Abstraction of Programming Models Across Multi-Core and GPGPU Architectures

Thomas H. BEACH [a], Ian J. GRIMSTEAD [a,1], David W. WALKER [a] and Nick J. AVIS [a]

[a] *Cardiff School of Computer Science, Cardiff University, Wales, UK*

Abstract. Work in the field of application acceleration devices is showing great promise, but still remains a tool largely for computer scientists with domain knowledge, given the complexity of porting existing algorithms to new architectures or environments. Such porting is hindered by the lack of abstraction available.

We present our latest work in the development of a novel solution to this abstraction problem; an intelligent semi-automatic porting system. This allows a higher level of abstraction where the user does not have to intervene or annotate their source code, while maintaining reasonable levels of performance. We present comparisons between manual and automatic code ports on two different platforms (NVIDIA CUDA and ClearSpeed C^n), showing the versatility of this approach.

Keywords. Application Acceleration, ClearSpeed, GPGPU, Performance Comparison, Semi-Automatic Porting

Introduction

Application speed-up is moving away from reliance on the increasing clock speed on serial processors (which is beginning to level off and be replaced with multiple cores [1]), and instead alternative approaches are required. Alternative hardware is becoming more popular, ranging from Field Programmable Gate Arrays (FPGA [8]) to Application Specific Integrated Circuits (ASICs) such as General-Purpose Graphics Processing Units (GPGPUs [12]), ClearSpeed accelerators[6] and the CELL Broadband engine[14].

In this paper we first summarise the current work in using these "application accelerator" devices and the related problems that prevent their wider acceptance in High Performance Computing (HPC). We then present our work in the development of a feasible solution to these problems, an intelligent semi-automatic application porting system. We describe the prototype that has been constructed, the latest results and discuss remaining future work to facilitate the acceptance of application acceleration devices into mainstream HPC.

[1]Corresponding Author: Dr. Ian J. Grimstead, Cardiff School of Computer Science, Cardiff University, Queens Buildings, 5 The Parade, Roath, Cardiff, Wales, UK, CF24 3AA; E-mail: I.J.Grimstead@cs.cardiff.ac.uk

1. Background

In recent years various high level tools have been developed to aid programming of application accelerators. To reduce the overhead of the unusual programming model, NVIDIA and AMD/ATI have both released their own high level programming languages for their GPGPUs. However, each of these languages is only compatible with specific series of hardware. There have been several tools developed to aid FPGA development, including Mitrionics who have developed the Mitiron-C language for FPGAs and Nallatech who have developed their DimeC language. To compound this, further application accelerators have been revealed: Intel have announced their Larrabee architecture [15] (a GPGPU consisting of many x86 cores in parallel), and Convey have announced their x86 application coprocessor [7].

Although high level tools have been successful with many applications being accelerated; such as Blast on the FPGA using Mitrion-C[10] and MolPro for Clearspeed[5], there is still a need for compatibility between the various classes of application accelerators and standardisation between languages for a class of device, i.e. FPGA and GPGPU. Brook from Stanford[3] and Rapidmind[13] are two other high level tools that have been developed, these tools enable cross compatibility between the two GPU manufacturers and supports a much wider variety of GPUs. The OpenCL [9] initiative has also been announced to produce a cross-platform programming environment, but still requires detailed knowledge of the abstracted architecture. Overall, the user has to supply explicit annotation within their existing code or to port their code directly to a new language; there is still much work to be done in this field.

2. Research Problem and Aims

We believe the main factor limiting the adoption of these application acceleration devices is the lack of a suitably abstracted programming interface for users of these devices [2]. To successfully port an application to an accelerator expertise is needed not only in the applications domain but the computer science domain also. This limitation is a backwards step from the service orientated view HPC currently takes and will become infeasible in the future.

