Delivering Faster Results Through Parallelisation and GPU Acceleration

Matthew Newall, Violeta Holmes, Colin Venters and Paul Lunn

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

M. Newall · V. Holmes (✉) · C. Venters
High Performance Computing Research Group, University of Huddersfield, Queensgate, Huddersfield HD1 3DH, UK
e-mail: v.holmes@hud.ac.uk

P. Lunn
Birmingham City University, Franchise Street, Birmingham B42 2SU, UK

K. Arai et al. (eds.), *Intelligent Systems in Science and Information 2014*,
Studies in Computational Intelligence 591, DOI 10.1007/978-3-319-14654-6_19

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 [7–9]. In response to this new GPU architectures have been designed specifically for general purpose processing, such as Nvidias TESLA series, shown in Fig. 1.

To explore the potential for speed up in scientific applications, two existing software cases have been examined for sections appropriate for parallelisation.

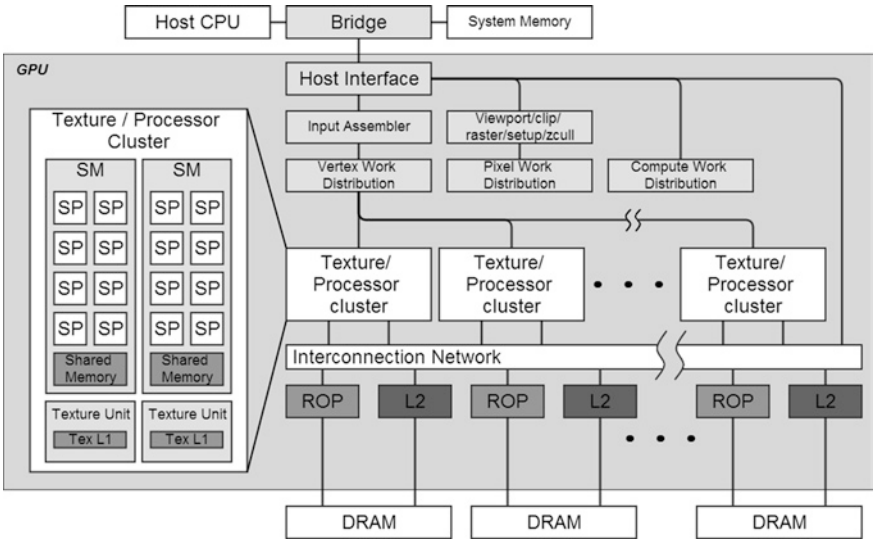

Fig. 1 Detail of the TESLA graphics and computing GPU architecture. Terminology: *SM* streaming multiprocessor; *SP* streaming processor; *Tex* texture, *ROP* raster operation processor [10]

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 accessible, 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 h to process an 8 GB 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 h per 8 GB 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.

