SCIENTIFIC COMPUTING ON STREAMING PROCESSORS

A Thesis Outline Presented

by

SANDEEP MENON

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE

September 2007

Mechanical and Industrial Engineering

SCIENTIFIC COMPUTING ON STREAMING PROCESSORS

A Thesis Outline Presented

by

SANDEEP MENON

Approved as to style and content by:

J. Blair Perot, Chair

David P. Schmidt, Member

Moon K. Kim, Member

Donald Fisher, Graduate Program Director
Mechanical and Industrial Engineering

ABSTRACT

SCIENTIFIC COMPUTING ON STREAMING PROCESSORS

SEPTEMBER 2007

SANDEEP MENON

B.E., PSG COLLEGE OF TECHNOLOGY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor J. Blair Perot

High performance streaming processors have achieved the distinction of being very efficient and cost-effective in terms of floating-point capacity, thereby making them an attractive option for scientific algorithms that involve large arithmetic effort. Graphics Processing Units (GPUs) are an example of this new intiative to bring vector-processing to desktop computers; and with the advent of 32-bit floating-point capabilities, these architectures provide a versatile platform for the efficient implementation of such algorithms. To exemplify this, the implementation of a Conjugate Gradient iterative solver for PDE solutions on unstructured two- and three-dimensional grids using such hardware is described. This would greatly benefit applications such as fluid-flow solvers which seek efficient methods to solve large sparse systems.

The implementation has also been succesfully incorporated into an existing object-oriented CFD code, thereby enabling the option of using these architectures as efficient math co-processors in the computational framework.

TABLE OF CONTENTS

Page

ABSTRACT . . . . . **iii**

LIST OF FIGURES . . . . . **vii**

CHAPTER

1. INTRODUCTION **1**

1.1 Motivation 1

1.1.1 The Memory Bottleneck 2
1.1.2 Graphics Processors and the Stream Processing Paradigm 4
1.1.3 Multiple Cores 6

1.2 Drawbacks 8
1.3 Prior work on Streaming Processors 9

2. MAPPING ALGORITHMS TO GRAPHICS PROCESSORS **12**

2.1 An Example 14

2.1.1 Setting up the graphics API 15
2.1.2 Creating arrays on the GPU 15
2.1.3 Defining the kernel program 17
2.1.4 Compiling the kernel program 18
2.1.5 Specifying inputs to the GPU 19
2.1.6 Specifying outputs for the GPU 20
2.1.7 Setting the viewport 22
2.1.8 Running the kernel program 23

2.2 Results 25
2.3 Comments 27

3. ARRAY LAYOUTS **28**

3.1 Algorithm 29
3.2 Results 31

4. REDUCTIONS **34**

4.1 Algorithm 36
4.2 A different approach 38
4.3 Results 41
4.4 Comments 44

5. SPARSE MATRIX OPERATORS **46**

5.1 Mesh data-structures 46
5.2 Results 51
5.3 Handling of Boundary Conditions 57

6. THE CONJUGATE GRADIENT ALGORITHM **61**

6.1 Node-based Discretization 63
6.2 Performance Results 64

7. IMPLEMENTATION OF THE NAVIER STOKES EQUATIONS ON GRAPHICS PROCESSORS **67**

7.1 Equations 67
7.2 Discretization 68
7.3 The Classical Fractional Step Method 71
7.4 The Exact Fractional Step Method 72
7.5 Performance Results 73

8. FUTURE WORK **77**

APPENDICES

A. MEMORY HANDLING ON THE GPU **79**

A.1 Creating Arrays 79
A.2 Transferring data from main memory to GPU arrays 81
A.3 Transferring data from GPU arrays to main memory 82
A.4 Algorithm: Mapping arrays on memory to GPU arrays 83

B. SOURCE CODE FOR REDUCTIONS **84**

B.1 Sum reduction of rectangular arrays 84

BIBLIOGRAPHY **86**

LIST OF FIGURES

Figure **Page**

1.1 Schematic representation of a cache-based architecture 3

1.2 Peak floating-point performance of GPUs over the last four years 6

1.3 Memory bandwidth comparison 7

1.4 Gather operations are natively supported on the GPU, while scatter operations are not 8

2.1 Memory layouts on the GPU vs. the CPU 12

2.2 Simplified represenation of stages in the Graphics Pipeline 13

2.3 Comparison of internal formats for GPU Arrays 16

2.4 Linear interpolation of two-dimensional rectangular array indices 24

2.5 Performance of the axpy operation vs. problem size 26

3.1 Simplified reduction example: Finding the array-maximum 30

3.2 Percentage padding vs. requested array size (32x32) 32

3.3 Percentage padding vs. requested array size (64x64) and (128x128) 33

4.1 Reduction of rectangular arrays: Padding approach 35

4.2 Interpolation of indices in a sum-reduction 37

4.3 Reduction methods: (a) local reduction (b) quarter reduction 39

4.4 Performance of the sum operation 42

4.5 Performance of the sum operation for various cases of R_{min} 43

4.6 Performance of the dot-product operation with different approaches 44

5.1 Unstructured tetrahedral mesh of a crankshaft (from NetGen). This particular mesh consists of 37151 nodes, 178486 cells, 370319 faces and 228983 edges. 47

5.2 Performance of the gradient operator. Problem Size denotes the number of faces in the mesh. 52

5.3 Performance of the divergence operator. Problem Size denotes the number of cells in the mesh. 53

5.4 Performance of the curl operator. Problem Size denotes the number of edges in the mesh. 54

5.5 Performance of the interpolation operator. Problem Size denotes the number of cells in the mesh. 55

5.6 Performance of the integration operator. Problem Size denotes the number of faces in the mesh. 56

6.1 Dual mesh cell (formed by the bold lines and shown with dual-face normals) represents a nodal control volume for the enclosed node. 63

6.2 Typical mesh used for performance evaluation 64

6.3 Performace comparison of the Conjugate Gradient solver using the node-based discretization of the Poisson equation. Problem size denotes the number of nodes in the mesh. 65

6.4 Contour plot for temperature along the Crankshaft 66

7.1 Unstructured staggered mesh scheme for the incompressible Navier Stokes equations 69

7.2 Typical mesh used for performance evaluation 74

7.3 Classical Fractional Step - Performance comparison of the Conjugate Gradient solver for the Momentum equation. Problem size denotes the number of faces in the mesh. 75

7.4 Classical Fractional Step - Performance comparison of the Conjugate Gradient solver for the Pressure equation. Problem size denotes the number of cells in the mesh. 76

7.5 Exact Fractional Step - Performance comparison of the Conjugate Gradient solver for the Streamfunction equation. Problem size denotes the number of edges in the mesh. 76

CHAPTER 1

INTRODUCTION

1.1 Motivation

Most engineering and scientific problems today are analyzed by solving the governing physical equations that define their underlying behaviour. Simulation of phenomena such as fluid-flow, heat-transfer and structural mechanics provides a cost-effective alternative to tests on physical models, and are also versatile as they can account for several modifications to the test environment without incurring significant overhead costs.

The numerical simulation process involves the discretization of the partial differential equations (PDEs) that govern the physical process. The discretization process divides a continuum domain into many smaller elements, thereby yielding a set of algebraic equations that must be solved to obtain the values of the variables at each element. In most cases, the solution techniques for these systems are iterative in nature; where a set of operations are repetitively applied on an initial guess for the unknowns, until the system converges to a solution within a specified tolerance. These simulations typically involve arithmetic operations on large sets of data, sometimes spanning several thousands or even millions of unknown variables; and tend to consume a significant amount of time on computers dedicated to the task. For large-scale simulations like weather-prediction, several computers work on the given task in parallel, in an effort to minimize this computation time - a paradigm that is commonly referred to as parallel processing. A Beowulf cluster is a variant of the parallel-processing paradigm that uses inexpensive computer hardware elements in order to

achieve better performance at a fraction of the cost. However, individual elements of these clusters consist of processors that were designed for efficient implementation of sequential code, and not for the manipulation of large sets of data, which is the case in numerical simulations. Thus, aside from the fact that these clusters are inexpensive, they prove to be a very poor fit for the kinds of tasks being performed on them.

Supercomputers, like the older Crays, were well-suited to scientific applications mainly because they implemented vector-processing capabilities which were optimized for performing mathematical operations on large arrays of data simultaneously. General-purpose tasks, however, involve much more complex operations like task-switching and branch-prediction, which is the prevalent form of work-loads dominating personal computing today. Given the narrow user-base for vector processing, it is only natural that the market forced vector-processors out of contention, thereby giving way to scalar processors that are expected to perform tasks involving heavy data reuse and infrequent accesses to memory.

1.1.1 The Memory Bottleneck

These days, scalar processors operate at much higher clock frequencies than the memory modules for a conventional computer system, and it is not uncommon to see a 3GHz processor being used in conjunction with a memory module that operates at a clock-rate of 400MHz, as determined by the front side data bus. The front side bus is a physical data bus that relays information between the processor and other devices such as the random-access-memory (RAM) and hard-disks. It has been an order of magnitude slower than the processor in terms of clock frequency for the past few years. Thus, despite the fact that a processor can execute individual instructions quickly, it must wait for many processor-cycles before data is retrieved from main-memory. For situations where memory accesses are more common, these processors

attempt to address the memory-fetch latency issue with the use of on-chip memory caches.

