Job Parallelism using Graphical Processing Unit Individual Multi-Processors and Localised Memory

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

April 2013

Abstract

Graphical Processing Units(GPUs) are usually programmed to provide data-parallel acceleration to a host processor. Modern GPUs typically have an internal multi-processor (MP) structure that can be exploited in an unusual way to offer semi-independent task parallelism providing the MPs can operate within their own localised memory and apply data-parallelism to their own problem subset. We describe a combined simulation and statistical analysis application using component labelling and benchmark it on a range of modern GPU and CPU devices with various numbers of cores. As well as demonstrating a high degree of job parallelism and throughput we find a typical GPU MP outperforms a conventional CPU core.

Keywords: GPU; task parallelism; data parallelism; hybrid parallelism; multi-processor.

1 Introduction

A great deal of the present research and developmental effort going into processor development is in increasing the number of cores that can be used concurrently on a single processor chip package. At the time of writing there are two complementary approaches being adopted. The first is addition of high capability central processing unit (CPU) cores, where each core presents computational capabilities to the applications programmer that individually appear very much the same as a traditional single core CPU. This approach is very much linked to the processor product development approach taken by Intel and AMD and at the time of writing is typified by devices with 4, 6, 8 core with recent devices announced fielding 16 and 32 such cores. The other approach is that typified by the GPU devices fielded by companies like NVidia. To a large extent the recent success of GPUs for general purpose (non graphical) programming has been due to the data parallelism possibilities offered by the large and rapidly growing number of simpler compute cores available. Recent GPUs have fielded 512 and 1536 such cores.

In this paper we explore the idea that one can also program GPUs in a manner closer to that of the traditional CPU core by focusing on the streaming multi-processors (MPs) and the resources available to them. A modern GPU has a broadly similar number of MPs as the number of compute cores on a modern CPU. In this respect it appears that vendors like Intel and NVidia are approaching the same problem but from different directions. This has interesting implications for future and hybrid devices.

We are interested in how one can use GPU MPs using a job parallelism approach. In separate work we have explored how jobs can be placed on completely separate GPU accelerators run by the same CPU host program, but in this present paper we explore independent jobs running on the MPs of a single GPU. There are a number of appropriate simulation models for which this is a powerful paradigm for enhancing throughput.

We present performance analysis based on an example such as simulating the 2D Game of Life (GoL) cellular automaton (CA) model [5, 7], but we also incorporate a sophisticated model analysis algorithm using cluster component labelling and histogramming [11].

Component labelling [6, 22] is a long standing problem of interest on parallel computers with a range of parallel approaches reported in the literature [19]. We have reported prior work of our own in achieving fast component labelling on a single GPU [8] where memory was not at a premium. In this present paper we include a report of our new work in achieving component labelling performed within the memory resources of a single MP.

Using bit-wise programming instructions and data structures we are able to cram a combined simulation model and its statistical component analysis software into the memory of individual MPs. The hosting CPU is thus able to manage independent jobs across all the MPs of its accelerating GPU device, and furthermore this can be extended if more than one GPU device is available.

The problem of managing job parallelism on modern multi-core devices is not a new one and much has been shown to depend of the efficiency and ease of programming using multiple threads of control [2, 3, 9, 10, 17]. These technologies are

typically needed to manage parallelism within a multi-cored CPU, with a language and environment such as NVidia's Compute Unified Device Architecture (CUDA) [13, 14] being used to program the GPU accelerator code. A coarser grain parallelism is also available across different host nodes altogether using hybrids of message passing and GPU accelerator techniques [23].

Our article is structured as follows: In Section 2 we summarise the simulation model and component analysis ideas that are central to this paper and which we aim to run independently upon the individual MPs of one or more GPUs. In Section 3 we review the various relevant technical characteristic GPU models which allow us to device various compact and bit-packing programming implementations which we describe in Section 4. We present a selection of performance results achieved with different GPU models in Section 5 and discuss the implications for this job parallelism paradigm in Section 6. We offer some concluding ideas and areas for further work in Section 7.

