From CPU to GP-GPU: Challenges and Insights in GPU-based Environmental Simulations

Jools Chadwick
Lancaster University
Infolab21, Bailrigg
Lancaster, LA1 4YW, UK
+44 (0)1524 510340
j.chadwick2@lancaster.ac.uk

Francois Taiani
Lancaster University
Infolab21, Bailrigg
Lancaster, LA1 4YW, UK
+44 (0)1524 510338
f.taiani@lancaster.ac.uk

Jonathan Beecham
Cefas
Pakefield Road, Lowestoft
Suffolk, NR33 0HT, UK
+44 (0)1502 524541
jonathan.beecham@cefas.co.uk

ABSTRACT

From economics to natural sciences, many disciplines use complex models and simulations to better understand the world, but the unknown parameters of these models can be difficult to find. Looking to optimise the search for such parameters, many turn to the high parallelism afforded by general purpose Graphical Processing Unit (GP-GPU) programming. This paper discusses the challenges faced and lessons learned when porting such a marine ecology simulation from a pure-CPU implementation to make use of GPU technology. While this is a specific implementation, many of the problems we encountered apply generally to GPU-based simulations. They therefore hint at the potential for reusable solutions to GPU-based environmental simulations, and pave the way for a generic GPU-middleware for natural sciences.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; I.6.7 [**Simulation And Modelling**]: Simulation Support Systems

General Terms

Performance, Design, Experimentation

Keywords

Markov Chain Monte Carlo, Differential Evolution, Graphical Processing Unit, Optimisation, NVIDIA, CUDA, GP-GPU

1. INTRODUCTION

While Graphical Processing Units (GPUs) were once specifically tailored to merely render graphics, due in part to the introduction of on-GPU physics calculations, modern GPUs are becoming increasingly suited to massively-parallel General-Purpose (GP) programming. These methods are adopted in the natural sciences, economics and any other discipline where duplicate computational tasks are carried out on large quantities of data.

This growing interest in GP-GPU has been matched by rapid advances in the capabilities of graphic cards. NVIDIA's Fermi architecture for instance introduced several technical advances that aid GP-GPU programming. GPUs were given up to 512 cores, along with greater support for double precision calculations (8 times faster than pre-Fermi). NVIDIA introduced global atomic operations to aid synchronisation. There was also the addition of a configurable 64KB shared memory and cache assigned to each streaming multiprocessor (SM). Each SM is consists of 32 of the GPU's total number of cores [6].

There have been many examples of successfully optimised GP-GPU algorithms. Zhu designed a Differential Evolutionary (DE) pattern search and saw a speed up of 4 to 7 times for the DE component and drastic speed ups of over 300 times for the pattern search component [8]. Bakhtiari et al used GPUs to optimise Monte Carlo based radiation dose calculations and found that very often the calculations were 150 times faster [1]. The most closely related work is probably that of Zhu & Li, who devloped a GPU-accelerated Differential Evolutionary Markov Chain Monte Carlo (DEMCMC) method for multi-objective optimization over continuous space, seeing a speedup of 100 [10].

In this paper we present another DEMCMC implementation which aims to discover the best parameter set for a model to predict the evolution of herring populations. Rather than implementing a new program on both CPU and GPU, the proposed project was to port a specific, pre-existing CPU implementation, to discover the challenges that arise in doing this. The program was authored independently of this project by members of CEFAS and had already been successfully optimisation through multithreading to take advantage of multicore CPUs.

Challenges explored include: the generation of random numbers; the structuring of data, both on and off the GPU; where the different memory types of the GPU architecture can be taken advantage of, as well as where they may not be enough; how modern parallel computation techniques can be used to optimise inherently serial processes. These challenges apply generally to GPU-based simulations. They therefore hint at the potential for reusable solutions to GPU-based environmental simulations, and more generally argue for a generic GPU-middleware for natural sciences.

The rest of this paper is structured as follows: The paper firstly discusses the motivation for the pre-existing system and then talks in detail about its underlying theory (Section 2). We then provide an introduction to GP-GPU programming and GPU architectures (Section 3), before moving on to discuss the challenges we faced when porting the simulation from a CPU implementation to a GP-GPU environment (Section 4). Finally we present the results (Section 5), discuss their implications (Section 6) and conclude (Section 7).