Our research is working on the development of a solution to this limitation; an intelligent semi-automatic application porting system, we focus particularly on the creation of a programming abstraction layer across a range of acceleration devices and enabling the intelligent selection of an appropriate device based on algorithm characteristics—not the automatic parallelization of sequential code. Our system aims to increase the level of abstraction while maintaining performance and be able to perform the following tasks:

- Locating acceleration devices.
- Selecting an appropriate device.
- Porting the application to the device.
- Allowing additional hand tuning of the code.

3. System Overview

The system is designed to cope with the application accelerators being distributed and not all resident in the host machine, such as more powerful accelerators such as Tesla GPUs[11], ClearSpeed CATS accelerators and FPGAs boards are unlikely to present in commodity workstations; even with data transfer overheads applications will still benefit by utilising these more powerful acceleration devices. Web services were selected to support this distributed environment, as web services are relatively mature and a well-understood and supported tool-set.

The overall compilation flow involved in creating an automatic port to an application accelerator from standard sequential C code is presented in [2]; in summary:

- Parse program code to locate candidate kernels
- Extract kernel features
- Use kernel descriptions to determine the appropriate device for acceleration
- Locate appropriate device (supported by UDDI server)
- Port the code to the accelerator device
- Monitor performance of application

In addition to this standard compilation flow the following features are also supported:

- Full profiling of an application across all application acceleration devices
- Bootstrapping a new application accelerator using stored applications

These tasks are divided between several separate components: UDDI server (for device location), client on user's workstation, program classifier and back-ends to support each type of application accelerator. These components are now described in more detail in the following sections.

3.1. System Client

The client is the front-end of the system, coded in C++. The client is responsible for the analysis and preparation of the program for execution on a acceleration device. Program code passed to the client will go through the following stages of analysis:

1. Syntax Check - A simple syntax check that confirms that the input code is correct.
2. Kernel Formation - Constructs a control flow graph of the input code allowing the location of natural loops within the code[16]. Each natural loop is then marked as a candidate kernel and is separated from the program code, with a placeholder being put in its place. Kernel acceleration feasibility is not yet analysed, as the target acceleration device and its restrictions are not known at this stage.
3. Kernel Analysis - Conducts a code analysis on the kernel code to determine various features of the kernel (such as data flow graph, characteristics of the loops formed, data items to be loaded and/or copied back to/from the device, and various metrics for the program classifier).
4. Initial Kernel Filtering - Performs some initial kernel filtering removing loops that are not suitable for execution, such as infinite loops.
5. Output Phase - The client then outputs a modified version of the input program, with the kernel code separated from the host code. The results of the kernel analysis are output as a kernel description file.

3.2. Program Classifier

The program classifier is a machine learning system, built using the WEKA machine learning workbench [17]. The classifier retains a list of known kernels each with associated metrics to provide a representation of the kernel:

- Required data precision.
- Mathematical intensity of calculations.
- Number of iterations and branches present.
- Amount of data copied to/from the device.

These metrics are generated during kernel analysis by the client, and passed to the program classifier for association with performance data supplied from the back-ends when the kernel is executed.

The program classifier performs decision making based on these data at the kernel level and then the program level. Firstly, if a kernel has sub-kernels (kernels that occur within the code of the kernel) at which level kernel should acceleration take place; secondly, which is the best device to accelerate the kernel in question. These decisions are made using a decision tree algorithm, where the kernel description is analysed to determine which device, based on previous experience, a kernel with those characteristics will perform best on. It is entirely possible at this point for the system to decide the best device to execute the kernel on is the CPU.

The classifier then examines at the overall program level; the optimum device for each kernel is now known, so the significance of each kernel within the program is used to decide which single device will give the best overall performance. If the classifier is unable to make a decision (due to a lack of training data), then the porting system executes the application on all available devices to produce the required data.

Note that the integration of new devices into the system is handled by the classifier. A set of test programs, of which performance data are known, are passed to the new device to enable performance data to be gathered. This will bootstrap the new device into the system and allows its selection as a viable candidate for accelerating programs.