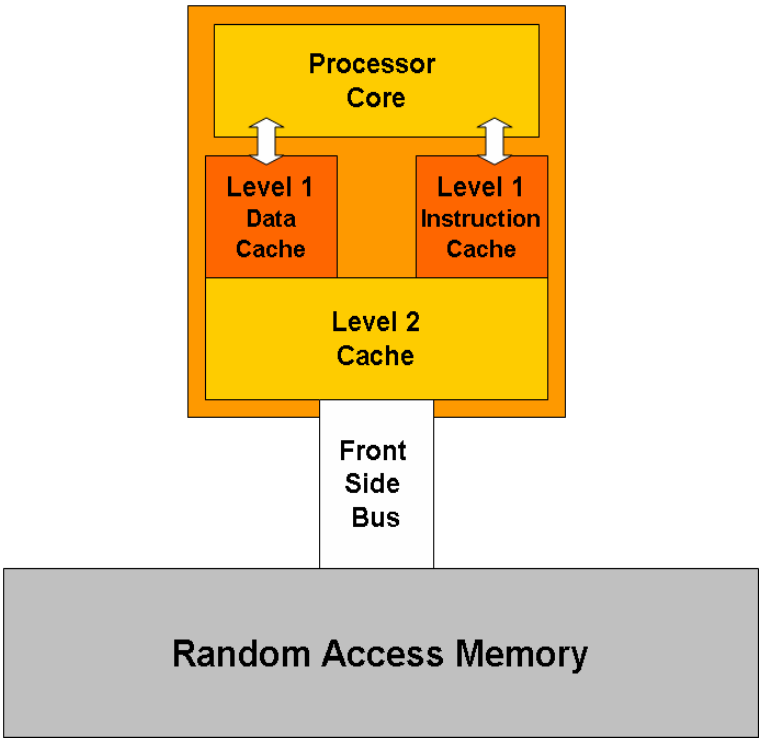


Figure 1.1. Schematic representation of a cache-based architecture

When a memory-fetch is requested, data from the memory-locations adjacent to the requested location is also typically brought in to a local cache on the processor. This is done in anticipation of the situation where the next memory-fetch instruction would request data from a location that is already fetched into the cache, thereby avoiding the round trip latency. These local caches are somewhat limited in terms of size, which in turn limits the amount of data that can be kept on them at any given time. Therefore, the data layout in main memory must be *spatially coherent* for this strategy to be effective. (A typical front-side-bus rating of 400MHz includes spatial locality. Random memory access in such cases is much slower.) Additionally, it is always desirable that data in cache be used repeatedly over time to avoid multiple

memory fetch requests, thereby forcing the data in main memory to be *temporally coherent* as well.

Cache-based CPU architectures work well for tasks that involve relatively small quantities of data that are frequently reused, such as word-processing for instance. But considering the nature of numerical simulations and other applications such as audio, video, and graphics-intensive multimedia, which involve large data sets, the limited cache of the processor does not suffice, and so the cache-based strategy is effectively useless in such a paradigm. These applications are unlikely to involve any form of data re-use which is local in time, thereby obviating the temporal aspect as well. In such a situation, the system is completely memory-bound, and the effectiveness of the computation is determined solely by the clock-rate of the front side bus. Thus, a processor that has a theoretical peak of close to 3 GigaFlops now performs at only a tenth of that rate.

1.1.2 Graphics Processors and the Stream Processing Paradigm

The rapid growth of the entertainment industry in recent times has driven the need for commodity hardware that is capable of efficiently processing media-rich applications like video decoding and graphics-intensive computer games. These applications work on large fields of audio-visual data. This leads to a situation where vector-processing capabilities are desired at relatively low-cost. Graphics Processing Units (GPUs) were designed with precisely that idea in mind.

Graphics Processing Units fall into the stream-processing paradigm, which is a variant of parallel-computing. In this case, a fixed set of operations, called a *kernel*, is applied to a each element in a continuous stream of data, like a large array, for instance [10] [14]. In most situations, these kernels consist of operations that involve very little dependencies on other elements in the stream, and so, the data stream can also be split into several streams over multiple processing elements, thereby em-

phasizing the data-parallel nature of this paradigm. These situations frequently arise in graphics applications and in scientific algorithms as well. Such operations involve either very little or no data re-use, and so cache-sizes in these architectures are minimized to provide only basic functionality. Since the need for large caches is now no longer necessary, transistor resources that were dedicated to memory can now be used for increased computational resources on the chip, and this results in performance growth-rates that outperform the oft-quoted Moore's Law for CPUs, which predicts transistor densities doubling every eighteen months. The survey by Owens et al [23] for instance, estimates peak graphics hardware performance doubling roughly every six months, and Fig. 1.2 provides a rough estimate of the rate at which peak floating-point performance for graphics processors has progressed over the years, when compared to traditional cache-based processors. However, these statistics do not represent practical situations as the ideal conditions required for such performance numbers rarely exist. Thus, even though the figure indicates performance numbers in range of hundreds of Gigaglops, it is an order of magnitude less in reality.

Graphics processors also benefit from improved memory clock-rates, owing to the Graphics Double Data Rate (GDDR) technology that achieves clock-speeds as high as 2.0GHz at the time of writing. This type of memory is almost exclusively set to either read-only or write-only mode at any given time, thereby relieving the processor from having to manage memory contention issues that plague conventional CPU architectures. There are certain caveats, however. Consider the statistics in Fig. 1.3 for the memory bandwidth comparison between an NVidia GeForce 7800 GTX chipset and a Pentium 4 processor. For a caching strategy involving sufficient temporal and spatial coherence, both processors perform close to their theoretical peaks. The performance-gap tends to widen for the sequential memory access pattern where temporal coherence is no longer available. The trend continues for the random access pattern which eliminates any form of coherence, and is prevalent in the sparse-matrix

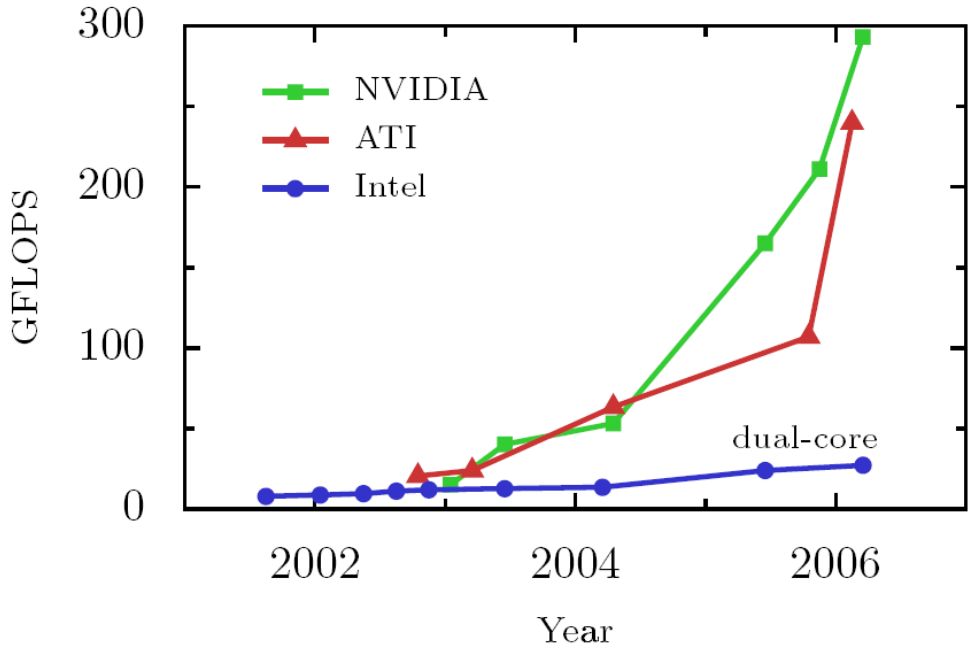


Figure 1.2. Peak floating-point performance of GPUs over the last four years

operations described later. For memory-bound algorithms, the primary bottleneck on both processors is the memory throughput of the hardware, rather than the floating-point processing capacity. However, the throughput of the GPU for the non-local data always encountered in scientific processing is atleast 3 times higher.

1.1.3 Multiple Cores

Currently, processors are fabricated using a 65-nanometer process that may soon transition to 45 nanometers in the second half of 2007. However, the fundamental atomic limit still holds, and the problems involved with going to nanoscales hardly justify the escalating costs of fabrication. In short, Moore's Law for current microprocessors is gradually approaching its limit. A different approach to this architectural conundrum is to step towards multiple cores on the same chip, with either a localised or shared-cache strategy. This approach is also well-suited to stream-processing, and so, graphics processors have used it to good effect. Having realized the potential

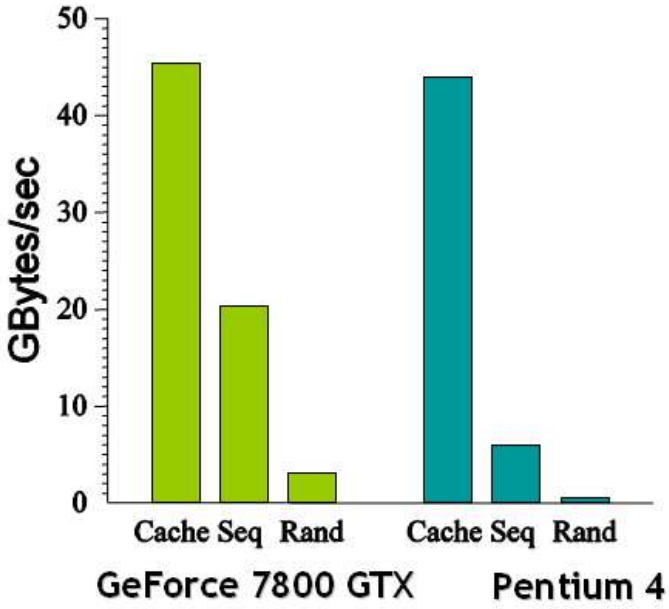


Figure 1.3. Memory bandwidth comparison

benefits, companies like Intel and AMD are also taking a step in this direction, but with a more generic application-base.

The Cell processor is the result of a collaboration by Sony, IBM and Toshiba to bring improved vector-processing capabilities to game-consoles and other multimedia-intensive applications like high-definition televisions. The architecture comprises of 8 vector-units and one general-purpose unit. However, the Cell focusses on running fewer threads as opposed to the large number of lightweight streams that are generated on a graphics processor. Although this initiative appears to be directed primarily towards niche-markets like graphics-intensive games on the PlayStation 3, a more generic application-base is envisioned.

Given the amount of computational horsepower and the data-parallel nature of the algorithms that they were designed for, architectures like those on the GPU and the Cell would be useful as cost-efficient math coprocessors.

1.2 Drawbacks

Despite the fact that graphics processors are well-suited to scientific computing, they present certain challenges. GPUs have been tuned to the specific requirements of graphics applications, and were restricted in terms of programmability until 2001 [20]. Over the past few years, the increasing flexibility of GPUs have allowed more general-purpose algorithms to be ported onto them. However, this task requires algorithms to be reformulated for use on hardware that was originally intended for graphics applications, and programmers are expected to be aware of the capabilities and limitations of the processor in order to achieve efficient implementations. This unusual programming model therefore hinders its adoption into the scientific community. Graphics processors also lack certain capabilities that are taken for granted on conventional hardware, and the workarounds for these tasks sometimes hamper the overall performance of the algorithm. For instance, consider a memory-gather operation, which is an indirect read from memory, of the form: `x=a[i]`, where `i` denotes the array-index. A scatter operation, on the other hand, is an indirect write to memory, of the form: `x[i]=a`. Gather operations are natively supported on the graphics processor, while scatter operations are not.