2. DIFFERENTIAL EVOLUTIONARY MARKOV CHAIN MONTE CARLO (DEMCMC)

Fisheries scientists typically have to make critical decisions about how much fishing will be allowed or whether to close a fishery altogether on data that has a degree of uncertainty associated with it. The first thing to be done is typically to estimate the size of a population of a species of fish (in this instance, herring) in an area or stock based on information about catches and from surveys. They are often faced with a choice of models based on how they think uncertainty might be distributed between the parameters of the model. We use a machine learning method using a Bayesian DEMCMC method to choose between models.

This section will break down each element of DEMCMC algorithm and describe in part how they relate to the given system.

2.1 Markov Chain Monte Carlo

Monte Carlo uses random numbers to estimate answers to intractable problems. A simple example of an intractable problem is finding out the mean height of the people in a country. This would require collecting and summing millions of heights and dividing by the population size. It would be a much better idea to take a large enough random sample of heights and divide by the sample size to give a fair estimate.

Rather than looking an average height, we are looking for the best parameter set in a multidimensional parameter space. Starting at random places, individual parameter sets act as walkers, carrying out a Markov Chain taking small steps through the parameter space. To prevent these walks being entirely random a differential evolutionary technique is used to guide the walkers.

2.2 Differential Evolution

The “population”, upon which the DE operation occurs, is in no way related to a “population of herring”, but it instead refers to a group of parameter-sets (referred to as individuals). In our system the population size was 256, with each individual having a number of genes made up of a number of gene parts. With each generation potential individuals are generated. If they are an improvement on the original they more likely to be accepted and so replace the original in the next generation. This describes how the population evolves, but not how the potential individuals are generated. The generation process involves a differential operation.

For purpose of visualisation it is easier to assume that an individual contains only two parameters, giving it a position in a two dimensional parameter space, similar to a battleship board-game. Successful individuals have a tendency to be clustered together in similar patterns within the parameter space, like the squares that battleships occupies. This means that when looking for successful individuals, taking

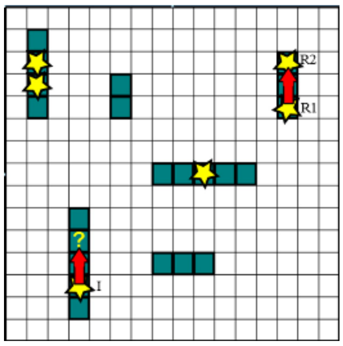

Figure 1: Using Vectors in a Parameter Space

the vector between two of individuals (such as stars $R1$ and $R2$) and applying it another accepted individual (such as I in Figure 1), it is more likely to land on a successful point in the parameter space, possibly pushing the individual deeper into an area of good parameter sets.

This is applied to each and every individual in the population, where R1 and R2 are two different individuals randomly selected from the entire population. As mentioned by Zhu, will cause a slower convergence than simply selecting from the best n individuals, but it will help prevent a premature convergence [10]. The example shows a fortunate selection of R1 and R2. It is entirely possible that the selected individuals would be at opposite, unrelated ends of the parameter space. The success or acceptance of the new individuals is decided by a calculation of their fitness.

2.3 Fitness and Acceptance

Fitness assessment involves calculating the probability that the model’s given parameter-set will produce an output similar to observed historical data. This is probably the most time consuming section of the algorithm. It is carried out on the parameter set, both as it was before mutation and as it is following the differential operation. In the provided system, the two parameter sets are expanded before being evaluated. This sometimes involves taking the natural exponent of the parameter set values.

Each of the expanded parameter sets is then evaluated to give likelihood value for reproducing real-world data. Whether the new parameter set (given after the differential operation) is accepted has a random element, but it is more likely to be accepted if it has a greater likelihood value than the original parameter set. If accepted this parameter set replaces the old one in the next generation/iteration.

With multiple Populations the likelihood values of all the parameter sets are summed, so that the best fitting Population can be determined.

