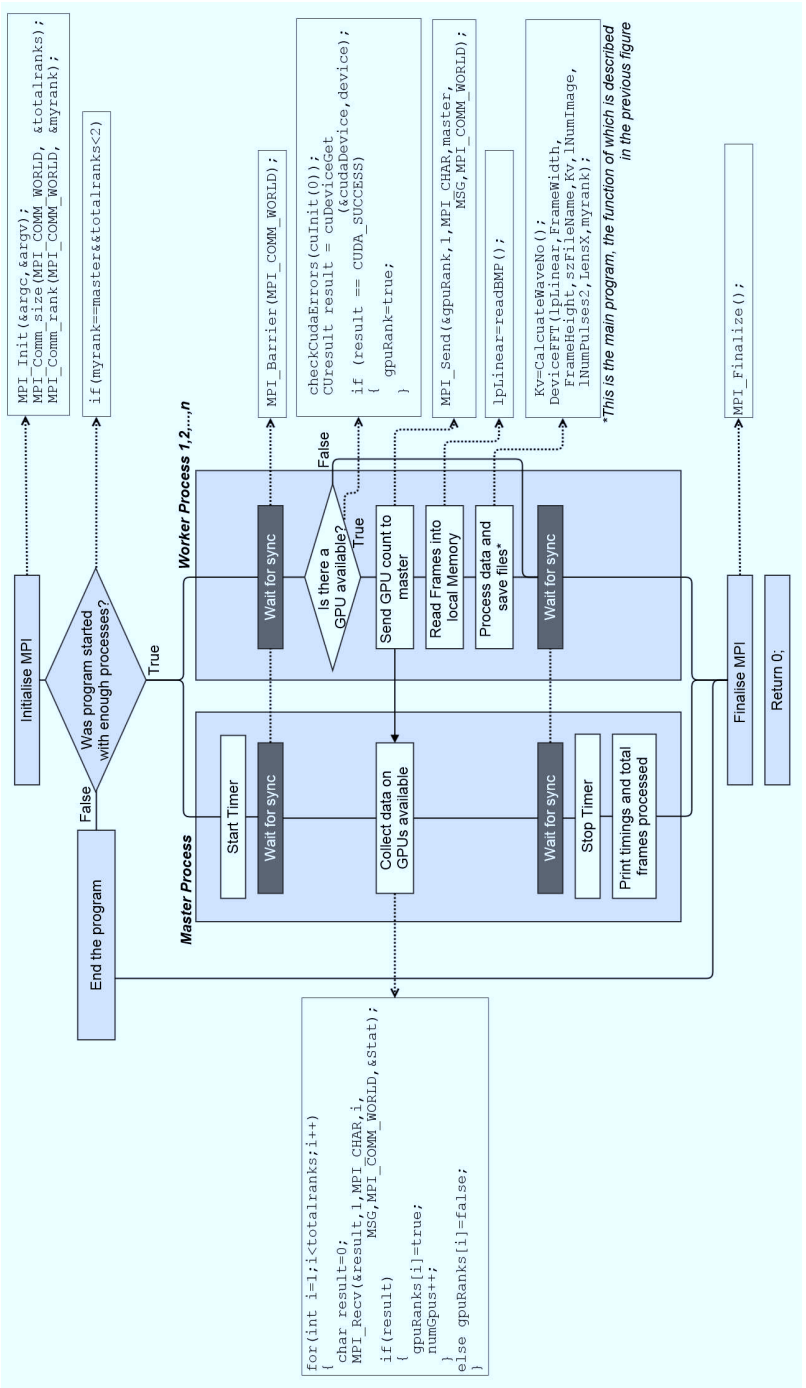


Fig. 5. Program flow for the MPI version using multiple GPUs

4.1 Evaluation of results

The graph in Fig. 6 compares total runtime for a single GPU versus two. When running on one GPU 256 frames are processed, when running on 2 GPUs 512 frames are processed. It can be seen that running on 2 GPUs adds an overhead of approximately 400 milliseconds, however Fig. 7 shows that running on 2 GPUs significantly reduces the per-frame processing time, being 1.9 times faster.

