Computational Science Technical Note **CSTN-073**

Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA

D.P. Playne and K.A. Hawick

2009

Computational scientific simulations have long used parallel computers to increase their performance. Recently graphics cards have been utilised to provide this functionality. GPGPU APIs such as NVIDIA's CUDA can be used to harness the power of GPUs for purposes other than computer graphics. GPUs are designed for processing two-dimensional data. In previous work we have presented several two-dimensional Cahn-Hilliard simulations that each utilise different CUDA memory types and compared their results. In this paper we extend these ideas to three dimensions. As GPUs are not intended for processing three-dimensional data arrays, the performance of the memory optimisations is expected to change. Here we present several three-dimensional Cahn-Hilliard simulations to explore the challenges and the performance of the different memory types in three-dimensions. The results show that the simulation design with the best performance in three-dimensions uses a different memory type to the optimal two-dimensional simulation.

Keywords: GPU; CUDA; scientific simulation; field equation

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-073,
        author = {D.P. Playne and K.A. Hawick},
        title = {{Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation
                on GPUs with CUDA}},
        booktitle = {Proc. 2009 International Conference on Parallel and Distributed Processing
                Techniques and Applications (PDPTA'09)},
        year = {2009},
        pages = {104-110},
        address = {Las vegas, USA},
        month = {13-16 July},
        organization = {WorldComp},
        institution = {Massey University},
        keywords = {GPU; CUDA; scientific simulation; field equation},
        owner = {kahawick},
        timestamp = {2009.02.28}
}
```

Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA

D.P. Playne and K.A. Hawick

Institute of Information and Mathematical Sciences

Massey University – Albany, North Shore 102-904, Auckland, New Zealand

Email: {d.p.playne,k.a.hawick}@massey.ac.nz

Tel: +64 9 414 0800   Fax: +64 9 441 8181

February 2009

Abstract

Computational scientific simulations have long used parallel computers to increase their performance. Recently graphics cards have been utilised to provide this functionality. GPGPU APIs such as NVidia's CUDA can be used to harness the power of GPUs for purposes other than computer graphics. GPUs are designed for processing two-dimensional data. In previous work we have presented several two-dimensional Cahn-Hilliard simulations that each utilise different CUDA memory types and compared their results. In this paper we extend these ideas to three dimensions. As GPUs are not intended for processing three-dimensional data arrays, the performance of the memory optimisations is expected to change. Here we present several three-dimensional Cahn-Hilliard simulations to explore the challenges and the performance of the different memory types in three-dimensions. The results show that the simulation design with the best performance in three-dimensions uses a different memory type to the optimal two-dimensional simulation.

Keywords: GPGPU; CUDA; Cahn-Hilliard; Data-Parallel.

1 Introduction

With the release of several GPGPU APIs in recent years, utilising Graphical Processing Units or GPUs for scientific simulations has become increasingly popular [1–9]. GPUs provide a great deal of computational power compared to a traditional CPU. Programs that execute on a GPU can operate many times faster than their CPU counter-parts. However, this performance gain depends heavily on the ability to decompose the program into many threads that can execute independently of one another. High speed-up factors are also strongly affected by the memory type and access patterns used [10]. This article extends our previous research on GPU methods for simulating the Cahn-Hilliard (CH) equation [11] by extending the simulation to three-dimensions.

In Sections: 2 and 3 we discuss the capabilities and architecture of GPUs and the role of NVidia's CUDA [5] with specific focus on the GPU memory types it exposes. Sections: 4 and 5 provide a brief introduction to the Cahn-Hilliard field equation and the basic method of decomposing a CH simulation onto a GPU. Section: 6 describes 5 kernel designs (A-E) for simulating the Cahn-Hilliard equation in three-dimensions. And finally a discussion of the simulations, their results and a summary are presented in Sections: 7 and 8.

2 Graphical Processing Units

Graphical Processing Units or GPUs are specialised processors designed for calculating the complex graphics pipeline required for three-dimensional computer graphics in real-time. GPUs contain several multiprocessors, each of which contain many stream processors (at the time of writing a typical high-end graphics card contains 240 stream processors). These stream processors can execute instructions in parallel and provide GPUs with many times the computational power of a traditional CPU.

