DESIGNING NUMERICAL SOLVERS FOR NEXT GENERATION HIGH PERFORMANCE COMPUTING

A transfer report submitted to the University of Manchester towards the degree of Doctor of Philosophy in the Faculty of Engineering and Physical Sciences

2011

By
Mark James Mawson
Mechanical, Aerospace & Civil Engineering
Contents

Abstract vii
Declaration viii
Copyright ix

1 Introduction 1

2 Graphics Processing Units and Hardware Accelerators 3
  2.0.1 GPU Hardware Evolution . . . . . . . . . . . . . . . . . . . 3
  2.1 NVIDIA CUDA Accelerators . . . . . . . . . . . . . . . . . . . 5
    2.1.1 Threads and Blocks . . . . . . . . . . . . . . . . . . . . . 7
    2.1.2 GPU Memory . . . . . . . . . . . . . . . . . . . . . . . . . 8
    2.1.3 Bandwidth and FLOPs for Performance . . . . . . . . . . 9
    2.1.4 Code Optimisation . . . . . . . . . . . . . . . . . . . . . . 10
  2.2 Other Hardware Accelerators . . . . . . . . . . . . . . . . . . . 10
    2.2.1 Field Programmable Gate Arrays (FPGA) . . . . . . . . . 11
    2.2.2 The Intel ‘Many Integrated Core’ Architecture . . . . . . 11

3 The Algebraic Multigrid Method 13
  3.1 Application of the Multigrid Principles to unstructured domains . 13
    3.1.1 Coarse Grid Selection . . . . . . . . . . . . . . . . . . . . 14
  3.2 Parallelisation . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

4 The Lattice-Boltzmann Method 19
  4.1 The Boltzmann Transport Equation . . . . . . . . . . . . . . . . . 19
  4.2 The BGK Approximation . . . . . . . . . . . . . . . . . . . . . . 21
  4.3 The D2Q9 Lattice and Iterative Solutions to the Boltzmann Trans-ports Equation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.4 Test Case - LBM for Isothermal Incompressible 2D Navier-Stokes equations ........................................ 24
4.4.1 Lid Driven Cavity Test Case - Preliminary Results Using PGI Accelerator Compiler ................. 24

5 Optimization of a Geometric Multigrid Method For Nvidia Fermi Based GPUs (Draft Paper) .......... 30

6 Conclusion and Future Work ........................................ 38
6.1 Further Work ......................................................... 38

Bibliography .............................................................. 42
List of Tables

4.1 Execution Times for 100 Iterations of LBM on a Lid Driven Cavity 26
4.2 Execution Times For LBM With Data Regions Inside Main Loop 27
4.3 Execution Times For LBM With Data Regions Inside Main Loop 27
List of Figures

2.1 Architecture of a NVIDIA GeForce 6 GPU [55] . . . . . . . . . . . 4
2.2 A Unified Shader Architecture . . . . . . . . . . . . . . . . . . . . 5
2.3 CPU & GPU Architectures [50] . . . . . . . . . . . . . . . . . . . 6
2.4 Quadro 6000 Architecture . . . . . . . . . . . . . . . . . . . . . . 6
2.5 Blocks and Threads in CUDA . . . . . . . . . . . . . . . . . . . . . 7
2.6 An Intel MICA Accelerator. . . . . . . . . . . . . . . . . . . . . . 12

4.1 D2Q9 Lattice . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2 Boundary Lattice Points . . . . . . . . . . . . . . . . . . . . . . . 25
4.3 Summary of All LBM Implementations . . . . . . . . . . . . . . . 28
4.4 Memory Transfers . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.5 Accelerator Data Region Inside Main Loop . . . . . . . . . . . . . 29
4.6 Accelerator Data Region Outside Main Loop . . . . . . . . . . . . 29
List of Algorithms

1  V-Cycle Multigrid Method ........................................ 13
2  Coarse Grid Selection .............................................. 15
3  Modification of $\omega_i$ ............................................. 18
Abstract

High Performance Computing (HPC) is moving towards massive scales of parallelism. The changes in hardware towards large scale on chip parallelism requires the re-writing of existing solvers for various Computational Fluid Dynamics (CFD) problems. The aim of the project is to write and optimise novel solvers for various common CFD numerical problems that can take advantage of this increased parallelism on new platforms as they develop. Of special interest is the use of Graphics Processing Units (GPUs) which have only recently become available for use in HPC. GPU computing represents a significant change in the architectures used for scientific computing, and offers a factor of ten increase in peak performance/price ratio when compared with traditional HPC clusters.

This report focusses on the optimisation of the geometric multigrid method on GPU and presents a literature review on the algebraic multigrid method and algorithms for parallelising this method. Work using a directive based GPU compiler to develop a GPU implementation of the Lattice Boltzmann is also shown.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the “Copyright”) and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii. The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the “Intellectual Property Rights”) and any reproductions of copyright works, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and exploitation of this thesis, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of Mechanical, Aerospace & Civil Engineering (or the Vice-President).
Chapter 1

Introduction

High Performance Computing (HPC) is coming to a crossroads in terms of the hardware being used. Traditional clusters of Central Processing Units (CPUs) are being supplemented with accelerator hardware, massively parallel devices designed with HPC in mind. These devices pose a new challenge for software developers as optimisation of code for accelerator architecture is a relatively new field.

The field of Computational Fluid Dynamics (CFD) will be affected by this change in hardware, as many CFD problems require the use of large computational resources. The writing and optimisation of CFD algorithms for accelerator technology is therefore becoming an important topic; while the numerical methods for solving CFD problems improve at a relatively slow rate, Moore’s law [46] indicates that the number of components on a chip (a good indicator of chip performance) doubles roughly every two years. In previous years this has equated to a doubling of the clock speed of the processor, but due to the amount of heat generated running at very high clock speeds this is no longer deemed an acceptable method of increasing performance. Hardware manufacturers now increase the number of cores on a chip to maintain Moore’s law, culminating in accelerator technology which foregoes the complicated cores found in CPUs in favour of many simpler cores. With a well written program, an accelerator can consistently out-perform current CPUs by an order of magnitude or more [64, 67, 61, 59, 48, 25, 16].

This report will present the optimisation of a geometric multigrid method introduced in the previous years report, and the use of directive based compilers to generate GPU code for a Lattice Boltzmann test case. Chapter 2 contains an
updated review of literature for the GPU hardware used within the report, Chapter 3 contains background theory on the algebraic multigrid method, Chapter 4 contains background theory for the Lattice Boltzmann method and a test case using a directive based GPU compiler, Chapter 5 contains a draft of the paper ‘Optimisation of a Geometric Multigrid Method for Nvidia Fermi based GPUs’, and Chapter 6 provides conclusions and the scope for further work.
Chapter 2
Graphics Processing Units and Hardware Accelerators

The use of accelerator technology, in particular Graphics Processing Units (GPUs), in HPC has increased in recent years [54]. The move from GPUs only capable of performing fixed programs to fully programmable pipelines [3] has allowed HPC to take advantage of the large scale on chip parallelism in GPUs. This is becoming increasingly important for projects such as the International Exascale Project [18], which are looking to achieve exa-FLOP ($10^{18}$) performance by 2018. As well as providing performance increases over CPUs, GPUs also use less power/FLOP [57]. Current predictions for exa-FLOP machines estimate a power consumption of $\approx 20$MW [43], making power consumption a primary concern when designing these large scale systems.

This chapter gives details of the evolutions of GPU hardware which has lead to their use in HPC and other accelerator hardware which takes advantage of on chip parallelism.

2.0.1 GPU Hardware Evolution

Prior to the ATI Xenos GPU [3] in 2005, the hardware in a GPU was constructed to match the graphics pipeline. Fig. 2.1 shows the architecture of a GeForce 6 series GPU from 2004, the different stages of the graphics pipeline are clearly implemented as different pieces of hardware. Work in [6] and [23] describes the use of programmable vertex and fragment (pixel) processors to solve multigrid problems.
A common problem with this architecture is that load imbalance between vertex and fragment processors is easy to come by. Large, simple shapes will cause little utilisation of the vertex processors but high use of the fragment processors, and vice-versa with small, complicated shapes. To overcome this problem a single collection of cores capable of executing any stage of the graphics pipeline is now used [41], this is known as a unified shader architecture (See Fig. 2.2).