3. NVIDIA CUDA & GPU ARCHITECTURE

The system was optimised through the use of the CUDA framework, which was developed by NVIDIA to aid GP-GPU science. Prior to these frameworks people were programming on GPUs but by mentally interpreting their program into graphical relate commands e.g. RGB values as a generic number space. CUDA is accessible through multiple programming languages including as Fortran and C; the latter being the one used in this work.

While modern CPUs have up to four cores, GPUs can have thousands. These GPU cores can handle many more thousands of threads executing the same set of instructions (a kernel) in parallel. Cores are grouped into SMs, which simultaneously execute small groups of threads (warps) for a single instruction. For this reason GP-GPU programming is often referred to as a Single Instruction Multiple Data (SIMD) paradigm. Threads belong to larger groups known as blocks. While a block is merely a software grouping, the execution of a block will never be shared between SMs, though one SM may handle multiple blocks.

The GPU architecture has many types of memory. Global memory is slowest to access, but is accessible by all threads. Local memory is physically the same as Global memory and is potentially as slow to access, but it is coalesced for optimised simultaneous access (see section 4.3). Simple variables are held in the much faster (x100) on-chip registers, whereas arrays are held in local memory; both accessible only by the thread declaring them. There is a small amount (64kb per SM) of on-chip memory, which is used as both an L1/L2 cache and shared memory. For each kernel call the division of this on-chip memory can be configured to give 48kb for one and 16kb for the other. Shared memory is accessible by all threads in a given block and is potentially as fast to access as the on-chip registers. There is also 64kb of constant memory which is globally accessible, but with the benefit of a dedicated 8kb cache per SM.

4. ADAPTING TO THE GPU

Many developers are put off GPGPU programming as it requires learning a new language. While different languages may only require learning a new syntax, GPGPU programming also requires learning a new architecture and how said architecture can be used efficiently. It is important to consider how data is structured in the program to aid both its transferal to the GPU and how the GPU threads access it once it is there.

Regardless of whether code is being parallelised through CPU multithreading or GPGPU programming, the existing code needs to be heavily altered, with special consideration given areas that are inherently serial. This underlines the importance of good software engineering with maintainability in mind. Many organisations spend the majority of their budgets on software maintenance [2], partly because this requires a strong understanding of the existing code as well as how new features will be adapted. GPGPU programming is no exception.

4.1 The Original CPU Bayesian DEMCMC

The original system, developed primarily by Dr. Jonathan Beecham of CEFAS, was authored in C++ it makes strong use of object-oriented (OO) concepts. Within an instance of an Individual object, a parameter-set is stored as large arrays of doubles, along with relevant methods. Population objects contain references to hundreds of unique Individuals. The Population object also contains the main iteration method, which steps through the DEMCMC algorithm. Within this iteration there are several key stages: a mutation of the parameter-sets, the differential operation, fitness and acceptance evaluation and a collation of results. These stages each occur in serial, handling one Individual after the other. It is at this level that GPU parallelism was first proposed, dedicating a thread to each Individual.

Previous attempts at optimisation through the use of multi-threading take advantage of multi-core CPUs to run multiple Populations' DEMCMC iterations in parallel, while within each Population, Individuals are still processed in serial.

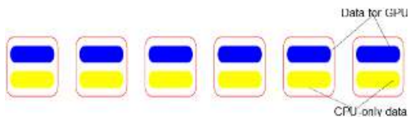

Figure 2: Encapsulating data in objects

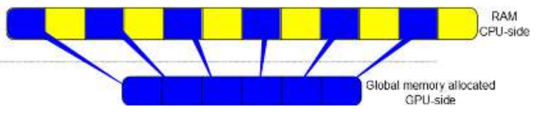

Figure 3: Copying broken memory to the GPU

4.2 Preparing data for the GPU

Encapsulation is a corner stone of object-oriented (OO) programming, but the bundling of data into single objects runs counter to how that data will be selected for use on the GPU.

It may well be the case that the pieces of data to be accessed by each GPU-thread are each held in their own object. Figure 2 shows how data to be used for the GPU can be broken up and encapsulated with data that is not going to be used on the GPU.

The problem becomes clear when you view the objects as they exist in linear memory; shown in the upper half of Figure 3. In order to copy the required data to the GPU, a separate copy call must be made for each object or GPU-thread, which will likely be thousands. It may seem better to copy the unused data to the GPU as well, simply using more memory GPU-side but only making one call to copy it. This ignores the facts that the unused data may be many times larger than the used data, making it inefficient as well and that this is not a suitable way to structure data on the GPU (see section 4.3).