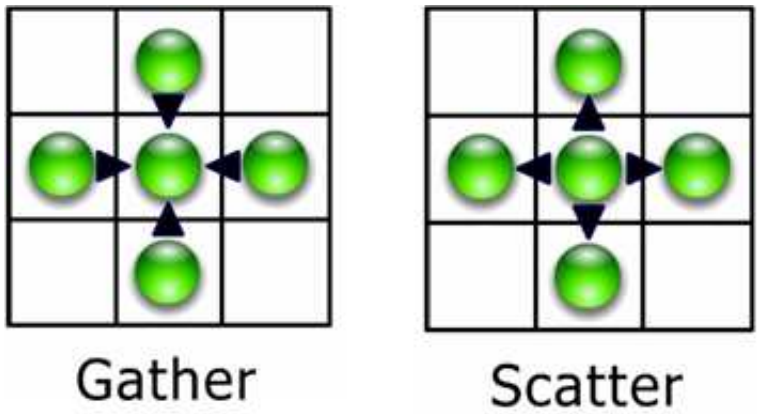


Figure 1.4. Gather operations are natively supported on the GPU, while scatter operations are not

Graphics hardware is also restricted in terms of video memory that can be dedicated for data storage on-board, and can sometimes be a limitation for large simulations (The nVidia 6600GT used in this study has 128MB of graphics memory). Additionally, the numerical precision offered by the manufacturers is restricted to 32-bit floating-point, which does not adhere to the IEEE-754 standard for reasons of efficiency. Double-precision capabilities are on the horizon, but since accuracy is not a critical issue for graphics applications, this aspect remains low on the priority list; although there have been a few efforts to emulate it [8]. Nevertheless, the benefits offered by the hardware are significant enough to justify their use for computation.

1.3 Prior work on Streaming Processors

The use of graphics processors for general-purpose computation has increased steadily since 2001, when programmability was first introduced. Harris et al [13] have implemented a coupled map lattice approach on graphics processors to simulate several physical phenomena such as chemical reaction-diffusion, cloud formation, convection and boiling. This work, however, involves structured domains that require the number of grid points to be powers of two in each dimension. A frequently cited work for fluids simulation on graphics hardware is that of Stam [30]; using an FFT approach to solve the Poisson equation with periodic boundaries. The solution time per time-step is low enough to allow real-time interaction with the fluid using an input device, and is unconditionally stable owing to fact that the time-stepping is fully implicit. Stam's approach is however, restricted to structured grids without internal boundaries and only first-order accurate in time. Scheidegger et al [28] have used GPUs to simulate fluid-flow in general rectangular domains using a simplified Marker and Cell approach; a scheme that was originally proposed by Harlow and Welch [11]. This method uses a staggered-grid along with the Jacobi method to solve the Poisson equation for pressure. Jacobi iterations are frequently associated with poor conver-

gence rates, and the authors agree that a better solution technique is warranted. Similarly, Bolz et al [1] and Goodnight et al [9] have implemented a Multigrid solver on the graphics processor to solve the two-dimensional stream-vorticity formulation of the Navier-Stokes equations on domains that do not involve walls. Lattice-Boltzmann Methods have also been implemented on graphics processors to simulate fluid-flow, like the work carried out by Li et al [34].

Kruger and Westermann [18] have also provided a set of linear algebra operators for the implementation of numerical algorithms on graphics hardware, to allow more complicated algorithms to be developed on a basic framework. As a primer on the applicability of graphics processors for general-purpose use, Mark Harris [12] has provided the use of the PUG library, which forms the basic framework of his Coupled Map Lattice approach and also deals with aspects like reductions. On similar lines, Galoppo et al [6] have implemented efficient dense-linear solvers on the GPU. These include a multi-pass Gauss-Jordan implementation for matrix-inversion, and an LU-decomposition algorithm with partial and full-pivoting. Dense-matrix inversion is a process that is particulary attractive for any hardware, as it involves very good spatial coherence, and is therefore highly cache-efficient. Larsen and McAllister [19] have done similar work involving matrix-matrix multiplication. However, such systems rarely occur in complex simulations; and is a good example of a technology demonstration with little practical significance. All the work stated above involve two-dimensional memory layouts for the arrays - restricted to power-of-two dimensions.

Some work on particle-systems have also been carried out, including that by Kipfer at al [15] and Kolb et al [16]. These implementations typically update the position and velocity of a large number of particles using Newton's laws and explicit time-stepping. Since fast animation frame-rates are a requirement in this application, they also implement sorting algorithms to determine the visibility of individual par-

ticles when they are drawn to the screen. Krakiwski et al [17] have implemented the finite-difference time-domain algorithm for solutions to Maxwell's time-dependent curl-equations. They analyze a simple two-dimensional domain with staggered computational nodes and explicit time-stepping. Being a cartesian mesh, the computational stencils are simplified, and they claim to achieve speedups to the order of about 7x over the CPU.

On a different note, Buck et al [2] recently developed a data-parallel programming language called Brook to allow scientific researchers to implement their algorithms on the GPU using high-level code, thereby providing a sufficient layer of abstraction from the low-level graphics constructs. Brook extends C to include data-parallel constructs, and involves an extra compilation stage which may not produce the most optimal implementation of the algorithm at hand. A similar attempt has been made by McCool et al, called Sh (which in turn led to a spin-off called RapidMind). This implementation also involves an extra compilation step, and because it was originally intended for graphics applications, is incapable of performing more generic operations like reductions and data-gather [2, 21]. Tarditi et al [32] have provided the Accelerator library as a stream-programming abstraction for the GPU. Their implementation converts high-level stream-constructs into kernels that run on the hardware through library calls, and also involves extra compilation. The PeakStream API, a commercial venture based on Brook, allows porting of algorithms to run on GPUs and the Cell Processor.

CHAPTER 2

MAPPING ALGORITHMS TO GRAPHICS PROCESSORS

On conventional CPUs, data is represented in the form of one-dimensional arrays, with two and three- dimensional arrays also represented as stacked one-dimensional layouts. Arrays on graphics processors, however, are represented natively in two-dimensions (also called Textures), as shown in Fig. 2.1. One- and three-dimensional arrays are also supported, but two-dimensional layouts are preferred mainly for reasons of hardware efficiency. As a result, two indices are required to access an element from a GPU array, which are either precomputed on the CPU or calculated on-the-fly for algorithms that require indirect memory addressing. Additionally, these indices, also called texture-coordinates, are required to be of the floating-point datatype, since the GPU used in this study does not support integers.

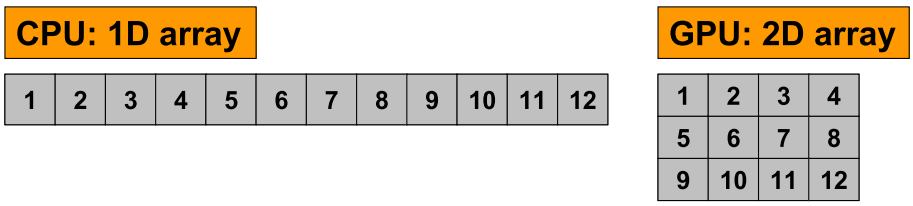


Figure 2.1. Memory layouts on the GPU vs. the CPU

Once the input and output arrays are allocated in video-memory, a kernel-program for the computation is defined. This program contains the set of instructions that must be executed for every element in the array. Following this step, a rectangular viewport is defined on the screen, with dimensions equal to that of the output array. The actual computation is triggered by drawing a rectangle to the screen (of the

same size as the viewport), defined by its four corner vertices in addition to four coordinate-locations representing the corners of the output array. When these vertices are defined, they are first passed into the vertex processors (which consist of about 6 elements working simultaneously) for manipulations that may be required on them. In typical graphics applications, additional information like normal-vectors, texture-coordinates or colors are also specified at each vertex, and this data sometimes requires changes before they are passed on to later stages, therefore the need for the vertex-processing stage. For general-purpose computation, the vertex-processing stage is sometimes useful for off-loading a small amount of the workload for efficiency, but mostly, the vertices are passed on to the next stage without any manipulations.

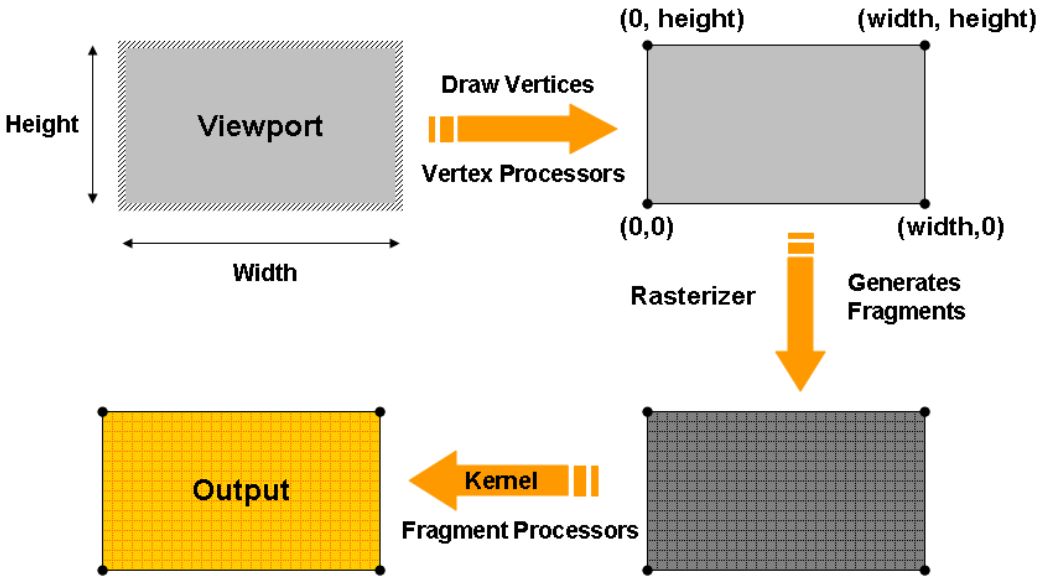


Figure 2.2. Simplified represenation of stages in the Graphics Pipeline

The next stage, known in graphics terms as the *rasterization* stage, generates elements (called pixels or fragments), representing each memory location in the output array. This stage is also required by design to interpolate per-vertex information to each fragment that is generated, like color for instance. Since coordinate-locations

were specified at each of the four vertices, this results in a linear-interpolation of coordinates across the array that now serve as two-dimensional indices for each fragment that is generated by the rasterizer. The rasterizer generates two-dimensional indices for an equal amount of elements. These generated elements are now passed into the fragment processors (consisting of about 12 elements working simultaneously), where the actual computations occur. At this stage, the instructions in the kernel program are applied to each element generated by the rasterizer. These instructions can include basic arithmetic operations as well as memory-fetches from other GPU arrays. At the end of this stage, each element consists of a value representing the result of the computation, and is finally written to the output array. This is a very simplified representation of the Graphics Pipeline, when seen from the perspective of general-purpose computation. A schematic of the various stages is shown in Fig. 2.2.

