An Analysis of Programmer Productivity versus Performance for High Level Data Parallel Programming

Alex COLE [a], Alistair McEWAN [a] and Satnam SINGH [b]
[a] *Embedded Systems Lab, University Of Leicester*
[b] *Microsoft Research, Cambridge*

Abstract. Data parallel programming provides an accessible model for exploiting the power of parallel computing elements without resorting to the explicit use of low level programming techniques based on locks, threads and monitors. The emergence of Graphics Processing Units (GPUs) with hundreds or thousands of processing cores has made data parallel computing available to a wider class of programmers. GPUs can be used not only for accelerating the processing of computer graphics but also for general purpose data-parallel programming. Low level data-parallel programming languages based on the Compute Unified Device Architecture (CUDA) provide an approach for developing programs for GPUs but these languages require explicit creation and coordination of threads and careful data layout and movement. This has created a demand for higher level programming languages and libraries which raise the abstraction level of data-parallel programming and increase programmer productivity. The Accelerator system was developed by Microsoft for writing data parallel code in a high level manner which can execute on GPUs, multicore processors using SSE3 vector instructions and FPGA chips. This paper compares the performance and development effort of the high level Accelerator system against lower level systems which are more difficult to use but may yield better results. Specifically, we compare against the NVIDIA CUDA compiler and sequential C++ code considering both the level of abstraction in the implementation code and the execution models. We compare the performance of these systems using several case studies. For some classes of problems, Accelerator has a performance comparable to CUDA, but for others its performance is significantly reduced; however in all cases it provides a model which is easier to use and enables greater programmer productivity.

Keywords. GPGPU, Accelerator, CUDA, comparisons.

Introduction

The emergence of low cost and high performance GPUs has made data-parallel computing widely accessible. The hundreds or thousands of processing cores on GPUs can be used not only for rendering images but they may also be subverted for general purpose data-parallel computations. To relieve the programmer for thinking in terms of graphics processing architectural features (e.g. textures, pixel shaders and vertex shaders) the manufacturers of GPUs have developed C-like programming languages that raise the abstraction level for GPU programming beyond pixel and vertex shaders to the level of data-parallel operations over arrays. However, these languages still require the programmer to think in fairly low level terms e.g. explicitly manage the creation and synchronization of threads as well as manage data layout and movement and the programmer also has to write code for the host processor to manage the movement of data to and from the graphics card and to correctly initiate a sequence of

operations. For some programmers it may be important to control every aspect of the computation in order to achieve maximum performance and it is justifiable to expend significant effort (weeks or months) to devise an extremely efficient solution. However, often one wishes to trade programmer productivity against performance – i.e. implement a data-parallel computation in a few minutes or hours and achieve around 80% of the performance that might be available from a higher cost solution that takes more effort to develop.

Originally, writing data-parallel programs for execution on GPUs required knowledge of graphics cards, graphics APIs and shaders to set up and pass the data and code correctly. Over time, libraries and languages were developed to abstract these details away. The Accelerator library from Microsoft [1] is one such library. Accelerator provides a high level interface to writing data parallel code using parallel array objects and operations on those arrays. This interface hides target details, with the array objects and operations representing generic data so that Accelerator can be retargeted with little effort. Note that a "target" is the device on which code is run and Accelerator supports more than just a GPU. A Just In Time compiler (JIT) is used to convert high level descriptions to target-specific instructions at run time.

As the popularity of General Purpose GPU computing (GPGPU) increased, GPU manufacturers started to develop systems with dedicated hardware and associated GPGPU software such as NVIDIA's CUDA [2]. Older methods encoded data and code in graphics terms and ran through the full graphics pipeline, including parts which were only relevant to graphics. These newer systems provide direct access to the major processing elements of a Graphics Processing Unit (GPU) and a native programming model.

Although these systems provide more direct access, they have returned to requiring lower level knowledge of the (non-graphics specific) hardware. On the other hand, more abstract systems still require no detailed knowledge. CUDA code is often tightly matched to a device on which it is to run for optimisation purposes, Accelerator code can be retargeted to different classes of hardware very quickly. The question is how do these systems compare, and how do they compare to no data parallel code at all (i.e. sequential C++ code)? Are any penalties incurred by using the JIT in Accelerator, and how do development efforts compare for similar performances? Do the gains outweigh any penalties or is the abstraction simply too high?

This paper presents an overview of the history of data parallelism with a focus on GPGPU in Section 1. It then examines Accelerator and compares it with CUDA on a GPU and sequential C++ code on a multi-core Central Processing Unit (CPU). These comparisons are performed for a number of case studies (Section 2), namely convolution (Section 3) and electrostatic charge map estimation (Section 4). Each case study includes an introduction to the algorithm, results and a conclusion; with the paper finished by a discussion on development (Section 5) and conclusions (Section 6). The following contributions are made:

- A comparison of the programming models used in CUDA, Accelerator and C++ code executed on a regular processor.
- A demonstration that very good speed-ups can be obtained using Accelerator from descriptions that are high level and easier to write than their CUDA counterparts.
- A demonstration that the Accelerator model provides a higher level technique for data-parallel computing that permits good performance gains with a greater degree of programmer productivity than CUDA.

1. Background

1.1. Data Parallelism

Data-parallel programming is a model of computation where the *same operation* is performed on every element of some data-structure. The operation to be performed on each element is

typically sequential although for nested data-parallel systems it may itself be a data-parallel operation. We limit ourselves to sequential operations. Furthermore, we limit ourselves to operations that are *independent* i.e. it does not matter in which order we apply the operation over the data-structure. This allows us to exploit data-parallel hardware by performing several data-parallel operations *simultaneously*. Key distinguishing features of such data-parallel programs is that they are *deterministic* (i.e. every time you run them you get the same answer); they do not require the programmer to explicitly write with threads and locks and synchronization (this is done automatically by the compiler and run-time); and the programmer's model of the system in essence needs only a single 'program counter' i.e. this model facilities debugging.

This technique allows us to perform data-parallel operations over large data sets quickly but requires special hardware to exploit the parallel description. There are multiple types of data parallel systems, including Single Instruction Multiple Data (SIMD) and Single Program Multiple Data (SPMD). The former is a single instance of a program operating on multiple sets of data at once; an example of this is the Streaming SIMD Extensions (SSE) instruction set on modern x86 processors. The latter is multiple instances of the same program running in parallel, with each instance operating on a subset of the data. The former performs multiple operations in lockstep, the latter may not. One important aspect of data parallel code is the independence of the data—performing the required calculation on one section of the data before another section should give the same results as performing the calculations in a different order.

1.2. Graphics Processing Units

To render an image on a screen may require the processing of millions of pixels at a sufficiently high rate to give a Frames Per Second (FPS) count which is smooth to the eye. For example a 60Hz 1080p HDTV displays over 124 million pixels per second. This is a massively compute intensive task which involves converting 3D scene descriptions to 2D screen projections. A GPU performs this task in parallel by utilising a highly specialised processing pipeline which can apply custom rendering effects in the form of shaders to large numbers of pixels. This GPU can be part of a dedicated graphics card or integrated on to the main board of a PC, laptop or even now phones. The processing pipeline of the GPU, and the memory in dedicated systems are well suited to high data throughput with emphasis on the number of parallel operations rather than the time for one operation. The memory system may take hundreds of clock cycles to complete a read request but then delivers a batch of data at once. The memory system can have multiple requests in various stages of completion at once.