GPUs also contain several different types of memory that are optimised for specific tasks within the graphics pipeline. GPGPU libraries can utilise these types of memory for purposes other than those they were initially intended for. Choosing the memory type with

optimisations best suited to the requirements of a task can greatly increase the performance of a GPU program [10].

The low-cost and high-computational power that GPUs provide has lead to a rise in GPGPU in recent years. GPGPU or General-Purpose computation on GPUs aims to harness the power of GPUs for purposes other than computer graphics. One obvious application of GPGPU is in the area of scientific simulation. If a simulation can be decomposed into a GPU program, it can execute many times faster, allowing for larger simulations producing more accurate results. Writing a program on GPU is not as simple as for a CPU. Writing programs that execute on GPUs requires a GPGPU API [1–5]. There are several GPGPU APIs currently available, the API used in this research is NVIDIA's CUDA.

3   NVIDIA's CUDA

CUDA is a software environment that allows developers to write programs in a C style language that will execute on an NVIDIA GPU using the SIMT (single instruction multiple thread) paradigm [5]. This paradigm is similar to SIMD as each thread in a CUDA program will execute the same piece of code or kernel. CUDA can provide the best performance when the task is split into many threads organised into blocks, each containing a maximum of 512 threads each. The exact computation each thread performs will depend on its thread and block number. CUDA handles the organisation and scheduling of these threads which is described in detail in the CUDA Programming Guide [5].

CUDA Memory Types   This article is concerned with the types of memory that CUDA allows developers to access and how they affect performance for three-dimensional data sets. There are four types of memory in CUDA that can be used for these data sets, each of which have different optimal access patterns. Using the correct memory or combination of memory types is vital to achieving the optimal kernel performance. The memory types are:

Global Memory   is the largest type of device memory but also has the slowest access times (approx 200-400 ms). Although this is the slowest access time of any type of memory, there is a way to reduce the time required for several memory transactions. If 16 sequential threads in a thread block access 16 sequential and aligned addresses in memory, the 16 memory transactions can be combined into a single memory transaction with a process known as coalescing. Any memory transactions that fill these constraints will automatically be coalesced by CUDA.

Shared Memory   can only be accessed by threads within the same block. The host program cannot access shared memory and thus it cannot be used to read the values from the field. However, it can reduce the total number of global memory transactions required. As neighbouring cells in the field must access values from the same cells, there are many duplicated global memory reads. Each thread can read the value of its allocated cell from global memory and write it into shared memory. All neighbouring threads can then read this value from the fast shared memory as opposed to the slow global memory.

Texture Memory   is not exactly a separate type of memory but rather a cached method of accessing global memory. When a value is first read from texture memory, it will read the value and the surrounding values out of global memory into the texture cache. From this point on any value within this area can be read directly out of the fast texture cache. If there is a cache miss (ie the required value is not in the cache) the texture memory cache will be reloaded according to the new value. If all the threads in a block access value from within a close spatial locality, this texture memory can greatly decrease the time required for memory access.

Constant Memory   is similar to texture memory in that it caches values from global memory. However, the constant cache is optimised for multiple threads reading the same value at the same time. If all the threads within a half-warp (a set of 16 sequential threads [5]) access the same value from constant memory, the transaction will be as fast as accessing a value from registers. As this situation does not occur often within our simulation, the constant cache is not useful and no implementation makes use of it.

4   Cahn-Hilliard Equation

The Cahn-Hilliard equation models the phase separation and domain growth of a binary alloy by approximating the ration of A- and B-atoms within a discrete macroscopic cell. This equation is used by metallurgists to explore quenching methods and temperatures to produce the strongest alloys possible. The Cahn-Hilliard

equation can be simulated in any number of dimensions in real-space using finite differences [12] or in Fourier space [13]. This model is governed by the dimensionless equation:

$$\frac{\partial \phi}{\partial t} = m\nabla^2 \left(-b\phi + u\phi^3 - K\nabla^2\phi\right) \qquad (1)$$

A CH simulation (like most field-equations) is well suited to execution on a GPU. Both the CH simulation and image processing involve computations on regular arrays with regular access patterns. Like the operations involved in computing the graphics pipeline, updating the cells of the CH equation can be performed independently and is thus well suited to parallel execution. Optimising a GPU implementation of a CH simulation depends highly on the memory type and access patterns used. To understand how useful each type of memory will be to the simulation, the memory access requirements of the CH model must be understood.