2 Model and Analysis Background

In this paper we focus on bit-wise models where there is a well-defined time-stepping procedure to update the bit-field based on a some localised calculation. A bit-field in d-dimensions can be analysed into its individual component clusters, which can be categorised by size and histogrammed appropriately. This is a particularly interesting analysis to incorporate into our performance testing since component labelling is no longer a localised calculation but must propagate information – about which site is connected to which cluster component – across the whole of the memory structure used for the bit-field. Although this approach applied to a wide class of complex systems simulation models, cellular automaton models are particular useful as concise benchmarks to discuss n this paper.

Cellular Automata (CA) models [1, 18] have long played an important role in exploring and understanding of the fundamentals of complex systems [21]. One classic CAs that provide a basis for much other work is Conway's Game of Life(GoL) [5]. There is a space of similarly formulated automaton rules [15] in the same family as GoL [12] but the Conway precise specification turns out to be particularly special in the rule set space of the family, having highly complex behaviour.

Much work has been done on studying the coherent structures that occur in GoL and its variants [4]. It is possible to implant specific patterns such as gliders, glider guns and so forth to obtain specific sequences of GoL automata configurations. However, in this present paper we investigate GoL automata systems that have been randomly initialised with an initial fraction of live cells. Providing we simulate a large enough sample of large enough automata systems, many different individual patterns can occur by chance, will interact and the system will eventually arrive a static or dynamical equilibrium.

The Game of Live is implemented using the Moore neighbourhood on a $d = 2$ dimensional array of cells, where the number of (Moore) Neighbouring sites $N_M = 8$, for $d = 2$. We define the (square) lattice Length as L and hence the number of sites N, typically $N = L^d$. We define the number of live sites N_L, and so the metric fraction $f_l = N_L/N$ and similarly the number of dead sites N_D, and fraction $f_D = N_D/N$.

Algorithm 1 Synchronous **automaton update** algorithm.

for all i, j in (L, L) **do**
  gather Moore Neighbour-hood $\mathcal{M}_{i,j}$
  apply rule $b[i][j] \leftarrow \{s[i][j], \mathcal{M}_{i,j}\}$
end for
for all i, j in (L, L) **do**
  copy $b[i][j] \rightarrow s[i][j]$
end for

In prior work [7] we have explored various simple metrics that can be applied to the GoL bit-field, but in this paper we focus on the size distribution of components of live cells as a function of time after random initialisation. Our long term interest is in building up histograms of this size distribution averaged over many independent configurations so we can track the time dependent behaviour to explore theoretical predictions such as Becker Doring [16].

We do not discuss the computational science and statistical mechanics aspects further in this present paper, but focus on the GPU and performance aspects using this applications model as a representative benchmark. The key aspects of the model as we employ it here are that although we only need a bit-field of $N = L^2$ elements for the simulation, we need an integer field to hold the working site labels when we perform the component analysis. This gives rise to various memory and size tradeoffs as described in Section 4 and the tradeoff space is different for different combinations of MPs and non-shared memory in the different GPU models described below.

3 GPU Architectural Background

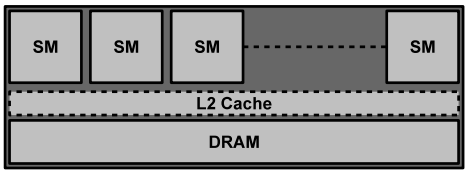

Figure 1: Overview of the GPU Architecture showing the role of the Streaming Multi-Processors (MPs). This scalable model easily allows additional MPs to be added to the GPU.

Graphical Processing Units have proved themselves as powerful and effective for large scale computational simulations. Acceptance of the GPU is shown by the number of GPU accelerated machines in the June 2012 TOP500 list [20]. Of the top 100 machines from this list, 15% are GPU accelerated, given the relatively recent rise of GPU computing this is a significant percentage.