Older GPUs contained multiple user programmable stages within the pipeline for custom processing (called "shaders"). One for calculating custom lighting and colouring effects across a complete primitive (triangle) and another for calculating per-pixel effects. The operations within these stages are relatively simple when compared to the capabilities of many modern CPUs; however, one GPU will contain hundreds or thousands of these simple cores (the NVIDIA GeForce GTX480 contains 480 [3], the ATI Radeon HD 5970 contains 3200 [4]). Each core is assigned a fraction of the total workload, leading to the desired high throughput. More modern GPUs still contain this configurable ability but the range of operations available to each stage has increased, leading to a convergence allowing the stages to be combined into a "unified shader".

1.3. General Purpose Graphic Processing Unit Computing

GPGPU is using a GPU for calculations other than just graphics, i.e. *general purpose* calculations [5,6]. This field largely took off with the advent of configurable graphics cards, though some work had been done as far back as 1978 [7] and more recently using fixed function

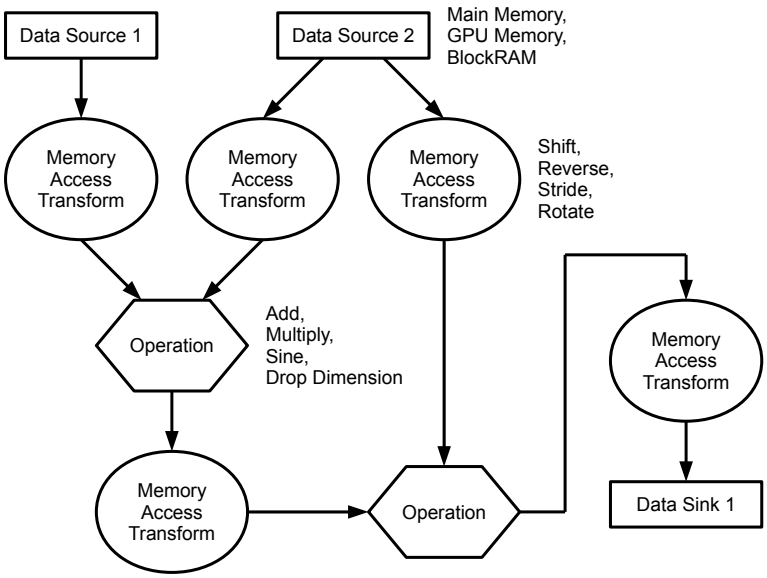

Figure 1. Expression Graph

pipelines (those without shaders) [8,9,10]. These configurable cards allowed a great deal of transformations to be applied to data, though only when presented as graphics data and graphics shader operations. This involved converting data to textures and operations to shaders, then passing both to a standard graphics pipeline to render an output which was saved as the result. Many systems such as Sh [11], Brook [12] and Scout [13] were developed to abstract the graphics APIs used for the processing. These systems meant that users were not required to learn about the underlying hardware and this is where Microsoft's Accelerator library [1] is targeted.

More modern GPGPU developments use a *unified shader* architecture. As the capabilities of the various pipeline shader stages increased their abilities largely converged, allowing the physical hardware for the stages to be combined into a single block of cores. This allowed for better load balancing and design optimisation, but also led to the development of direct access systems which better utilised this shader for GPGPU. This model allows a programmer to pass data and code to the GPU shaders more directly, removing the reliance on the graphics pipeline and graphics concepts. Direct access systems include CUDA [2] by NVIDIA, Close To Metal (CTM)/Stream by ATI (discontinued), DirectCompute by Microsoft and OpenCL by the Khronos group. These systems are still very low level. To get the best performance requires writing code to take into account low level designs issues such as memory location and thread layout.

1.4. Accelerator

Accelerator is based around a collection of data-parallel arrays and data-parallel operations which are used in a direct and intuitive style to express data-parallel programs. The Accelerator system performs a significant number of powerful optimizations to produce efficient code which is quickly JIT-ed into GPU code via DirectX or into SIMD SSE4 code using a customized JIT-er. The data-parallel computation to be instantiated on a target like an FPGA is represented as an expression tree that contains nodes for operations and memory transforms (e.g. see Figure 1). The Accelerator system supports several types of data-parallel arrays (floating point, integer, boolean and multi-dimensional arrays) and a rich collection of data-parallel operations. These include element-wise operations, reduction operations and rank changing operations. A very important aspect of Accelerator's design is the provision of

Section $(b_i, c_i, s_i, b_j, c_j, s_j)$	$R_{i,j} = A_{b_i} + s_i \times i, b_j + s_j \times j$
Shift (m,n)	$R_{i,j} = A_{i-m,j-n}$
Rotate (m,n)	$R_{i,j} = A_{(i-m) \bmod M,(j-n) \bmod N}$
Replicate (m,n)	$R_{i,j} = A_{i \bmod m,\ j \bmod n}$
Expand (b_i, a_i, b_j, a_j)	$R_{i,j} = A_{i-b_i \bmod M,(j-b_j) \bmod N}$
Pad (m, a_i, m, a_j, c)	$R_{i,j} = \begin{cases} A_{i-m,j-n} & \text{if in bounds} \\ c & \text{otherwise} \end{cases}$
Transpose(1,0)	$R_{i,j} = A_{j,i}$

Figure 2. Examples of transform operations of size $M \times N$ arrays

operators that specify memory access patterns and these are exploited by each target to help produce efficient vector code, GPU code or FPGA circuits. Examples of memory transform operations are shown in Figure 2.

Even on a low end graphics card, it is possible to get impressive results for a 2D convolver. All 24 cores of a 64-bit Windows 7 workstation are effectively exercised by the x64 multicore target, which exploits SIMD processor instructions and multithreading. Stencil-style computations [14] are examples of problems that map well to Accelerator.

As a concrete example, we show a very simple F♯ Accelerator program that performs the point-wise addition of two arrays using a GPU (Listing 1). When executed, this program uses the GPU to compute a result array containing the elements 7; 9; 11; 13; 15. To perform the same computation on a multicore processor system using vector instructions, we write the same program but specify a different target (Listing 2).

```
open System
open Microsoft.ParallelArrays
[<EntryPoint>]
let main(args) =
    let x = new FloatParallelArray
        (Array.map float32 [|1; 2; 3; 4;  5 |])
    let y = new FloatParallelArray
        (Array.map float32 [|6; 7; 8; 9; 10 |])
    let z = x + y
    use dx9Target = new DX9Target()
    let zv = dx9Target.ToArray1D(z)
    printf "%A\n" zv
    0
```

Listing 1. F♯ Accelerator code targeting GPU.

```
use multicoreTarget = new X64MulticoreTarget()
let zv = multicoreTarget.ToArray1D(z)
```

Listing 2. F♯ Accelerator code targeting X64 CPU *(only the two changed lines are shown).*

The FPGA target does not work in an on-line mode and does not return a result. Instead it generates VHDL circuits which need to be implemented using FPGA vendor tools. A key point here is that we can take the same computation and instantiate it on three wildly different computational devices.