Memory Access

The CH equation (equation 1) calculates the change in a CH cell's concentration value depending to the values of the cells surrounding it. The memory access requirements are determined by the Laplace and Laplace^2 operators ∇^2 and ∇^4. These stencil operators can be applied to any number of dimensions. The access requirements for one- two- and three-dimensions can be seen in Figure: 1.
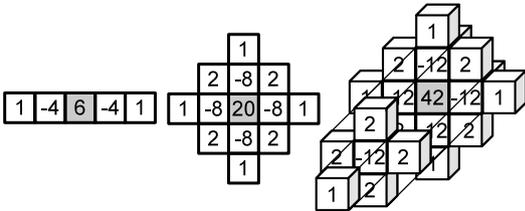

Figure 1: ∇^4 in one- two- and three-dimensions.

5 Two-Dimensional Cahn-Hilliard Simulations

In our previous work, it was found that the the most efficient memory type for a two-dimensional CUDA CH simulation is texture memory [10]. Texture memory caches all the values required by each block of threads which severely increased performance. However, as only global memory can be used to output data, each block of threads required a width of 16 so that all writes to output memory where coalesced. Using this method,

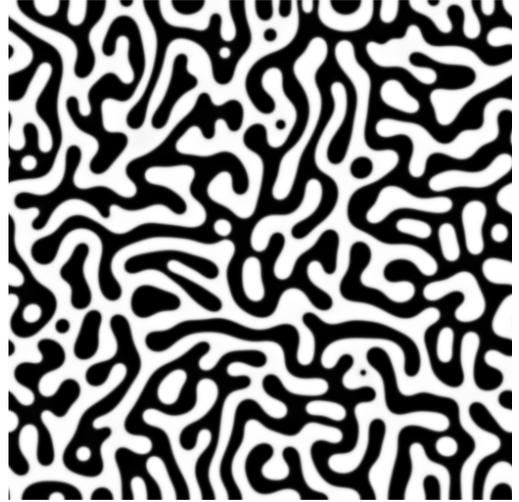a speed-up factor of 50x was achieved as compared to a traditional CPU simulation.

Figure 2: A visualisation of a 512x512 two-dimensional Cahn-Hilliard field at t = 131072.

As previously mentioned, these two-dimensional systems are very similar to the image processing tasks that graphics cards are optimised for. The texture memory was very well suited to the task and offered the best performance although shared memory did offer a speed-up factor of 30x as compared to the global memory speed-up of 13x. This research aims to examine how GPU memory optimisations scale to three (or potential higher) dimensions.

6 Three-Dimensional Cahn-Hilliard Simulations

While two-dimensional CH simulations are useful, three-dimensional simulations are more representative of a real-world metal alloy (see Figure: 3). However, these three-dimensional simulation are computationally more complex. Three-dimensional simulations are also harder to migrate onto GPU architectures. GPUs are optimised mainly for two-dimensional data processing and are thus less well-suited to processing three-dimensional fields. However, CUDA-enabled graphics cards do support some optimisations for three-dimensions such as three-dimensional texture memory [5].

There are several options available for computing a three-dimensional CH simulation with CUDA. Presented here are five different designs (Kernels A-E) for this CUDA simulation. Their results and performance gains are presented in Section: 7.
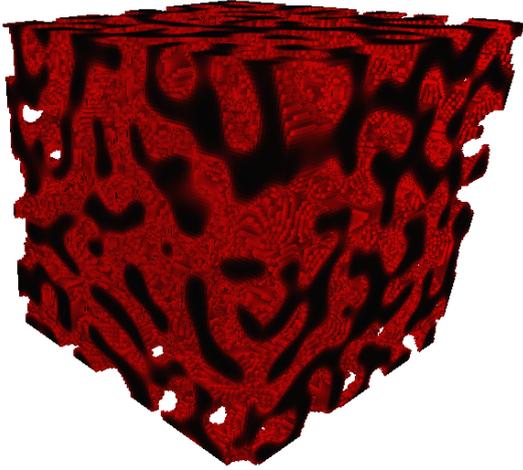
3

Figure 3: A visualisation of a three-dimensional Cahn-Hilliard field.