The more general nature of programmable unified processors not only allows graphics programmers to write their own implementations of the various pipeline stages, e.g. implementing their own lighting models, but also allows GPUs to be programmed for non-graphical operations. This is shown in Fig. 2.2 by the direct link between GPU Memory and ‘compute programs’. Application Programming Interfaces (APIs) to take advantage of this were released soon after unified shader architectures became popular; BrookGPU in 2007 [1], OpenCL in 2008 [37] and DirectCompute as part of DirectX 11 in 2009. NVIDIA also allow direct programming of their GPUs using their own API, CUDA).

More recently CAPS have provided an API which abstracts away from having to write GPU code directly [12]. Code to be executed on GPUs is framed by
Figure 2.2: A Unified Shader Architecture

directives specifying whether to generate CUDA code to run on NVIDIA GPUs or OpenCL code to run on NVIDIA or ATI GPUs. The HMPP compiler then generates the appropriate API code and compiles it using either the CUDA or OpenCL compiler. PGI also offers a similar compiler, which is used in Chapter 4.

An open source compiler for NVIDIA GPUs is currently being developed by PathScale [20]. It claims their compiler is better optimised for HPC than CUDA and OpenCL, as the CUDA drivers support traditional GPU use (gaming and video decoding) as well as HPC use, and OpenCL optimisation is generally poor as cross-platform support takes priority.

2.1 NVIDIA CUDA Accelerators

As can be seen from Fig. 2.3, the cores within a GPU are grouped together into blocks to share caches and control logic. NVIDIA call these blocks Streaming Multiprocessors (SM).

For the GPU used in this report (a NVIDIA Quadro 6000), each SM contains 32 cores, known as Thread Processors and four special Function Units (SFU) for fast calculation of common math functions. Double precision operations are performed by sharing the operation between two Thread Processors, effectively halving the performance. The remaining components of the SM include a 64KB
Figure 2.3: CPU & GPU Architectures [50]

Shared Memory/L1 Cache, registers and control units for threads, warps and instructions. Fig. 2.4 shows a single SM, and how fourteen SMs make up the architecture of a Quadro 6000. A 768KB L2 cache is shared between all SMs, and 6GB of RAM is provided on the GPU (This is known as global memory).

Figure 2.4: Quadro 6000 Architecture
2.1.1 Threads and Blocks

Threads are issued in groups of 32, known as ‘warps’, and up to 1536 threads can be assigned to a SM at any one time. A group of threads assigned to a SM is known as a block, and the amalgamation of all the blocks scheduled to run on the GPU is known as the grid (See Fig. 2.5). It is through blocks that problems are parallelized, e.g. a matrix can be split into sub-matrices and each sub-matrix is assigned to a block, with each thread in the block operating on one element of the sub-matrix. A warp of threads operates in Single Instruction Multiple Thread (SIMT) mode; threads within a warp may execute different instructions by way of branching (If tests, exiting loops, etc) but threads which do not execute the branched instructions will stall until the branch is completed. I.e. threads will only run in parallel while they execute the same instructions. It is therefore important to minimise branching operations to allow threads to run in parallel as often as possible.

Figure 2.5: Blocks and Threads in CUDA
2.1.2 GPU Memory

The Quadro 6000 contains four kinds of memory that can be used directly in programming (other memory types exist but these are registers or overspill in cache for registers.); global memory, cache, shared memory and texture memory. It is recommended to initially program using global memory and then maximise use of the other types where possible as they have a higher bandwidth [21, 61, 27, 22, 23, 16, 52].

Global Memory

Global memory is the largest of all the memories on the Quadro 6000 GPU at 6GB. As the global memory is off-chip (see Fig. 2.4), access is relatively slow (a latency between 400-600 clock cycles is common [50], and a bandwidth of approximately 89GB/sec is found on the Quadro 6000). The slow access speed is offset by the simplicity of programming access to global memory. This makes global memory useful when first writing code, as the inherent parallelism of the target problem can be explored before optimisation of memory use. In order to use shared or texture memory, the variables must be initialised on the host computer and passed to global memory. From here the variables can be moved to shared or texture memory. In order to transfer results back to the host computer, they must be passed back to global memory. Global memory is cached by default, and access times can be improved by coalescing the memory transactions within a warp of threads. This can be achieved when threads within a warp access the same cache line. Specifics of how coalesced transfers are executed can be found in the CUDA Programming Guide [51].

Shared Memory/Cache

Each SM contains a combined 64KB shared memory/L1 cache that all cores within the SM can access. This memory can be split 48Kb-16Kb in favour of one memory type. If L1 cache is preferred as the larger memory, then no further programming is needed as global memory is already cached by default. The contents of the cache will be flushed at the end of the GPU code to ensure data coherency in global memory. If shared memory is preferred, then extra code is required to explicitly gather the data and then to return it to global memory. Bandwidth for both of these memory types is approximately 400 GB/sec [49].
Texture Memory

Texture memory is another form of cache for global memory, more specifically it is the cache used by a processing unit known as the Texture Unit. This unit is shared between several SMs and is responsible for the mapping of a 2D image (texture) onto a 3D surface [31], as used in the rendering of 3D objects for video games and CGI. Although the CUDA programming model does not allow access to the texture unit, it does allow access to the Texture Memory. In order to use texture memory a piece of global memory must be bound to it; this has the effect of allowing the bound piece of Global Memory to use Texture memory as a cache. Texture Memory is read only from the perspective of the GPU and only natively supports IEEE-754 standard [35] single precision float values, which have a machine precision of $1.192092896 \times 10^{-7}$ (1). This makes its use within scientific computing limited. It should be noted that it is possible to store the bit values of a 64-bit double precision number as two 32-bit values and then recombine them once they have been read from texture memory, but a time penalty is incurred whilst the double precision value is reconstructed. For this reason texture memory will not be used within the work presented in this report.

2.1.3 Bandwidth and FLOPs for Performance

The Quadro 6000 has a peak performance of approximately 1 TFLOP (1 Multiply-Add operation/cycle/core × 448 cores × 1.15GHz clock speed) in single precision and 500 GFLOPS in double precision. Assuming a program could be written with a 100% cache hit rate (an impossibility, but it serves to illustrate a point), $50 \times 10^9$ double precision pieces of data could be moved each second. This gives a factor of ten shortcoming in memory bandwidth versus computational speed, meaning that 10 floating point operations would need to be performed for every memory transaction to achieve peak performance. In many applications this is unlikely, and therefore performance will rarely approach peak values. In Chapter 5 this is discussed in detail, and a recommendation to base performance on peak

---

1 The machine precision is defined as the smallest value that can be added to 1 without the answer being rounded to 1 [17]. This value can be calculated using the formula $\beta^{1-p}$, where $\beta$ is the base number (in the case of binary computing, 2) and $p$ is the number of bits used to represent the mantissa (the fractional part of the value, with the remaining bits being used as the exponent of the fractional value). The values of $1.192092896 \times 10^{-7}$ for single precision and $2.220446049 \times 10^{-16}$ for double precision were found to be the machine precisions for the default rounding mode of the C1060 through the use of a linear search algorithm.
memory bandwidth rather than peak FLOPs is made.

2.1.4 Code Optimisation

Due to the highly parallel architecture of GPUs and the variety of memories available, there are several approaches to optimising code. Cohen & Molemaker [16] recommend implementing as much of the code as is possible on the GPU, and then optimising as required. Whilst this is a valid method, it poses problems when attempting to port existing CPU code as many programming hours may be spent porting sections of code that may have very small execution times. In this case it is possible for the time penalty of transferring data to and from the GPU to outweigh the benefit of faster execution.

Frigaard [21] demonstrated a method of accelerating existing CPU code by locating the computationally intensive subroutines within a program and porting them to CUDA C. Both approaches recommend that the initial version of code should use global memory and that changes in algorithms should be considered first in order to exploit any hidden parallelism inherent in the code. Common alterations to the algorithm include grouping data so that the threads in a block all follow the same execution path and loop unrolling. Once this is completed, memory optimisation takes one of four forms; using shared memory or cache, coalescing global memory accesses, the use of texture memory and minimising transfers across the PCI bus. Using these techniques Bell & Garland achieved a $1.5 \times$ speedup for matrix-vector multiplications compared with the original GPU code [22]. Simek et. al [61] a speedup of $9 \times$ for modelling of atmospheric equations using a method of lines solved by a Runge-Kutta method and Harris presented a speedup of $21 \times$ for a reduction operation [27]. More recently NVIDIA has released a set of optimisation guidelines for current GPUs [52].