Accelerator running on the GPU currently uses DirectX 9, which introduces a number of limitations to the target. First is the lack of data-type support, with only Float32 data-types that are not quite compliant with IEEE 754 (the floating point number standard). Secondly the code is quite limited in performance by both shader length limits, which restrict the amount of

code which can be run; and memory limitations. In DirectX 9 there is no local shared memory as in CUDA, limited register files and limited numbers of textures in which to encode input data.

1.5. CUDA

NVIDIA CUDA provides direct access to an NVIDIA GPU's many hundreds or thousands of parallel cores (termed the "streaming multiprocessor" in the context of GPGPU), rather than being required to run code through the graphics pipeline.

In CUDA one writes programs as functions (called kernels) which operate on a single element, equivalent to the code in the inner loop of sequential array code. Threads are then spawned, one for every element, each running a single kernel instance. When a program is executed through CUDA on the GPU the programmer first declares how many threads to spawn and how they are grouped. This allows the system to execute more threads than there are cores available by splitting the work up into "blocks". The low level nature of CUDA allows for code to be very highly tailored to the device it is running on, leading to optimisations such as using thread local memory (which is faster than global memory), or configuring exactly how much work each thread does. The CUDA kernel code to add two arrays, equivalent to the code in Section 1.4, is shown in Listing 3. This code adds a single set of elements together after first determining the current thread ID for use as an index.

```
__global__ void
    DoAddition(float aCudaA[], float aCudaB[],
        float aCudaTot[], int iSize)
{
    const int
        index = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (index < iSize)
    {
        aCudaTot[index] = aCudaA[index] + aCudaB[index];
    }
}


void
    main()
{
    float
        arrayOne[5] = {1, 2, 3, 4, 5 },
        arrayTwo[5] = {6, 7, 8, 9, 10},
        arrayOut[5];
    Link(arrayOne, arrayTwo, arrayOut, 5);
}


extern "C" void
    Link(float a1[], float a2[], float ao[], int size)
{
    void *
        cudaArrayOne = 0,
        cudaArrayTwo = 0,
        cudaArrayOut = 0;
    // Allocate GPU memory for the arrays.
    cudaMalloc(&cudaArrayOne, size);
    cudaMalloc(&cudaArrayTwo, size);
    cudaMalloc(&cudaArrayOut, size);
```

```
    // Copy the input data over.
    cudaMemcpy(cudaArrayOne, a1, size, cudaMemcpyHostToDevice);
    cudaMemcpy(cudaArrayTwo, a2, size, cudaMemcpyHostToDevice);
    // Call the GPU code from the host.
    dim3
        dimBlocks(1),
        dimThreads(size);
    DoAddition<<<dimBlocks, dimThreads>>>(
        cudaArrayOne, cudaArrayTwo, cudaArrayOut, size);
    // Save the result.
    cudaMemcpy(ao, cudaArrayOut, size, cudaMemcpyDeviceToHost);
}
```

Listing 3. CUDA addition code

CUDA code has three parts—host code, which runs on the CPU and is regular C/C++; link code, which invokes a kernel and has custom, C-based, syntax; and the kernel code itself, which also has custom syntax. The example code shows the kernel first ("`DoAddition`"), then the host code ("`main`") and finally the link code ("`Link`"). In this instance the link code initialises the CUDA arrays and spawns many instances of the kernel function, each of which calculates a single output result based on it's unique thread and block ID. The "`DoAddition<<<dimBlocks, dimThreads>>>`" code is custom CUDA syntax and requires a special compiler; however, everything else in that function is regular C code and could equally be placed in the device code.

2. Case Studies

The CUDA programming model is designed to allow maximum flexibility in code, requiring in-depth target knowledge but allowing for lots of optimizations. Conversely the Accelerator programming model provides easy to use, high-level access to data but includes a JIT, which is an overhead not present in CUDA. We believe that despite this Accelerator gives reasonable speed for many classes of problems and we have instrumented the overhead of the JIT and found it to be very small (less than 3%) for realistic workloads. We also believe that the development effort involved is lower in Accelerator and so justifies some reduced performance. Indeed, it may be possible for the JIT-bases scheme to produce *faster* code because it can exploit extra information about the execution environment which is only available at run-time. The motivation behind this work was to find out just how much of an improvement or overhead Accelerator has compared to other data parallel and sequential programming models.

Two case studies were used to test Accelerator's performance against other systems: convolution and electrostatic charge map generation. Each algorithm was run in CUDA on a GPU, C++ code on a CPU and Accelerator on both a GPU and a CPU. The Accelerator tests were run on both platforms using the same code for two different implementations. The CUDA tests were run for two different implementations and the C++ tests for three. These studies were run in a common framework to share as much code as possible and reduce the areas which could affect timings. The experiments were run on an AMD 1055T CPU at 2.8GHz (the Phenom II X6) with 8Gb of DDR3 RAM and an NVIDIA GTX 460 GPU (the Palit Sonic) with 2Gb of GDDR5 RAM. Every individual test was run ten times for every target with the results displaying the totals for all ten runs. Memory and initialisations were all reset between every run.

The case studies are aimed at determining the speed ups available for a reasonable amount of effort. While it is technically possible to write code by hand that will match any-

thing generated by a compiler the effort involved is unreasonable. JIT compilers can use advanced knowledge of the data to be processed through techniques such as branch prediction and memory layout optimisation, writing generic code by hand to exploit such dynamic information is very difficult although some optimisations can still be applied. The optimisations applied to CUDA code were limited to those found in the Programming Massively Parallel Processors [15] book. This was assumed to be a reasonable estimation of the ability of a non-specialist.

3. Convolution Case Study

3.1. Introduction

Convolution is the combination of two signals to form a third signal. In image processing a blur function is a weighted average of a pixel and its surrounding pixels. This is the convolution of two 2D signals—an input image and a blur filter which contains the weightings. In a continuous setting both signals are theoretically infinite. In a discrete setting such as image processing both signals are clipped. Figure 3 shows a 1D convolution example. The filter (`a`) is applied to the current element "7" of the input (`m`) and its surrounding elements (highlighted). These are multiplied to give array `ma` and all the elements are summed together to give the current element in the output array (`n`). This is shown generically in Equation (1) where $\mathtt{m}_\mathtt{t}$ and $\mathtt{n}_\mathtt{t}$ are the current input and output points respectively, `N` is the filter radius and `a` is the filter.

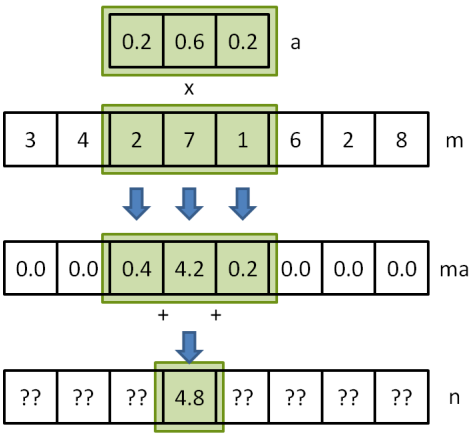

Figure 3. Convolution of a radius 1 1D filter and an 8 element input array with one operation highlighted.

