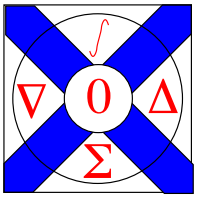

Computational Science Technical Note **CSTN-165**

On-Demand Generating and Scheduling Optimised Parallel Applications on Heterogeneous Platforms

K. A. Hawick and D. P. Playne

2013

Scheduling applications tasks across heterogeneous clusters is a growing problem, particularly when new upgraded components are added to a parallel computing system that may have originally been homogeneous. We describe how automatic and just-in-time source code generation techniques can be used to make the best parallel decomposition for whatever resource is available in a heterogeneous system consisting of graphical processing unit accelerators and multi-cored conventional CPUs. We show how a high level domain specific language approach to our set of target simulation applications can be used to cater for a variety of different GPU and CPU models and scheduling circumstances. We present some performance and resource utilisation data illustrating the scheduling issue for heterogeneous systems in computational science. We discuss the future outlook for this approach in eScience more generally.

Keywords: eScience; computational science; on-demand code generation; simulation; code reuse; GPUs; multi-core

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-165,
        author = {K. A. Hawick and D. P. Playne},
        title = {On-Demand Generating and Scheduling Optimised Parallel Applications
                on Heterogeneous Platforms},
        booktitle = {Proc. 12th Int. Conf. on Software Engineering Research and Practice
                (SERP'13)},
        year = {2013},
        number = {CSTN-165},
        pages = {SER2424},
        address = {Las Vegas, USA},
        month = {22-25 July},
        organization = {WorldComp},
        institution = {Computer Science, Massey University},
        keywords = {eScience; computational science; on-demand code generation; simulation;
                code reuse; GPUs; multi-core},
        owner = {kahawick},
        timestamp = {2012.12.01}
}
```


Prof Ken Hawick, Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand.

Technical Report CSTN-165

On-Demand Source Code Generation & Scheduling Optimised Parallel Applications on Heterogeneous Platforms

K.A. Hawick and D.P.Playne
Computer Science, Massey University, North Shore 102-904, Auckland, New Zealand
email: { k.a.hawick, d.p.playne }@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

March 2013

ABSTRACT

Scheduling applications tasks across heterogeneous clusters is a growing problem, particularly when new upgraded components are added to a parallel computing system that may have originally been homogeneous. We describe how automatic and just-in-time source code generation techniques can be used to make the best parallel decomposition for whatever resource is available in a heterogeneous system consisting of graphical processing unit accelerators and multi-cored conventional CPUs. We show how a high level domain specific language approach to our set of target simulation applications can be used to cater for a variety of different GPU and CPU models and scheduling circumstances. We present some performance and resource utilisation data illustrating the scheduling issue for heterogeneous systems in computational science. We discuss the future outlook for this code generation approach in software engineering.

KEY WORDS
software engineering; on-demand code generation; code reuse; computational science; simulation; GPUs.

1 Introduction

Scheduling application jobs [2,7] across heterogeneous parallel computing systems is a long standing problem in computational science, with renewed efforts and work reported for distributed systems [1, 3, 20, 21] and grid systems. Cluster computing has also attracted a lot of scheduling research efforts [10,12,17,18]. As soon as it is given its first upgrade any computer cluster or systems typically becomes heterogeneous unless great care and planning moves are made to obtain exact replacement or upgrade components. Clocks speeds move on, memory speeds and capacities improve in performance and price performance and so do disk capacities. As nodes are added to any existing compute cluster there is firstly a strong temptation to upgrade with improved price performance or improved performance components. Secondly, as a system ages it may become effectively impossible to source the older components, even if there is an intention to maintain the original systems' homogeneity.

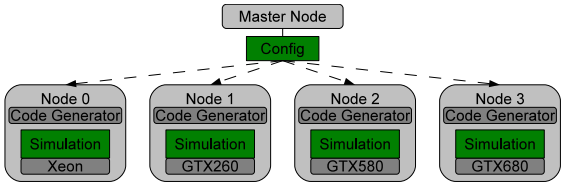

Figure 1: An illustration of a master node distributing a configuration file to four different nodes, each of which will generate code optimised to run it's specific hardware.