2.2 Other Hardware Accelerators

GPUs are not the only technology available to accelerate computation, as the concept is not a new one. Early FPUs were accelerators in the form of ‘coprocessors’, which were physically separate from the CPU. This section gives details of a commonly used type of accelerator, Field Programmable Gate Arrays, and an upcoming accelerator from Intel known as the ‘Many Integrated Core’ architecture (MIC).
2.2.1 Field Programmable Gate Arrays (FPGA)

An FPGA is a collection of configurable logic blocks (AND, OR, etc), digital signal processing cores (for floating point operations) and traditional CPU cores [10]. Connections between the various components are reconfigurable through software, allowing custom hardware pipelines to be built which can be specific to an application.2

GPU pipelines have previously been implemented on FPGAs in work by Gu et al [26] and Kim et al [38], and FPGA performance for a two-point angular correlation function (used in astronomy to denote how the number of celestial bodies near to a target body varies with angle) was compared with GPU implementations in work by Kindratenko et al. [39]. This report found that the FPGA provided a modest speedup over a CPU implementation, but much less than that of a GPU. Brodtkorb et al. [10] concluded that FPGAs perform best on fixed point (integer) operations, and cite cost and limited floating point performance as the reason why GPUs are being adopted in their place.

2.2.2 The Intel ‘Many Integrated Core’ Architecture

Intel are currently developing and testing an accelerator architecture, known as ‘Many Integrated Core’ (MIC), based on an abandoned GPU project known as Larrabee [58]. The MIC architecture uses superscalar cores capable of performing 64 bit operations (Similar to current Intel CPUs). Each core also contains a vector unit for floating point math operations, capable of accepting 512 bit vectorised inputs [47] (e.g. sixteen single precision or eight double precision values) and performing fused multiply-add operations, i.e. two operations per clock cycle. Due to their compatibility with current Intel cores, these cores as known as Intel Architecture (IA) cores. A two level data coherent cache is provided for each core, and the cores/cache can share data via an interconnect. Fig. 2.6 shows the architecture of a MIC accelerator.

The current development hardware for MIC is known as ‘Knights Ferry’ and contains 32 IA cores running at 1.2GHz, each core capable of supporting four threads, with 8MB of cache between the cores and 1GB of high speed RAM. Peak single precision performance is therefore $32(\text{cores}) \times 1.2\text{GHz} \times 16(\text{vectorised})$.

---

2A processor designed for a specific application is known as an Application Specific Integrated Circuit (ASIC) [68]. FPGAs are often used to prototype ASICs as errors in the configuration can be easily fixed.
inputs)×2(operations/core/clock cycle) = 1.229 TeraFLOPS. The accelerator has been tested to achieve over one TeraFLOP performance on matrix multiplication algorithms.

The MIC architecture is also compatible with the current Intel instruction set [58], allowing code parallelised using Posix threads or OpenMP to be compiled for use on traditional parallel systems and MIC systems with no alterations.

Figure 2.6: An Intel MICA Accelerator.
Chapter 3

The Algebraic Multigrid Method

3.1 Application of the Multigrid Principles to unstructured domains

The multigrid method presented in last year’s report for solving the problem $A u = f$, where $u$ is unknown, can be summarised in the following algorithm: The

Algorithm 1 V-Cycle Multigrid Method

$u_L = MG(A_L, f_L, u_L, L)$

function $MG(A_i, f_i, u_i, l)$

for $i=1$ to Number of Pre Smoothing steps do

$u_i \leftarrow S_{i}^{\text{Pre}}(u_i, f_i)$

end for

$r_i = f_i - A_i u_i$

$r_{i-1} = R_i \bar{r}_i$

if $l-1=0$ then

Solve exactly $A_{l-1} \epsilon_{l-1} = r_{l-1}$

else

$\epsilon_{l-1} = MG(A_{l-1}, f_{l-1}, r_{l-1}, l)$

end if

$\epsilon_l = P_{l-1} \epsilon_{l-1}$

$u_l = u_l + \epsilon_l$

for $i=1$ to Number of Post Smoothing steps do

$v_i \leftarrow S_{i}^{\text{Post}}(u_i, f_i)$

end for

$\text{Call multigrid function at grid } L$

Prototype for the MG

Smooth $A u = f$

Calculate the residual

Restrict residual to next grid level

Solve coarsest grid

exactly

Recursive call to MG

using next coarsest grid

Prolong error to current grid

Add error to current solution

Remove interpolation errors
error in \( u \) is repeatedly coarsened and smoothed, and then the smoothed errors are summed together. This repeated coarsening allows the different frequencies within the error to be smoothed equally, increasing the convergence rate versus that of only smoothing on a fine grid. A full explanation of the geometric method can be found in last years report, and \([7, 9]\).

The application of this method to unstructured problems is simple in concept as it is very similar to the geometric approach. The novelty in algebraic multigrid comes in the restriction operation, as points for the coarser grid must be selected without knowledge of the geometry of the problem \([8]\). This applies to other aspects of the multigrid method as well, as stencil operations with known point-point distances must be replaced by operations using \( \mathbf{A} \) \([33]\). To avoid confusion, the term \( e \) will be used exclusively to represent the unknown function being solved at each level of the multigrid method from this point onwards, although in Alg. \( 1 \) \( e \) is re-cast as \( u \) when the multigrid function is recursively called.

### 3.1.1 Coarse Grid Selection

Selection of points for a coarser grid involves determining which points are ‘strongly’ connected to others, where \( e_i \) is ‘strongly’ connected to \( e_j \) if \(-a_{i,j} > \alpha \max_{k \neq i} -a_{i,k}\), where \( a_{i,j} \) is an element in \( \mathbf{A} \) and \( \alpha \) is some parameter, typically 0.25 \([32]\). In this way the set of points \( e_i \) ‘strongly’ depends on is defined as

\[
S_i \equiv \{ j : i - a_{i,j} \geq \alpha \max_{k \neq i} -a_{i,k} \} \tag{3.1}
\]

We also define

\[
S^T_i \equiv \{ j : i \in S_j \} \tag{3.2}
\]

the set of points which ‘strongly’ depends on \( e_i \). By defining the set of points in the coarse grid as \( C \) and the set of points which remain on the fine grid as \( F \), selection of points for the coarse grid then becomes a matter of satisfying the following criteria:

1. C1. For each \( e_i \in F \), each \( e_j \in S_i \) is either in \( C \) or \( S_j \) and \( C_i \) have non-null values.

2. C2. \( C \) should be a maximal subset such that no point in \( C \) is dependent on another point in \( C \).
The first condition guarantees that all values of $e_i$ are represented in interpolating $C$ back to $F$, the second condition is designed to minimize the number of points in $C$. A serial algorithm to build $C$ is as follows: This is repeated for as many

\textbf{Algorithm 2 Coarse Grid Selection}

\begin{verbatim}
for i=1 to Number of points in $u$ do
    $S_i \equiv \{ j \mid i \neq j, a_{i,j} \geq \alpha \max_{k \neq i} a_{i,k} \}$
    $\chi_i$=Number of points in $S_i$
end for

while Coarse grid selection criteria not satisfied do
    $C \leftarrow e_{\max \chi_i}$
    Put most strongly connected $u_i$ in $C$
    $F \leftarrow e_j$ in $S_i$
    Increase all $\chi_i$ for $S_i$ that contain new $F$ points
end while
\end{verbatim}

levels of grid coarsening are necessary. It should be noted that no values in $F$ contribute to those in $C$, $e_i$ in $C$ are injected directly from the fine grid. Adhering to the selection criteria for points in $C$ also ensures that the problem is coarsened in the directions of smoothness in $e$, this smoothness is characterised as algebraic, rather than geometrically smooth in the problem domain. In [19] Falgout derives the equation

$$\lambda = e^T A e = \sum_{i<j} (-a_{i,j}(e_i - e_j)^2 \leq 1 \quad (3.3)$$

to describe the eigenvalues of $e$ as a function of $A$ and $e$. It can be seen that larger values of $-a_{i,j}$ will correspond to larger eigenvalues which are associated with lower frequency (i.e. smooth) errors [24]. Work in [8, 53] provides further detail on this matter.