The filter (Equation (2)) used in this study is a discretised Gaussian curve, rounded to zero beyond the stated radius (generally very small values). The "radius" of the filter represents the number of non-zero values—a 1D filter with a radius of 5 will yield 11 total non-zero values, a (square) 2D filter of radius 5 will yield 121. As it is a constant for every element in the input signal, the filter values are pre-computed prior to running any timed code. This is a part of the previously discussed common test framework and ensures that all the implementations run off the same initial data.

$$n_t = \sum_{k=0}^{2N+1} a_k m_{(t+k-N)} \tag{1}$$

$$a_k = \left(\frac{(k-N)^2}{2\sigma^2} \right) / \left(\sum_{i=0}^{2N+1} \frac{(i-N)^2}{2\sigma^2} \right) \tag{2}$$

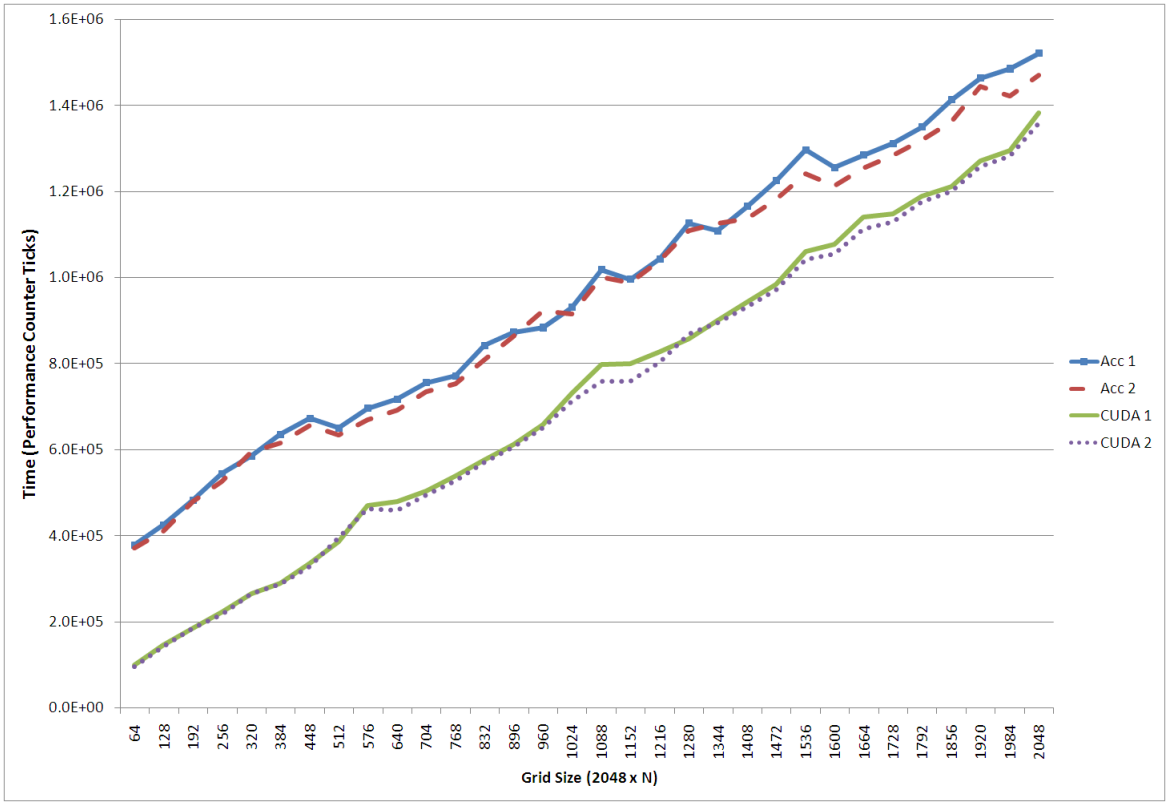

Figure 4. Convolution case study *GPU* results for a radius 5 filter on grids from 2048×64 to 2048×2048 elements.

A 2D convolver has a 2D input and a 2D filter (in this case a square of width $2N + 1$). A separable convolver is a special case of 2D convolver in which the result is the same as applying a 1D convolution to every row of original input and to every column of those results applying a second 1D convolution. A radius five filter on a 2D input array implemented as a separable convolver would require only $11 + 11 = 22$ calculations per output, compared to $11 \times 11 = 121$ for basic 2D convolution.

When calculating results close to the edge of the input data the radius of the filter may extend beyond the limits of known data, in this case the nearest known value is used. This zone is called the "apron", as is the zone of input data loaded but not processed by CUDA when splitting the data up into processing chunks [16]. Accelerator does not permit explicit array indexing so convolution is implemented as whole array operations using array shifts where every element in the input array has the effect of one element in the filter calculated in parallel.

3.2. Results

Figures 4 and 5 show the results for a 2D separable convolver using a radius five Gaussian filter (Equation (2)) on grids of size 2048×64 to 2048×2048. Figure 4 shows the results for the various GPU targets ("Acc" is "Accelerator"). Both Accelerator and CUDA ran two different implementations. Figure 5 shows the CPU target results with "C++" using three different implementations of varying complexity.

3.3. Discussion

For the convolver the Accelerator GPU version is only marginally slower than the CUDA version. The CUDA code used here was based on public documentation [16] which included optimisations based on loop unrolling and usage of low latency shared memory. While the