This problem of managing a resource that inevitably becomes less homogeneous in nature with time is therefore common. In this paper we consider the particular issues concerning GPU-accelerated clusters that are growing in heterogeneity [26]. We describe how software engineering techniques such as automated code generation from a higher-level problem specification can improve code reuse and extend its lifetime.

Hybrid systems of multi-cored conventional CPUs as well as GPU-accelerated systems are also quite common and scheduling for highly heterogeneous environments is a particular challenge [4]. At the time of writing GPU systems are becoming widespread but we believe this issue of growing heterogeneity and other legacy system effects are still relatively new for GPU system owner/operators. The heterogeneous system effect will not go away and cannot be simply addressed as an economic issue. The pragmatic approach is to consider what ideas, terminologies, and software technologies and solutions are available to help quantify the scheduling inefficiencies and aid us to make better and effective use of heterogeneous resources as part of a managed process.

Many groups, like our own, will be working with a mix of optimisation goals. We are interested in parallel computing systems from an experimental computer science systems perspective as well as a computational science one. That means

we actively collect disparate systems with different processors, CPU models, memory configurations and so forth as that gives us a wide range of experimental system data points with which to explore parallel algorithms, system capabilities, efficiencies, speed ups, cache effects and similar experimental systems effects. However, to make good use of the capital investment in such equipment we also like to keep our systems busy doing number crunching and running simulations and other science applications – when we are not deliberately reconfiguring them.

In this paper we discuss our ongoing work on scheduling dynamically generated applications codes on various parallel computing platform configurations. We are exploring the idea of applications software that is (re)compiled at run time for the particular platform that the scheduler deems appropriate and available. This is not new idea in general, and OpenCL [27] contains the notion of just-in-time compilation, particularly aimed towards heterogeneous systems [13]. Similarly, many attempts have been made over recent years to come up with compilation transformation tools based around compiler directive for example, that will allow the relatively straightforward re-targeting of source code to a specific platform or configuration such as a GPU [9].

We go a step further in the work we report here and show how some of the emerging domain-specific programming language technologies and ideas [8, 11, 14–16] can be used to tackle this issue. We show how numerical simulations - admittedly in a very specific applications domain - can be written in a high level language that can be used to generate highly optimised implementations for parallel paradigms and platforms. Specifically, we discuss how a set of field equation-based simulations that have been formulated in this way can be used to explore the idea of dynamically generating optimised parallel versions for different models or families of models of GPU devices or for multi-cored CPUs.

In this present paper we consider the new notion of dynamically interrogating the heterogeneous components of the cluster to determine which node or nodes best satisfy the parallelism capabilities that match the task as well as scheduling availability of the hardware resource in question. This model has restricted utility - it obviously involves an overhead to generate and compile the application code and the latency of carrying this out has to be weighed against the time to execute the task that is being optimised. Nevertheless we believe that for departmental resources in scenarios like ours, this model has great value.

The heterogeneity of GPU devices is hard to overcome due to the drastic change in architecture between different generations. To get the maximum performance out of a GPU, code must be specifically tuned to make best use of the device's specialized memory. Even a small change in memory patterns or problem decomposition can have a large impact on performance. While it is possible to write general GPU implementations that work on devices of all generations, these codes tend to be large, complex and still cannot fully utilize the device.

The novelty of our system comes from the software architectural notions shown in Figure 1. The assumption is that complex equation based code has been formulated in a high level form that is input to a source code generator. The output of the generator is conventional C/C++ source code that might have: embedded compiler directives; generated message passing calls; generated multi-threading management; or of most recent interest – specialist GPU kernel calls in CUDA or a similar language, also generated automatically. The resulting (human-readable) source code is then compiled in the usual way by the vendor or platform-specific tools and the job appropriately launched and run.

Dynamical source-to-source code generation is still a relatively unusual approach, with most reported work on generated code on-demand appearing in the mobile computing literature [6, 22]. In this paper we add to its novelty by doing it on-demand – effectively at run time, but with a high-level application-specific language specification of the core algorithmic aspects.