GPUs can achieve a high computational throughput by provid-

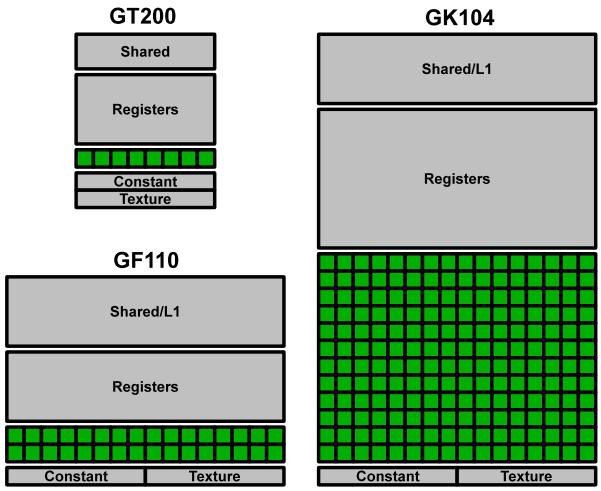

Figure 2: Streaming Multi-Processor Architecture showing three generations - Tesla (GT200), Fermi (GF110) and Kepler (GK104). It can be seen that although the number of cores on each MP has grown significantly in each generation

ing a large number of simple cores and small cache structures. This approach contrasts with modern CPU design which instead contains a small number of powerful cores that are kept busy by large cache hierarchies. GPUs also control thread management and scheduling in hardware which allows them to manage a large number of threads running at a time. Various thread scheduling techniques can be used to hide memory latencies and allow GPUs to perform efficiently when executing a large number of threads.

NVIDIA GPUs each contain a number of Streaming Multiprocessors (MPs), each of which contains a number of cores as well as registers and shared memory. The main memory area of a GPU is global memory which can be accessed by both the MPs of the GPU and also by the host through the PCIe bus. The Fermi and Kepler architecture GPUs also have an area of L2 cache that is shared between all of the MPs on the device The general GPU architecture can be seen in Figure 1. The main difference between the generations of GPU is the configuration of the MPs themselves.

Tesla generation (G80, GT200) MPs each contained 8 cores, either 32KB or 64KB of registers and 16KB of shared memory. The Fermi architecture (GF110) improved on this with 32 cores per MP with 128KB of registers and 48KB of shared memory. It also introduced an L1/L2 cache which automatically cached read/write transactions to global memory (the main device memory). The recently release Kepler architecture (GK104) GPUs contain SMX units (next generation MPs) with 192 cores, 256KB of registers and 48KB of shared memory. A illustration of the different MP configurations is shown in Figure 2.

At this point in most GPU articles, a detailed description of global memory accesses and the various caches for accessing global memory (constant, texture, L1/L2) would be described. However, this research takes a different approach to GPU-based computation. Instead of achieving high computational throughput by creating a large number of threads and processing a large data set like most GPU simulations, we instead investigate how GPU MPs can be used to process **small** data sets.

Traditionally GPUs struggle with small data sets because it limits the number of threads that can be active at once and the memory transactions to global memory cannot be hidden effectively. In this article we investigate how simulations with small data sets can be computed entirely on a single MP. This means the data set must fit into shared memory and the simulation computed entirely by a single block of threads. This obviously places strict restrictions on the maximum size of the system and presents a set of challenges not normally faced by GPU developers. The advantage of this approach is that the only slow global memory transactions are the writes to output the results of the simulation. All other memory transactions are through either the very fast registers or relatively fast shared memory.

Device Model (NVidia)	Multi Procs	CUDA Cores	Global Mem per GPU (MBytes)	GPU Clock (GHz)
GTX 260+	24	216	896	1.40
GTX 580	16	512	1,536	1.59
GTX 590	2x16	2x512	1,536	1.22
GTX 680	8	1536	2,048	0.71
M2075	14	448	5,375	1.15

Table 1: Configurations and relevant properties of the GPU devices.

In Table 1 we list the relevant memory and core configurations and clock speeds of the various GPUs we used. The GTX "Gamer" devices were hosted with Intel CPUs in conventional desktop computers and the M2075 was hosted in a blade unit with Xeon CPU host.

CPU Model (Intel)	CPU Cores	Cache (MBytes)	CPU Clock (GHz)
Q8200 Core2	4	4	2.33
X5675 Xeon	6	12	3.06
2600K Core-i7	4	8	3.40

Table 2: CPU relevant Properties.

Table 2 lists the relevant core numbers, cache sizes and clock speeds of the conventional CPU devices we experimented with.