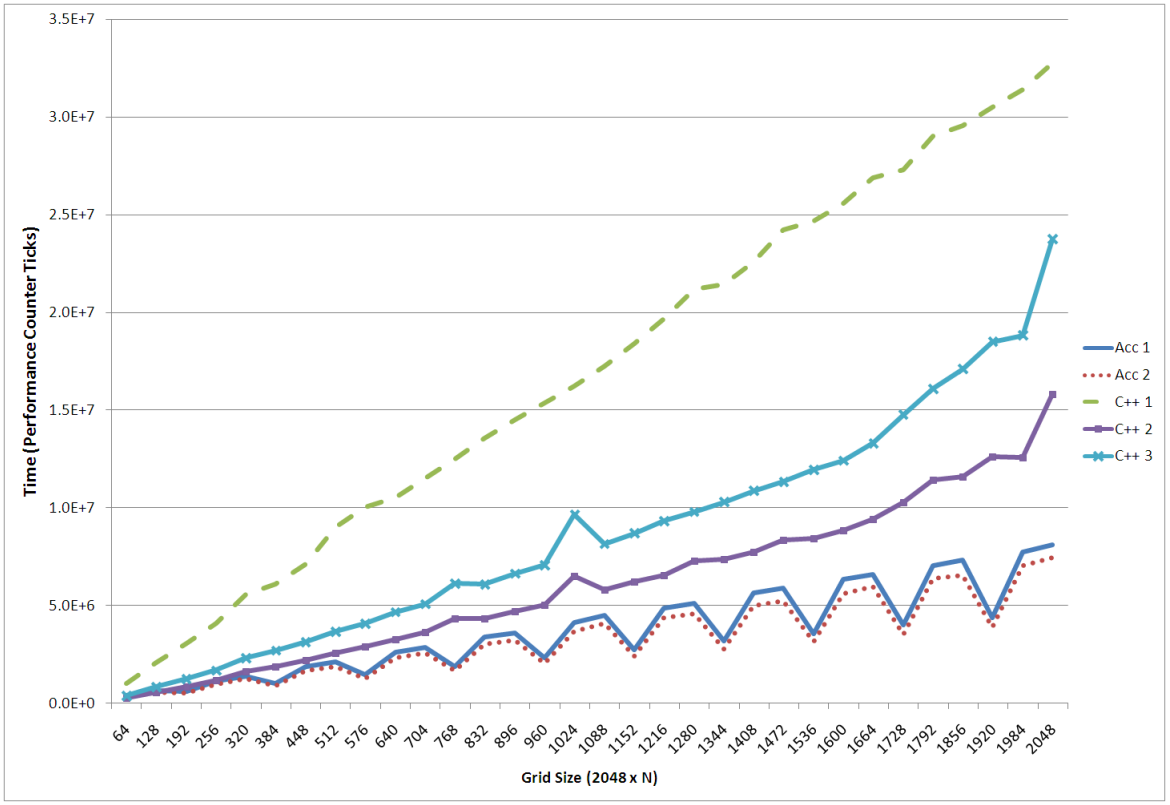

Figure 5. Convolution case study *CPU* results for a radius 5 filter on grids from 2048×64 to 2048×2048 elements.

speed is slower the development efforts here are significantly different. Examples of the code required to implement a convolver in Accelerator, C++ and CUDA can be found in Appendix A. Additional examples can be found in the Accelerator user guide [17] and CUDA convolution white paper [16]. One point to note, clearly visible on the GPU graph, is the constant overhead from the Accelerator JIT.

The performance of Accelerator on the CPU was significantly better than the original C++ sequential code ("C++ 1") and slightly better than the more advanced versions ("C++ 2" and "C++ 3"). These versions performed the apron calculations separately from the main calculations, rather than using a single piece of generic code. Generic code requires branches to clamp the requested input data to within the bounds of the available data. The Accelerator code here was between two and four times faster than the C++ versions, and with significantly less development effort than "C++ 2" and "C++ 3". Both Accelerator CPU implementations display a very interesting and consistent oscillating graph which requires further investigation.

All the alternate code versions ("Acc 2", "CUDA 2", "C++ 2" and "C++ 3") rely on the fact that the filter in use (a Gaussian curve) was symmetrical and so performed multiple filter points using common code. The only place where the alternate code gives significant speed improvements over the original is in the "C++" implementations and the number of other optimisations applied there implies that using symmetry made little difference.

4. Charge Map Case Study

4.1. Introduction

Electrostatic charge maps are used to approximate the field of charge generated by a set of atoms in space. The space in question is split up into a 3D grid and the charge at every point

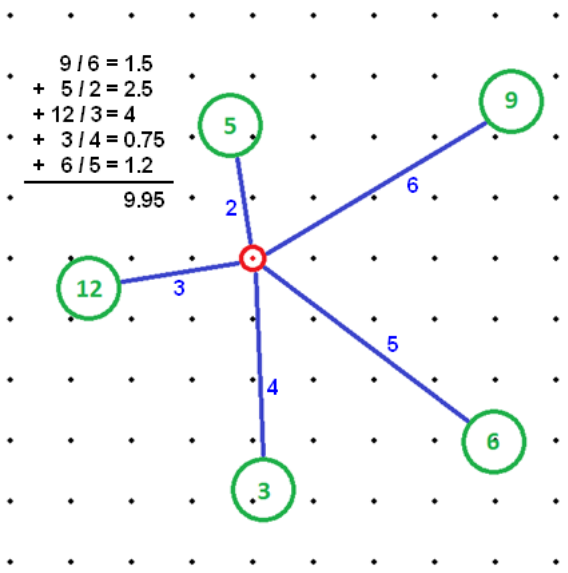

Figure 6. Grid of points showing the full calculation for charge at one point.

in that grid is calculated by dividing the charge of every atom by their distance from the point and summing all the results. The finer the grid the better the approximation as space is continuous in reality. The basic calculation is given in Equation (3), where `N` is the number of atoms, C_i is the charge of atom `i` and `dist(i, xyz)` is the distance between atom `i` and grid point `xyz`. The total (G_{xyz}) is the final charge at that point in space. The 3D world grid is divided up into slices with each 2D layer calculated independently. For this test only one slice was calculated but all the atoms were used, regardless of their location in 3D space. This is demonstrated in Figure 6 for one point (circled). The large circles with numbers in are atoms with their charges, the numbers by the lines are the distance between one atom and the currently calculated point and the sum in the corner is the overall calculation.

$$G_{xyz} = \sum_{i=1}^{N} \frac{C_i}{dist(i,xyz)} \tag{3}$$

There are two obvious methods for parallelising this algorithm. The first is to loop through the atoms sequentially and calculate the current atom's effect on every point in the grid in parallel. The second is the reverse—loop through grid points and calculate every atom's effect on that point in parallel. This latter option would require a parallel addition, such as a sum-reduce algorithm and would also generate very long code in Accelerator, due to loops being unrolled by the compiler. The number of atoms should be small compared to the number of grid points being calculated and may be small compared to the number of GPU processing elements available. The lack of parallel addition, shorter programs and greater resource usage makes the former option the only realistic option. A third option, calculating every atom's effect on every grid point simultaneously, is not possible as at present Accelerator does not provide the 3D arrays required to store all the atom offsets from all the points in the 2D grid.

DirectX 9, upon which the Accelerator GPU target is currently based, has relatively low limits on shader lengths; however, the Accelerator JIT can split programs into multiple shaders to bypass this limit. New Accelerator GPU targets are alleviating this restriction. The original algorithm was found in the Programming Massively Parallel Processors book [15] and the CUDA code is based on the most advanced version of the code in there. Two Accelerator implementations were produced, the second pre-computing a number of constants to simplify the distance calculations based on the fact that the distance between two atoms is