This problem was addressed by restructuring the data so that the GPU-relevant data lay outside of the Individual objects. Individuals maintain pointers to said data allowing CPU-side code to continue to function, as seen in Figure 4.

4.3 Structuring Data on the GPU

Figure 5 shows how two GPU-threads might access their data in Global Memory. They simultaneously access their respective data indexes 0, 1, 2 etc. This means that the data being accessed simultaneously is strided by the size of an Individual's data (in this example, 10 elements).

The NVIDIA's OpenCL Best Practices Guide states that

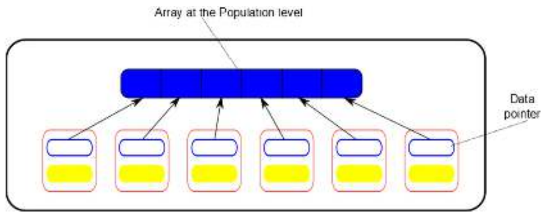

Figure 4: Restructured Population data

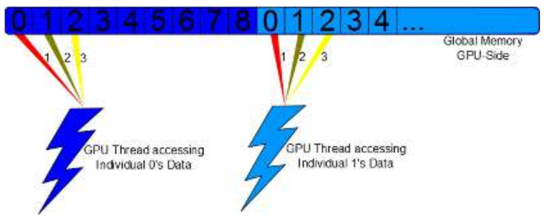
0 1 2 3 4 5 6 7 8 0 1 2 3 4 ...
Global Memory GPU-Side
1 2 3
1 2 3
GPU Thread accessing Individual 0's Data
GPU Thread accessing Individual 1's Data

Figure 5: Simultaneous Data Access

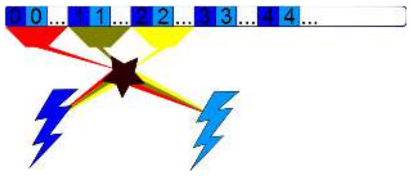

Figure 6: Coalescing data on the GPU

the coalescing of global memory is "perhaps the single most important performance consideration in programming for the CUDA architecture" [5]. This because the GPU architecture is optimised to take advantage of its high bandwidth memory buses, by grouping requests to adjacent memory addresses into a single request.

Figure 6 illustrates how data might be restructured to take advantage of this optimisation. Rather than all of a thread's data being together, different threads' data elements, which are likely to be accessed at the same time are coalesced; all of the threads' index 0 data elements are adjacent, followed by index 1 data elements and so on. This means that when all the threads access their comparable indexes, they will be accessing a single block of memory, which the SM will recognise and group requests into fewer larger requests.

This lead to a restructuring of data used so that the least significant the least significant index of any large data structure was the Individual or GPU-thread index. This means any piece of data that will be read within GPU-thread sits in memory adjacent to the corresponding pieces of data of all other threads. This means that none of the data of a given Individual is adjacent. This is of course the least effective way to access data with a CPU and RAM so this restructuring was only adopted for the data that was used only by the GPU. Input and output data remained structurally unchanged.

Frameworks such as GPUPhysBAM (GPU Physics Based Modelling) [4] have been developed to address this problem. This framework aims to automatically restructure data as it is transferred between the CPU and GPU, so that it optimised for access in both CPU code and the GPU kernel, though it was not adopted for this work.

4.4 Random Number Generators (RNGs)

There are a number of different approaches to using random numbers in GPU code and as the architecture evolves, the best approach may change. One approach is to generate all the needed random number prior to the running of the GPU threads. This can be done either CPU side or on the GPU. Generating on the GPU may be more efficient and it means that the numbers are created in GPU memory ready to be used.

A problem with either approach is that one needs to know how many numbers will be required. Iterative methods conditioned on the numbers generated have the potential to require an infinite number of numbers. This outcome is usually just incredibly unlikely. In this situation one would tend to generate some kind of maximum expected requirement that would almost always be enough.