4 Implementation Method

There are a number of important aspects to implementing the simulation and component labelling code on GPUs. In this section we discuss storage issues; the update algorithm; CUDA aspects; and the component labelling algorithm.

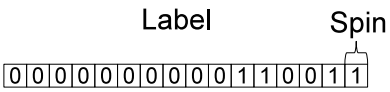

Figure 3: Bit-packing used to store a cell's spin and label in the same 16-bit unsigned short int. This cell is 'alive' and has the label 25.

4.1 Storage Requirements

The storage requirements for the Game of Life are relatively small, each cell has only two possible state - alive or dead. Thus a single bit per cell is sufficient to represent the system. For the different threads in the kernel to be able to access this system, it must be stored in shared memory - 16KB on a Tesla or 48KB on a Fermi/Kepler GPU. To make full use of this shared memory area to store for the state of the system, the state of the cells must be packed into an unsigned char, unsigned int or a similar data type. This means the maximum system size that can be stored in shared memory is $\approx 360^2$ on a Tesla and $\approx 620^2$ on a Fermi/Kepler GPU.

However, for this application there are additional storage requirement. Not only does the storage for the Game of Life need to fit into shared memory but also the space required for the component labeling. In order to support the number of labels required, an unsigned short int is used for each label. This data type gives a maximum of 2^{16} or 65536 labels. This limits the maximum system size to $\approx 150^2$ which gives a maximum of $22,500$ cells. In turn this means only 15 of the 16 bits in the short integer are required for the label. The extra bit can be used to store the spin for the Game of Life system. This is shown in Figure 3.

This allows both the labels and the Game of Life system to be stored in a lattice of unsigned short ints and give a maximum size of $\approx 150^2$ for a Fermi/Kepler device or $\approx 90^2$ for a Tesla device.

4.2 Update Method

Unfortunately this maximum storage size only takes the memory to store a single system into account. As the Game of Life uses a synchronous update method, two system states are usually used to compute the model. One is used to store the current state and to calculate the next state of each cell, this new state is written to the second system. However, this method immediately doubles the memory required to store a Game of Life system. For most implementations this would not cause a major problem, however in this case there is a very limited amount of memory available and another method must be used.

In this method only one full system is stored and two buffers are used to store rows of previous states. The system is updated one row at a time and writes the new value of each row directly updates the system. However, before each new row is written to memory the old value is copied to a buffer. The values from this buffer can then be used to calculate the update for the next row. Because this simulation uses periodic boundaries, another buffer is also required to store the original value of the first row as it is required for the calculation of the final row. This update method is illustrated in Figure 4.

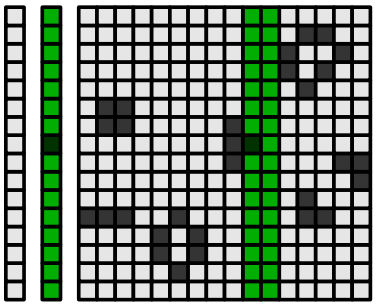

Figure 4: Update of a Game of Life system using buffers. The highlighted green rows are the rows currently being using in the update. This example shows how a blinker can be correctly updated by using the buffer to store the previous row.

4.3 CUDA implementation

Normally CUDA kernels create one thread for each cell, but unfortunately in this case it would also overrun the available registers. The easiest way to implement this update method in CUDA is to create one row of threads, each of which is responsible for update one column of cells. Before the update of the system begins, the first row and last rows are copied into the buffers. The buffer containing the first row will remain untouched until the last stage of the update while the other buffer will be continually updated. Each thread then reads three values from the buffer (representing the previous row) and the other six values from the system. Using these values it will calculate the new value of the cell. Once the new value has been calculated it will write the old value to the buffer and the new value to memory. All the threads must synchronize after each row has been updated to ensure that the buffer contains the correct values. This implementation is given in Algorithm 2.

However, this implementation will limit the number of threads to a maximum of L. This will limit the maximum performance of the implementation as the MPs use thread scheduling to hide memory latency. A better approach is to create a block of threads that process the field several rows at a time. In this implementation only the first row of the block will read from the buffer and only the last row will write to it. This allows a greater number of threads to run at once and makes better use of the MPs. The optimal number of rows updated by the block will depend on the total size of the system.