3.3. Back-Ends

Currently three back-ends have been developed, each targeting: single core CPU, NVIDIA CUDA and ClearSpeed C^n. The CPU back-end acts as a control to allow comparison between execution on an acceleration device and on a standard sequential processor. This back-end is fully integrating into the porting system to allow this comparison to be made ignoring communication overheads introduced by the porting system itself.

The NVIDIA CUDA back-end and ClearSpeed C^n back-ends have a common structure, where compilation entails the following steps:

1. Kernel Validation - Validates that each kernel is able to execute on the target device and merges kernels that can not be accelerated back into the host code. This including checking for data dependencies within kernels that will prevent acceleration in their current form.
2. Kernel Code Generation - Generates a CUDA/C^n device function based on the original code for the kernel.

3. Host Code Generation - Generates host code, based on the data from the kernel description, this host code will allocate memory storage on the device and copy data from/to the acceleration device.
4. Creation of Build Scripts - Creation of a build script to handle the CUDA/C^n and C/C++ compilation and the linking of system libraries.

These steps enable the system to abstract the actual compilation of the code, hiding any complexities such as the need to run sections of the device code through multiple compilers. The back-ends also control the execution of the application and provide performance monitoring (through the addition of optional instrumentation code) where kernel execution time can be monitored and stored by the program classifier.

3.4. Other Platforms

The back-end can be ported to support any architecture, as long as it is possible to translate C code into an appropriate language supported by the new architecture. One example is the potential to produce an OpenMP back-end to support multi-core CPUs, or direct hardware support for a Larrabee based accelerator.

4. Results

This section discusses the results obtained by using our prototype system with two applications when executed on the GPGPU, CPU and ClearSpeed. A General Matrix Multiplication (GEMM) and an N-Body simulation were chosen as they fall into a separate dwarf category, based on the seven dwarfs taxonomy of applications[1]. The GEMM is a dense linear algebra problem while the N-Body simulation falls into the N-Body methods category. Both examples were initially developed in C and then passed to the porting system. Once the porting was complete, five test datasets were created to test the performance of the ported code.

To determine how successful the porting system has been in generating efficient code, the generated code was compared against the original C implementation and a manually ported implementation of the examples. Wall clock time was measured, to reveal the overall impact of acceleration with full system overheads taken into account rather than just the accelerated portion of the algorithm.

The CPU tests were carried out on an Intel Xeon E5472 3.0GHz (1600GHz FSB) with 16GB RAM (identical CPUs used with the application accelerators). The GPU Tesla C1060 featuring 240 cores and 4GB of memory, rated at 933 single-precision GFLOPs and 78 double-precision GFLOPS. The ClearSpeed device used was a ClearSpeed e710 card, featuring 2GB of RAM and two CSX700 accelerator processors (each with 96 cores), rated at 96 single-precision GFLOPS, 96 double-precision GFLOPS.

4.1. General Matrix Multiplication

This application utilised the standard GEMM formula $C \leftarrow \alpha AB + \beta C$, where the new matrix C is computed based on the product of two matrices A and B and the old matrix C. The dataset size n signifies the size of matrices A, B and C is $n * n$. The results from these tests are presented in Table 1, showing performance figures from NVIDIA CUDA and ClearSpeed C^n, where "SP" represents Single Precision (i.e. SGEMM), and "DP" represents Double Precision (i.e. DGEMM).