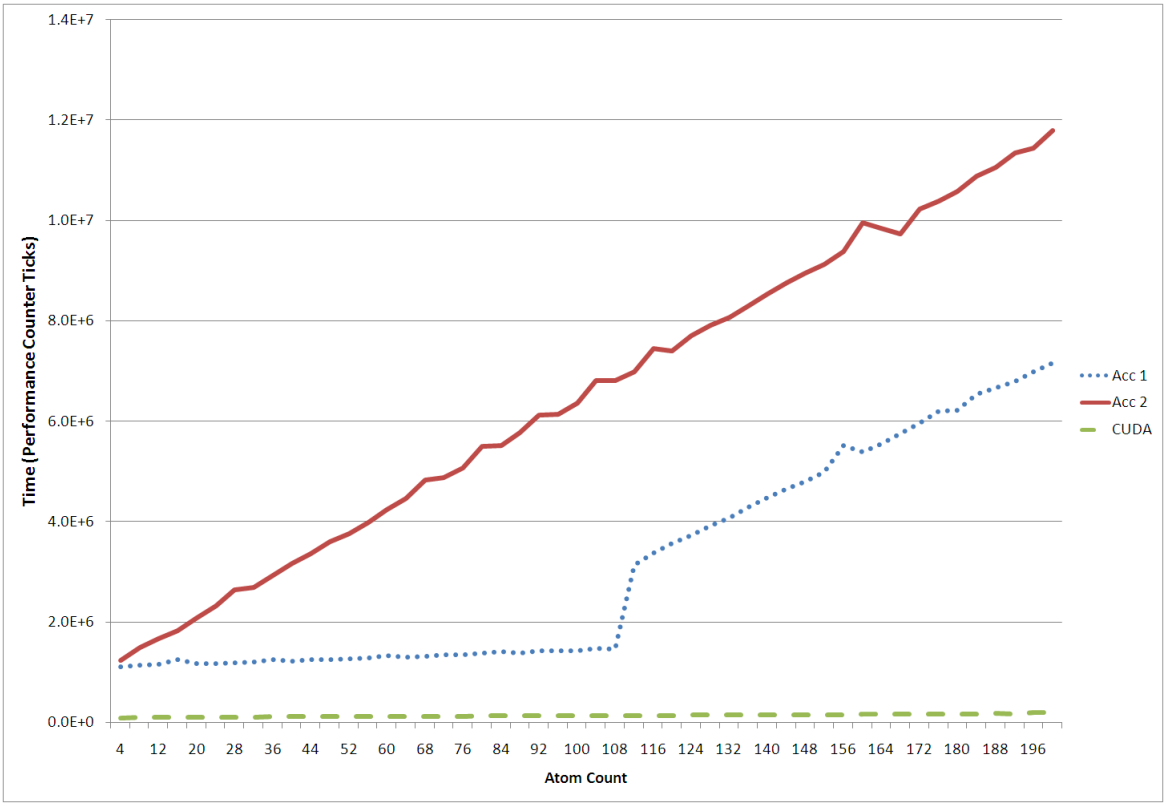

Figure 7. Electrostatic charge map case study *GPU* results for 4-200 atoms with a grid size 1024 x 1024.

constant. This fact was used with a rearrangement of the standard Pythagorean equation to get the distance to one atom based on the distance to the last atom.

4.2. Results

Figures 7 and 8 show the results for the electrostatic charge map experiment. These are for a range of atom counts placed in a constant size grid. Both graphs were generated with the same set of randomly placed input atoms for consistency. The grid was a single 1024×1024 slice containing just over 1,000,000 points. Results were run for 4 to 200 atoms in 4 atom steps with results re-run for the "Accelerator 1 GPU" target between 106 and 114 atoms in 1 atom intervals (not shown). The experiments were only run to 200 atoms because the CUDA target stopped running beyond that point.

"Acc 1" is a basic Accelerator implementation performing the full distance calculation on the GPU for every point in parallel. "Acc 2" is the alternate distance calculation, the timings here include the longer expression generation phase for pre-computing constants. Similarly "C++ 1", "C++ 2", "CUDA 1" etc. show the results for different implementations run on a given target.

4.3. Discussion

Figure 8 show the results for Accelerator and C++ running on a CPU. Here the optimised C++ versions ("C++ 2" and "C++ 3") were the fastest. Although they were very slightly faster than the basic Accelerator CPU version far more effort was used to write them. In terms of development effort "Accelerator 1" was on a par with "C++ 1", and the benefits there are clear to see. The CPU results for "Accelerator 2" are also interesting. Extra effort was put into this version to attempt to make the calculations run on the GPU (or multi-core CPU) faster at the expense of running more calculations at code generation time (see electrostatic charge

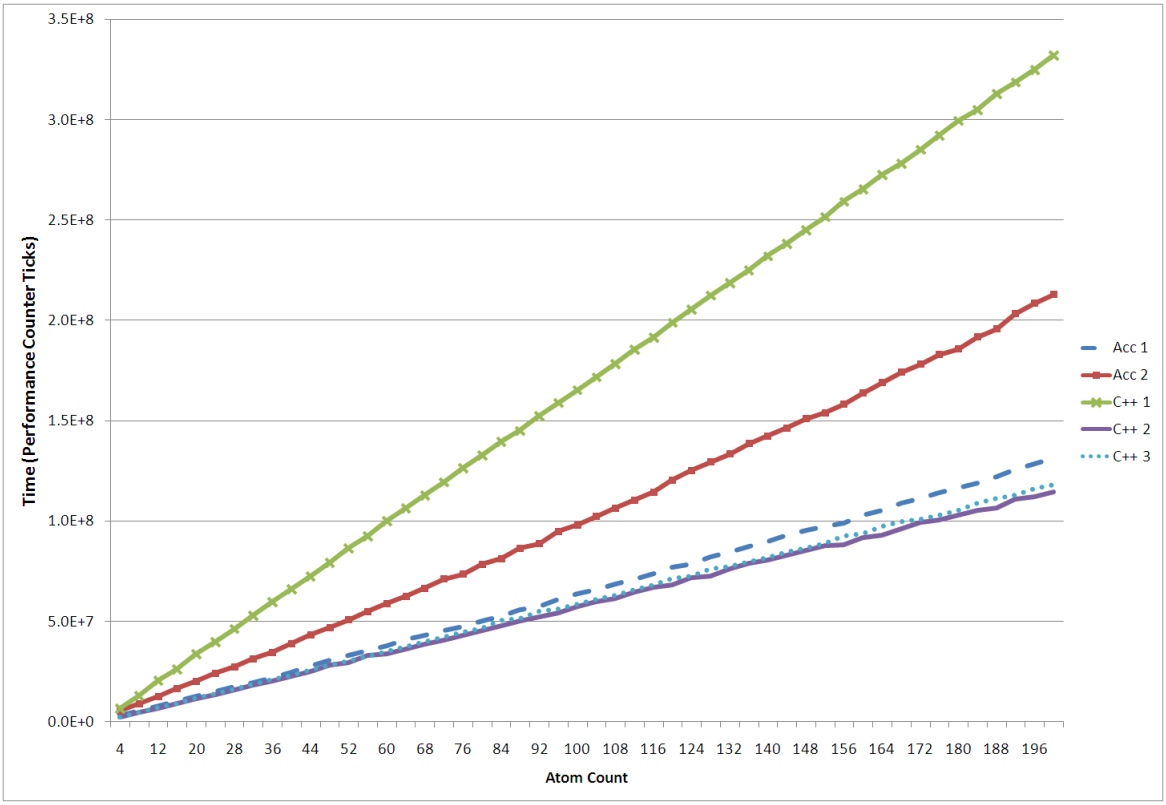

Figure 8. Electrostatic charge map case study *CPU* results for 4-200 atoms with a grid size 1024 x 1024.

map introduction). This was not worth the effort as the results there are significantly slower than the basic Accelerator version.

For the GPU results (Figure 7) CUDA is unparalleled in speed—almost parallel to the x axis but it is important to note that far more development effort was used in that version compared to the Accelerator version. For Accelerator the results again show the JIT overhead seen in the Convolution study, and for "Accelerator 1" also show a discontinuity between 108 and 112 atoms. The gradient of "Accelerator 1" before this discontinuity is around seven times greater than the gradient of "CUDA" and after is around ten times greater, with "Accelerator 2" consistent throughout.