4.4 Connected Component Labelling

The other stage of the kernel is to label the connected components of the system. The same method of iterating through the cells used by the simulation can be used by the labeling phase. The labeling method used by this implementation is based on previous implementations and is outlined in algorithm 3.

This algorithm is used by both the GPU and CPU implementations albeit with some minor implementation modifications.

Algorithm 2 CUDA implementation of the buffer update method.

```
declare __shared__ s[L][L]
declare __shared__ b0[L]
declare __shared__ b1[L]
declare xm1=(threadIdx.x == 0) ? L-1 : threadIdx.x-1
declare xp1=(threadIdx.x == L-1) ? 0 : threadIdx.x+1
for t = 0 to t_e do
    copy s[0][threadIdx.x] to b1[threadIdx.x]
    copy s[L-1][threadIdx.x] to b0[threadIdx.x]
    for y = 0 to L do
        declare sum = 0
        sum += b0[xm1] + b0[threadIdx.x] + b1[xp1]
        sum += s[y][xm1] + s[y][xp1]
        if y == L-1 then
            sum += b0[xm1] + b0[threadIdx.x] + b0[xp1]
        else
            sum += s[y+1][xm1] + s[y+1][threadIdx.x] + s[y+1][xp1]
        end if
        syncthreads
        copy s[y][threadIdx.x] to b0[threadIdx.x]
        if sum == 3 then
            s[y][threadIdx.x] = 1
        else if sum < 2 or sum > 3 then
            s[y]threadIdx.x] = 0
        end if
    end for
end for
```

Algorithm 3 Component labelling algorithm outline.

```
declare change = true
for y = 0 to L do
    label[y][threadIdx.x] = y*L + threadIdx.x
end for
while changed == true do
    changed = false
    for y = 0 to L do
        ln = find lowest connected label in neighbours
        if ln < label[y][threadIdx.x] then
            label[y][threadIdx.x] = ln
            changed = true
        end if
    end for
    for y = 0 to L do
        declare l0 = label[y][threadIdx.x]
        while l0 != label[l0/L][l0 % L] do
            l0 = labe[l0];
        end while
        label[y][threadIdx.x] = l0
    end for
end while
```

Because the GPU implementation still has a degree of parallelism to it some synchronizations are required in order to ensure that the connected components are correctly labelled.

4.5 CPU Threading Implementation

In order to make a fair comparison and assessment of our approach on GPUs we also implemented the simulation and component labelling using a threading system across the (conventional) high capability cores of various CPU models. Intel's Threading Building Blocks [9] has been used to implement this multi-threading for the CPU implementations. Effectively the Game of Life simulations are distributed as jobs to the available cores on the CPU. This allows each simulation to remain as local as possible to the core and make best use of the available cache while still making use of the CPU cores.

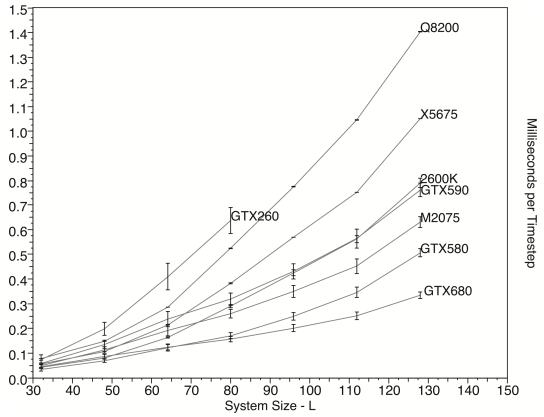

Figure 5: Comparison of CPU cores and GPU MPs computing a single Game of Life simulation with Connected Component Labelling.