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);
```

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.

The program was rewritten using MPI, to allow it to take advantage of multiple GPUs. Figure 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.

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.

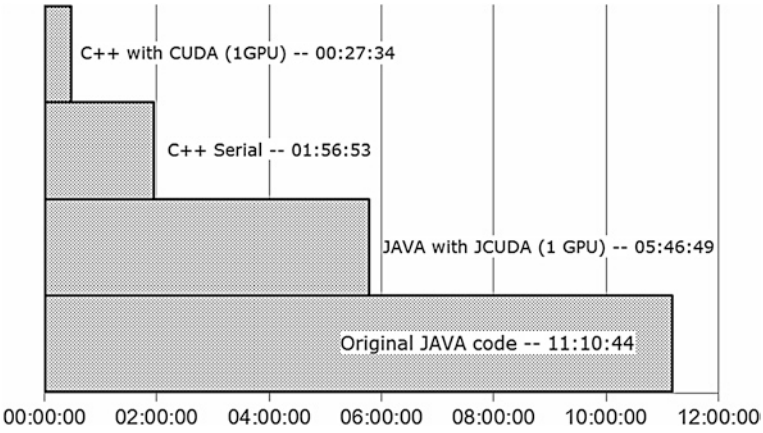

Fig. 2 Run time of each method

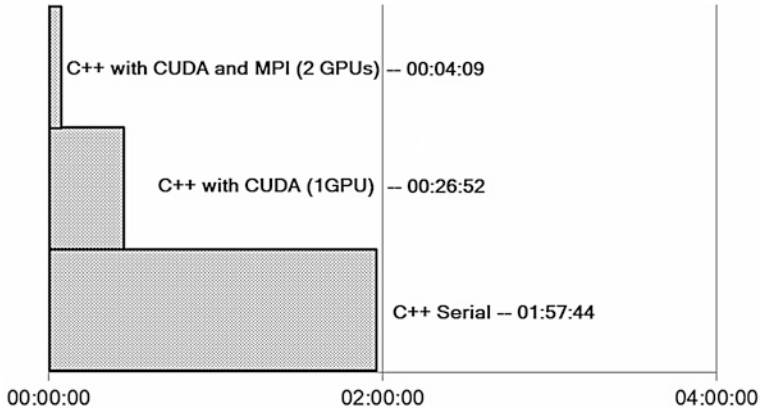

Fig. 3 Run time of the parallel/FFT part of each method

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. Figure 4 shows the main function of the program, Fig. 5 describes the MPI program which allows the CUDA program to executed on multiple GPUs.

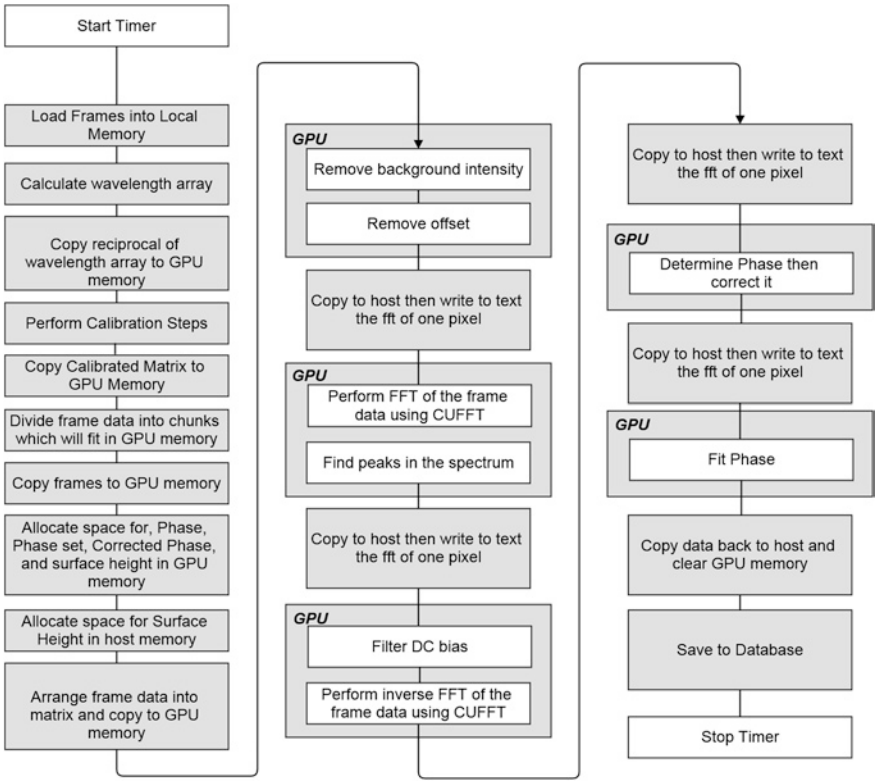

Fig. 4 Program flow for the original CUDA code

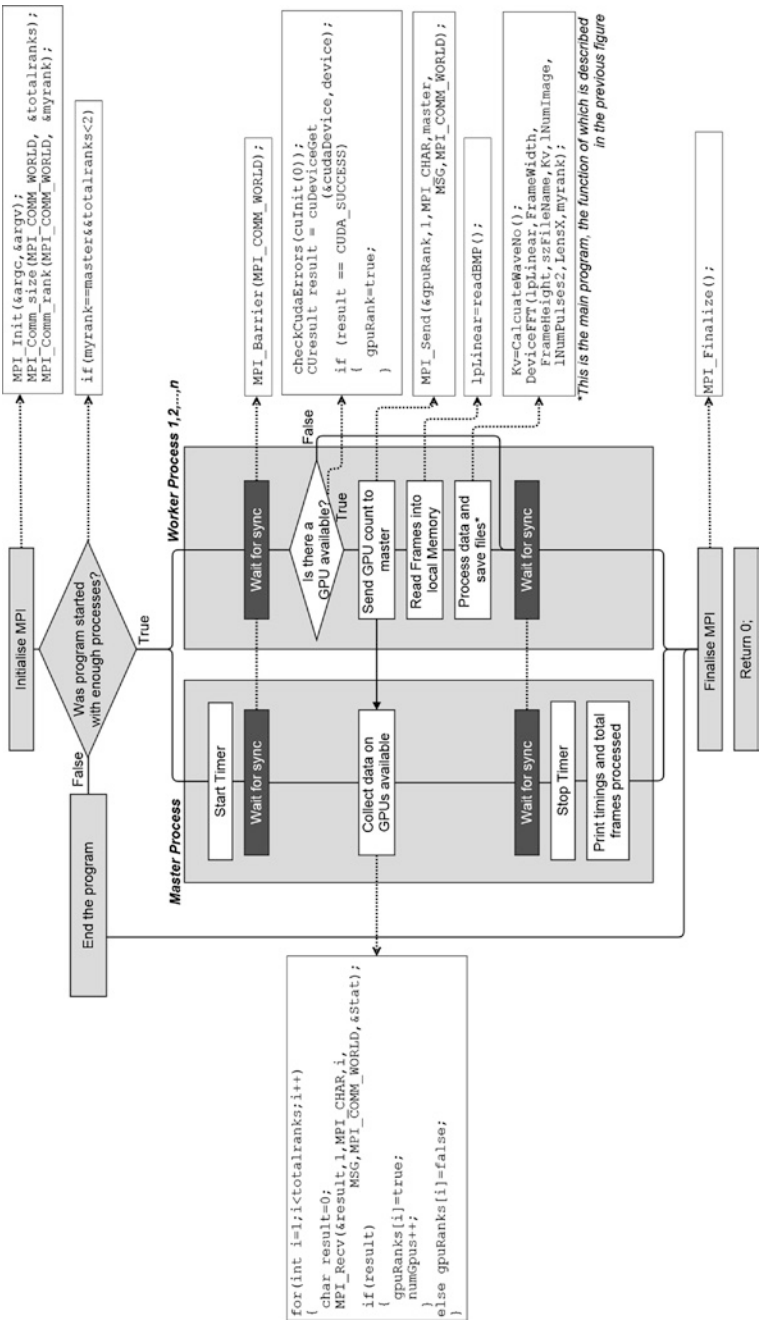

Fig. 5 Program flow for the MPI version using multiple GPUs

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 ms, however Fig. 7 shows that running on 2 GPUs significantly reduces the per-frame processing time, being 1.9 times faster.

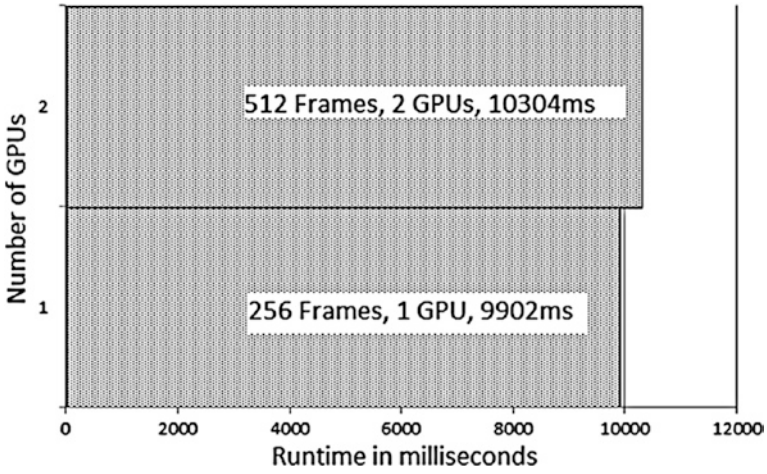

Fig. 6 Total run time

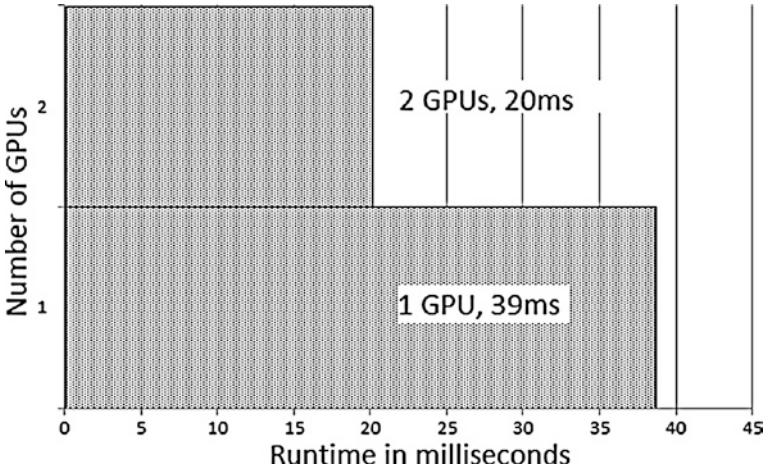

Fig. 7 Processing time per frame

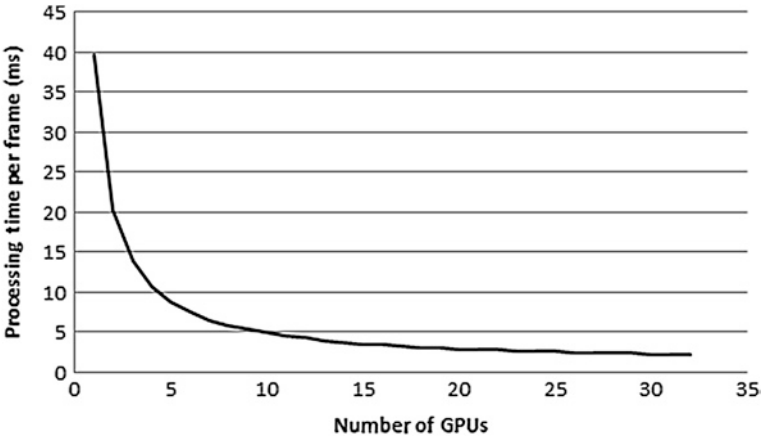