Interpolation in the algebraic multigrid method is usually of the form

$$e_i = \sum_{j \in C_i} w_{i,j} e_j \quad (3.4)$$

where $C_i$ is the set of variables in $C$ that also belong to $S_i$. If $e_i$ is also in $C$ then the corresponding value in $C$ is directly copied and Eqn. 3.4 is unnecessary. $w$ is a weighting factor, and is calculated as follows; Call $N_i$ the set of all points to which $e_i$ is connected, $D_i$ the set of points to which $e_i$ is weakly connected and $F_i$ the set of points in $F$ to which $e_i$ is strongly connected. Approximating $e_i$ as
the weighted average of its neighbours gives

\[
a_{i,i}e_i \approx -\sum_{j \in C_i} a_{i,j}e_j - \sum_{j \in D_i} a_{i,j}e_j - \sum_{j \in F_i} a_{i,j}e_j \tag{3.5}
\]

As values of \(\sum_{j \in D_i} a_{i,j}\) will be small [32], substituting \(e_i\) for \(e_j\) allows a rearrangement

\[
\left(\sum_{j \in D_i} a_{i,j} + a_{i,i}\right) e_i \approx -\sum_{j \in C_i} a_{i,j}e_j - \sum_{j \in F_i} a_{i,j}e_j \tag{3.6}
\]

The values of \(e_j\) in \(F_i\) can also be approximated by summing across values of \(e_k\), where \(k \in C_i \cap C_j\) [62, 11]. This approximation is the equivalent of representing a strong connection between two elements in \(F\) using points in \(C\) that both are strongly connected to. This approximation yields

\[
\left(\sum_{j \in D_i} a_{i,j} + a_{i,i}\right) e_i \approx -\sum_{j \in C_i} \left( a_{i,j} - \sum_{m \in F_i} a_{i,m}a_{m,j} \sum_{k \in C_i} a_{m,k} \right) e_j \tag{3.7}
\]

Grouping together sums across \(C_i\) gives

\[
\left(\sum_{j \in D_i} a_{i,j} + a_{i,i}\right) e_i \approx -\sum_{j \in C_i} \left( a_{i,j} - \sum_{m \in F_i} a_{i,m}a_{m,j} \sum_{k \in C_i} a_{m,k} \right) e_j \tag{3.8}
\]

and solving for \(e_i\) yields

\[
e_i \approx \frac{\sum_{j \in C_i} \left( a_{i,j} - \sum_{m \in F_i} a_{i,m}a_{m,j} \sum_{k \in C_i} a_{m,k} \right) e_j}{\sum_{j \in D_i} a_{i,j} + a_{i,i}} \tag{3.9}
\]

Substituting of this into Eqn. 3.4, it follows that the interpolation weights are [34]

\[
w_{i,j} = -\frac{a_{i,j} - \sum_{m \in F_i} a_{i,m}a_{m,j} \sum_{k \in C_i} a_{m,k}}{\sum_{j \in D_i} a_{i,j} + a_{i,i}} \tag{3.10}
\]
3.2 Parallelisation

While the interpolation function for a given point is inherently parallel, and parallel algebraic smoothers are well known (See [33] for a comprehensive list of examples), the restriction operation given in Alg. 2 is inherently sequential. In order to allow parallelism C2 of the restriction criteria must be relaxed, i.e. points in C may be connected in N. This may cause C to be larger than necessary, but allows C1 of the restriction criteria to be fulfilled. This ensures that correct interpolation is preserved. Henson & Yang [32] present several implementations of parallel restriction, in this report we examine the CLJP [15] restriction operation as an introduction to such methods.

Create a matrix, $S_{i,j}$, such that $S_{i,j} = 1$ if $j \in S_i$. The $i$th row gives $S_i$ and the $j$th column gives $S^T_i$, therefore $S_{i,j}$ gives us the graph of strongly connected points in $N$. We then assign to each point in $N$ a value $\omega_i = |S^T_i| + \sigma_i$, the number of points influenced by $e_i$ plus a random number in $(0, 1)$ (used to prevent ties between points with the same number of influences). A maximally independent set of points, $P$, is selected based on the criteria $e_i \in P$ if $\omega_i > \omega_k$ for all $k \in S_i \cap S^T_i$[36]. These points are points in C. The following heuristics are then applied to $S_{i,j}$:

1. Values at C points are not interpolated, therefore neighbours that influence a C point are less valuable as C points themselves.

2. If $e_k$ and $e_j$ both depend on $e_i$ (a point in C), and $e_j$ influences $e_k$, then $e_j$ is less valuable as a potential point in C, as $e_k$ can be interpolated from $e_i$.

A point $e_i$ in N that is deemed to be ‘less valuable’ has its value of $\omega_i$ decremented. When $\omega_i < 1$, the point $e_i$ in N becomes a point in F. after values of $\omega_i$ have been updated $P$ is recalculated using the new $S_{i,j}$. This process is repeated, with points in $P$ at every iteration being added to C until all points are either in C or F. The application of these heuristic detailed above is shown in Alg. 3.
Algorithm 3 Modification of $\omega_i$

for each $e_i \in P$ do
  for each $e_j \in S_i$ do
    decrement $\omega_j$
    set $S_{i,j} \leftarrow 0$
  end for
  for each $e_j \in S^T_i$ do
    set $S_{j,i} \leftarrow 0$
    for each $e_k \in S^T_i$ do
      if $e_k \in S^T_j$ then
        decrement $\omega_j$
        set $S_{k,j} \leftarrow 0$
      end if
    end for
  end for
end for

Remove edge $S_{i,j}$

Remove edge $S_{j,i}$

Remove edge $S_{k,j}$
Chapter 4

The Lattice-Boltzmann Method

Aside from the Eulerian methods of solving fluid flow problems (where the fluid is treated as a continuum and the properties of the fluid passing through a part of the continuum are calculated), fluid flow can also be described in a Lagrangian frame of reference. In this case the velocities and locations of the particles in a fluid are calculated, and the particles themselves are ‘tracked’ across the problem space [4]. The Lattice Boltzmann method exists as a middle ground between the two methods. Calculating the properties of individual particles is computationally intensive, instead we calculate a probability distribution function (PDF) at discrete points describing the distribution of particles along specified directions. This chapter introduces the underlying equations of the Lattice-Boltzmann equation.

4.1 The Boltzmann Transport Equation

A system of particles can be described by a distribution function \( f(r, v, t) \), where \( f(r, v, t) \) is the number of particles at time \( t \) with positions between \( r \) and \( r + dr \) and velocities between \( v \) and \( v + dv \). If an external force \( F \) is applied to a particle then its velocity and position will change from \( v \) to \( v + Fdt \) and from \( r \) to \( r + vdt \) respectively. Assuming there are no collisions between particles then the effect of \( F \) on the system is [45].

\[
f(r + vdt, v + Fdt, t + dt)drdv - f(r, v, t)drdv = 0
\]  

(4.1)
With collisions between particles, there will be a change in the number of particles in $drdv$. These collisions are accounted for by an integral operator, known as $\Omega$:

$$f(r + vdt, v + Fdt, t + dt)drdv - f(r, v, t)drdv = \Omega(f)drdvdt \quad (4.2)$$

At the limit $dt \to 0$ and dividing by $dt dr dv$, Eqn. 4.2 becomes

$$\frac{df}{dt} = \Omega(f) \quad (4.3)$$

Eqn. 4.3 states that the rate of change of $f$ is equal to the rate of collision. A change in $f$, $df$, can be expressed as

$$df = \frac{\delta f}{\delta r} dr + \frac{\delta f}{\delta v} dv + \frac{\delta f}{\delta t} dt \quad (4.4)$$

Dividing by $dt$ gives

$$\frac{df}{dt} = \frac{\delta f}{\delta r} \frac{dr}{dt} + \frac{\delta f}{\delta v} \frac{dv}{dt} + \frac{\delta f}{\delta t} \quad (4.5)$$

where $\frac{dr}{dt} = v$ and $\frac{df}{dv} = a = F/m$ ($a$ is acceleration and $m$ is the molecular mass).