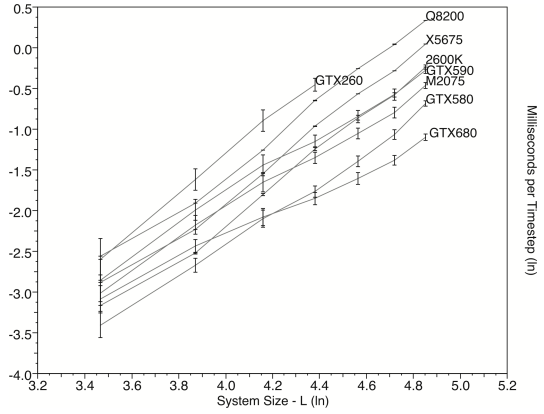

Figure 6: ln-ln scale comparison of CPU cores and GPU MPs computing a single Game of Life simulation with Connected Component Labelling, with least-squares fitted slopes between 1.8 and 2.1.

5 Performance Results

The performance of the GPUs described in Table 1 and CPUs from Table 2 have been compared for two different situations - computing a single Game of Life simulation and computing 100 simulations. The error-bars shown in the plots were obtained from standard deviations obtained by averaging time measurements over multiple runs with independently seeded initial conditions for the model.

The first comparison allows the performance of a single CPU core to be compared with a single MP of a GPU. This comparison is shown in Figure 5 both of which show the time taken by a single core of the CPUs and and single MP of the GPUs to compute a step of a Game of Life simulation with connected component labelling across various system sizes. This is also shown in log-log scale in Figure 6

The second compares the CPUs and GPUs as an entire processing architecture. It is expected that the GPUs will be able to provide higher performance when computing many simula-

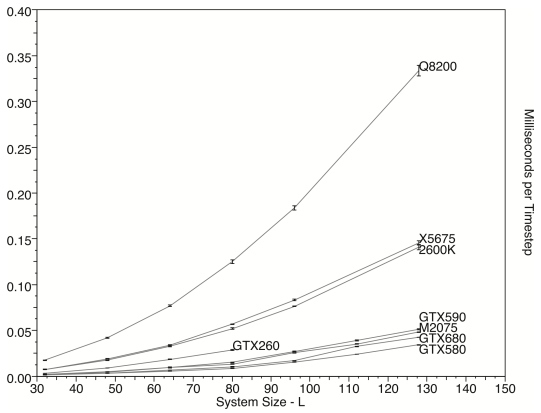

Figure 7: Comparison of CPUs and GPUs computing one hundred Game of Life simulations with Connected Component Labelling.

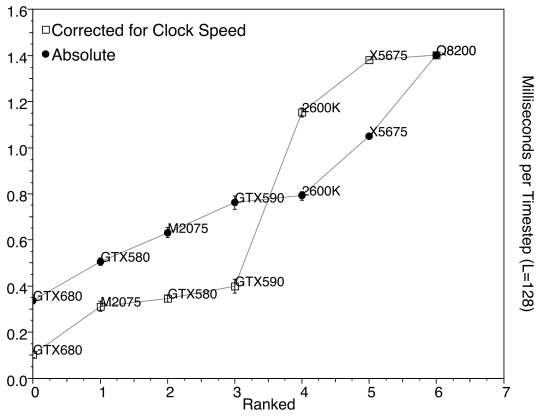

Figure 8: Comparison of CPUs computing one hundred Game of Life simulations with Connected Component Labelling.

tions as they generally contain many MPs compared to CPUs which generally contain 2-6 cores. This can be seen in Figure 7 which shows the time taken by all the available cores of the CPUs and all available MPs of the GPUs to compute a step of a Game of Life and label the connected components. Results are shown across a range of system sizes $L = 32...128$.

6 Discussion

The software architectural model we adopt is essentially a CPU master program that controls as many slave programs as practically possible given the available CPU cores, GPU devices and MPS available on the GPUs. Our outer loop therefore tries to schedule M_j slave jobs across N_{mp} multiprocessors or N_T threads across N_D devices on each of N_N individual nodes. There is therefore a significant choice in which level of parallelism to exploit to maximum the number of jobs that can be executed at once. In this paper we limit our investigation to a single GPU device, driven by a single hosting node, but there is obviously scope for deploying multiple nodes with multiple devices.

Figure 8 shows the mean timing result for the model system size $L = 128$ ranked in ascending time and shows how the various single CPU cores and single GPU MPs compared. The solid points plotted are the absolute times and show a steady increase in time from the fastest device we tested (the Kepler architecture GTX 680 GPU MP) up to the slowest (the Core2 CPU). As one might expect faster clock speeds in the CPU devices lead to a improved times. There is approximately a ratio of 3.5 between the times for slowest and fastest. Averaged over a larger number of jobs that can be scheduled across many CPU cores or many GPU MPs, improves this ratio even further to a factor of approximately 8.