Fig. 8 Projected per-frame runtime on multiple GPUs

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 9,902 ms, and the addition of a second GPU adds a 400 ms overhead, it is not unreasonable to suggest that 16 GPUs may be able to process 4,096 frames in around 14 s (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.

5 Conclusion and Further Work

In this chapter we have presented our work in parallelising 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., Xu, J.: The blind men and the elephant: towards an empirical evaluation framework for software sustainability journal of open research software. J. Open Res. Softw. **2**, e8 (2014)
4. Top500: Titan - cray xk7, opteron 6274 16c 2.200ghz, cray gemini interconnect, nvidia k20x (2013)
5. Nickolls, J., Dally, W.J.: The GPU computing era. Micro IEEE **30**(2), 56–69 (2010)
6. McKenney, P.E.: Is parallel programming hard, and, if so, what can you do about it? (2011)
7. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program GPUs for general-purpose uses. In: Proceedings of the 12th International Conference on Architectural, pp 325–335 (2006)
8. Harris, M.: Mapping computational concepts to GPUs. In: ACM SIGGRAPH 2005 Courses, SIGGRAPH '05, ACM, NY, USA (2005)
9. Takizawa, H., Kobayashi, H.: Hierarchical parallel processing of large scale data clustering on a pc cluster with GPU co-processing. J. Supercomput. **36**(3), 219–234 (2006)
10. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: Nvidia tesla: a unified graphics and computing architecture. Micro IEEE **28**(2), 39–55 (2008)
11. Newall, M., Holmes, V., Venters, C., Lunn, P.: GPU cluster for accelerated processing and visualisation of scientific data (2014)
12. Nvidia: Opencl (2013)
13. Nvidia: Introduction to cuda (2008)
14. Nvidia: The cuda parallel computing platform (2013)
15. Bonebright, T., Cook, P., Flowers, J., Miner, N., Neuhoff, J., Bargar, R., Barrass, S., Berger, J., Evreinov, G., Tecumseh Fitch, W., 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. Muhamedsalih, H., Jiang, X., Gao, F.: Accelerated surface measurement using wavelength scanning interferometer with compensation of environmental noise. In: Procedia Engineering: 12th CIRP Conference on Computer Aided Tolerancing, Apr 2012
19. Eager, D.L., Zahorjan, J., Lozowska, E.D.: Speedup versus efficiency in parallel systems. IEEE Trans. Comput. **38**(3), 408–423 (1989)
20. Oxford University: Emerald: e-infrastructure south GPU supercomputer (2013)
21. Innovate UK (2014)