This problem is worsened by the fact that the GPU threads cannot efficiently share a list of random numbers, because they would need to maintain an atomic index indicating which number should be read next. Instead one must generate a maximum requirement for each thread. This could increase the total requirement drastically and it is likely that most of these numbers would never be used.

The approach adopted in our work was to generate the numbers in parallel on demand, using the same seed but with unique sequence numbers for each thread, which act as offsets into the generators computation. While generating numbers beforehand used to be the preferred option as on-demand parallel methods were not efficient enough [9]. Improvements in later versions of CUDA have made this a much more viable option [10] and so it was decided to adopt the parallel XORWOW generator found in the CURAND library.

4.5 Reducing

Some aspects of the DEMCMC algorithm involve an inherently serial interaction between Individuals' threads, such as the summing of fitness/likelihood values (see section 2.3). Reducing or folding methods are useful in making this process more parallel. Using shared memory, half of the threads can add their value to the values of the other half. By repeating this process, the sum can be found in $Log_2 N$ steps, where N is the number of values being summed. Between each step threads can be synchronised to ensure all values have been written using the syncthreads method. Syncthreads is part of the CUDA framework and will act as a barrier for the threads provided that they belong to the same block. The pattern of writes also benefits from the coalescing optimisations described in section 4.3.

4.6 CPU Multithreading meets GPU

Given that the system already implemented multithreading, it was decided to maintain this to preserve the optimisation of any remaining CPU-side code. In the system, each thread handles a single population. A population of 256 Individuals will launch 256 GPU-threads in a kernel call. A much greater number of GPU-threads could be launched if multiple populations could collate their Individuals to run in parallel.

We developed a system that uses barriers to synchronise CPU threads before informing the GPU methods that they will be running multiple populations. The GPU kernels were slightly altered so that it can run for multiple populations. The key difference is that the differential operation kernel must only occur within a population rather than between the increased numbers of GPU-threads, while other kernels can remain fairly agnostic due to their isolated nature.

Damage to memory coalescing should be minimal as the data being simultaneously read within a population is much greater the bandwidth of coalescing optimisations.

Another approach would be to make use of concurrent kernel launches. The CUDA C programming guide states that some GPUs with a compute capability of 2.0 or greater

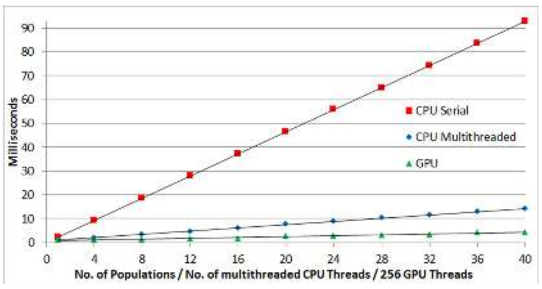

Figure 7: Mutate method CPU vs MT-CPU vs GPU

can execute multiple kernels concurrently [7]. It would be interesting to examine this feature because while it obviously allows you to launch multiple different kernels, it may or may not be beneficial to launch multiple instances of the same kernel as opposed to one large kernel.

5. RESULTS

The system was tested and compared with both serial execution and the pre-existing multithreading (MT) optimisation. The MT-CPU version was parallelised at the Population level, while the GPU solution was parallelised at the Individual level, meaning the latter used 256 times as many threads. While all methods presented a significant optimisation when compared to serial execution, one of the benchmarks methods proved to be less effective than the CPU multithreaded solution.

Separate kernel methods were developed as a result of the incremental CPU to GPU porting process, but they were later grouped to avoid the overheads of unnecessary kernel launches. Unfortunately, the need for a guaranteed global synchronisation necessitated a division of kernels. This is the case with the differential operation, as it involves each thread randomly reading globally shared memory, which is written to in the mutation stage. Given that this division of kernels is for the purpose of synchronisation, data is not transferred between the GPU and CPU for each kernel call, but is left in global memory until each kernel has run several thousand times.

5.1 Mutation

The simplest of the GPU methods was Mutate and it presented an immediate optimisation with only 256 GPU threads, as shown in Figure 7. With 10240 GPU threads there was a x21 speedup compared to the serial execution. Even when compared to the multithreaded CPU code there was a x3 speedup.