6.1   Global Memory Approach

This approach uses global memory for all storage requirements. It is important that the every set of 16 sequential threads access 16 sequential cell values to ensure that all aligned memory transactions are coalesced. As global memory is the only memory that can be written to, this is an important requirement for any kernel. It is expected that this approach should be faster than the CPU version but slower than the other kernels as it requires the most global memory transactions. However, it still represents a valuable baseline to demonstrate the importance of correct memory use.

Kernel A   makes use of global memory for all data, and each thread is responsible for calculating the change in concentration of one cell. Each thread will read one value at a time from global memory. Provided that the width of the thread block is a multiple of 16, all the aligned global memory accesses will be coalesced. A code sample of this kernel is shown in Listing: 1.

Listing 1: Code showing the input and output of Kernel A (the reading of only five values are shown here, the others are accessed in the same manner). (z,y,x) is the position of the thread's cell and (NZ, NY, NZ) is the size of the field.

```
//Read Input
float czyxm1 = global[z*NY*NX + y     *NX + x-1];
float czym1x = global[z*NY*NX + (y-1)*NX + x   ];
float czyx   = global[z*NY*NX + y     *NX + x   ];
float czyp1x = global[z*NY*NX + (y+1)*NX + x   ];
float czyxp1 = global[z*NY*NX + y     *NX + x+1];
...
//Perform Calculation
...
//Write output
global_output[z*NY*NX + y*NX + x] = result;
```

6.2   Shared Memory Approach

Shared memory can be used to minimise the number of global memory transactions required. In this approach, each thread reads one value from global memory and then the rest from shared memory. There is a problem with this method, the threads on the edge of the block cannot read all surrounding values from shared memory. This problem was overcome in two-dimensions by having a border of two threads around the edge of the block that simply read values into the shared memory but perform no computation themselves.

In a two-dimensional simulation, the thread block size chosen was 16x16 which still fits within the 512 limit. Of these 256 threads (16x16) and 144 are computational threads (12x12) after removing a 2 thread border required by the memory access of the model (see Figure: 1). This was found to work well in two-dimensions [10] as aligned global memory writes would still be coalesced (block width of 16), however this approach does not scale well to three-dimensions.

For our three-dimensional thread block, we wish to have width of 16 threads is applied to ensure coalesced memory access. However, the best arrangment that is 16 threads wide and fits within the 512 maximum is 16x5x5, which leaves 12 out of 400 as computational threads. This is inefficient and so methods to improve the ratio of computational to total threads were explored. Kernel B is the simple shared memory approach with 12 per 400 computational threads and Kernel B and C are two kernels that implement methods of improving this ratio.

Kernel B   uses the simple method of having one thread per cell and maintaining a block size of 16x5x5. This is the only thread arrangement that allocates one cell per thread and has coalesced global memory access. A section of code showing the the use of shared memory in Kernel B can be seen in Listing: 2.

Listing 2: Code showing the input and output of Kernel B. (tz,ty,tx) is the thread's id and (BZ, BY, BX) is the size of each thread block.

```
//Read Input
float crc    = global[z * NY * NY + y   * NX + x];
shared[tz*BY*BY + ty*BX + tx] = crc;
__syncthreads();
float czyxm1=shared[tz*BY*BX +  ty   *BX + x-1];
float czym1x=shared[tz*BY*BX + (ty-1)*BX + x   ];
float czyp1x=shared[tz*BY*BX + (ty+1)*BX + x   ];
float czyxp1=shared[tz*BY*BX +  ty   *BX + x+1];
...
//Perform Calculation
...
//Write output
global_output[z*NY*NX + y*NX + x] = result;
```

4

Kernel C makes more efficient use of shared memory by breaking the coalesced memory access constraint. All global memory reads/writes will not be coalesced. Without this constraint, the best way to arrange the threads for maximum computation threads is into a cube of size 8x8x8 which still fits within the 512 maximum. Out of these 512 thread, 64 (4x4x4) will be computational threads. The code for Kernel C will be identical to that of Kernel B (See Listing: 2) but the block size will simply be 8x8x8 rather than 16x5x5.