Our present article is structured as follows: In Section 2 we summarise the general problem of scheduling and lay out a notation for performance timing. We summarise the particular class of numerical simulations applications we use for our benchmarks and performance analysis in Section 3 and give a description of our prototype just-in-time source code generator in Section 4 and present some performance timing results in Section 5. We discuss the implications for scheduling applications on heterogeneous systems in Section 6 and offer some conclusions and areas for further work in Section 7.

2 Scheduling Systems

Scheduling jobs has been an important area of research throughout the history of computing. It is usual to split the subject from two usually different perspectives: firstly individual users or programmers aim to get a particular job to complete in the shortest time possible - either by having it start running as soon as possible in any given queue system and/or having it run on the fastest and most appropriate resource available. The organisation that owns and operates the resources usually has the possibly conflicting goal of having the resource be as well utilised as possible. Economic considerations can provide another axis of interest but in our discussion we focus only on the first two points.

We have realised that as a computer science research group we need to combine the two goals. Much of our "computer science" systems oriented research work involves experimenting with our systems, often deliberately reconfiguring them to try different combinations of processors, memory, accelerators, and communications system. When we are not doing this however we want to make it as easy as possible to deploy number crunching applications that will soak up as many compute cycles and other resources as possible while producing "computational science" outcomes in the form of completed numerical experiments and analysis and so forth. These two

complementary aspects of computational science need to co exist and this present paper is a manifestation of some prototypical software management and scheduling analysis.

There are many good software systems for managing jobs on cluster computers. We focus here on the latency overhead issues concerned with giving a scheduler the additional capability of generating applications source code and compiling it "just in time" before running it in the usual way.

Suppose we have a number of compute jobs labelled by index j that are to be scheduled to run on the most appropriate of a set or resources index by r. Generally a scheduler or job management system will have one or more queues that are usually managed as first in first out streams of jobs. They need not of course be executed in the order of submission as there may be any number of economic and socio-political priority considerations in effect. The goals are either: to minimise the time to completion of all or some jobs; or to maximise the resource utilisation efficiency. Scheduling a homogeneous collection of resources is a relatively well known problem and often modern resources have the capability of running more than one job at once, with some degree of process level parallelism managed straightforwardly by the operating system software.

In the scenarios we consider in this paper, we are often interested in jobs that are being timed or benchmarked as part of an exploration of a resource configuration or as part of parallel algorithmic development work. Consequently we often want exactly one job running per resource at any given time, to minimise job-job interference through resource contention and so forth.

However in the heterogeneous systems we are interested in there is some extra information is available to the scheduler concerning the resources and their capabilities. They can be queries or polled dynamically to determine what their availability is, but they can (and need to) have a much richer and more detailed capability specification that would a plain ordinary CPU. These include floating point capability, number of GPU devices, number of low level cores per device and other parameters which as we find in our results can make an order of magnitude in difference in run time if not properly catered for.

Our approach to this problem has been to consider how much information can be made available to the scheduler about the application properties as well as the compute resources, and to consider how this information needs to be expressed and how it can best be matched by a smart scheduler.

3 Field Equation Examples

To focus on specific applications for which we can measure and demonstrate improved performance, we report on some simulation model calculations based upon a field equation formulation.

Three example field equations are used to evaluate the on-demand code generation system - the Heat (1), Ginzburg-Landau (2) and Cahn-Hilliard (3) equations. These equations were chosen for several reasons. First of all the code generation system used in this research is designed for field-equations that can be numerically simulated on N-dimensional regular lattices, using finite-difference methods and explicit Runge-Kutta integration methods [24]. These three equations all fit into this category and can be automatically generated.

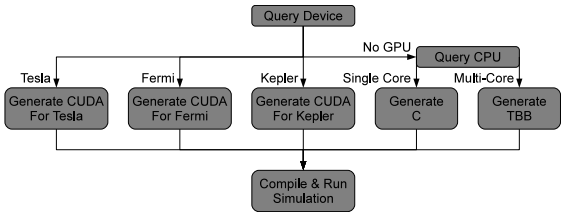

Figure 2: Code generator process to select a suitable processing device and generate code for the simulation.