Eqn. 4.5 can be expressed as

$$\frac{\delta f}{\delta t} + \frac{\delta f}{\delta r} \cdot v + \frac{F}{m} \cdot \frac{\delta f}{\delta v} = \Omega \quad (4.6)$$

This is the Lattice-Boltzmann transport equation, and for a system without external forces it can be simplified to

$$\frac{\delta f}{\delta t} + \nabla f \cdot v = \Omega \quad (4.7)$$

When discretised into a lattice, the values of $v$, $f$ and $\Omega$ are only valid along $k$ links between lattice points, and Eqn. 4.7 becomes

$$\frac{\delta f_i}{\delta t} + \nabla f_i \cdot v_i = \Omega_i \quad (4.8)$$

1This operator is a double integral and is one of the key problems in solving Lattice-Boltzmann problems. A simplification of the collision integral is introduced in Section 4.2, but further details can be found in [13, 5]
4.2 The BGK Approximation

The collision operator $\Omega$ is difficult to solve [13], but can be approximated as

$$\Omega_i = \omega (f_i^{eq} - f_i) = \frac{1}{\tau} (f_i^{eq} - f_i)$$

(4.9)

This is known as the Bhatnagar, Gross and Crook (BGK) operator [5], where $f_i^{eq}$ is an equilibrium distribution for the system and $\tau$ is $\frac{1}{\omega}$, where $\omega$ is the collision frequency. The Lattice-Boltzmann transport equation can now be expressed as

$$\frac{\delta f_i}{\delta t} + \nabla f_i \cdot v_i = \frac{1}{\tau} (f_i^{eq} - f_i)$$

(4.10)

This equation can be non-dimensionalised by using a characteristic length scale $L$, a reference speed $U$, a reference density $n_r$ and the time between particle collisions $t_c$. With these values a lattice velocity, $c_i$ is defined as $v_i/U$, $\nabla$ is replaced by $\hat{\nabla} = \nabla L$, $\hat{\tau} = \tau \cdot U/L$ and replaces $\tau$, $\hat{f}_i = f_i/n_r$ and replaces $f_i$. Eqn. 4.10 then becomes [66]

$$\frac{\delta \hat{f}_i}{\delta \hat{t}} + \hat{\nabla} \hat{f}_i \cdot c_i = \frac{1}{\hat{\tau} \epsilon} (\hat{f}_i^{eq} - \hat{f}_i)$$

(4.11)

where $\epsilon = t_c U/L$, known as the Knudsen number [44]. Discretising this equation (and setting $\Delta r_i / \Delta t = c_i$ for each directional component of $r$) leads to

$$\frac{\hat{f}_i(\hat{r}, \hat{t} + \Delta \hat{t}) - \hat{f}_i(\hat{r}, \hat{t})}{\Delta \hat{t}} + \frac{\hat{f}_i(\hat{r} + c_i \Delta \hat{t}, \hat{t} + \Delta \hat{t}) - \hat{f}_i(\hat{r}, \hat{t} + \Delta \hat{t})}{\Delta \hat{t}} = \frac{1}{\hat{\tau} \epsilon} \left[ \hat{f}_i^{eq}(\hat{r}, \hat{t}) - \hat{f}_i(\hat{r}, \hat{t}) \right]$$

(4.12)

Choosing $\Delta t = t_c$ and multiplying by $\Delta \hat{t}$ gives

$$\hat{f}_i(\hat{r} + c_i \Delta \hat{t}, \hat{t} + \Delta \hat{t}) = \hat{f}_i(\hat{r}, \hat{t}) + \frac{1}{\hat{\tau} \epsilon} \left[ \hat{f}_i^{eq}(\hat{r}, \hat{t}) - \hat{f}_i(\hat{r}, \hat{t}) \right]$$

(4.13)

an explicit equation that can be solved iteratively.

More information on the equilibrium distribution function can be found in chapters 2, 4, and 5 of [45] and [28], although it generally takes the form of

$$f_i^{eq} = \Phi w_i \left[ A + B c_i \cdot u + C (c_i \cdot u)^2 + Du^2 \right]$$

(4.14)
where $\Phi$ is a scalar parameter equal to the sum of all the distribution functions, $w$ is a weighting value for each direction of $c$, $u$ is the macroscopic velocity at $\hat{f}$ and $A$, $B$, $C$ and $D$ are constants dependent on the equations of conservation for mass, momentum and energy of the system (See page 24 of [45]). The value of $u$ at $\hat{f}$ can be recovered by first calculating the density at $\hat{f}$, $\rho$

$$\rho = \sum_{i=0}^{k} \hat{f}_i$$  \hspace{1cm} (4.15)

then the momentum

$$\rho u = \sum_{i=0}^{k} \hat{f}_i c_i$$  \hspace{1cm} (4.16)

and dividing the second by the first.

### 4.3 The D2Q9 Lattice and Iterative Solutions to the Boltzmann Transports Equation

The problem domain is discretised into a lattice, where lattice points are arranged according to the ‘DmQn’ format, in which the lattice has $m$ dimensions and each lattice point is linked directly to $n$ others. For the purpose of illustration a D2Q9 lattice will be used in this report, more information on the construction of lattices can be found in [40].

A lattice point in D2Q9 lattice has nine linkages (See Fig. 4.1), denoted by $\hat{f}_i$, including link ‘0’ for the stagnant component of $\hat{f}$. By setting $\Delta \hat{x} = \Delta \hat{y} = \Delta \hat{t}$ (by convention these are set to 1) the velocities of $\hat{f}_i$, $c_i$ can be expressed as

\begin{align*}
c_0 &= (0, 0) \\
c_1 &= (1, 0) \\
c_2 &= (0, 1) \\
c_3 &= (-1, 0) \\
c_4 &= (0, -1) \\
c_5 &= (1, 1) \\
c_6 &= (-1, 1) \\
c_7 &= (-1, -1) \\
c_8 &= (1, -1)
\end{align*}

(4.17)

22
The accompanying values for $w_i$ in Eqn. 4.14 are \[2\]

\[
\begin{align*}
    w_0 &= 4/9 \
    w_{1,2,3,4} &= 1/9 \
    w_{5,6,7,8} &= 1/36
\end{align*}
\]

With this in place, Eqn. 4.13 can be solved by iteratively solving

\[
\hat{f}_i(\hat{r}, \hat{t} + \Delta \hat{t}) = \hat{f}_i(\hat{r}, \hat{t}) + \frac{1}{\tau} \left( \hat{f}_i^{eq}(\hat{r}, \hat{t}) - \hat{f}_i(\hat{r}, \hat{t}) \right)
\]

which performs the function of updating $\hat{f}_i$ with the collision operator and

\[
\hat{f}_i(\hat{r} + c_i \Delta \hat{t}, \hat{t} + \Delta \hat{t}) = \hat{f}_i(\hat{r}, \hat{t} + \Delta \hat{t})
\]

which streams the value of $\hat{f}_i$ along $c_i$ to a neighbouring lattice point.
4.4 Test Case - LBM for Isothermal Incompressible 2D Navier-Stokes equations

To solve the 2D Navier Stokes equations for incompressible, isothermal flow (See Chapter 4 of [65]) Eqn. 4.14 becomes

\[ f_i^{eq} = \rho w_i \left[ 1 + \frac{c_i \cdot u}{c^2_s} + \frac{1}{2} \left( \frac{(c_i \cdot u)^2}{c^4_s} - \frac{1}{2} \frac{u^2}{c^2_s} \right) \right] \]  

(4.21)

where \( c_s = c_i / \sqrt{3} \) [14, 29, 60]. As Eqn. 4.13 has been non-dimensionalised, the number of lattice points needed is dependent on the Reynolds number of the problem domain. This is given by

\[ Re = \hat{U} \hat{L} / \nu \]  

(4.22)

where \( \nu \) is the kinematic viscosity of the fluid and \( \hat{U} \) and \( \hat{L} \) are the characteristic velocity and length of the problem domain respectively. The number of lattice points in the direction of the characteristic length is given by \( N = \hat{L} / \Delta x \), as we have already set \( \Delta x = 1 \) it follows that \( N = \hat{L} = L \) and the Reynolds number can be expressed as

\[ Re = \hat{U} N / \nu \]  

(4.23)

It should be noted that the characteristic velocity and length of the lattice \( U \), and the kinematic viscosity associated with the lattice, \( \nu_l \), can take any values provided the Reynolds number is the same as that of the original problem domain, i.e.

\[ Re = \hat{U} \hat{L} / \nu = U N / \nu_l \]  

(4.24)

This also applies to the lattice characteristic length \( L \), if \( \Delta x \neq 1 \).

4.4.1 Lid Driven Cavity Test Case - Preliminary Results Using PGI Accelerator Compiler

A lid driven cavity test case was used to determine the suitability of the LBM for use on GPUs. \( Re \) was set to 1000(\( \hat{L} = 0.2m, \hat{U} = 6m/s \), the lid velocity and \( \nu = 1.2 \times 10^{-5} m^2/s \)), and in the first instance 100 lattice points were used in each direction, allowing values for \( U_l \) and \( \nu_l \) to be 0.1 and 0.01 respectively.
**Boundary Conditions**

The treatment of boundary conditions for non-moving boundaries is simple, lattice points are placed at the boundary, and and values of \( \hat{f}_i \) that would be streamed past the boundary are ‘bounced back’ into the domain i.e.