2.1 An Example

The following section elaborates on the various steps that constitute the mapping process by taking an example.

```
for (int i=0; i<N; i++)
    Z[i] = Y[i] + a * X[i];
```

The code segment shown above is commonly known as the *axpy* operation in linear algebra, used to denote *'**a** times **x** plus **y**'*. This is a fairly straightforward operation in conventional programming - the same operation is performed N times, where N is the size of the arrays X and Y; and **a** is a multiplication factor. Any given element doesn't depend on information from other locations on the same array and so, it is also a trivially parallel problem because it requires no particular effort to divide the workload into a number of efficient parallel tasks. The example will assume that C++ is used as the underlying CPU base-code, with specialized routines for handling data that is offloaded to the GPU for computation.

2.1.1 Setting up the graphics API

Firstly, it is essential to set up a graphics API for interaction with the graphics processor. In this regard, there are two options - DirectX and OpenGL. DirectX is a collection of libraries provided by Microsoft that allows programmers to access certain hardware elements like graphics cards directly rather than being routed through the Windows interface, thereby attaining better performance for applications like games and multimedia. OpenGL is an API with similar functionality, but with the exception that it is open-source and platform-independent. Since platform-independence is an important issue, OpenGL will be used here. In addition to this, two other utilities are also required - the GL Utility Toolkit (GLUT), and the GL Extension Wrangler (GLEW). The toolkit provides the necessary routines to implement simple windows and menus; and also to handle mouse and keyboard events. GLUT is required to initialize a graphics-related context, to 'trick' the application into believing that it is intended for graphics, whereas it is actually being used for numerical computation. GLEW provides a handy way of accessing certain features of the graphics-hardware through functions, rather than having to deal with the low-level assembly routines. At this point, it would suffice to say that these two utilities have to be initialized in order to use the GPU for computation; and the following code segment shows how it is done.

```
#include <GL/glew.h>
#include <GL/glut.h>
// Sets up GLUT, creates a "window", taking arguments from main()
glutInit ( &argc, argv );
windowHandle = glutCreateWindow("FakeWindow");

// Sets up GLEW
int err = glewInit();
```

2.1.2 Creating arrays on the GPU

GPU arrays, or textures, can have many data formats in graphics memory with each element in a GPU array holding up to four 32-bit floating-point values. This is

owing to the fact that textures in a graphics application are typically used to store the color and opacity information of a fragment. Colors in the spectrum can be represented by combinations of the three basic color components - Red, Green and Blue; and an additional component Alpha to determine how opaque the fragment should be. These four components are typically stored together for a total of 128 bits per pixel element, since graphics operations sometimes operate on all four components at one time. Alternatively, the GPU can also store a single data item at each element, which saves memory when only one item is required. Each array that is allocated on the GPU is identified by a unique integer reference, similar to a pointer in C. Memory management on the GPU, and tasks like data-transfer between the GPU and main memory are enumerated in detail in Appendix A, along with code.

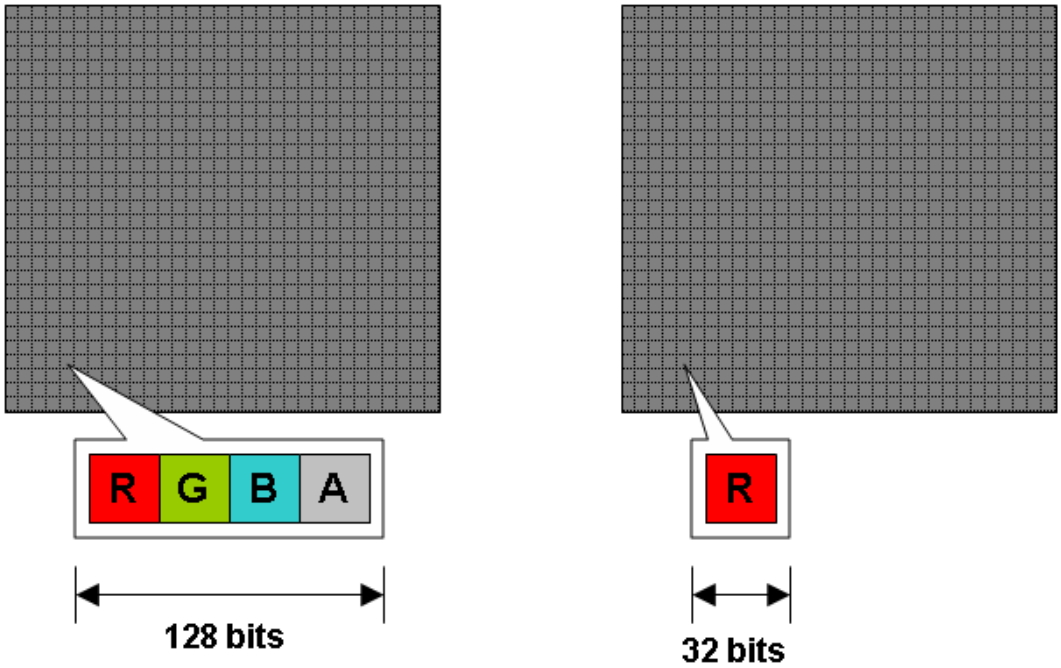


Figure 2.3. Comparison of internal formats for GPU Arrays

Since GPU arrays are two-dimensional in nature, the choices for memory layouts are restricted to two options as dictated by the hardware - square arrays with dimensions that must be powers-of-two; or rectangular arrays which can be of ar-

bitrary dimensions for width and height. Considering Fig. 2.1 for the moment, a one-dimensional array of size 12 is mapped to a two-dimensional array of width 4 and height 3. However, an array of size 13 doesn't have a perfect two-dimensional equivalent and therefore, the next optimal dimensions must be selected; like a width of 7 and height of 2, for instance. The extra elements would then have to be padded with zeros to prevent garbage values from interfering with calculations. Choosing an optimal mapping configuration with minimal padding in discussed in a subsequent chapter.

2.1.3 Defining the kernel program

Since programmability of the graphics pipeline is a fairly new concept, OpenGL 2.0 was introduced with an additional feature known as GLSL, or the GL Shading Language. This follows from the use of the term *Shader* to denote programs that are written for execution on either the vertex or fragment processors. The language closely resembles C in a lot of ways, with a few constructs that are specific to graphics. This saves the user from the drudgery of having to program the processors at the assembly level, and represents a fairly decent level of abstraction. For more information on GLSL, refer the Orange Book [27] or the Lighthouse3D tutorials [5]. The kernel for the *axpy* operation in this case, is defined as a shader which runs on the fragment processors:

```
// Shader for the axpy operation using rectangular textures

uniform samplerRect texY;
uniform samplerRect texX;
uniform float a;

void main(void) {
    float y = textureRect(texY,gl_TexCoord[0].xy);
    float x = textureRect(texX,gl_TexCoord[0].xy);
    gl_FragColor = y + a*x;
}
```

`texX` and `texY` are GPU arrays containing the values for the X and Y arrays, respectively. The `uniform` type-specifier is used to indicate that the data is persistent (or "uniform") for all fragments generated by the rasterizer. Uniform variables are typically variables that are specified by the CPU, like arrays and constants. The other alternative is the `varying` type-specifier to indicate variables that are passed into the fragment shader from the vertex-processors, rather than the CPU. The `gl_TexCoord` variable is a built-in register value that always contains the coordinates of the current fragment, which is essentially a two-dimensional version of an array index. This variable is a `float4` data-type that contains four floating-point values, very similar to a `struct` in C. The `.xy` is an access technique to obtain the first two values (and therefore the coordinates) stored in the `float4` variable. Individual elements can be accessed by combinations of (x,y,z,w), (r,g,b,a) or (s,t,p,q). Thus, the textureRect function uses the current coordinate to index a location on the array, and stores that value in a variable. The final statement is a requirement for every fragment program, since it determines the final output of the fragment.

2.1.4 Compiling the kernel program

Kernel programs are typically stored in external text files, read into a character-string, compiled at runtime, and then attached to the appropriate vertex/fragment processors when needed. The compilation process is encapsulated in the following routine:

```
int CompileKernel(const char* fragment_source, const char* vertex_source)
{
   // Create the program
   int program = glCreateProgramObjectARB();

   // Create shader object (vertex shader) and attach to program
   int vertex_shader = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
   glAttachObjectARB (program, vertex_shader);
   // Attach the vertex-program to the shader
   // vertex_source is the string containing the source
```

```
    glShaderSourceARB(vertex_shader, 1, &vertex_source, NULL);
    // Compile the vertex program
    glCompileShaderARB(vertex_shader);

    // Create shader object (fragment shader) and attach to program
    int fragment_shader = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
    glAttachObjectARB (program, fragment_shader);
    // Attach the kernel source-program to the shader
    // fragment_source is the string containing the source
    glShaderSourceARB(fragment_shader, 1, &fragment_source, NULL);
    // Compile the fragment program
    glCompileShaderARB(fragment_shader);

    // Link the program
    glLinkProgramARB(program);

    // Return the reference to the kernel-program
    return program;
}
```

The axpy operation does not involve the vertex processors in any way and so, in the absence of a vertex shader, the vertex processors behave as a fixed-function unit - where no manipulations are performed on the vertices passed into the vertex processors. This is reminiscent of the functionality of graphics processors before they were programmable. To activate the kernel program on the GPU, the `glUseProgramObjectARB` is used. This command takes the integer reference of the kernel program as its argument.