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u \tag{1}$$

$$\frac{\partial \psi}{\partial t} = -\frac{p}{i}\nabla^2\psi - \frac{q}{i}\left|\psi\right|^2\psi + \gamma\psi \tag{2}$$

$$\frac{\partial \phi}{\partial t} = m\nabla^2\left(-b\phi + u\phi^3 - K\nabla^2\phi\right) \tag{3}$$

The three equations also have different computation intensities and memory halos. The the Heat equation is a very simple equation with a small memory halo and can be represented by a scalar field. The Ginzburg-Landau equation also has a small memory halo but requires a field of complex numbers to represent the field. Finally the Cahn-Hilliard equation is represented by a scalar field but requires the use of the biharmonic operator resulting in a larger memory halo. These equations are provided to the generator as ASCII representations in an equation file. An example ASCII representation of the Heat equation can be written as follows:

```
floating a;
floating[] u;
d/dt u = a * Laplacian{u};
```

The generator will then parse this file and combine it with the appropriate stencil and integration method as defined by the configuration file. This process is shown in Figure 3.

More details on the workings of our code generator and domain-specific field equation language are described in [19, 23] from the perspective of the applications domain. In what follows, we focus on the aspects of generation for different accelerator devices and capabilities.

4 Code Generation On-Demand

The Code Generation component of this system does not consider the problem of scheduling but will simply run the simulation on it's hardware as it sees fit. It assumes all scheduling

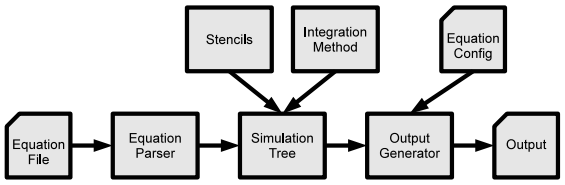

Figure 3: A diagram of the structure of the code generator. The generator take information on the equations, stencils and integration method to construct an abstract tree representation of the simulation. This tree and configuration is given to the output generator that produces code for a specific target language.

To launch a simulation on a machine, the master node sends the configuration file to the node and launches the code generator. This configuration file contains the details of the simulation, model parameters, numerical methods, time scale etc. When the code generator is launched it will first query the device to determine what computing resources it has available. First it will determine whether or not there is a suitable Graphical Processing Unit available.

If no GPU device exists the simulation must be run on the CPU in which case it will query the CPU to determine if it is a single- or multi-core CPU. The generator will then create a C implementation for a single-core or an implementation using Thread Building Blocks (TBB) [25] for a multi-core CPU. This is not a restriction of the generator, C and TBB were chosen simply due to previous experience with this language and library. Generators could easily be written for other languages or multi-threading libraries.

Listing 1: Code snippet of the Tesla implementation allocating texture memory and fetching values from texture memory for the point (ix,iy).

```
//Create texture
texture<float, 2, cudaReadModeElementType> texture_u;

//Create and bind array
texture_u.normalized = false;
texture_u.filterMode = cudaFilterModePoint;
cudaArray *array_u;
cudaChannelFormatDesc u_descriptor =
     cudaCreateChannelDesc<float>();
cudaMallocArray(&array_u, &u_descriptor, X, Y);
cudaBindTextureToArray(texture_u, array_u);

//Fetch values from texture memory
float u0ym1x = tex2D(texture_u, ix, ym1);
float u0yxm1 = tex2D(texture_u, xm1,iy);
float u0yx   = tex2D(texture_u, ix, iy);
float u0yxp1 = tex2D(texture_u, xp1,iy);
float u0yp1x = tex2D(texture_u, ix, yp1);
```

If a suitable NVIDIA GPU is available on the machine, the generator will run a small program to query the device(s) to determine their capabilities. The generator can be configured to find a suitable device to run the simulation. This includes selecting a device with sufficient memory for the simulation etc. If no suitable device is found to run the simulation, the generator will fall-back to using the CPU. If there is more than one suitable device, the generator will select the latest generation device. This process of querying the machine and generating code is shown in Figure 2.

The major difference in code generation lies between the Tesla and Fermi/Kepler generation cards. This is due to the introduction of L1/L2 cache in the Fermi and subsequent generation devices. Prior to this change in memory architecture, texture memory provided the best performance for the type of access pattern used by these simulation. The use of texture memory to fetch values from the field is shown in Listing 1. However, in later generations the higher bandwidth of L1/L2 cache provides the best performance (See Listing 2). This change in memory type requires some significant changes to simulation code. The difference between Fermi and Kepler devices is less as they are more similar architectures.