![Figure 4.2: Boundary Lattice Points](image)

where, for example, at the Southern boundary \( \hat{f}_7, \hat{f}_4 \) and \( \hat{f}_8 \) are ‘bounced’ to \( \hat{f}_5, \hat{f}_2 \) and \( \hat{f}_6 \) respectively. In the lid driven cavity test case lattice points on the Northern boundary have a prescribed, and non-dimensionalised against \( U_l \), macroscopic velocity \( u \) (in this particular example \( U_l \) is the velocity of the lid and therefore \( u_x = 1 \) and \( u_y = 0 \)). By taking the unknown values of \( \hat{f}_i, \hat{f}_{7,4,8} \), and applying the equilibrium scheme [42, 30] which assigns an unknown \( \hat{f}_i \) to be it’s corresponding \( \hat{f}_i^{eq} \) the ‘bounceback’ scheme can be used to give

\[
\hat{f}_{7,4,8} - \hat{f}_{5,2,6} = \hat{f}_{7,4,8}^{eq} - \hat{f}_{5,2,6}^{eq}
\]  

(4.25)

Calculating the values of \( \hat{f}_i^{eq} \) using Eqn. 4.21 yields

\[
\hat{f}_4 = \hat{f}_2
\]  

(4.26)

\[
\hat{f}_7 = \hat{f}_5 - \frac{\rho u_x}{6}
\]  

(4.27)

\[
\hat{f}_8 = \hat{f}_6 + \frac{\rho u_x}{6}
\]  

(4.28)

\[
(4.29)
\]
The PGI Accelerator Compiler

The PGI Accelerator Compiler allows code to be executed on a GPU by using a set of directives, in a similar manner to how OpenMP [56] allows for parallelisation across multiple CPU cores. Code to be executed on the GPU is surround by ‘acc region’ and ‘end acc region’ pragmas. These also notify the compiler that any data within the ‘acc region’ needs to be copied to the GPU and then back to the host at the end of the ‘acc region’. The simplicity of executing code on the GPU allowed the GPU implementation of the test case to be coded quickly based on the CPU version. More detail on the PGI Accelerator Compiler can be found in[63].

Results

100 iterations of the Lattice Boltzmann method were completed on a Fortran based CPU code (running on an Intel Xeon 5620) and the PGI Accelerator code (running on a Fermi based Nvidia Quadro 6000 GPU). As the lattice size was increased the value of \( U_l \) was adjusted to maintain a Reynolds number of 1000.

<table>
<thead>
<tr>
<th>Number of Lattice Points</th>
<th>CPU Code</th>
<th>Accelerator Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.5s</td>
<td>1.13s</td>
</tr>
<tr>
<td>250000</td>
<td>10.68s</td>
<td>10.42s</td>
</tr>
<tr>
<td>1000000</td>
<td>47.17s</td>
<td>35.48s</td>
</tr>
<tr>
<td>4000000</td>
<td>198.7s</td>
<td>134.51s</td>
</tr>
<tr>
<td>16000000</td>
<td>837.5s</td>
<td>530.24s</td>
</tr>
</tbody>
</table>

Table 4.1: Execution Times for 100 Iterations of LBM on a Lid Driven Cavity

Optimisation-Data Movement in Memory

As Table 4.1 shows, initial performance of the accelerator code is poor, with a maximum speedup over serial CPU code of 1.5\( \times \). Use of the analysis tool present in the Nsight Parall Debugger reveals that a vast majority of the execution time is spent passing data to and from the host/GPU.

By using a ‘data region’ directive to explicitly copy \( \hat{f}, \hat{f}^{eq}, u, v \) and rho to the GPU at the beginning of each iteration of the main loop and back to the host at the end, memory transactions between the host and GPU within each iteration are removed. The memory transactions for ten iterations using this method
are shown in Fig. 4.5. The removal of the intra-iteration memory transactions improves the performance of the code, for the same 100 iterations test used in Table 4.1 the execution times now become:

<table>
<thead>
<tr>
<th>Number of Lattice Points</th>
<th>Data Region used Inside Main Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.67s</td>
</tr>
<tr>
<td>250000</td>
<td>3.41s</td>
</tr>
<tr>
<td>1000000</td>
<td>9.56s</td>
</tr>
<tr>
<td>4000000</td>
<td>32.31s</td>
</tr>
<tr>
<td>16000000</td>
<td>122.58s</td>
</tr>
</tbody>
</table>

Table 4.2: Execution Times For LBM With Data Regions Inside Main Loop

Further host-GPU memory transactions can be removed by placing the ‘data region’ outside the main loop of the program. This currently means that values of the calculations cannot be monitored on-screen whilst the program runs, future work will involve moving a single values from GPU-host to fulfil this purpose. With the further reduction of memory transactions execution time can be improved to:

<table>
<thead>
<tr>
<th>Number of Lattice Points</th>
<th>Data Region Used Outside Main Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.35s</td>
</tr>
<tr>
<td>250000</td>
<td>1.59s</td>
</tr>
<tr>
<td>1000000</td>
<td>3.75s</td>
</tr>
<tr>
<td>4000000</td>
<td>10.26s</td>
</tr>
<tr>
<td>16000000</td>
<td>37.176s</td>
</tr>
</tbody>
</table>

Table 4.3: Execution Times For LBM With Data Regions Inside Main Loop

Execution times for all implementations are summarised in Fig. 4.3.
Figure 4.3: Summary of All LBM Implementations
Figure 4.4: Memory Transfers

Figure 4.5: Accelerator Data Region Inside Main Loop

Figure 4.6: Accelerator Data Region Outside Main Loop
Chapter 5

Optimization of a Geometric Multigrid Method For Nvidia Fermi Based GPUs (Draft Paper)
Optimization of a Double Precision Geometric Multigrid Method For Nvidia Fermi Based GPUs

M. J. Mawson, A. Revell, R. Prosser

ABSTRACT

This paper presents the implementation and optimization of a geometric multigrid method on a Fermi based Nvidia Quadro 6000 GPU. Comparisons are made with a serial CPU implementation, and performance metrics are compared against theoretical limits. Optimization improved performance by approximately 50% and was found to be limited by the bandwidth of the GPU.

1. INTRODUCTION

High Performance Computing (HPC) is coming to a crossroads in terms of the hardware being used. Traditional clusters of Central Processing Units (CPUs) are being supplemented with accelerator hardware, massively parallel devices designed with HPC in mind. These devices pose a new challenge for software developers as optimisation of code for accelerator architecture is a relatively new field.

The field of Computational Fluid Dynamics (CFD) will be affected by this change in hardware, as many CFD problems require the use of large computational resources. The writing and optimisation of CFD algorithms for accelerator technology is therefore becoming an important topic; while the numerical methods for solving CFD problems improve at a relatively slow rate, Moore’s law [Moore 1998] indicates that the number of components on a chip (a good indicator of chip performance) doubles roughly every two years. In previous years this has equated to a doubling of the clock speed of the processor, but due to the amount of heat generated running at very high clock speeds this is no longer deemed an acceptable method of increasing performance.

Hardware manufacturers now increase the number of cores on a chip to maintain Moore’s law, culminating in accelerator technology which foregoes the complicated cores found in CPUs in favour of of simpler cores. With a well written program, an accelerator can consistently out-perform current CPUs by an order of magnitude or more [Cohen and Molemaker 2009, Griebel and Zaspel 2010].

This paper confirms the suitability of optimization techniques suggested in [NVIDIA 2010b] for the geometric multigrid solver, and provides a theoretical limit for the performance of such a solver based on the bandwidth available.

2. BACKGROUND

2.1 GPU Technology

Modern GPUs utilize a Unified Shader Architecture [Luebke et al. 2007], in which blocks of processing cores capable of performing operations within all parts of the graphics pipeline are favoured over architectures in which the hardware architecture matches the flow of the graphics pipeline. The generic architecture of a GPU is shown in Fig.1 and its comparison with a generic CPU highlights the following key differences:

- GPUs sacrifice larger amounts of cache and control units for more processing cores.
- Cores are grouped together into blocks known as Stream Multiprocessors (SM).
- The cores of a SM take up less die space than those of a CPU, and are therefore simpler in design.