2.1.5 Specifying inputs to the GPU

Once the kernel has been compiled and linked, inputs for the uniform variables have to be specified from the CPU. The location of the uniform variables in the kernel program will first have to be determined using the `glGetUniformLocationARB` function. This function compares the character-string with uniform variables specified in the kernel, and once it is found, returns an integer reference which can then be used by the `glUniformARB` function to specify the GPU array to be associated with that variable. For instance,

```
int getInput(int program, char *var_in_prog)
{
   // Get the variable location in the program
   return glGetUniformLocationARB(program, var_in_prog);
}

void setInput(int location_handle, int FieldID)
{
   // Enable the texture unit and bind the array to it
   glActiveTexture(GL_TEXTURE0);
   glBindTexture(GL_TEXTURE_RECTANGLE_ARB,FieldID);

   // Specify the texture unit associated with the location
   glUniform1iARB(location_handle,0);
}
```

In the example shown above, after obtaining the location for `texX` in the kernel, the array associated with X in the axpy operation is bound to a 'texture unit' - a hardware parameter that defines the maximum number of simultaneous array inputs to a program. A GeForce 6600GT, for instance, is limited to 16 texture units. In this case, the array is bound to `GL_TEXTURE0`, assuming that `FieldID` is the reference to the array associated with X. `glUniform1iARB(location_handle,0)` specifies that the array attached to `GL_TEXTURE0` is to be associated with the variable referenced by `location_handle`. To specify a single float variable, like the value of **a** in the axpy operation, the statements would be:

```
// Get the variable location in the program
int loc = getInput(program, "a");

// Specify the float-variable associated with the location
glUniform1fARB(loc,a_cpu);
```

2.1.6 Specifying outputs for the GPU

The final stage of the graphics pipeline after the fragment-processing stage is represented by the framebuffer, a portion of memory that stores the pixel information temporarily before displaying it on the screen. This is typically double-buffered to reduce flickering, so that one buffer could be displayed on-screen while the other

buffer is being written to. In a conventional graphics application, the contents of the framebuffer are usually displayed on the screen for a fraction of a second and then discarded for the next frame. In the computational paradigm however, the results of a calculation are usually used as inputs for a subsequent computation-step, and therefore it is vital to store this information to an array, rather than discard it to the screen. OpenGL implements an extension known as the Framebuffer Object, which allows results to be redirected to an array rather than the actual framebuffer. The following routine can be called to initialize the framebuffer object.

```
int fb;
// Generate a reference to the new Framebuffer object
// [integer 1 signifies a request for only one object]
glGenFramebuffersEXT(1, &fb);

// Bind the framebuffer to this object
// [i.e., skip the window-specific target]
// (*,0) = real display
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
```

Now that the object has been defined, a GPU array has to be attached to the framebuffer so that results of the computation are redirected to the array rather than the actual display. This is a simple process:

```
void setOutput(const int FieldID)
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT0_EXT,
    GL_TEXTURE_RECTANGLE_ARB,
    FieldID,
    0);
}
```

The first argument to the `glFramebufferTexture2DEXT` function is mandatory. Each framebuffer object can currently have up to four arrays attached to it, thereby allowing up to four outputs for each kernel. Each output could be a 4-vector float, so the actual limit is 16 output arrays. This is a feature known as Multiple Render Targets (MRT), and more information on this area can be found in the GPGPU

tutorial [7]. Since only one output is currently considered here, the first attachment defined by the `GL_COLOR_ATTACHMENT0_EXT` enumerant is used. For information on the third argument, refer to Appendix A. The fourth argument is the integer reference to the GPU array (representing the array `Z`), while the last argument is a mipmap level that is irrelevant here.

Attaching another array to the framebuffer removes the current one from being a target for redirected output. Generating a framebuffer object is an expensive process and should be typically done only once in the program for efficiency. Attaching arrays to an existing framebuffer object is relatively inexpensive.

2.1.7 Setting the viewport

Setting a viewport with at least the same dimensions as the output array is required to ensure that fragments generated by the rasterizer are not clipped off. This is done by the following OpenGL routine:

```
void setGPUview(const int width, const int height)
{
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0, width, 0.0, height);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glViewport(0.0, 0.0, width, height);
}
```

Looking at this routine one statement at a time:

- GL_PROJECTION and GL_MODELVIEW are standard transformation matrices provided by OpenGL to project vertices in three-dimensional space to two-dimensional coordinates on the screen. Both these matrices are invoked using the `glMatrixMode` function.

- The `glLoadIdentity` function is required to reset the current matrix, as it might have been used in a previous operation.

- `gluOrtho2D` performs an orthographic projection of the viewing region, using the standard GL_PROJECTION matrix.

- The dimensions of the viewport are defined using the `glViewport` function, which ensures that fragments are not generated outside the region defined in the function arguments. Notice the use of `glLoadIdentity` before the call to this function.

When provided with the dimensions of the Z array, this routine now ensures an orthographic projection of the viewport on the screen, with the specified dimensions.

2.1.8 Running the kernel program

The actual computation process is invoked by drawing a rectangle with dimensions equal to that of the output array. This is done by the following lines of OpenGL code:

```
void RunProg(float v_width, float v_height, float f_width, float f_height)
{
    glBegin(GL_QUADS);

       glTexCoord2f(0.0, 0.0);
       glVertex2f(0.0, 0.0);

       glTexCoord2f(f_width, 0.0);
       glVertex2f(v_width, 0.0);

       glTexCoord2f(f_width, f_height);
       glVertex2f(v_width, v_height);

       glTexCoord2f(0.0, f_height);
       glVertex2f(0.0, v_height);

    glEnd();
}
```

If `width` and `height` to be the dimensions of the `X` and `Y` arrays, the axpy operation is actually performed by issuing the following sequence of statements:

```
// Code assumes that three GPU arrays (X, Y, and Z) are created
float a;
int prog = CompileKernel(fprog, "");
glUseProgramObjectARB(prog);
int xloc = getInput(prog, "texX");
int yloc = getInput(prog, "texY");
int aloc = getInput(prog, "a");
setInput(xloc, X);
setInput(yloc, Y);
glUniform1f(aloc, &a);
setOutput(Z);
RunProg(width, height, width, height)
```

The `glVertex2f` function specifies the four vertex-coordinates of the rectangular region being drawn; while the glTexCoord2f function specifies the four end-coordinates that are to be linearly interpolated across the rectangle. By passing these four vertices, the rasterizer generates a stream of fragments that equal the number of elements in the output array. These elements are then processed by the fragment processors, which read a value of **a**, **X** and **Y** for each index; finally resulting in an output for each element of Z.

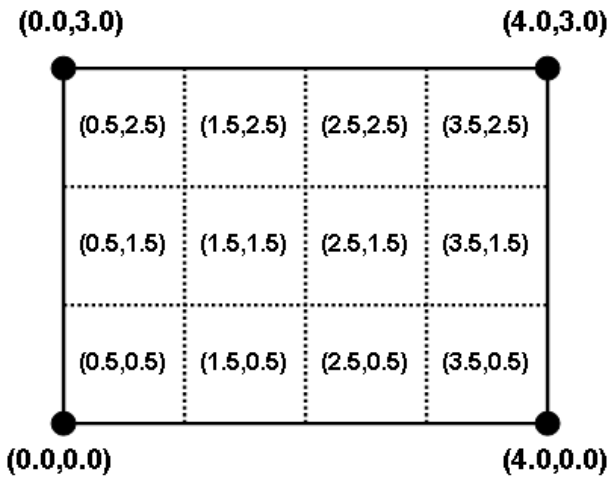


Figure 2.4. Linear interpolation of two-dimensional rectangular array indices

The interpolated coordinates for an array of 12 elements is shown in Fig. 2.4. The indices always reference the center of the fragment, and vary linearly in the orthogonal

directions. These indices are also obtained by the `gl_TexCoord[0].xy` instruction in the kernel program.

2.2 Results

The axpy operation was timed for performance results over a range of problem-sizes. Since the resolution of the system timer is poor for small times, the operation is performed repeatedly in a loop with an outer timer and then finally averaged to obtain the cost for one operation. It is imperative to make an OpenGL `glFinish` call before invoking the system timer, since the GPU works asynchronously with the CPU and could therefore be in the process of completing a task when the timer call is made, thereby leading to erroneous results. `glFinish` is a blocking call that returns control to the CPU only when all outstanding commands in the graphics pipeline have been completed. The floating-point computational capacity in terms of gigaflops is given by the following relation:

$$GFlops = \frac{OpsPerElement \times NumIterations \times NumElements}{TotalTime \times 10^9} \tag{2.1}$$

For an axpy operation involving an add and a multiply operation per element, the value of OpsPerElement is 2. The results shown in Fig. 2.5 depict the performance of a GeForce 6600GT graphics card against a 1.8GHz AMD Athlon processor with an FSB clock-rate of 400MHz.

The graphics hardware (represented by triangles) performs at close to 2GFlops for large problem sizes as opposed to 0.33GFlops on the CPU (represented by squares) - an increase of about 8x which is clearly quite significant. For very small problem sizes where the whole problem fits in cache, the performance only compares favorably with the CPU, which can be attributed to the fixed cost involved with tasks like viewport initialization, etc. The caching strategy of the CPU is also clearly visible in the plot

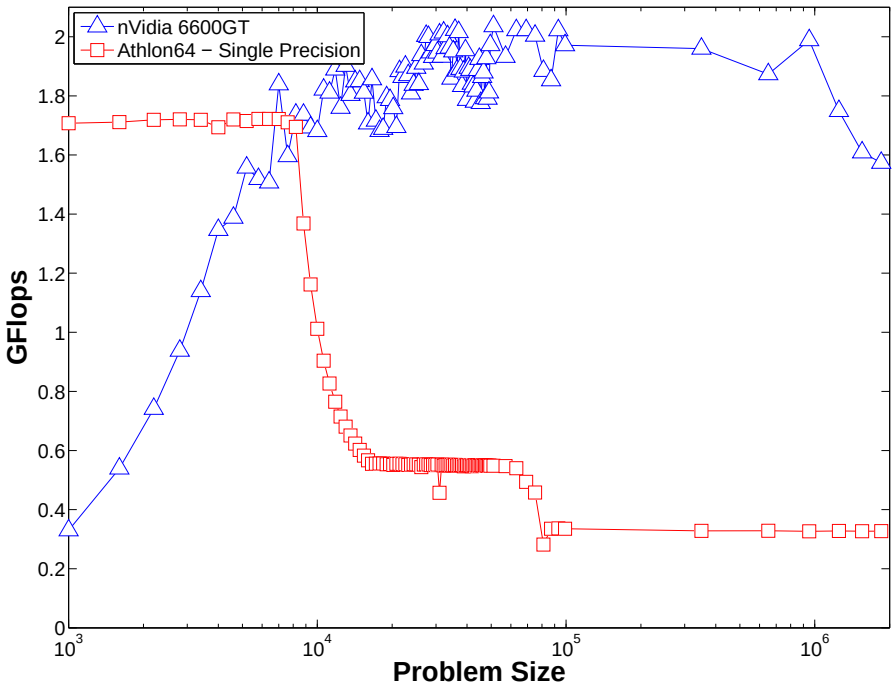


Figure 2.5. Performance of the axpy operation vs. problem size

- which shows three distinct plateaus for variations in problem sizes, depicting the Level-1, Level-2 caches and accesses to main-memory respectively.