Listing 2: Code snippet of the Fermi/Kepler implementation fetching values from global memory through L1/L2 cache and calculating the heat equation for the point (ix,iy).

```
//Create global memory array
float *u0;
cudaMalloc((void**) &u0, X*Y*sizeof(float));

//Fetch values from global memory
float u0ym1x = u0[ym1*X + ix];
float u0yxm1 = u0[iy*X  + xm1];
float u0yx   = u0[iy*X  + ix];
float u0yxp1 = u0[iy*X  + xp1];
float u0yp1x = u0[yp1*X + ix];
```

Once the generator has determined the device it is going to run the simulation on. It can produce code tailored specifically for that device. This includes using different memory types, grid/block sizes based on the number and type of multiprocessors in that GPU etc. Currently this system only makes use of a single GPU however it can be extended to utilize mGPU machines including systems with multiple GPUs of different architectures.

The details of this code generation system is described in [24] but the general structure of the generator is shown in Figure 3.

There will be a slight overhead when running simulations using this code generation method. Obviously there will be some communication required to send the configuration file to the compute node, the configuration is a small file and the time to send it to the node is negligible. Copying the results back from the simulation may require more communication but this is dependent on the simulation not the code generator and thus is not considered.

The overhead comes about from the fact that rather than distributing and running a program, the nodes must read the configuration file and generate the code for the simulation. The exact generation time will vary based on the hardware and computational load of the node, the complexity of the equation and numerical methods and implementation of the generator. For the most part this generation completes in the order of seconds and generally much less than the run-time of the

simulation. This generation time is discussed further in Section 5 below.

5 Results

The best way to assess the feasibility of the just-in-time code generation approach is to measure the performance attainable on different system configurations. Although one is normally interested in the scalability and how the performance of an application changes with the number of parallel components or with some measure of the problem size, in this paper we are especially interested in determining accurately the latency or overhead that arises from the code generation and recompilation. A reliable way to determine the "zero sized job" time is to plot run times with increasing job size and use a least-squares fit, weighted by the standard deviation on the completion times. the slope of such a fit gives us the normal indications of speed scaling, but more usefully here, the intercept - accurately extrapolates back to zero job size and gives us the latency overhead.

Figure 4 shows the compute time vs number of simulations steps run for a 1024x1024 cell sized Cahn Hilliard simulation, integrated using the RK2 integration method on four different nodes. These four nodes all have different compute devices - a Tesla GPU (GTX260), a Fermi GPU (GTX580), a Kepler GPU (GTX680) and a multi-core Xeon (X5675). Analysing the intercepts of these plots shows that the overhead of code generation is $\approx 1...6$ seconds depending on the node.

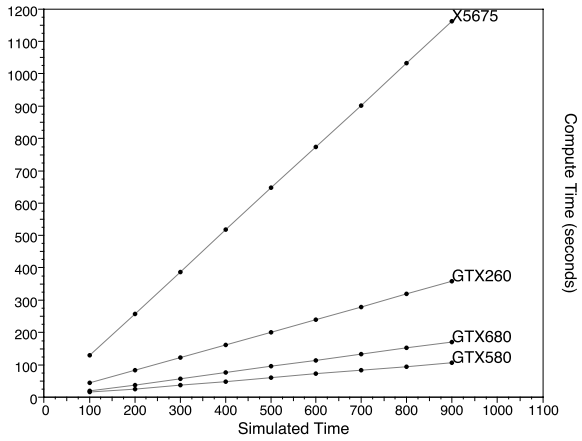

Figure 4: Plot of simulation time vs compute time for a Cahn-Hilliard simulation using RK2 numerical integration on a 1024x1024 field. Results shown for a GTX260, GTX 580, GTX680 and a four-core Xeon X5675.

Another comparison can be drawn between generic CUDA code and code that has been specifically targeted for a certain type of card. Obviously if all GPU cards were of the same generation, it would be much easier to create a general implementation that could run reasonably efficiently on all of them. However, to be able to run a simulation on all generations of card, the implementation must be built for the most general case. In this case we compare a Cahn-Hilliard simulation that can be run on any CUDA capable card and the implementations created by the code generator (Gen). It can be seen from Figure 5 that the generated code provides significant performance benefits over the general version.