Data Size	Execution Time (Seconds)									
	CPU		Manual GPU Port		Automatic GPU Port		Manual Clearspeed Port		Automatic Clearspeed Port	
	SP	DP	SP	DP	SP	DP	SP	DP	SP	DP
200	0.16	0.16	1.43	1.47	1.47	1.54	0.15	0.19	0.21	0.19
800	4.40	4.46	3.34	3.52	3.42	3.53	3.24	4.45	3.14	4.39
1200	12.91	13.56	6.54	6.98	6.57	7.03	9.79	12.71	9.56	12.75
1600	28.85	29.85	12.48	13.60	12.52	13.71	18.06	25.43	15.25	25.27
2000	53.99	55.78	18.31	21.89	20.99	24.35	28.57	47.43	27.52	47.51
2800	139.94	146.30	45.05	54.26	45.83	67.74	82.95	145.72	77.28	145.51

Table 1. Results of GEMM port to NVIDIA CUDA and ClearSpeed C^n

No. Bodies	Execution Time (Seconds)							
	CPU		Manual GPU Port		Automatic GPU Port		Manual ClearSpeed Port	
	SP	DP	SP	DP	SP	DP	SP	DP
50	0.02	0.02	1.29	1.32	1.31	1.33	0.58	0.60
500	0.76	0.93	1.45	1.64	1.46	1.77	1.02	1.24
1000	3.02	3.77	1.54	2.07	1.56	2.19	2.10	3.21
2000	12.05	15.02	1.71	2.88	1.84	3.00	6.67	10.72
4000	47.97	60.66	2.32	6.09	2.46	6.11	26.08	39.57
8000	189.74	239.86	4.17	16.78	5.27	20.57	102.42	153.33

Table 2. Results of N-Body Simulation Port to NVIDIA CUDA and ClearSpeed C^n

4.2. N-Body Simulation

The N-Body simulation we are using is the all-pairs method outlined in [4], the initial inputs to the problem are a set of n bodies $b_1 ... b_n$, each body i has mass m_i, velocity v_i and position p_i. The distance between any two bodies is written d_{ij} and the force on body i due to body j is written f_{ij}.

The algorithm then carries out the following steps

- Compute f_{ij} for all pairs of bodies. $f_{ij} = \frac{G m_i m_j r_{ij}}{|r_{ij}^3|}$ where $i \neq j$
- Compute total force on each body $f_i = \sum_{i,j \neq i} f_{ij}$
- Update the position p_i of each body $p_i = p_i + v_i \Delta t + \frac{\Delta v_i}{2} \Delta t^2$
- Update the velocity v_i of each body $v_i = v_i + \frac{f_i \Delta t}{m_i}$
- In our experiments this algorithm will be carried out for 100 time steps with a varying number of bodies in the system.

Initial results from these tests are presented in Table 2, showing performance figures from NVIDIA CUDA and ClearSpeed C^n. Note that "SP" represents Single Precision and "DP" represents Double Precision variations of the algorithm. Timings of a manual port only were available for ClearSpeed, as the automatic porting is incomplete at present.

4.3. Analysis of Results

The same trends are seen in the results of both examples considered. With the smallest dataset the CPU implementation outperforms the accelerated implementations, this is because the dataset is too small for the speedup achieved to overcome the overhead of moving the dataset into the accelerator's memory.

As the size of the datasets increase the accelerated ports of the applications both begin to outperform the CPU version. The performance for the automatic port compared to the manual port for the GEMM example is very similar, showing the efficiency of the approach. The ClearSpeed accelerator initially ran faster than the GPGPU, then rapidly tailed off at a data size of 800, performed roughly 30% slower than the GPGPU in single precision. This appears to be reasonable given that ClearSpeed device performs double precision at the same speed as single precision (it is designed primarily for double precision).

However, ClearSpeed performed poorly in double precision mode when compared to the GPGPU; this was initially surprising, as a ClearSpeed device should show a slight performance edge given its higher GFLOP rating (96 double-precision GFLOPS compared to 78 GFLOPS on the C1060). It would appear to be caused by on/off device memory bandwidth, as the automatic port and the manual port do not make use of the on-chip swazzle network to reduce off-chip memory accesses (which supports 160Gbyte/sec on-chip, versus 6.4Gb/sec to the accelerator's 2Gb DRAM). For comparison, the NVIDIA C1060 has 102 Gbyte/sec memory bandwidth.