![Figure 1: GPU and CPU architectures](image)

These attributes make GPUs suitable for performing computation on large datasets where little control is needed, i.e. the same task is performed across the entire dataset. It is this aspect which has created interest in GPU computing for...
HPC, and CFD in particular as the same set of governing equations is applied to each point in the dataset.

2.1.1 The Quadro 6000 GPU

The GPU used in this paper (Nvidia Fermi based Quadro 6000) consists of fourteen Stream Multiprocessors (SM), each containing 32 Single Instruction Multiple Thread (SIMT) processing cores, a 64KB joint shared memory/cache and various control and special function units. Fig 2 illustrates the architecture of a single SM inside a Quadro 6000. A 768KB layer of L2 cache is shared between all SM’s to provide cache coherency. All SMs have access to 6GB of on-board Device RAM (DRAM)

![Figure 2: A Fermi Streaming Multiprocessor](image)

2.1.2 Threads and Blocks of Processing

Code written for GPUs in the CUDA programming language is parallelized at two levels; computation is divided into blocks which contain component parallel threads (Fig. 3). A single block of threads is allocated to a SM at any one time, with the component threads divided into groups of 32 called ‘warp’. Each warp of threads is executed in SIMT fashion on a SM; this can lead to performance issues when branching, which is discussed in Section 4.1.1. Fig. 3 shows an example of threads and blocks being allocated two-dimensionally; division of threads and blocks can be performed in one to three dimensions depending on the problem.

![Figure 3: Threads and Blocks in CUDA Programming](image)

2.2 The Geometric Multigrid Method

The geometric multigrid method is used to solve \( \Delta u = f \) of the form \( \Delta u = f \) where \( \Delta \) denotes a square matrix, \( f \) defines a known vector and \( u \) the unknown vector to be found. The implementation presented in this paper utilizes a finite difference method on a square domain of unit dimensions, allowing the \( \Delta u = f \) to be discretized into discrete points.

\[
A^{-1}f - u^m
\]

(1)

\[
M^{-1}(f - Au^m)
\]

(2)

Where \( M \) is easier to invert that \( A \) and \( M^{-1}A \) approaches the identity. This approximation of the error can be removed from \( u \), and a new value of can be calculated.
Complete introductions to the multigrid method can be found in [Brandt 1977, Briggs 1987, Wesseling 1992]. The method presented in this paper utilizes the ‘V-Cycle’ form of multigrid, whereby is repeatedly coarsened to the smallest grid, and then interpolated back through the grid levels. Various patterns of coarsening and interpolation can be used, though these will not be covered in this paper (See Chapter 8 of [Wesseling 1992] for more information). Pseudocode for the V-Cycle implementation is given in Alg. 1.

**Algorithm 1: V-Cycle Multigrid Method**

\[ u_t = M G(A_t, f_t, u_t, L) \]

Call multigrid function at grid \( L \)

\[ \text{function } M G(A_t, f_t, u_t, L) \]

Prototype for the MG

for \( i = 1 \) to \( \text{Number of PreSmoothing steps} \) do

\[ u_t \leftarrow S_{P_c}^M(u_t, f_t) \]

Smooth \( A u = f \)

end for

\[ r_t = f_t - A_t u_t \]

Calculate the residual

\[ r_{t-1} = R_t f_t \]

Restrict residual to next grid level

if \( l = 1 \) then

Solve exactly \( A_{t-1} u_{t-1} = r_{t-1} \)

else

\[ e_{t-1} = M G(A_{t-1}, f_{t-1}, r_{t-1}, L) \]

Recursive call to MG using next coarsest grid

end if

\[ t = P_{t-1} t_{t-1} \]

Prolong error to current grid

\[ u_t = u_t + t \]

Add error to current solution

for \( i = 1 \) to \( \text{Number of PostSmoothing steps} \) do

\[ v_t \leftarrow S_{P_c}^M(u_t, f_t) \]

Remove interpolation errors

end for

3. IMPLEMENTATION

The initial implementation of the multigrid method is shown in this section, optimizations of the various kernels are discussed in Section 4.1. The program solves a finite difference approximation of the 2D Laplace problem

\[
\frac{d^2 u}{dx^2} = f(x)
\]  

on a square grid of unit dimensions.

3.1 Smoothing Operation - Red-Black Gauss-Seidel

By choosing \( M = D - L \) in the basic iterative method given in Eq. 3 the Gauss-Seidel iteration is formed. For the Laplace problem discretised using a 2D finite difference scheme this can be expressed as [Zhang 1996]

\[
\begin{align*}
   u_{i,j}^{m+1} &= \frac{1}{4} \left[ u_{i+1,j}^m + u_{i-1,j}^m + u_{i,j+1}^m + u_{i,j-1}^m + h^2 f_{i,j} \right]
\end{align*}
\]  

where \( h \) is the spacing between grid points. The rate of con-
3.4 Interpolation

Whilst the residual calculation and the restriction are straightforward to parallelize, the prolongation operation must be performed in several steps. This requires synchronisation when implementing the operation in parallel. At an abstract level of implementation there must be at least two steps to the prolongation operation; the direct mapping of elements of the coarse grid to the fine grid and interpolation between the mapped points to complete the fine grid. The implementation of the prolongation operator presented in this work uses linear interpolation between the mapped points. In this way the interpolation operation consists of three stages:

\[\text{Algorithm 2: Interpolation}\]

\[u_{ix,iy} = \frac{1}{2}(u_{i,x+1} + u_{i,x-1}) \quad \text{Horizontal Interpolation}\]

4.1 Analysis and Optimization

Use of the Nvidia’s analysis and debugging tools allowed performance analysis and optimization of the multigrid method to be performed. Use of the analysis tool and guidelines from [NVIDIA 2010b, Sinek et al. 2009, Cohen and Molemaker 2009] highlighted the following possibilities for optimization:

- Removal of branching operations.
- Use of cache and/or shared memory.
- Improving memory access times by coalescing accesses.

4.1.1 Removal of Branching Operations

CPU parallelization of the Red Black method simply involves looping across the \(x\) and \(y\) directions of \(u\) with a branching operation to determine whether or not the current element is ‘red’ or ‘black’. Launching a thread for every element of \(u\) and branching proves to be very inefficient, as a thread which does not satisfy a branching condition stalls, effectively halving the number of active threads. This can be remedied by performing iterations on two elements of \(u\) in a single thread, a ‘red’ or ‘black’ element and it’s North-East or North-West neighbour respectively. This allows ‘red’ or ‘black’ elements to be indexed directly, without branching. For ‘red’ elements of \(u\) the indexing is performed as follows:

```c
// Thread Indices
int x = (2 * blockIdx.y * blockDim.y) + (2 * threadIdx.y);
int y = (2 * blockIdx.x * blockDim.y) + (2 * threadIdx.y);
```

4. RESULTS

100 iterations of the V-Cycle were performed on a 2D problem with varying grids of size \(N \times N\), for comparison a CPU implementation was also executed on an Intel Xeon 5620. The initial results are presented in Table 1, a speedup of approximately 8x can be seen on the largest grid size.

4.1.1 Removal of Branching Operations

CPU parallelization of the Red Black method simply involves looping across the \(x\) and \(y\) directions of \(u\) with a branching operation to determine whether or not the current element is ‘red’ or ‘black’. Launching a thread for every element of \(u\) and branching proves to be very inefficient, as a thread which does not satisfy a branching condition stalls, effectively halving the number of active threads. This can be remedied by performing iterations on two elements of \(u\) in a single thread, a ‘red’ or ‘black’ element and it’s North-East or North-West neighbour respectively. This allows ‘red’ or ‘black’ elements to be indexed directly, without branching. For ‘red’ elements of \(u\) the indexing is performed as follows:

```c
// Thread Indices
int x = (2 * blockIdx.y * blockDim.y) + (2 * threadIdx.y);
int y = (2 * blockIdx.x * blockDim.y) + (2 * threadIdx.y);
```

Listing 1: Non-branching Indexing for Red Elements

Table 2 shows the execution time to update all the ‘red’ elements on a grid of size 4097×4097 when branching to differentiate between ‘red’ and ‘black’ elements and using direct indexing.

<table>
<thead>
<tr>
<th>Indexing Method</th>
<th>Execution Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branching</td>
<td>7367.298</td>
</tr>
<tr>
<td>Non-branching</td>
<td>3561.042</td>
</tr>
</tbody>
</table>

Table 2: ‘Red’ Sweep with Branching and Non-Branching Indexing.