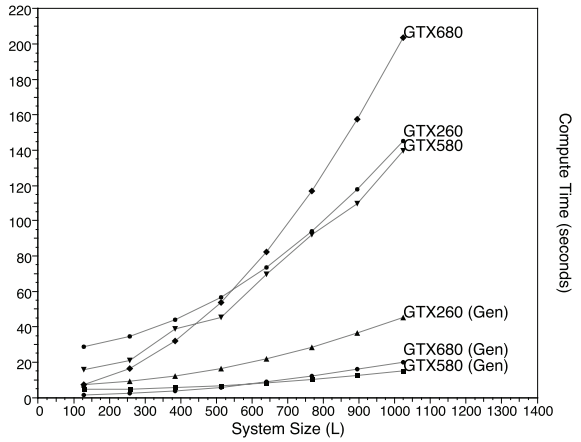

Figure 5: Comparison between code generated for specific devices (Gen) and device independent code. The simulation used to compare these implementations is a Cahn-Hilliard Simulation using RK2 (h=0.01) and system size of L=$\{128...1024\}$ run for a simulation time of 100.

Table 1 shows the performance variations of different devices computing the three example simulations using float and double data types.

Table 1: Performance variations (in seconds) across different devices. Accurate to $\pm 0.5s$

Precn.	Eqn.	TBB	Tesla	Fermi	Kepler
float	Heat	224s	57s	22s	18s
	TDGL	720s	114s	41s	32s
	CH	920s	133s	46s	30s
double	Heat	339s	119s	33s	44s
	TDGL	984s	305s	93s	129s
	CH	1161s	358s	106s	171s

As can be seen from the table, computing simulations with double precision requires more compute time. This increase in compute time is especially noticeable in the old (Tesla) and new (Kepler) generations of GPU.

6 Discussion

The code generation overhead times as shown in Figure 4 and from Table 1 are significant in absolute terms, but in relative terms and for the sort of numerical simulation job that might typically take more than a few minutes at least, and more

likely take more than an hour, the overheads are not significant. The application codes were have focused one are have been shown to be good representative benchmark codes based on past work. The data suggest therefore that the dynamical code generation approach is quite feasible an practicable.

The different GPU models show quite significant variations in performance. These models are all still relatively recent - they are still commercially available and are only separated by a year or so. This is indicative of advances in the field for accelerator technologies like GPUs. It underlines the importance of planning for heterogeneous systems. Our University plans its computer depreciation on a four year cycles, and arguably for supercomputer cluster equipment one might even expect components to have a usable life-cycle of 5-6 years. It there are component performance changes as significant as nearly an order of magnitude still occurring within a single year, then it is vital to plan for heterogeneity in the system.

At the time of writing we still believe that GPU and similar accelerator devices are especially prone to this effect. Conventional multi-cored CPUs may well exhibit it too as developments in their technology are accelerating. It is entirely feasible to build 4, 6, 8, 16 cored CPU nodes in 2012, and its is likely that 32-cored CPUs will be commonplace and commodity priced by 2014. This same heterogeneity effect will quite likely influence even conventional cluster computer purchases and plans over that time-scale.

In this short present paper we have focused on a small class of numerical simulations that we already understand well and for which the code generation approach works well. Our code generation system obviously has greater scope than the issues discussed here. It allows relatively simple use of quite complex higher order numerical integration schemes with all the boilerplate communications and data structures management code generated automatically. This is useful to be able to experiment with different algorithms, but is beyond the scope of this present paper. Here we have just experimented with different GPU and multi-core tuning aspects rather than major algorithmic aspects.

We have seen in Figure 5 that we can obtain very good performance on GPU-accelerated nodes for this class of numerical simulation, but also that we can improve further by nearly an order of magnitude by tuning for the right device with the right properties. Furthermore as table 1 shows, the overheads accrued are insignificant next to the typical run times of production level jobs. There are a myriad of different GPU model available and this "horse for courses" approach is likely a good vendor strategy and will certainly persist to serve different market segments well. The area of floating point precision and precision capability available to each core will likely continue to be a major market segmentation aspect.