5.2 Differential Operation

With only 256 GPU threads, the GPU implementation was actually significantly slower than serial execution. Increasing the number of threads (to 10240), the GPU code quickly became a more efficient solution giving a x2 speedup. However the MT-CPU optimisation proved to be more effective, giving a x4 optimisation over serial execution and x2 speedup over the GPU implementation.

5.3 Fitness

Similar to the Differential Operation method, the initial GPU implementation for fitness with only 256 threads was

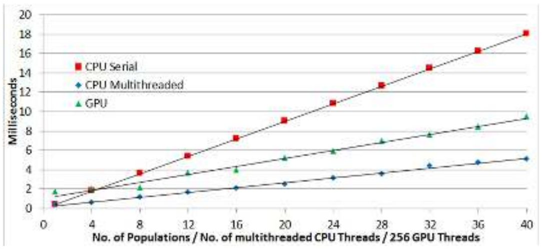

Figure 8: Differential Operation CPU vs MT-CPU vs GPU

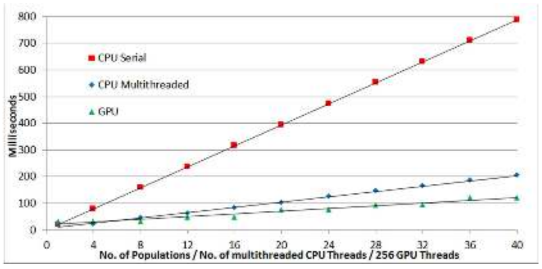

Figure 9: Fitness CPU vs MT-CPU vs GPU

significantly less efficient than serial execution. Figure 9 shows how the GPU solution quickly became the more efficient solution. Conversely to Differential Operation, the GPU implementation of Fitness showed superior scaling and eventually presented both a x6 speedup over the serial execution and a x2 speedup over the MT-CPU implementation.

6. DISCUSSION

One of the core difficulties of programming for GPU architectures pertains to their parallelism, which translates as threads in terms of programming. From a design point of view, however, GPU threads differ starkly from their CPU counterparts. While CPU threads can handle any kind of concurrency, GPUs are vector processing units optimised for Single Instruction Multiple Data (SIMD) loads.

The simplicity of the SIMD model makes GPU multithreading generally easier to grasp than multithreading on the CPU. On the CPU, multithreading has long been identified as difficult to program correctly by developers. Lee claimed for instance that asking developers to program in threads was comparable to asking them to be insane [3]. To appreciate and handle the non-determinism of multithreading they must understand that they are repeating the same action many times expecting different results. Lee went on to suggest that we should move away from traditional CPU multithreading to explore other, more intuitive parallel techniques; some of which were parallel architectures like the GPUs used here.

We experienced this strong difference between CPU and GPU multithreading in our own work. One example occurred in the seemingly simple task of profiling methods in multithreaded execution. While timing the execution of a GPU kernel is a relatively simple task because of the SIMD philosophy of GPU, and is directly supported as a result by the CUDA framework, the MT-CPU implementation required barriers and counters to ensure that all threads started the method at the same time and that the first and last threads started and stopped the timer respectively.

This difference between the traditional model of CPUs and that of GP-GPUs suggests that a modular middleware for GP-GPU-based environmental simulations should focus on interaction paradigms that directly capture the SIMD parallelism of GPUs. Lee's work on alternatives to CPU multithreading [3] should in this respect provide interesting avenues which we plan to pursue.

Another lesson learnt from our work is that not all parts of a MT-CPU program are equally suited for GPU-type parallelism. For instance, whilst the GPU implementation of the differential operation was more efficient than serial execution, it differed from other methods tested, in that it was slower than the MT-CPU version (see section 5.2). This suggests that this method is not suited to the GPU architecture, but there are successful examples of DE algorithms for the GPU [10]. It is likely that this dissimilarity is due to both the large data size of the parameter sets used by our system and the number of parameter sets selected for forming vectors. The differential operation used in [10] only took vectors from the 10 best fitting Individuals to give a faster convergence, whereas our work used any Individual in the population to prevent a fast convergence to a local maximum. This design and the size of our data meant that the limited cache of the GPU units was very probably overwhelmed and the data being read was much less likely to be coalesced (Section 4.3).