Kernel D increases the computational to total thread ratio by assigning multiple cells to each thread. Because each thread is responsible for multiple cells, the thread block does not need to be a three-dimensional structure. Like the two-dimensional simulation, each thread block is arranged in a 16x16 square structures. Each of these threads is responsible for 10 cells each. The 10 cells each thread is responsible for adds the three-dimensional element to the block and still allows good use of shared memory.

10 cells per thread may seem to be a strange size given that most sizes are a power of two; however, there is a sound logical reason for this choice. Of the 16x16 block of threads, only the internal 12x12 threads will be computational threads (like the two-dimensional version), thus the field size must be a multiple of 12. It would be desirable for the height of the computational thread block to also be 12; however, this would result in a 16x16x16 block of float values for the cells or 16KB. Although the shared memory of the GPU is 16KB [5], some of this memory is used by CUDA and not all 16KB can be accessed. The height of 10 threads yields 6 computational threads, which will fit perfectly into any multiple of 12 field size.

In this kernel, each thread will load 10 cell values into shared memory and then synchronise with the other threads to ensure all values have been correctly loaded. The threads will then go through and compute the change in concentration of the 6 internal cells and write the results to global memory. A code sample of this kernel can be seen in Listing: 3.

Listing 3: Code showing the input and output of Kernel D. (tz,ty,tx) is the thread's id and (BZ, BY, BX) is the size of each thread block. (TBZ,TBY,TBX) is the size of the inner block of computational threads.

```
//Read Input
for(int i = 0; i < BZ; i++) {
    int z = (bz * (TBZ) + i - 2);
    ...
    shared[i*BY*BX + ty*BX + tx] =
        global[z*NY*NX + y*NX + x];
}
```

```
__syncthreads();

for(int i = 2; i < BZ-2; i++) {
    float czyxm1 = shared[i*BY*BX + ty*BX + x-1];
    float czym1x = shared[i*BY*BX + (ty-1)*BX + x ];
    float czyp1x = shared[i*BY*BX + (ty+1)*BX + x ];
    float czyxp1 = shared[i*BY*BX + ty*BX + x+1];
    ...
    //Perform Calculation
    ...
    //Write output
    global_output[z*NY*NX + y*NX + x] = result;
}
```

6.3 Texture Memory Approach

Texture memory is designed for image processing when rendering a textured object. Multiple accesses to a texture within the same spatial locality are common. For this reason, the texture cache will store values of the texture within a spatial locale. This cache is mainly used in two-dimensions but is also supported for three-dimensional data.

The interface CUDA provides for interacting with three-dimensional textures is not as easy-to-use as in two-dimensions. Textures are bound to `cuda_arrays` example, the code to copy from global device memory to the cuda-array bound to a texture requires more explicit instructions. The global device memory used must also be a `cudaPitchPtr` which defines a pitch (in bytes) and a width and height in cells.

Although initialing correctly and interacting with three-dimensional texture memory is a complicated process, there are great advantages in utilising the texture memory cache. It was found in our previous work [10] that the texture memory was the most efficient type of memory for a two-dimensional Cahn-Hilliard simulation but this was in an application very close to the intended use of texture memory. It is unknown how well the performance of texture memory will scale to three-dimensions.

Kernel E uses three-dimensional texture memory to speed up the CH simulation. As the caching is performed entirely by CUDA, every thread within the thread block can be used to compute change in concentration values. This overcomes some of the issues that the shared memory approach faced with border threads. Listing: 4 shows a part of the code in kernel E that reads in values from memory.

Listing 4: Code showing the input and output of Kernel E (the reading of only five values are shown here, the others are accessed in the same manner). (z,y,x) is the position of the thread's cell and (NZ, NY, NZ) is the

size of the field.

```
//Read Input
float czyxm1 = tex3D(texData, x-1, y, z);
float czym1x = tex3D(texData, x, y-1, z);
float czyx   = tex3D(texData, x, y, z);
float czyp1x = tex3D(texData, x, y+1, z);
float czyxp1 = tex3D(texData, x+1, y, z);
...