We believe the dynamic code generation approach could be incorporated into a more conventional scheduler software framework. What appears to need more work however is to develop a constraint specification language so that the user or the code generator can impart impart further information about the sort of compute resource that best suits the simulation. As we have commented, OpenCL encourages this notion within a limited scope at run time, but the notion of a domain-specific high level application language opens up this idea further to a greater range of device preferences.

We have not reported in the various scheduling heuristics [5] and other queue parameter tuning that could be done to optimise resource utilisation efficiency. The main point arising from our present work is that a scheduler with the extra information we have described about device specifics and the means to recompile a tuned application will be able to apply economic and other heuristics even more.

7 Conclusion

We have described how source code generation technologies can be used to schedule performance tuned parallel simulation applications on heterogeneous clusters of GPU-accelerated nodes and conventional multi-cored CPUs. Our data indicates that very significant performance enhancement comes from using specially tuned GPU-model specific codes instead of general versions.

We have been able to demonstrate these effects since we have focused on a well-defined set of field-equation based numerical simulation applications. The domain-specific high-level problem formulation works well on this class of problems and we believe could be extended to other application domain families that share common algorithmic and data structural elements

We have shown effects relating to the presence or absence of floating point units; floating vs double precision equipped devices; as well as devices with varying numbers of low level cores. We have also shown effects related to some problems that have greater computational intensities than others.

This notion of custom compilation for an available device is a powerful one. OpenCL environments aim towards having some capabilities for low level device optimisation but since our application domain specific language allows more control at the different levels of the software stack, we have been able to demonstrate quite high benefits with just source to source transformations.

We conclude that while GPUs are already known to greatly accelerate some problems, it is important to consider the particular device model and its availability in scheduling runs to give best turn-around run time and/or best resource utilisation. Allowing a scheduler access to the extra information available in a stack of automatically generated application source code opens up new scheduler optimisation potential especially for heterogeneous clusters. We believe there is scope for further work in incorporating these code generation and management ideas within the framework of existing cluster computing scheduler software systems and that this approach will be useful for improving utilisation of computational resources, particular for research groups like our own

with mixed computer systems science and applied computational science goals.

References

[1] Abramson, D., Giddy, J.: Scheduling large parameteric modelling experiments on a distributed meta-computer. In: Proc. PCW '97 (September 1997)

[2] Adam, T.L., Chandy, K.M., Dickson, J.R.: A comparison of list schedules for parallel processing systems. Communications of the ACM 17(12), 685–690 (December 1974)

[3] Berman, F., Wolski, R., Figueira, S., Schopf, J., Shao, G.: Application-level scheduling on distributed heterogeneous networks. In: Supercomputing '96 (November 1996)

[4] Blazewicz, M., Brandt, S.R., Diener, P., Koppelman, D.M., Kurowski, K., Loffler, F., Schnetter, E.: A massive data parallel computational framework for petascale/exascale hybrid computer systems. arXiv 1201.2118v1, Poznan Supercomputing and Networking Center, poland (10 January 2012)

[5] Braun, T.D., Siegel, H.J., Beck, N., Boloni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., BinYao, Hensgen, D., Freund, R.F.: A comparison of eleven static heuristics for mapping a class of independent taks onto heterogeneous distributed computing systems. J. Parallel and Distributed Computing 61, 810–837 (2001)

[6] Carzaniga, A., Picco, G.P., Vigna, G.: Is code still moving around? looking back at a decade of code mobility. In: 29th IEEE Int. Conf. on Software Engineering (ICSE'07). pp. 9–20. Minneapolis, USA (20-26 May 2007)