If we attempt to normalise the times by clock speed, we see a marked separation of the data in Figure 8 into two very definite clusters - one for the GPU MP and one for the CPU core. This is plotted as the open data points, which again, we have ranked in order to highlight the changes. Obviously this approach only gives a very broad brush comparison between individual GPU MP and CPU core but within the two categories it does highlight some interesting effects.

One might expect the improved Kepler architecture GPU (680) with it greater number of cores to outperform the Tesla architecture devices (2075, 590 and 580 models). The M2075 has error-checked memory (which is why it needs a slower clock speed than the other Fermi devices - 580 and 590) and the M2075 generally outperforms the GTX devices on floating point calculations. Our benchmark application however primarily uses bit-wise, integer and memory manipulations, but the M2050 is still the best Fermi GPU architecture when clock speed is factored out.

Analysing the slopes from Figure 6 also shows some interesting results. Plotting the log of the time versus the log of the problem size L normally indicates the computational complexity exponent ν from the slope and the relationship $t \approx \mathcal{O}(L^{\nu})$. We obtain values for the exponent ν from inverse error-weighted least-squares fitting to the whole curves.

The slope of the CPU performance plots are all similar at ≈ 2.1. This is expected as the complexity of computing the Game of Life should increase with $\approx L^2$ but the connected component labeling should be slightly greater than L^2. However, the Fermi architecture GPUs all show a similar slope of $\approx 1.8 - 1.9$. This suggests that the computational power of the MPs is still being under utilized and that the GPU would be able to provide more efficient computation if a larger system size could be fit into shared memory. This is even more pronounced in the Kepler GPU which had a slope of ≈ 1.4.

While the GPU architectures provide very efficient computation they are still severely limited by the available memory and especially by the available cache. Unfortunately this problem appears to be amplified in the latest GPU architecture which has increase the number of cores per MP from 32 to 192 with no increase in shared memory.

7 Conclusion

We have described our use of streaming multi-processors of various NVidia GPU device models to implement independent job parallelism for accelerating complex systems simulations. We have showed that even quite sophisticated applications can be fit into the memory of an MP and that both the simulation itself as well as a more elaborate component analysis calculation can use this approach. We have also implemented the same application algorithm for individual single CPU cores scheduling across the multiple cores in Intel CPUs.

We have discussed the tradeoff space offered by the particular memory and MP mix available on current GPUs. We anticipate this space offering even more memory per MP on future devices and therefore this paradigm is likely to become viable for other application problems and algorithms. This approach is in contrast to the high successful data-parallel approach often used by applications programmers on GPU accelerators. We observe the broadly similar numbers of conventional CPU cores available from 2012 Intel chip offers and the number of MPs available on NVidia GPUs available in 2012. We postulate some convergence and continued growth of these numbers as hardware and production capabilities improve.

We observe that generally the present trend is still towards multi-core CPUs being somewhat easier to program than are GPUs using CUDA even when we have to use multi-threading code and a library like Intel TBB. However with appropriate parallel programming and bit-packing we obtained higher performance using the GPU MPs than from the CPU cores.

In this paper, we have focused on the MP parallelism with a single GPU. However, combining all sources of parallelism: from multiple host nodes in a cluster; where each node is itself a multi-cored CPU being accelerated by multiple GPUs; and where each GPU has multiple MPs; and each MP is capable of data-parallelism across its cores and threads, offers great potential for job thoughput speedup in the applications context we have described. Not all applications nor applications programmers will be able to or have the time and inclination to exploit all these levels of parallelism however. There is therefore scope for code generation and compiler directives and other tools to help achieve this. We expect to be able to deploy this approach for many of the complex systems applications of interest to us, extending the work to higher dimensional model systems and other forms of statistical measurements including time correlation analyses.