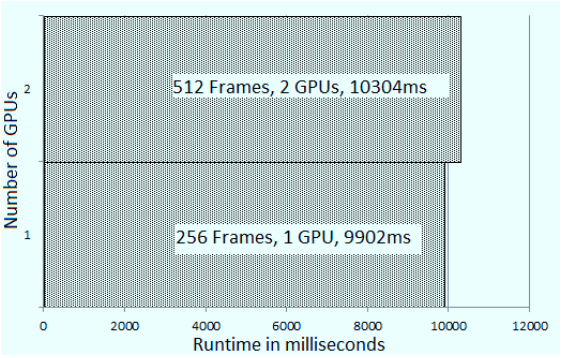


Fig. 6. Total run time

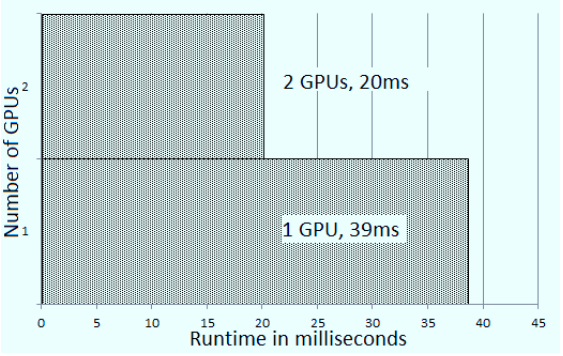


Fig. 7. Processing time per frame

While only 2 GPUs were used in this case, our system has a capacity for 16. It can be speculated, given the results already gained, what the potential speed-up would be if 16 GPUs were used. Given that a single GPU processes 256 frames in 9902 milliseconds, and the addition of a second GPU adds a 400 millisecond overhead, it is not unreasonable to suggest that 16 GPUs may be able to process 4096 frames in around 14 seconds (when including inevitable network overhead)

- an 11 fold increase in throughput over processing on a single GPU, and a 5 fold increase over 2 GPUs. As the software already utilises MPI, were the hardware available the software could run at this scale without modification. The law of diminishing returns will apply here however, as network overhead increases with the number of processes it would be come less beneficial to keep adding more GPUs. Using these assumptions we can predict system performance, as shown in Fig. 8, which illustrates that as we add more GPUs the relative benefit is less every time. This is where it is important to consider speed versus efficiency. Using the methods outlined in [19] we can identify that the efficiency of the software, based on these projections, peaks at 5 GPUs, after which the improvements tend towards zero. Hence, while speed up does continue to increase after this point, the resources required to do this might be best used for other tasks.

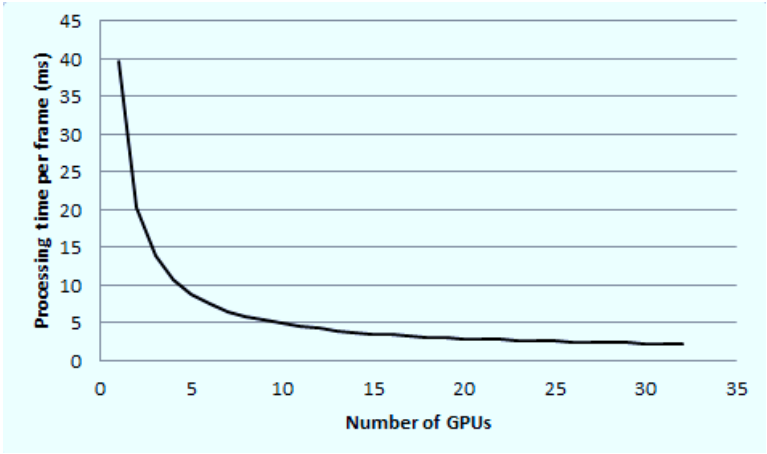


Fig. 8. Projected per-frame runtime on multiple GPUs

5 Conclusion and Further Work

In this chapter we have presented our work in parellelising existing codes for processing radio telescope and surface metrology data. Writing sustainable code for modern, multi-core, multiprocessor systems still presents a challenge. Existing programming environments for parallel and distributed platforms do not provide software developers with the tools necessary to test programs for the newest most powerful hardware.