The axpy is a good example of a memory-bound operation, since it involves only two floating-point operations per element. The processing work per element is insufficient to hide the memory-fetch latency costs and so, the processor is more likely to be idling for long periods, waiting for data to be fetched. These results, like most scientific computations, depend predominantly on the memory bandwidth of the hardware rather than the floating-point performance. The 100's of GFlops peak performance of the GPU (shown in Fig. 1.2) is *not* obtained. The performance is better captured by the memory access speeds shown in Fig. 1.3.

2.3 Comments

Another option to be considered is the use of various internal formats for GPU arrays to determine possible benefits. The options in this case would be the use of either the 128 bits-per-element(RGBA) or 32 bits-per-element(R only) layout as shown in Fig. 2.3. Using an RGBA internal format would involve an intermediate routine which allocates a GPU array with a quarter of the number of elements, which is essentially packing scalars into vector components. This approach doesn't seem to provide any significant advantages, and since it involves an intermediate packing step, the approach was discarded.

It is worthwhile to note that while the axpy operation in the BLAS library is defined as $\mathbf{y} = \mathbf{y} + a * \mathbf{x}$, it is actually implemented as $\mathbf{z} = \mathbf{y} + a * \mathbf{x}$ on the GPU. This was done to overcome a limitation of the hardware, which demands that memory be designated as either read-only or write-only. It may be argued that by attaching an array to the framebuffer and designating it as an input array, it should be possible to read and write to the array simultaneously. This approach was experimented as well, but was quickly discarded as it leads to incorrect results. To achieve the effect of reading and writing to the same array, the axpy is split into two steps:

```
y2 = y1 + a.x
y1 = y2 + a.x
```

This approach is an example of a common technique known as ping-ponging. However, this approach wastes memory and time. An alternative is to use loop unrolling. Loop-unrolling is described later in the context of the Conjugate Gradient algorithm.

CHAPTER 3

ARRAY LAYOUTS

Arrays on the GPU work most efficiently in two-dimensional layouts, although one- and three-dimensional arrays are supported. This efficiency can be attributed to the fact that two-dimensional arrays are closely coupled with the classical graphics applications. One-dimensional arrays are often limited by the maximum size that can be allocated, which makes them inadequate for large problems; and writing to slices of three-dimensional arrays is considered to be highly inefficient [23]. Additionally, arrays can be represented either in a rectangular layout with arbitrary dimensions, or in a square layout with dimensions that are strictly power-of-two. Given the available options for memory layouts, deciding on an appropriate representation for a one-dimensional array to a two-dimensional GPU array becomes a mapping problem.

Square layouts have certain drawbacks which limit their use:

- Since square textures are limited to dimensions that are powers of two, the primary issue is that of memory wastage. For instance, a one-dimensional array consisting of 16 elements can be readily mapped to a two-dimensional square array with 4x4 elements. However, an array of 17 elements would require a square array with 8x8 elements (the next power-of-two) - a wastage of 276%; and this becomes worse with larger problem sizes.

- When extra elements occur, they have to be initialized (or 'padded') with zeros to prevent garbage values from interfering with the computation. Since the graphics processor works on the entire array, resources are unnecessarily

wasted on the padded elements; leading to a reduction in efficiency. Considering the amount of padding in square arrays, this loss in efficiency can be quite significant.

- Indices generated by the rasterizer are always normalized between 0 and 1 for square arrays, regardless of the actual array dimensions. This can be counter-intuitive, especially for someone without a graphics background.

Rectangular array support was introduced in OpenGL 2.0 to alleviate these restrictions. They can be of arbitrary dimensions; and indices(coordinates) are always between zero and the actual height/width of the array. As a consequence, they also minimize the amount of padding that would be required, and are therefore more efficient. Performance of rectangular arrays on nVidia graphics hardware have also been shown to be superior, thereby justifying its use for this implementation. However, being a recent addition, care must be taken to ensure that the hardware can actually support this format.

3.1 Algorithm

The objectives for the array-layout algorithm are:

- Determine an optimal two-dimensional layout for a single-dimensional array of size N, with a minimal amount of padding.

- In addition, the array should be partially reducible. This situation arises in the case of reduction operations which act on the entire array to produce a single result, like its sum or maximum value.

- Layouts that are close to square would be preferred.

A typical reduction on the GPU would involve successive divisions of the array to smaller dimensions, finally yielding a single result. A simplified example would be

the reduction of a 8x8 array to a 1x1 array to find its maximum value; as shown in Fig. 3.1. In this case, the local maximum of a 2x2 block is found at each stage and passed on until a single element remains.

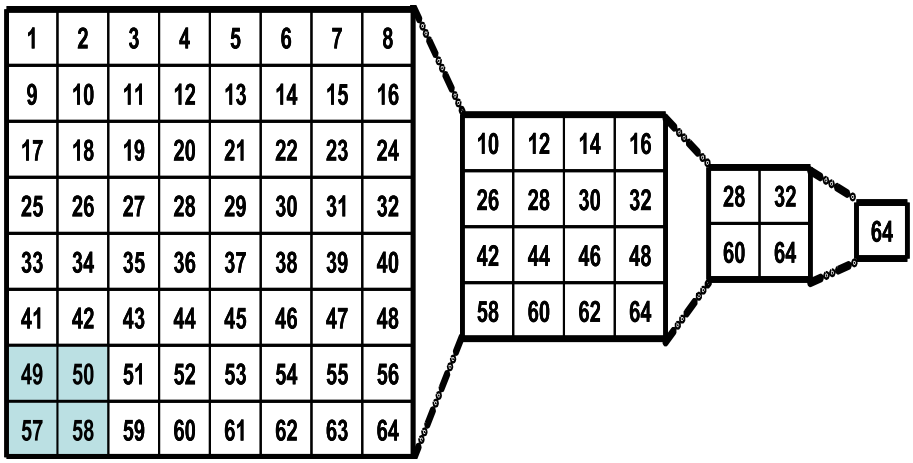


Figure 3.1. Simplified reduction example: Finding the array-maximum

For a rectangular array, achieving a single value after several stages of 2x2 reductions is not possible. For rectangular arrays, the goal is to determine the minimum number of 2x2 reductions that would have to be taken until a sufficiently small rectangle is achieved. This smaller rectangle can then be processed using several techniques without incurring significant costs. Reduction operations and their implementation are discussed at length in a subsequent chapter.

The first step is to define the dimensions of the smallest rectangle, R_{min}. This can be set to an arbitrary value, depending on the size of the arrays that might be expected. For example, choosing dimensions of 16x16 means that any array of a size less than 256 elements has to be allocated 256 elements.

Now that the minimum dimensions are defined, the next step is to determine the number of 2x2 reductions that need to be taken until the array is reduced to a rectangle of dimensions less than that of R_{min}. This can be achieved by successive divisions of the problem size by two until the minimum size (say, S_{min}) is obtained. The entire

one-dimensional array can now be split into 'blocks', each of size 2^l elements, where l is the number of divides.

Determining the layout for the blocks is done by starting with $\sqrt{S_{min}}$ as an estimate for the y-dimension (an attempt to make the array close to square), checking whether S_{min} is divisible by y values varying between 2 and the maximum dimension of R_{min}; performed in a loop. Once a number is found, it can be designated the y-dimension of the block, and dividing S_{min} by y gives the x-dimension. If a number could not be found, or one of the dimensions exceed that of the R_{min}, the value of S_{min} is incremented and the check is repeated.

Individual steps of the algorithm are provided in Appendix A.4.

3.2 Results

Since the amount of padding is an important factor, the algorithm was tested for various array sizes up to 3 million, and the percentage of extra elements was plotted against array size. Three cases of R_{min} were taken into consideration - 32x32, 64x64, and 128x128. The maximum percentage of padding was found to be 2.85%, 1.02% and 0.244%; with average percentages of 0.4762%, 0.136% and 0.039% respectively. While the larger R_{min} leads to less padding, it also increases the work done in the 'final reduction'. The larger R_{min} also limits the smallest size problem the GPU will be effective on since arrays smaller than R_{min} are typically heavily padded. For this reason, the reduction operations in this work use $R_{min} = 32^2$

The plots for the various cases are shown in Fig. 3.2 and Fig. 3.3

This would indicate that the algorithm is a satisfactory fit for the mapping problem, in addition to satisfying the dimensional constraints brought in by the reduction operations.