The discontinuity between 108 and 112 atoms, which more fine-grained testing revealed to be located between 111 and 112 atoms, is consistent and repeatable. Accelerator is limited by DirectX 9's shader length limit but has the ability to split long programs up into multiple shaders to bypass this limit. The length of generated program in this case study depends on the number of atoms being processed; atoms are processed sequentially in an unrolled loop as Accelerator does not generate loops. It is believed that 112 atoms is the point at which this splitting occurs as the length of generated code exceeds the maximum. The time jump in the discontinuity is very close in size to the JIT overhead displayed earlier (less than twice the height), most likely resulting from multiple compilation stages or program transfer stages. The increase in gradient can be explained by requiring multiple data transfers between GPU and host (the computer in which the GPU is located).

5. Development

An important consideration for any program is the ease of development. The code in Appendix A helps demonstrate the differences in development efforts between CUDA, C++ and Accelerator, metrics were unavailable as portions of the code were based on existing examples. Even when an algorithm implementation is relatively constant between the various lan-

guages, for example convolution, much more work is required in CUDA before the operation can begin in terms of low-level data shifting between global and local memory. Due to its model the CUDA version does have more options for manual improvement—with Accelerator the user is entirely bound by the layout decisions of the JIT. This is not always a bad thing, however. It is always possible to write code at the assembly level but compilers for high-level languages exist because they are seen as an acceptable trade-off: Accelerator is no different.

CUDA uses a separate language with a separate compiler. Accelerator is embedded in C++; it is usable from languages with C extensions and can use operator overloading, making it possible to interchange Accelerator and sequential code with very little effort.

Listing 4 shows a function to add two values of type "`float_t`" together and defines two C input arrays and one C output array.

Listing 5 shows C code which defines the "`float_t`" type and uses the generic code wrapped in a loop to add the two input arrays together sequentially.

Similarly, Listing 6 shows Accelerator using the same generic function and input data, this time defining the type as an Accelerator array object and performing the calculation in parallel on the GPU.

```
float
    gInputArrayA[4] = {10, 20, 30, 40},
    gInputArrayB[4] = {9, 8, 7, 6},
    gOutputArray[4];


float_t DoCalculation(float_t a, float_t b)
{
    // More complex calculations can be used here with operators.
    return a + b;
}
```

Listing 4. Generic addition code

```
// Set the data to float.
typedef float float_t;

void main()
{
    // Loop over the data.
    for (int i = 0; i != 4; ++i)
    {
        gOutputArray[i] =
            DoCalculation(gInputArrayA, gInputArrayB);
    }
}
```

Listing 5. C++ use of generic addition code

```
// Set the data to FloatParallelArray.
typedef float FloatParallelArray;

void main()
{
    // Set up the target.
```

```
    Target *
        target = CreateDX9Target();
     // Convert the data.
    FloatParallelArray
        aParA(gInputArrayA, 4),
        aParB(gInputArrayB, 4),
        // Build expression tree.
        aParTot = DoCalculation(aParA, aParB);
    // Run expression and save.
    target->ToArray(aParTot, gOutputArray, 4);
    // Clean up.
    target->Delete();
}
```

Listing 6. Accelerator use of generic addition code

6. Conclusions and Future Work

We compared the programming models used in CUDA, the Accelerator library and C++ code and demonstrate that Accelerator provides an attractive trade-off between programmer productivity and performance. Accelerator allows us to implement a 2D convolver using high level data-parallel arrays and data-parallel array operations without mentioning detailed information about threads, scheduling and data layout yet it delivers almost the same performance as the hand written CUDA implementation.

Section 5 looked at the code for convolution using the Accelerator, CUDA and C++ systems. The CUDA code is the most complex, regardless of the advantages afforded by that additional complexity. The Accelerator code did introduce some restrictions (e.g. the use of whole array operations) but these restrictions allow the system to efficiently implement data-parallel operations on various targets like GPUs. Additionally the model means the code is complete—all further optimisation work is left to the compiler. Once past the few overheads the programming model is very similar to C++ code, providing operations on arrays in a manner similar to operations on individual elements. This is also shown in the development discussion by the example using common code for both systems (Listings 4, 5 and 6). The sequential C++ was the simplest to write, but offered no additional acceleration. The CUDA implementation was the fastest to run, though not always by much; so we believe that Accelerator gives a good balance between development and speed.

We also demonstrated reasonable speed ups can be obtained using the Accelerator library. The graphs of CPU results (Figures 5 and 8) show how Accelerator performed compared to C++ code. The only time where Accelerator was slower than the C++ code was within the electrostatic charge map implementation, and only when compared to heavily optimised and tweaked implementations. The GPU and CPU used for the tests were both around £200 which makes comparing their results directly justifiable from a performance/pound point of view. Given that the CPU and GPU Accelerator tests ran from the same implementations with different targets the GPU results show how much of an advantage is available over the CPU for these case studies. When these tests were first run Accelerator was significantly behind CUDA on the GPU, but re-runs with the latest versions of both have brought the two sets of results much closer together as the JIT in Accelerator improves to give better and better code outputs.

A range of algorithms were looked at for different classes of problems showing speed-ups in some. While a more in-depth study looking at categories such as “The Seven Dwarfs” [18], or the updated and more comprehensive “Thirteen Dwarfs” [19], is required; this work shows that some areas are well suited to Accelerator and that a comprehensive re-

view of algorithm classes would be useful work. In the cases where Accelerator can be used, further work is required to give a more complete picture of the situations to which it is well suited. The results for the electrostatic charge map case study are vastly in CUDA's favour, but the convolution results are arguably in Accelerator's favour as the performance gap is minimal and the development gap is huge. For this reason we believe that Accelerator is a useful system, but more work is required to determine problem classes that it is not suitable for.

One major optimisation method for CUDA is the use of shared memory, of which Accelerator has no knowledge currently. One possible avenue for speed up investigation is an automated analysis of the algorithm to group calculations together in order to utilise said shared memory. The GPU results were also produced despite the limitations caused by DirectX 9 listed in Section 1.4.

Several of the results have shown the overheads due to Accelerator's JIT. These tests were run without using Accelerator's "parameters" feature which can pre-compile an expression using placeholders (called "parameters") for input data and store the result. Every run in the results was repeated ten times and summed, however as it uses off-line compilation the CUDA code was only ever built once.

Acknowledgements

This work was carried out during an internship at Microsoft Research in Cambridge. The studentship is jointly funded by Microsoft Research and the Engineering and Physical Sciences Research Council (EPSRC) through the Systems Engineering Doctorate Centre (SEDC).

References

[1] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM.

[2] NVIDIA. CUDA homepage. `http://www.nvidia.com/object/cuda_home.html`, 2010.

[3] NVIDIA. GeForce GTX 480 Specifications, 2010.

[4] ATI. ATI Radeon HD 5970 Specifications, 2011.

[5] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[6] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[7] J. N. England. A system for interactive modeling of physical curved surface objects. *SIGGRAPH Comput. Graph.*, 12(3):336–340, 1978.

[8] Christian A. Bohn. Kohonen Feature Mapping through Graphics Hardware. In *In Proceedings of Int. Conf. on Compu. Intelligence and Neurosciences*, pages 64–67, 1998.

[9] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. *SIGGRAPH Comput. Graph.*, 24(4):327–335, 1990.