Using the examples detailed here, and by utilising our own GPU cluster, we have shown that speed-up of up to 30 times is possible even on a modest GPU system. This will enable scientists and researchers to process complex problems and large volumes of data in near real-time.

To further explore the challenges of parallelisation we will investigate how these software examples scale onto much larger systems by running them on EMERALD, the UKs largest GPU cluster at Rutherford Appleton Laboratory [20].

In order to address the energy efficiency of our code, and software sustainability with respect to energy efficiency, we will build on our current research project funded by the innovate UK (technology strategy board) in Energy-Efficient computing [21] . Our focus will be on energy efficient data structures and algorithms for GPU technology. The resulting software will be evaluated and will be optimised under energy efficiency constraints creating more efficient software for affordable and sustainable high performance computing.

References

1. EPSRC, 2014.
2. EPSRC. Software for the future ii, 2014.
3. Lau L. Griffiths M. Holmes V. Ward R. Jay C. Dibsdale C. Venters, C. and J. Xu. The blind men and the elephant: Towards an empirical evaluation framework for software sustainability journal of open research software. In *Journal of Open Research Software*, 2014.
4. Top500. Titan - cray xk7 , opteron 6274 16c 2.200ghz, cray gemini interconnect, nvidia k20x, 2013.
5. J. Nickolls and W.J. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, March 2010.
6. Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it?, 2011.
7. David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *in Proceedings of the 12th international conference on Architectural*, pages 325–335, 2006.
8. Mark Harris. Mapping computational concepts to gpus. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.
9. Hiroyuki Takizawa and Hiroaki Kobayashi. Hierarchical parallel processing of large scale data clustering on a pc cluster with gpu co-processing. *J. Supercomput.*, 36(3):219–234, June 2006.
10. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March 2008.
11. Paul Lunn Colin Venters Matthew Newall, Violeta Holmes. Gpu cluster for accelerated processing and visualisation of scientific data.
12. Nvidia. Opencl, 2013.
13. Nvidia. Introduction to cuda, 2008.
14. Nvidia. The cuda parallel computing platform, 2013.
15. Terri Bonebright, Perry Cook, John Flowers, Nadine Miner, John Neuhoff, Robin Bargar, Stephen Barrass, Jonathan Berger, Grigori Evreinov, W Tecumseh Fitch, et al. Sonification report: Status of the field and research agenda. 1997.
16. Nvidia. The allen telescope array, 2013.
17. Nvidia. Nvidia cuda zone, 2013.
18. Hussam Muhamedsalih, Xiang Jiang, and F. Gao. Accelerated surface measurement using wavelength scanning interferometer with compensation of environmental noise. *Procedia Engineering: 12th CIRP Conference on Computer Aided Tolerancing*, April 2012.

19. D. L. Eager, J. Zahorjan, and E. D. Lozowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423, March 1989.
20. Oxford University. Emerald: e-infrastructure south gpu supercomputer, 2013.
21. Innovate UK, 2014.

References

Horizon 2020. Horizon 2020, 2013. URL http://ec.europa.eu/programmes/horizon2020/.

Athanasios Anthopoulos, Ian Grimstead, and Andrea Brancale. Gpu-accelerated molecular mechanics computations. *Journal of Computational Chemistry*, 34(26): 2249–2260, 2013. ISSN 1096-987X. doi: 10.1002/jcc.23384. URL http://dx.doi.org/10.1002/jcc.23384.