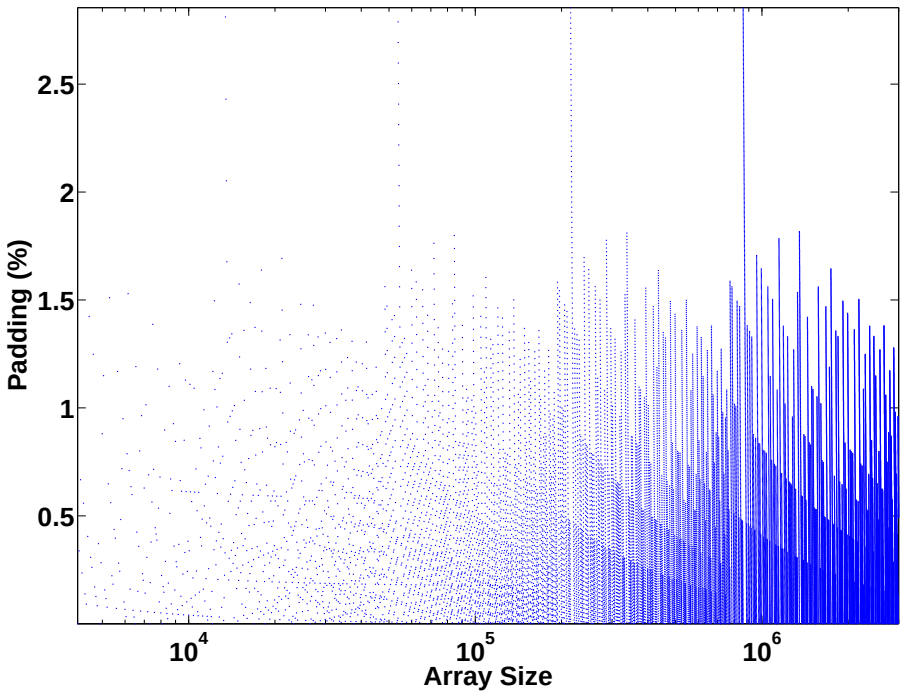
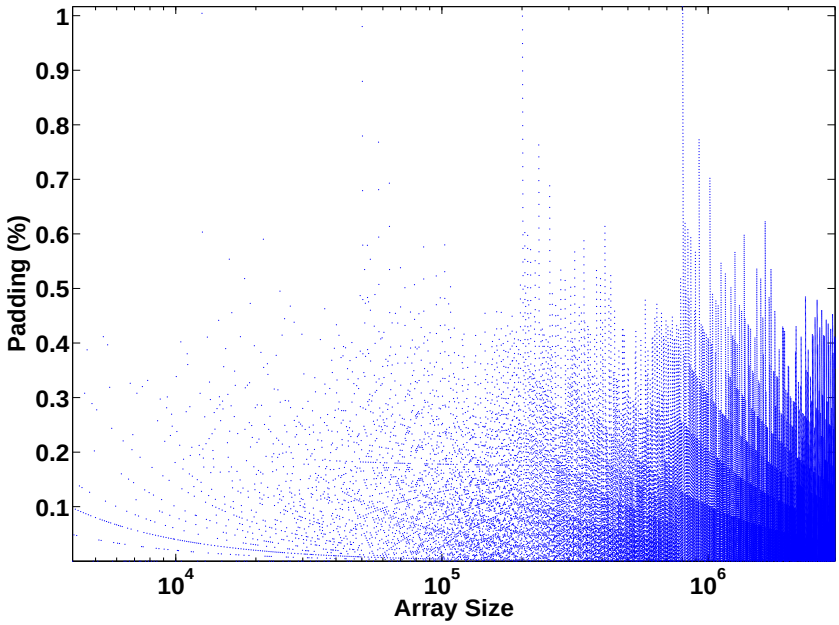
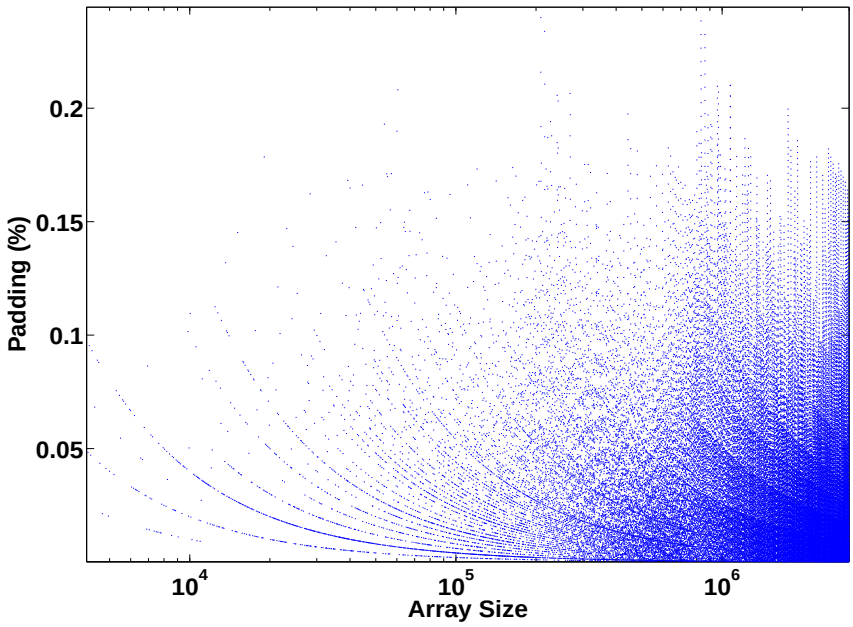


Figure 3.2. Percentage padding vs. requested array size (32x32)



(a) 64x64



(b) 128x128

Figure 3.3. Percentage padding vs. requested array size (64x64) and (128x128)

CHAPTER 4

REDUCTIONS

Reduction is the term used to describe an operation that takes an array of data and returns a single value as the output. Typical operations that fall under this category include the sum, minimum and maximum of an array, or the inner-product of two arrays. The operation is fairly trivial on a conventional CPU:

```
float sum = 0.0f, max = X[0], dot = 0.0f;
for (int i=0; i<N; i++){
    sum = sum + X[i];
    max = X[i] > max ? X[i] : max;
    dot = dot + X[i]*Y[i];
}
```

From the example shown above, it is clear that when a reduction operation is performed on the CPU, a variable is required to maintain the value of the reduction, as the program loops over the array. The value of this variable is constantly updated for each element being accessed in the array. This represents a scatter operation. Since this option doesn't exist on the data-parallel paradigm of the GPU, a workaround is required.

The approach taken on the GPU is to perform a stage-by-stage reduction of the array. For example, the first stage draws an array that is half the size of the original one in each dimension. By drawing a quadrilateral that is quarter the size, the rasterizer is forced to generate only a quarter of the number of fragments. In doing so, each fragment that is generated by the rasterizer is made to grab four fragments from the original array in the kernel program and then perform a local-reduction such as a sum or a max, as shown in Fig. 3.1. The results of this first stage must

then be stored onto a temporary array that is attached to the framebuffer. Once all the fragments have been processed, this temporary array is set as an input for a subsequent pass that generates a quarter of the fragments produced in the first pass; and the result of this pass is stored in a second temporary. This procedure is repeated several times between the two temporaries, until the array is reduced to a rectangle smaller than R_{min}, which can then be handled appropriately to provide the result of the reduction operation.

There are two important issues to be noted in the procedure described above - the use of temporary arrays and the handling of the final array after succesive reduction steps. The use of temporary arrays was necessary because of the read-only / write-only nature of the arrays on the GPU. Also, prior to a reduction operation, it is necessary to ensure that the temporary arrays are at least a quarter the size of the array that is to be reduced; and appropriate resizing is required if they are not.

The second issue is the handling of the final reduction-step. One approach is to pad the small rectangular array back to a square with the dimensions of R_{min}, and then proceed with more reduction steps until a single value is obtained; as shown in Fig. 4.1. Another approach is to read the results back to the main memory and perform the rest of the reduction on the CPU.

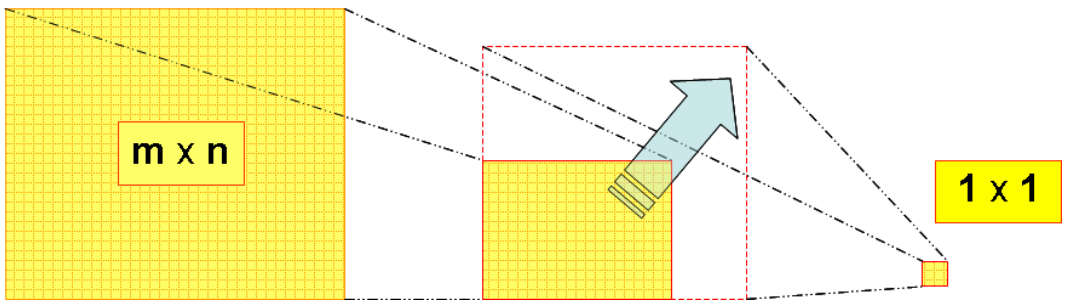


Figure 4.1. Reduction of rectangular arrays: Padding approach

4.1 Algorithm

The following algorithm describes a conventional sum-reduction operation on the graphics processor. This technique will use the vertex processors in addition to the fragment processors, two temporary arrays, and a read-back operation to the CPU for the final reduction. The first step is to compile the shader programs for both the vertex and fragment processors. The shader program for the vertex processors is given as:

```
// Vertex-shader program
void main()
{
    vec2 Coord = gl_MultiTexCoord0.xy;

    // Passes vertices straight through to the rasterizer...
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // Modify the coordinates that are passed into the fragment processor
    gl_TexCoord[0] = vec4(Coord + 0.5, Coord - 0.5);
}
```

The sum-reduction kernel, defined for the fragment processors, is given as:

```
// Fragment-shader program
uniform sampler2DRect Source;
void main(void)
{
    vec4 quad;

    quad.x = texture2DRect(Source, gl_TexCoord[0].xy).r;  // Top right
    quad.y = texture2DRect(Source, gl_TexCoord[0].zy).r;  // Top left
    quad.z = texture2DRect(Source, gl_TexCoord[0].xw).r;  // Bottom right
    quad.w = texture2DRect(Source, gl_TexCoord[0].zw).r;  // Bottom left

    gl_FragColor.r = dot( quad , vec4(1.0, 1.0, 1.0, 1.0) );
}
```

Both the shaders are compiled using the `CompileKernel` routine described in Chapter 2. Once this is done, the array to be reduced is defined as an input, using the `glUniformARB` GL call. The first temporary array (which must have dimensions of atleast width/2 and height/2), is then attached to the framebuffer as an output. The

viewport is set to the dimensions of the input array, using the `setGPUview` routine. However, the call to the `RunProg` routine is now different from the axpy call:

```
RunProg(width/2, height/2, width, height)
```

This call now draws the vertices at half the width and height of the original array, thereby forcing the rasterizer to generate a quarter of the number of fragments. However, since the coordinate locations (the third and fourth arguments) are still specified at the corners of the original array, the rasterizer must now interpolate the indices accordingly. As a result, the generated coordinate indices vary across the whole array as shown by the diamond symbols in Fig. 4.2.