//Perform Calculation
...
//Write output
global_output[z*NY*NX + y*NX + x] = result;
```

It is desirable that all the values accessed by the threads can fit within the texture cache so only one initial access to global memory is neccessary. As texture memory is relatively small (6-8KB) and there is no control (or even documentation) on the structure of the values loaded into the cache it is unknown on what the optimal arrangement will be for this thread block. There is still the constraint that the thread block be 16 threads wide to ensure the writes to global memory will be coalesced. To ensure the optimal results for this method, several block sizes where examined and the best results was with a thread block size of 16x4x4.

7 Results and Discussion

The results from testing these CUDA kernels are extremely favourable with the slowest GPU kernel providing a speed-up factor of 4x (See Figure: 4 and Table: 1). Unexpectedly, the slowest GPU kernel is Kernel B which uses shared memory. It was expected that Kernel A would be the slowest as all of its memory accesses are global transactions, however this kernel provides a speed-up of 18x. The advantage of utilising the shared memory in Kernel B was outweighed by the fact that most of its threads could not perform any real calculation and slowed the simulation significantly, once again showing the importance of proper use of memory. Interestingly the texture memory method (Kernel E) which was the fastest two-dimensional method does not scale well to three-dimensions. Even so, it was faster than Kernel A with a speed-up factor of 30x.

Table 1: The speed-up factors of the five GPU Kernels vs the CPU simulation.

Kernel	Speed-Up
A - Global	18x
B - Shared	4x
C - Shared	17x
D - Shared	50x
E - Texture	30x

The fastest GPU Kernel was the Shared Memory method Kernel D where each thread updated several

cells. It is believed that the direct control over the values cached and the high computational to total threads ratio is responsible for this kernel's higher performance. Kernel D provides a speed-up factor of 50x vs the CPU implementation, the same speed-up factor achieved by the texture method in two-dimensions [10].
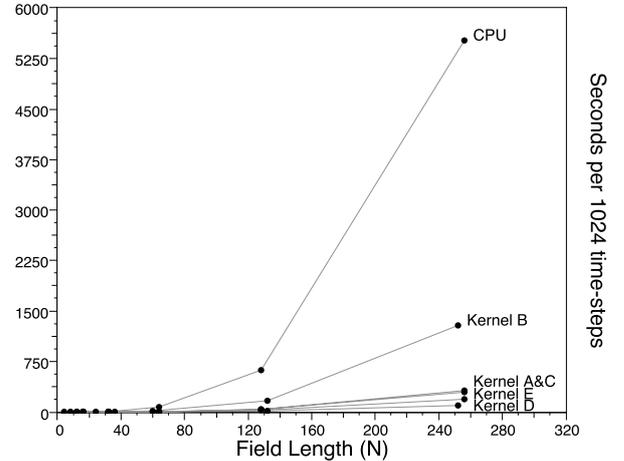

Figure 4: The performance results of the different kernels vs the CPU implementation.

In log-log scale, it is expected that all the performance of all the kernels will take a simple linear form (ignoring very small fields which are not large enough to utilise all the GPU cores). The time taken for this simulation is expected to be in the form $t = AN^d$ where d is the number of dimensions (in this case $d = 3$). It is thus expected that in ln-ln space this will form a straight line with slope $m = d$. This holds true for all the simulations which have a slope of $m = 3 \pm 0.02$ with the exception of Kernel E (See Figure: 5).
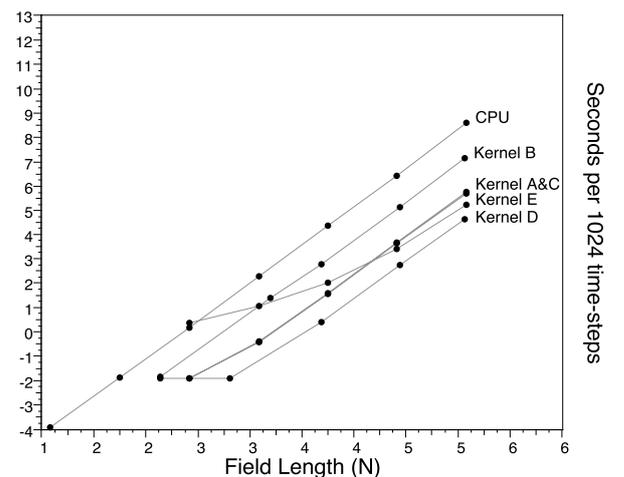

Figure 5: The performance results of the different kernels vs the CPU implementation in ln-ln scale.