Branching operations within the interpolation kernels (these can be seen in Alg. 3) were also replaced in a similar fashion. With branching operations only remaining to reinforce boundary conditions the execution times improved to:

<table>
<thead>
<tr>
<th>(N)</th>
<th>Original</th>
<th>Reduced Branching</th>
</tr>
</thead>
<tbody>
<tr>
<td>257</td>
<td>0.14s</td>
<td>0.138s</td>
</tr>
<tr>
<td>513</td>
<td>0.29s</td>
<td>0.278s</td>
</tr>
<tr>
<td>1025</td>
<td>0.81s</td>
<td>0.677s</td>
</tr>
<tr>
<td>2049</td>
<td>2.75s</td>
<td>2.19s</td>
</tr>
<tr>
<td>4097</td>
<td>10.48s</td>
<td>8.37s</td>
</tr>
</tbody>
</table>

Table 3: Execution Time for 100 V-Cycles M ultigrid Sweeps With Reduced Branching.
4.1.2 Cache and Shared Memory Use

L1 cache and shared memory are share the same physical space on Fermi GPUs. Of the 64KB available on each SM, it can be split 48KB-16KB between the two types of memory. Favoring cache allows performance to be improved with no changes to the program, whereas shared memory requires a specific implementation within each kernel that uses it. The execution times for favouring a larger L1 cache are shown in Table 4.

<table>
<thead>
<tr>
<th>N</th>
<th>Reduced Branching</th>
<th>Large L1 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>257</td>
<td>0.138s</td>
<td>0.127s</td>
</tr>
<tr>
<td>513</td>
<td>0.273s</td>
<td>0.246s</td>
</tr>
<tr>
<td>1025</td>
<td>0.677s</td>
<td>0.636s</td>
</tr>
<tr>
<td>2049</td>
<td>2.19s</td>
<td>2.08s</td>
</tr>
<tr>
<td>4097</td>
<td>8.37s</td>
<td>7.85s</td>
</tr>
</tbody>
</table>

Table 4: Execution Time for 100 V-Cycles Multigrid Sweeps With Large L1 Cache.

The use of shared memory involves a block of threads gathering all of the data it will need to execute and storing it in a single array that all threads in the block can access. In the case of stencil operations (such as the Red-
6. REFERENCES


---

Figure 8: Comparison of Speedups Against CPU for 100 Iterations of V-Cycle for Unoptimized/Optimized CUDA Implementations

GFLOPS. Given that the peak performance of the Quadro 6000 is 500 GFLOPS in double precision, this may seem like poor performance. However, the multigrid method is known to be memory bound [Goodnight et al. 2003, Göddeke and Strzodka 2010], therefore the metric against which performance should be measured is the memory bandwidth of the GPU. Assuming a 100% cache hit ratio (which is impossible but serves to illustrate a point) the memory bandwidth is approximately equal to 400 GB/sec [NERSC 2011]. The multigrid algorithm uses 52 load/store operations/element/cycle, giving a ratio of memory-to-floating point operations of 1.156. As each memory operation operates on a double precision value, 50×10⁶ values can be moved per second. Dividing by the memory/floating point ratio gives a theoretical performance limit of 43.25 GFLOPS. With an assumption of no cache hits, the performance is limited by the bandwidth of global memory (89 Gb/sec) and falls to 9.62 GFLOPS. A comparison with the 100% cache hit scenario shows our implementation to be approximately 25% efficient.

5. CONCLUSION

A double precision geometric multigrid method has been presented and optimization has been performed to improve speedup relative to a serial CPU implementation from 8× to 12×. The multigrid method has been shown to be bandwidth limited and theoretical performance limits have been calculated based on the number of memory operations required per element in the problem domain. Fig. 9 highlights the effects that different optimization methods have had on the execution time. It can be seen that the removal of branching operations has the largest impact on performance (approximately 25%), followed by padding arrays to increase coalesced memory accesses (approximately 9%). This supports optimization recommendations in [NVIDIA 2010b, Simek et al. 2009, Cohen and Molenaker 2009].
Figure 9: Comparison of Optimization Methods
Chapter 6

Conclusion and Future Work

This report has presented the following:

• An optimized double precision geometric multigrid method has been presented, and performance metrics and limits calculated. The final implementation yielded a $12 \times$ speedup over a CPU implementation.

• Initial work on the Lattice Boltzmann method which shown it to be suitable for parallelisation on GPU, yielding a speedup over a CPU implementation of $22 \times$ in single precision.

• An introduction to the theory of algebraic multigrids, building on the theory of geometric multigrids presented in last years report.

6.1 Further Work

Completion of a GPU based Algebraic Multigrid Method compatible with an open source CFD code (Code_Saturne or OpenFOAM) will make up the bulk of the remaining work, and can be broken down into several tasks. Approximation timescales are given:

• Reading and loading standard mesh files to the GPU (October-December 2011)

• Algebraic smoothing operations (January-April 2012).

• Parallel algorithms for grid restriction/interpolation (April-July 2012).
Several smaller tasks to run in parallel with programming the algebraic multi-grid include, but are not limited to:

- Expansion and continuation of work on Lattice Boltzmann methods.
- Profiling and optimising an existing SPH code.
- Further use of the PGI Accelerator to develop GPU implementations of existing CFD code.
Glossary

Computing Terminology

**ALU** Arithmetic Logic Unit - An execution unit responsible for integer arithmetic, logic and bit shifting operations and moving data to/from the CPU registers.

**Cache** Small, fast memory used to hide latency between a processor and main memory.

**Core** A unit containing all of the components of a CPU necessary to complete an instruction cycle, and some cache.

**CPU** Central Processing Unit - A processor responsible for execution of instructions and control of all other components.

**FPU** Floating point unit - An execution unit capable of performing floating point operations.

**FLOPS** FLoating point OPerations per Second.

**GPU** Graphics Processing Unit (Also commonly referred to as Graphics Card or Device). A piece of hardware responsible for rendering images, usually converting 3D vector information about a scene into a 2D representation with pixels.

**Main Memory** The memory a program (and all of its data) is stored in whilst the program executes. This is typically RAM.

**Processor** A chip containing a number of cores and any on chip memory hierarchy. In the case of a GPU a processor contains several groups of cores (See SM).
**RAM**  Random Access Memory.

**Register**  Small, high speed memory for storing the operands, instructions and results of CPU execution.

**SFU**  Special Function Unit - An execution unit designed to perform a specific task quickly, e.g. calculating square roots.

**Thread**  A single unit of executable work (i.e. a program).

**CUDA Terminology**

**Block**  A collection of several threads assigned to a single SM.

**CUDA**  Compute Unified Device Architecture.

**DP**  Double Precision core.

**Grid**  The collection of all blocks over which a kernel operates.

**Host**  The computer on which a GPU is mounted.

**Kernel**  The portion of a program that executes on one or more GPUs.

**SM**  Stream Multiprocessor - Several different types of cores grouped together to share resources.

**SP**  Single Precision core.
Bibliography


integral. Physics of Plasmas 6, 7 (1999), 2947.


selection for parallel algebraic multigrid. Solving Irregularly Structured Prob-

[16] COHEN, J., and MOLEMAKER, M. A fast double precision CFD code using 

[17] CORLESS, R., FRANK, G., and GRAHAM, J. Chaos and continued frac-

[18] DONGARRA, J., and BECKMAN, P. International Exascale Software 
(Hamburg, 2010).

and Engineering 8, 6 (Nov. 2006), 24–33.

[20] FELDMAN, M. PathScale Looks to One-Up CUDA, OpenCL with New 
GPU Compiler. http://www.hpcwire.com/features/PathScale-Looks-to-One-

Eie Frigaard Introduction to N-body Solvers.

[22] GARLAND, N. B. . M. Efficient Sparse Matrix-Vector Multiplication on 

[23] GOODNIGHT, N., WOOLLEY, C., LEWIN, G., LUEBKE, D., and 
HUMPHREYS, G. A multigrid solver for boundary value problems using programmable graphics hardware. In Proceedings of the ACM SIG-
GRAPH/EUROGRAPHICS conference on Graphics hardware (2003), Euro-
graphics Association Aire-la-Ville, Switzerland, Switzerland, pp. 102–111.


[27] Harris, M. Optimizing Parallel Reduction in CUDA. Presentation Slides.