OpenMP Architecture Review Board. Openmp, 2013. URL http://openmp.org/wp/.

Stephan Brumme. Portable memory mapping c++ class, 2014. URL http://create.stephan-brumme.com/portable-memory-mapping/.

Miguel Cardenas-Montes, Miguel A. Vega-Rodriguez, Christopher Bonnett, Ignacio Sevilla-Noarbe, Rafael Ponce, Eusebio Sanchez Alvaro, and Juan Jose Rodriguez-Vazquez. Gpu-based shear–shear correlation calculation. *Computer Physics Communications*, 185(1):11 – 18, 2014. ISSN 0010-4655. doi: http://dx.doi.org/10.1016/j.cpc.2013.08.005. URL http://www.sciencedirect.com/science/article/pii/S001046551300266X.

Dell. Poweredge c410x pcie expansion chassis, 2014. URL http://www.dell.com/us/business/p/poweredge-c410x/pd.

e-infrastructure South. Emerald, 2013. URL http://www.einfrastructuresouth.ac.uk/cfi/emerald.

EMMA. Electron microscopy and materials analysis research group, 2014.

EPSRC. E-infrastructure roadmap, 2014a. URL http://www.epsrc.ac.uk/research/ourportfolio/themes/researchinfrastructure/subthemes/einfrastructure/strategy/roadmap/.

EPSRC. Software for the future ii, 2014b.

Jianbin Fang, A.L. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225, Sept 2011. doi: 10.1109/ICPP.2011.45.

'The MPI Forum'. Message passing interface forum, 2014. URL http://www.mpi-forum.org/.

James Fung and Steve Mann. Openvidia: Parallel gpu computer vision. In *Proceedings of the 13th Annual ACM International Conference on Multimedia*, MULTIMEDIA '05, pages 849–852, New York, NY, USA, 2005. ACM. ISBN 1-59593-044-2. doi: 10.1145/1101149.1101334. URL http://doi.acm.org/10.1145/1101149.1101334.

Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27, 2008. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2008.57.

I.S. Haque and V.S. Pande. Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 691–696, May 2010. doi: 10.1109/CCGRID.2010.84.

M. Harris. Mapping computational concepts to gpus. In *ACM SIGGRAPH 2005 Courses*, page 50. ACM, 2005.

Ansys INC. About ansys, 2014. URL http://ansys.com/About+ANSYS.

Cleve Moler Jack Dongarra, Jim Bunch and Gilbert Stewart. Liinpack, 2013. URL http://www.netlib.org/linpack/.

Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and opencl. *CoRR*, abs/1005.2581, 2010. URL http://arxiv.org/abs/1005.2581.

Gregory Kramer. *Auditory Display: Sonification, Audification, and Auditory Interfaces*. Perseus Publishing, 1993. ISBN 0201626047.

Ibad Kureshi. Establishing a university grid for hpc applications. September 2010. URL http://eprints.hud.ac.uk/10169/.

LBL. Warewulf: Scalable, modular, adaptable systems management, 2014. URL http://warewulf.lbl.gov/trac/wiki/About.

E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.31.

LLNL. Silo users guide, 2010.

LLNL. About visit, 2013. URL https://wci.llnl.gov/codes/visit/about.html.

D. Luebke and G. Humphreys. How gpus work. *Computer*, 40(2):96–100, Feb 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.59.

Risman Adnan Mattotorang. Ms-mpi with visual studio 2008, 2009. URL http://blogs.msdn.com/b/risman/archive/2009/01/04/ms-mpi-with-visual-studio-2008.aspx.

Micorsoft. Microsoft hpc pack 2008 and hpc pack 2008 r2 tool pack, 2014. URL http://www.microsoft.com/en-us/download/details.aspx?id=8433.

Microsoft. Using microsoft message passing interface, 2014a. URL http://technet.microsoft.com/en-us/library/4cb68e33-024b-4677-af36-28a1ebe9368f.