The performance of Kernel D is strangely non-linear

and unfortunately our results only extend to a maximum field size of 256x256x256. Issues with display driver time-outs where encountered when larger systems were executed because each kernel call would take an increasingly long time to execute. Larger fields can be simulated by making multiple kernel calls on separate sections of the field, this extends the maximum field size but does not provide any more useful information with respect to the performance of the kernels.

These results show a marked difference between the performance of CUDA's memory types in two- and three-dimensions. Texture memory provided the highest performance for the two-dimensional CH simulation but was strangely non-linear in three. The API for three-dimensional texture memory was also markedly more complicated to use.

CUDA's limitation of 512 threads per block [5] did not cause a problem in two-dimension but severely disables several kernels in three-dimensions. It limits the effectiveness of caching methods as non-cube structures must be used if the block width is 16 to ensure coalesced memory accesses. The maximum size of the shared and texture cache also place restrictions on the kernels as they limit the number of values that can be cached. In two-dimensions all the values of a thread block could easily fit within the texture cache or shared memory, however the same is not true in three-dimensions. This shows once again the focus of GPU architectures towards processing two-dimensional data.

8 Summary & Conclusions

The results presented here and our previous results in [10] have shown a difference between the performance of the CUDA memory types in two-vs three-dimensions. Kernel designs where restricted by thread limits, cache sizes and coalesced memory access requirements. These kernels where more limited and more complex to design and implement than their two-dimensional counterparts which was reflected in their performance. These limitations will hopefully relax as the capabilities of GPUs increase but until such a time, programmers using CUDA for three-dimensional simulations must rely on more complex kernels to provide the high speed-up factors so easily enjoyed in two-dimensions.

References

[1] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware. ACM Trans. Graph. **23** (2004) 777–786

[2] AMD: ATI CTM Guide. (2006)

[3] McCool, M., Toit, S.D.: Metaprogramming GPUs with Sh. A K Peters, Ltd. (2004)

[4] Khronos Group: OpenCL (2008)

[5] NVIDIA® Corporation: CUDA^{TM} 2.0 Programming Guide. (2008) Last accessed November 2008.

[6] Fatahalian, K., Houston, M.: A Closer Look at GPUs. Communications of the ACM **51** (2008) 50–57

[7] Langdon, W., Banzhaf, W.: A SIMD Interpreter for Genetic Programming on GPU Graphics Cards. In O'Neill, M., Vanneschi, L., Gustafson, S., Alcazar, A.E., Falco, I.D., Cioppa, A.D., Tarantino, E., eds.: Proc. EuroGP. Volume LNCS 4971. (2008) 73–85

[8] Messmer, P., Mullowney, P.J., Granger, B.E.: GPULib: GPU computing in high-level languages. Computing in Science & Engineering **10** (2008) 70–73

[9] Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. ACM Queue **6** (2008) 40–53

[10] Leist, A., Playne, D., Hawick, K.: Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. Technical Report CSTN-065, Massey University (2008) To appear in Concurrency and Computation: Practice & Experience.

[11] Cahn, J.W., Hilliard, J.E.: Free Energy of a Nonuniform System. I. Interfacial Free Energy. The Journal of Chemical Physics **28** (1958) 258–267

[12] Hawick, K.A., Playne, D.P.: Modelling and visualizing the Cahn-Hilliard-Cook equation. In: Proceedings of 2008 International Conference on Modeling, Simulation and Visualization Methods (MSV'08), Las Vegas, Nevada (2008)

[13] Chen, L.Q., Shen, J.: Applications of semi-implicit Fourier-spectral method to phase field equations. Computer Physics Communications **108** (1998) 147–158