We expect vendors will continue to announce higher numbers of CPU cores and MPs in the course of the next few years and that the arena of programming language and associated software environment will be of growing importance for the continued exploitation of hybrid combinations of multi-core CPU and multi-processor GPUs by applications programmers. Finally, this work emphasises the current need to fit key data structures of applications into local uncontended fast memory wherever possible. GPU memory architecture appears to allow more opportunity for this with this application category.

References

[1] Adamatzky, A. (ed.): Game of Life Cellular Automata. No. ISBN 978-1-84996-216-2, Springer (2010)

[2] Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: Symp. Principles and Practice of Parallel Programming. pp. 207–216. ACM (1995)

[3] Bokhari, S., Saltz, J.: Exploring the Performance of Massively Multithreaded Architectures. Concurrency and Computation: Practice and Experience 22(5), 588–616 (April 2010)

[4] Evans, K.M.: Game of Life Cellular Automata, chap. Larger than Life's Extremes: Rigorous Results for Simplified Rules and Speculation on the Phase Boundaries, pp. 179–221. Springer (2010)

[5] Gardner, M.: Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life". Scientific American 223, 120–123 (October 1970)

[6] Hambrusch, S.E., Winkel, L.E.T.: A Study of Connected Component Labeling Algorithms on the MPP. In: Proceedings of 3rd Int. Conference on Supercomputing. vol. 1, pp. 477–483 (May 1988)

[7] Hawick, K.A.: Static and dynamical equilibrium properties to categorise generalised game-of-life related cellular automata. In: Int. Conf. on Foundations of Computer Science (FCS'12). pp. 51–57. CSREA, Las Vegas, USA (16-19 July 2012)

[8] Hawick, K.A., Leist, A., Playne, D.P.: Parallel Graph Component Labelling with GPUs and CUDA. Parallel Computing 36(12), 655–678 (December 2010), www.elsevier.com/locate/parco

[9] Intel: Intel(R) Threading Building Blocks Reference Manual. Intel (May 2010)

[10] Lamie, E.L.: Real-Time Embedded Multithreading - Using ThreadX and MIPS. No. ISBN 978-1-85617-631-6, Newnes - Elsevier (2009)

[11] Lumetta, S.S., Krishnamurthy, A., Culler, D.E.: Towards modeling the performance of a fast connected components algorithm on parallel machines. In: Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM). p. 32. ACM, New York, NY, USA (1995)

[12] Martinez, G., Adamatzky, A., Morita, K., Margenstern, M.: Game of Life Cellular Automata, chap. Computation with Competing patterns in Life-Like Automaton, pp. 547–572. Springer (2010)

[13] Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. ACM Queue 6(2), 40–53 (March/April 2008)

[14] NVIDIA® Corporation: NVIDIA CUDA C Programming Guide Version 4.1 (2011), http://www.nvidia.com/ (last accessed April 2012)

[15] de Oliveira, G.M.B., Siqueira, S.R.C.: Parameter characterization of two-dimensional cellular automata rule space. Physica D: Nonlinear Phenomena 217, 1–6 (2006)

[16] R.Becker, W.Doring: Droplet Theory. Ann.Phys. 24, 719 (1935)

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

[18] Sarkar, P.: A Brief History of Cellular Automata. ACM Computing Surveys 32, 80–107 (2000)

[19] Suzuki, K., Horiba, I., Sugie, N.: Fast Connected-Component Labeling Based on Sequential Local Operations in the Course of Forward Raster Scan Followed by Backward Raster Scan. In: Proc. 15th International Conference on Pattern Recognition (ICPR'00). vol. 2, pp. 434–437 (2000)

[20] TOP500.org: TOP 500 Supercomputer Sites. http://www.top500.org/, last accessed August 2012

[21] Wolfram, S.: Complex Systems theory. Tech. rep., Institute for Advanced Study, Princton, NJ 08540 (6-7 October 1984 1985), presented at Santa Fe Workshop on "A response to the challenge of emerging synthesis in science"

[22] Wu, K., Otoo, E., Suzuki, K.: Optimizing Two-Pass Connected-Component Labeling Algorithms. Pattern Anal. Applic. 12, 117–135 (2009)

[23] Yang, C.T., Huang, C.L., Lin, C.F.: Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. Computer Physics Communications 182, 266–269 (2011)