Microsoft. Microsoft mpi, 2014b. URL http://msdn.microsoft.com/en-us/library/bb524831%28v=vs.85%29.aspx.

Microsoft. Diy supercomputing: How to build a small windows hpc cluster, 2014c. URL http://social.technet.microsoft.com/wiki/contents/articles/2539.diy-supercomputing-how-to-build-a-small-windows-hpc-cluster.aspx.

MPICH. Mpich, 2014. URL http://www.mpich.org/.

Hussam Muhamedsalih, Xiang Jiang, and F. Gao. Accelerated surface measurement using wavelength scanning interferometer with compensation of environmental noise. *Procedia Engineering: 12th CIRP Conference on Computer Aided Tolerancing*, April 2012. URL http://eprints.hud.ac.uk/15357/.

Matthew Newall, Violeta Holmes, and Paul Lunn. Gpu cluster for accelerating processing and visualisation of scientific and engineering data, August 2014. URL http://eprints.hud.ac.uk/21907/.

Nvidia. Geforce 6 architecture, 2006. URL http://technet.microsoft.com/en-us/library/4cb68e33-024b-4677-af36-28a1ebe9368f.

Nvidia. Introduction to cuda, 2008. URL http://www.training.prace-ri.eu/uploads/txpracetmo/IntroductiontoCUDA.pdf.

Nvidia. The cuda parallel computing platform, 2013a. URL http://www.nvidia.com/object/cuda-parallel-computing-platform.html.

Nvidia. Cuda toolkit, 2013b. URL https://developer.nvidia.com/cuda-toolkit.

Nvidia. Download drivers, 2013c. URL http://www.nvidia.com/Download/index.aspx?lang=en-us.

Nvidia. Opencl, 2013d. URL https://developer.nvidia.com/opencl.

Nvidia. Nvidia tesla m2050 specification, 2013e. URL http://www.nvidia.com/docs/IO/43395/BD-05238-001v03.pdf.

Nvidia. Sonicseti website, 2014. URL https://developer.nvidia.com/nvidia-nsight-visual-studio-edition/.

ORNL. Oscar: Open source cluster application resources, 2005. URL http://www.csm.ornl.gov/oscar/.

ORNL. Parallel virtual machine, 2009. URL http://www.csm.ornl.gov/pvm/.

ORNL. Titan: Built for scince, 2011a.

ORNL. Ornl debuts titan supercomputer, 2011b.

Oxford University. Emerald: e-infrastructure south gpu supercomputer, 2013. URL http://people.maths.ox.ac.uk/gilesm/emerald.html.

P.Glaskowsky. Nvidia's fermi -the first complete gpu architecture. 2009. URL http://www.nvidia.co.uk/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf.

SETI. The allen telescope array, 2013a. URL `http://www.seti.org/ata`.

SETI. Sonicseti website, 2013b. URL `http://www.sonicseti.com/`.

STFC. The dl_poly molecular simulation package, 2014. URL `http://www.stfc.ac.uk/SCD/research/app/ccg/software/DL_POLY/44516.aspx`.

David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *in Proceedings of the 12th international conference on Architectural*, pages 325–335, 2006.

Top500. The top500 supercomputers, 2013a. URL `http://www.top500.org/`.

Top500. Titan - cray xk7 , opteron 6274 16c 2.200ghz, cray gemini interconnect, nvidia k20x, 2013b. URL `http://www.top500.org/system/177975#.U4-c5PldXW4`.

C. Venters, L. Lau, M. Griffiths, V. Holmes, R. Ward, C. Jay, C. Dibsdale, and J. Xu. The blind men and the elephant: Towards an empirical evaluation framework for software sustainability' journal of open research software. In *Journal of Open Research Software*, 2014. ISBN 2049-9647.

Nathan Whitehead and Alex Fit-florea. Precision and performance: Floating point and ieee 754 compliance for nvidia gpus, 2011.