This second point highlights the importance of an accompanying process to guide the design choices of developers and environmental modellers. Differential Evolution (DE) is a recurring method for parameter search estimation in environmental sciences, and would benefit from being provided as a reusable software building block. Yet, our experiment shows that the details of a DE strategy and the type of data it uses have a direct impact on its suitability for GP-GPU execution. This knowledge should ideally be embodied into any envisaged DE middleware to guide (and possibly automate) decisions regarding the distribution of code between the CPU and GPU.

Finally, our work has been quite limited in developing for a single model of GPU (Quadro 4000) with a single framework (CUDA). An exploration of the heterogeneity of GP-GPU enabled hardware would probably uncover many more challenges. Even within the subset of NVIDA GPUs, factors like the amount of memory available on the GPU may mean that a solution should be split over multiple GPUs or become serial. Another important consideration is the compute capability of such devices. These may not just require altering where the solution is ran; they may require using an entirely rewritten GPU kernel to avoid heavy performance penalties between incompatible compute capabilities. Again, an insulating software layer between a modeller's code and the moving reality of GPU capabilities seem to offer a promising route to avoid costly ad-hoc maintenance efforts on long-running environmental code, while benefiting from the latest advances in GPU capacities.

7. CONCLUSIONS AND FUTURE WORK

While this work has been a mild success in optimising an existing system, the numerous challenges faced have highlighted some of the common concerns about GP-GPU development. To be successful, developers are required to have a good knowledge of how to take advantage of the GPU architecture available, as well as strong understanding of the problem the system is addressing. Given that most of these problems lay outside of the field of computer science, there is a clear opportunity for middleware to make these technologies available to people who shouldnâĂŹt need to consider the underlying mechanics.

Heterogeneity of devices, such as that described in in section 6, is yet another obstacle that is frequently addressed by middleware. Ideally a middleware would provide a service to the user that required little or no knowledge of GPU hardware. This service may then compute a request as best it can with the hardware available.

Our work focused on a single and specific system. It would be useful to look at similar applications in order to discover commonalities and reusable components that such a middleware could provide. Design, producing and deploying such a middleware will be a challenging endeavour, but the result should be an invaluable tool for scientists.

8. ACKNOWLEDGEMENTS

This work is being supported by Centre for Environment, Fisheries & Aquaculture Science (CEFAS) and the EPSRC within the iCASE project entitled: SeaCloud: A GPU-Powered Cloud for Marine Ecology

9. REFERENCES

[1] M. Bakhtiari, H. Malhotra, M. D. Jones, V. Chaudhary, J. P. Walters, and D. Nazareth. Applying graphics processor units to Monte Carlo dose calculation in radiation therapy. *J Med Physics*, 35(2):120–122, 2010.

[2] C. Kim and S. Westin. Software maintainability: Perceptions of EDP professionals. *MIS Quarterly*, 12(2):167–179, 1988.

[3] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[4] P. Mistry, D. Schaa, B. Jang, D. Kaeli, A. Dvornik, and D. Meglan. Data Structures and Transformations for Physically Based Simulation on a GPU. *Computer Engineering*, 6449:162–171–171, 2011.

[5] NVIDIA. NVIDIA OpenCL Best Practices Guide. *Optimization*, 181(1.0):2175–2184, 2009.

[6] NVIDIA. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: FERMI. *ReVision*, pages 1–22, 2009.

[7] NVIDIA. Nvidia CUDA C Programming Guide Version 4.2. page 173, 2012.

[8] W. Zhu. Massively parallel differential evolution-pattern search optimization with graphics hardware acceleration: an investigation on bound constrained optimization problems. *J Global Optim*, 50(3):417–437, Aug. 2010.

[9] W. Zhu and Y. Li. GPU-accelerated differential evolutionary Markov Chain Monte Carlo method for multi-objective optimization over continuous space. *Proceeding of the 2nd workshop on Bioinspired algorithms for distributed systems BADS 10*, page 1, 2010.

[10] W. Zhu, A. Yaseen, and Y. Li. DEMCMC-GPU: An Efficient Multi-Objective Optimization Method with GPU Acceleration on the Fermi Architecture. *New Generat Comput*, 29(2):163–184, 2011.