[7] Casavant, T.L., Kuhl, J.G.: A taxonomy of scheduling in general-purpose distributed computing systems (May 1986)

[8] Chafi, H., Sujeeth, A.K., Brown, K.J., Lee, H., Atreya, A.R., Olukotun, K.: A domain-specific approach to heterogeneous parallelism. In: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming. pp. 35–46. PPoPP '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1941553.1941561

[9] Chen, L., Liu, L., Tang, S., Huang, L., Jing, Z., Xu, S., Zhang, D., Shou, B.: Unified parallel c for gpu clusters: Language extensions and compiler implementation. In: Proc. 23rd Int. Workshop on Languages and Compilers for Parallel Computing (LCPC2010). pp. 151–165. No. 6548 in LNCS, Springer (2010)

[10] Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: Gpu cluster for high performance computing. In: Proc. Supercomputing (SC'04). p. 47. Pittsburgh, USA (6-12 November 2004)

[11] Fowler, M.: Domain-Specific Languages. No. ISBN 0-321-71294-3, Addison Wesley (2011)

[12] Gan-El, M., Hawick, K.A.: Parallel containers - a tool for applying parallel computing applications on clusters. In: Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'06). pp. 764–767. No. PDP3889, CSREA, Las Vegas, USA (26-29 June 2006), iSBN 1-932415-86-6

[13] Gaster, B., Howes, L., Kaeli, D.R., Mistry, P., Schaa, D.: Heterogeneous Computing with OpenCL. Elsevier (2012), iSBN 978-0-12-387766-6

[14] Ghosh, D.: Dsl for the uninitiated - domain-specific languages bridge the semantic gap in programming. Communications of the ACM 54(7), 44–50 (2011)

[15] Hawick, K.A.: Engineering internal domain-specific language software for lattice-based simulations. In: Proc. Int. Conf. on Software Engineering and Applications. IASTED, Las Vegas, USA (12-14 November 2012)

[16] Hawick, K.A.: Fluent interfaces and domain-specific languages for graph generation and network analysis calculations. In: Proc. Int. Conf. on Software Engineering (SE'13). IASTED, Innsbruck, Austria (11-13 February 2013)

[17] Hawick, K.A., James, H.A.: Trends in cluster computing scheduling and the missing cycles. In: Proc. Int. Conf on Parallel and Distributed Processing Techniques and Applications (PDPTA'05). pp. 732–738. CSREA, Las Vegas, USA. (27-30 June 2005)

[18] Hawick, K.A., James, H.A., Scogings, C.J.: 64-bit architectures and compute clusters for high performance simulations. Tech. Rep. CSTN-024, Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand (April 2006), http://www.massey.ac.nz/~kahawick/cstn/024/cstn-024.pdf

[19] Hawick, K.A., Playne, D.P.: Automatically Generating Efficient Simulation Codes on GPUs from Partial Differential Equations. Tech. Rep. CSTN-087, Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand (July 2010), http://www.massey.ac.nz/~kahawick/cstn/087/cstn-087.pdf

[20] James, H.A.: Scheduling in Metacomputing Systems. Ph.D. thesis, The University of Adelaide (1999), forthcoming

[21] Krueger, P., Chawla, R.: The stealth distributed scheduler. Proc. IEEE 11th Int. Conf. Distributed Computing Systems pp. 336–343 (1991)

[22] Lau, F.C.M., Belaramini, N., Kwan, V.W.M., Siu, P.P.L., Wing, W.K., Wang, C.L.: Code-on-demand and code adaptation for mobile computing. In: The Handbook of Mobile Middleware. Auerbach (2006)

[23] Playne, D.P., Hawick, K.A.: Auto-generation of parallel finite-differencing code for mpi, tbb and cuda. In: Proc. International Parallel and Distributed Processing Symposium (IPDPS); Workshop on High-Level Parallel Programming Models and Supportive - HIPS 2011. pp. 1163–1170. IEEE, Anchorage, Alaska, USA (16-20 May 2011), in conjunction with IPDPS 2011, the 25th IEEE International Parallel & Distributed Processing Symposium

[24] Playne, D.P.: Generative Programming Methods for Parallel Partial Differential Field Equation Solvers. Ph.D. thesis, Computer Science, Massey University (2011)

[25] Reinders, J.: Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism. No. ISBN 978-0596514808, O'Reilly, 1st edn. (2007)

[26] Samuel, T.K., Baer, T., Brook, R.G., Ezell, M., Kovatch, P.: Scheduling diverse high performance computing systems with the goal of maximizing utilization. In: Proc. 18th International Conf on High Performance Computing (HiPC). pp. 1–6. Bangalore, India (18-21 December 2011)

[27] Stone, J.E., Gohara, D., Guochun, S.: Opencl: A parallel programming standard for heterogeneous computing systems. Computing in Science & Engineering 12(3), 66–73 (May-June 2010)