The N-Body problem shows a similar story, where with small datasets the ClearSpeed device performed faster than the GPGPU, but rapidly fell behind. Memory bandwidth would appear to be the issue, given the significant slowdown when moving from single to double precision, yet the raw GFLOP execution rate should be the same for both formats.

Overall, these results are very promising as they show that our porting system is producing code that outperforms the original CPU implementation and furthermore executes at a performance level only slightly depressed from that of manually ported code.

5. Future Work

The above results show the prototype system has performed well and is showing very promising progress towards its goals of increasing the programming abstraction level while maintaining acceptable performance. Future work will focus initially on completing the automatic porting to ClearSpeed devices.

The performance of automatic porting of additional example applications will be investigated and these examples will be selected from each category of the seven dwarfs of applications [1], allowing us to determine for each dwarf category if its applications are generally suitable or unsuitable for acceleration on a particular application accelerator device.

6. Conclusion

We have discussed the background to application accelerators in HPC and the obstacles to their wider adoption in the community. The main obstacle appears to be the lack

of a device-agnostic abstraction approach, for which we have presented our proposed solution: a semi-automatic porting system.

Our prototype solution has demonstrated the automatic porting of two C code examples (DGEMM and N-Body) that do not require the user to intervene or annotate their original source code. The results from these tests are promising with the automatically ported code outperforming standard, serial CPU implementations for moderate sized dataset and delivering performance within 15% of manually ported versions of the examples. Our results show that host-device memory bandwidth is a large factor with our two examples; further work aims to investigate other categories of the seven dwarfs to reveal which dwarfs are best suited to which application accelerators.

In summary, our system is a novel tool which has demonstrated promising initial results and as such will be of widespread interest to the HPC community.

References

[1] Krste Asanovic, Ras Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, University of California at Berkeley, 2006.

[2] Thomas H. Beach and Nicholas J. Avis. An intelligent semi-automatic application porting system for application accelerators. In *Proceedings of the ACM International Conference on Computing Frontiers, Workshop on UnConventional High Performance Computing*. ACM, 2009.

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Transactions on Graphics (TOG)*, 23:777–786, 2004.

[4] Francisco Chinchilla, Todd Gamblin, Morten Sommervoll, and Jan F. Prins. Parallel n-body simulation using gpus. Technical report, University of North Carolina at Chapel Hill, 2004.

[5] ClearSpeed. Clearspeed application note: Ground-breaking acceleration quantum chemical calculations using molpro. Technical report, 2007.

[6] ClearSpeed. Csx processor architecture. Technical report, 2007.

[7] Convey Computer Corporation. The convey hc-1 computer: Architecture overview. Technical report, 2008.

[8] Anders Dellson, Goran Sandburg, and Stefan Mohl. Turning FPGAS into supercomputers. Technical report, Cray Users Group, 2006.

[9] Khronos. Khronos launches heterogeneous computing initiative. Technical report, 2008.

[10] Mitrionics. Mitrion-accelerated ncbi blast. Technical report, 2007.

[11] NVIDIA. Nvidia tesla computing processor: Solve tomorrows computing problems today. Technical report, 2007.

[12] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general purpose computation of graphics hardware. *Eurographics*, 26:80–113, 2005.

[13] Rapidmind. Rapidmind product overview. Technical report, 2006.

[14] M. W. Riley, J. D. Warnock, and D. F. Wendel. Cell broadband engine processor: Design and implementation. *IBM Journal of Research and Development*, 51, 2007.

[15] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A manycore x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):Article 18, 2008.

[16] Jeffrey D. Ullman, Alfred V. Aho, and Ravi Sethi. *Compilers : principles, techniques, and tools*. Wokingham : Addison-Wesley, 1986.

[17] Ian H. Witten and Eibe Frank. *Data Mining : Practical machine learning tools and techniques*. Morgan Kaufmann, 2000.