[10] Kenneth E. Hoff, III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[11] Michael McCool and Stefanus Du Toit. Metaprogramming GPUs with Sh. `http://libsh.org/brochure.pdf`, 2004.

[12] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[13] Patrick S. McCormick, Jeff Inman, James P. Ahrens, Charles Hansen, and Greg Roth. Scout: A Hardware-Accelerated System for Quantitatively Driven Visualization and Analysis. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 171–178, Washington, DC, USA, 2004. IEEE Computer Society.

[14] M. Lesniak. PASTHA - parallelizing stencil calculations in Haskell. *Declarative Aspects of Muilticore Programming*, Jan 2010.

[15] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1 edition, 2010.

[16] Victor Podlozhnyuk. Image Convolution with CUDA. `http://developer.download.nvidia.com/compute/cuda/sdk/website/C/src/convolutionSeparable/doc/convolutionSeparable.pdf`, 2007.

[17] Accelerator Team. *Microsoft Accelerator v2 Programming Guide*. Microsoft Research, 2010.

[18] Phillip Colella. Defining Software Requirements for Scientific Computing. Presentation, 2004.

[19] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

A. Convolution Code

Presented here is the main "kernel" code used to perform a 1D convolution in CUDA, Accelerator and sequential C++. Code to set up and destroy data arrays and target interactions has been omitted. The code used for the case studies in Section 3 is based on multiples calls to the code presented here.

A.1. C++ code

Listing 7 is the reference C++ code for a 1D convolver. This code is presented first as it most clearly demonstrates the basic convolution algorithm. "`arrayInput`" is the input data as a C array, "`arrayOutput`" is the resulting C array, "`filter`" is an array containing the full filter (a "`filterRadius`" value of five results in eleven filter values). The code loops through every element in the input array (of which there are "`arrayWidth`"), for each one calculating the result according to all filter values and surrounding elements (clipped to the array size). The operation of "`Clamp`" (Listing 8) is the main basis of the two improved C++ implementations which separate the loops into three loops to deal with start, middle and end of array values separately and do away with branching in "`Clamp`".

```
for (int j = 0; j != arrayWidth; ++j)
{
    float
        sum = 0;
    for (int u = -filterRadius, p = 0; u <= filterRadius; ++u, ++p)
    {
        int
            J = Clamp(j + u, 0, arrayWidth - 1);
        sum += arrayInput[J] * filter[p];
    }
    arrayOutput[j] = sum;
}
```

Listing 7. C++ convolution code

```
int
    Clamp(int x, int min, int max)
{
    return (x < min) ? min : (x > max) ? max : x;
}
```

Listing 8. Additional C++ code

A.2. Accelerator code

Listing 9 is the code to perform a 1D convolution in Accelerator. The C++ sequential code loops over every input element in turn, calculating the effect of each surrounding element before moving on to the next. In contrast this code calculates the effect of one offset on every element using Accelerators "`Shift`" function which behaves much like the clamped array lookup in the C++ code, but for every element in parallel. Because Accelerator is a JIT system the main loop builds a large unrolled expression tree which is evaluated when "`ToArray`" is called for a specified target (here DirectX 9). This has the effect that all the filter values are known at compile time and become constants in the executed GPU code. In this code "`arrayTemp`" and "`arrayInput`" are Accelerator objects representing arrays on the GPU, the former is declared and initialised to 0 in the code given. "arrayOutput" is a C array to which the final result is saved after GPU execution.

```
size_t
    dims[] = {arrayWidth};
intptr_t
    shifts[] = {0};
FloatParallelArray
    arrayTemp(0.0f, dims, 1);
for (int u = -filterRadius, p = 0; u <= filterRadius; ++u, ++p)
{
    shifts[0] = u;
    arrayTemp += Shift(arrayInput, shifts, 1) * filter[p];
}
ParallelArrays::Target *
    target = MicrosoftTargets::CreateDX9Target();
target.ToArray(arrayTemp, arrayOutput, height,
    width, width * sizeof (float));
```

Listing 9. Accelerator convolution code

A.3. CUDA code

The CUDA code equivalent to the previous two examples is shown in Listing 10. This code is by far the most complex and its development is well documented in the "Image Convolution with CUDA" white paper by Victor Podlozhnyuk [16]. The code within the function "`DoOneRow`" does a 1D convolution, or part of one according to the number of available processing units on the GPU—if the input is larger the data is partitioned into blocks in "`DoRows`". "Part 1" (see in-code comments) is responsible for determining the limits of data for which the current block is responsible, including aprons which may be beyond the input data limits or may be beyond the processing limits of the current block. "Part 2" uses every thread in the current block to load one value from global memory to faster memory and waits for all other threads to complete. "Part 3" performs the actual convolution algorithm on one element, with many threads running at once. Unlike the C++ and Accelerator code listings, this version does not save its result to a C array, requiring explicit conversion after the main calculation.

```
__global__ void
    DoOneRow(float * const arrayOutput,
        const float * const arrayInput, const int width,
        const int pitch, const int radius)
{
    // Part 1.  Load all values for calculations into variables.
    __shared__ float
        fRowData[1280];
    const float * const
        filter = &gc_fFilter[radius];
    const int
        dataStart     = blockIdx.x * blockDim.x,
        apronStart    = dataStart - radius,
        apronClamp    = max(apronStart, 0),
        alignedStart  = apronStart & (-HALF_WARP),
        dataEnd       = dataStart + blockDim.x,
        apronEnd      = dataEnd + radius,
        dataEndClamp  = min(dataEnd, width),
        apronEndClamp = min(apronEnd, width),
        apronOffset   = apronStart & (HALF_WARP - 1),
        maxX          = width - 1;
    int
```

```
        load = threadIdx.x + alignedStart,
        pos  = threadIdx.x - apronOffset;
    // Part 2.  Copy data from global to local memory, clamped.
    while (load < apronEnd)
    {
        if (load >= apronEndClamp)
        {
            fRowData[pos] = arrayInput[maxX];
        }
        else if (load >= apronClamp)
        {
            fRowData[pos] = arrayInput[load];
        }
        else if (load >= apronStart)
        {
            fRowData[pos] = arrayInput[0];
        }
        load += blockDim.x;
        pos  += blockDim.x;
    }
    __syncthreads();
    // Part 3.  All data is loaded locally, do the calculation.
    const int
        pixel = dataStart + threadIdx.x;
    if (pixel < dataEndClamp)
    {
        float * const
            dd = fRowData + threadIdx.x + radius;
        float
            total = 0;
        for (int i = -radius; i <= radius; ++i)
        {
            total += filter[i] * dd[i];
        }
        arrayOutput[pixel] = total;
    }
}

// Non-truncating integer division.
#define CEILDIV(m,n) \
    (((m) + (n) - 1) / (n))

extern "C" void
    DoRows(float * arrayOutput, float * arrayInput, int width,
        int threads, int pitch, int radius)
{
    // Call the GPU code from the host.
    dim3
        dimBlocks(CEILDIV(width, threads)),
        dimThreads(threads);
    DoOneRow<<<dimBlocks, dimThreads>>>(
        arrayOutput, arrayInput, width, pitch, radius);
}
```

Listing 10. CUDA convolution code