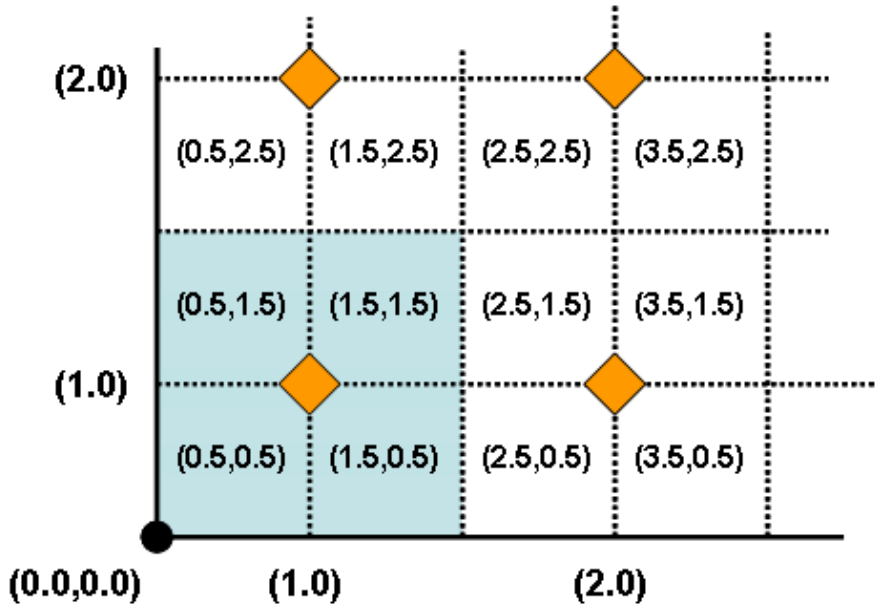


Figure 4.2. Interpolation of indices in a sum-reduction

The shaded portion in the figure signifies the output element, representing four elements from the original array. It is also worthwhile to note that the indices of the output array now vary in steps of 1 in each direction, rather than 0.5; whereas the indices of the input array are still varied in steps of 0.5. This variation of indices is

utilized in the vertex program to generate appropriate coordinates for the fragment program. By adding or subtracting a value of 0.5 from the output array indices in the vertex program, the coordinates of the four input array indices are obtained. These values are then written to the `gl_TexCoord[0]` variable, which is passed on to the fragment program. The fragment program uses combinations of these indices to access the four elements of the input array to perform a local sum before writing out the final result.

The output from this pass can now be processed in a subsequent pass that uses the same vertex and fragment programs. A second temporary array is attached to the framebuffer, and the first temporary is used as an input. The `RunProg` routine is called again with the first two arguments as width/4 and height/4, and the second two arguments as width/2 and height/2. Now that two temporary buffers are available, these passes can be performed in a loop until a sufficiently small rectangle is obtained.

This technique is a fairly efficient approach to the problem; however, certain factors must also be taken into consideration. For instance, there is always a small fixed-cost involved with the set up. There comes a stage in the computation when this cost is large when compared to the cost of performing a local-reduction for a small array. In such a case, it would be a sensible idea to read the remaining data back to the main memory and perform the rest of the reduction on the CPU, since it is relatively more efficient on small sets of data (Data is read back to the main memory using the `glGetTexImage` function). The source code for this technique is provided in Appendix. B

4.2 A different approach

Another approach to the reduction operation is to split the array logically into four quadrants. For each pass of the previous reduction approach, the value of each element of the output array is the sum of values from each of the four quadrants,

seperated by a distance of width/2 and height/2, as shown in Fig. 4.3b. This can be done in either one of two ways - a modification to the fragment program that is used for the sum operation, or by modifying the `RunProg` routine.

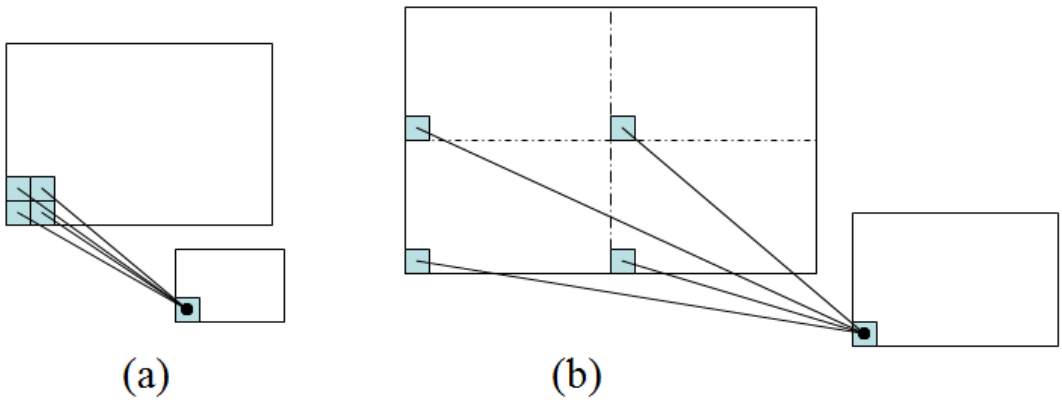


Figure 4.3. Reduction methods: (a) local reduction (b) quarter reduction

Considering the first approach, the fragment program can be modified as follows:

```
// Fragment-shader program
uniform sampler2DRect Source;
uniform float width;
uniform float height;
void main(void)
{
    vec4 quad;
    vec3 offset = vec3(0.0,width/2,height/2);

    // 1st quadrant
    quad.x = texture2DRect(Source, gl_TexCoord[0].xy).r;
    // 2nd quadrant
    quad.y = texture2DRect(Source, gl_TexCoord[0].zy + offset.xy).r;
    // 3rd quadrant
    quad.z = texture2DRect(Source, gl_TexCoord[0].xw + offset.yz).r;
    // 4th quadrant
    quad.w = texture2DRect(Source, gl_TexCoord[0].zw + offset.xz).r;

    gl_FragColor.r = dot( quad , vec4(1.0, 1.0, 1.0, 1.0) );
}
```

Now that the required offsets are computed in the fragment program, the use of a vertex program is now unnecessary. When this fragment program is used, the `RunProg` routine must now be called as follows:

```
RunProg(width/2, height/2, width/2, height/2)
```

In the previous technique, the rasterizer was forced to perform a linear interpolation of the coordinates; which tends to reduce performance. This aspect will be quite apparent when the performance results are considered. In this approach however, since the coordinates vary between zero and width/2 (or height/2) without any interpolation, the efficiency of the operation improves.

The second approach is to eliminate the offset computation in the fragment program altogether, in an effort to reduce the number of operations for efficiency. To achieve this, the fragment program can be provided with four sets of indices by modifying the `RunProg` routine as follows:

```
void RunProg(float v_width, float v_height, float f_width, float f_height)
{
    glBegin(GL_QUADS);

      glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0,       0.0);
      glMultiTexCoord2fARB(GL_TEXTURE1_ARB, f_width/2, 0.0);
      glMultiTexCoord2fARB(GL_TEXTURE2_ARB, 0.0,       f_height/2);
      glMultiTexCoord2fARB(GL_TEXTURE3_ARB, f_width/2, f_height/2);
      glVertex2f(0.0, 0.0);

      glMultiTexCoord2fARB(GL_TEXTURE0_ARB, f_width/2, 0.0);
      glMultiTexCoord2fARB(GL_TEXTURE1_ARB, f_width,   0.0);
      glMultiTexCoord2fARB(GL_TEXTURE2_ARB, f_width/2, f_height/2);
      glMultiTexCoord2fARB(GL_TEXTURE3_ARB, f_width,   f_height/2);
      glVertex2f(v_width, 0.0);

      glMultiTexCoord2fARB(GL_TEXTURE0_ARB, f_width/2, f_height/2);
      glMultiTexCoord2fARB(GL_TEXTURE1_ARB, f_width,   f_height/2);
      glMultiTexCoord2fARB(GL_TEXTURE2_ARB, f_width/2, f_height);
      glMultiTexCoord2fARB(GL_TEXTURE3_ARB, f_width,   f_height);
      glVertex2f(v_width, v_height);

      glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0,       f_height/2);
      glMultiTexCoord2fARB(GL_TEXTURE1_ARB, f_width/2, f_height/2);
```

```
      glMultiTexCoord2fARB(GL_TEXTURE2_ARB, 0.0,       f_height);
      glMultiTexCoord2fARB(GL_TEXTURE3_ARB, f_width/2, f_height);
      glVertex2f(0.0, v_height);

    glEnd();
}
```

The `glMultiTexCoord2fARB` call allows multiple coordinates to be specified per vertex; thereby generating several sets of indices which are accessed by the appropriate `gl_TexCoord` variable in the fragment program. The corresponding fragment program would be:

```
// Fragment-shader program
uniform sampler2DRect Source;

void main(void)
{
    vec4 quad;

    // 1st quadrant
    quad.x = texture2DRect(Source, gl_TexCoord[0].xy).r;
    // 2nd quadrant
    quad.y = texture2DRect(Source, gl_TexCoord[1].xy).r;
    // 3rd quadrant
    quad.z = texture2DRect(Source, gl_TexCoord[2].xy).r;
    // 4th quadrant
    quad.w = texture2DRect(Source, gl_TexCoord[3].xy).r;

    gl_FragColor.r = dot( quad , vec4(1.0, 1.0, 1.0, 1.0) );
}
```

4.3 Results

The performance results of the sum operation vs. the CPU is shown in Fig. 4.4. The plot shows a trend similar to the results from the axpy operation. Note the log scale on this figure exaggerates the importance of small vector lengths which are actually not typical in scientific computations. Scientific computations use vectors of 20k or more (about where the cross over in the performance of the sum occurs). The figure is based on the actual vector length necessary, not padded vector length. The

amount of padding affects the performance and makes the GPU results noisier. When optimally implemented, the sum operation can be performed at roughly 1.5 Gigaflops on the GPU for array sizes typical of scientific computations. This is contrasted with the performance obtained from a 2 GHz Athlon64 CPU. The CPU even uses the faster (but error prone) naive summation algorithm and obtains a performance of about 0.4 GFlops (note that the front-side bus runs at 400 MHz on this machine).

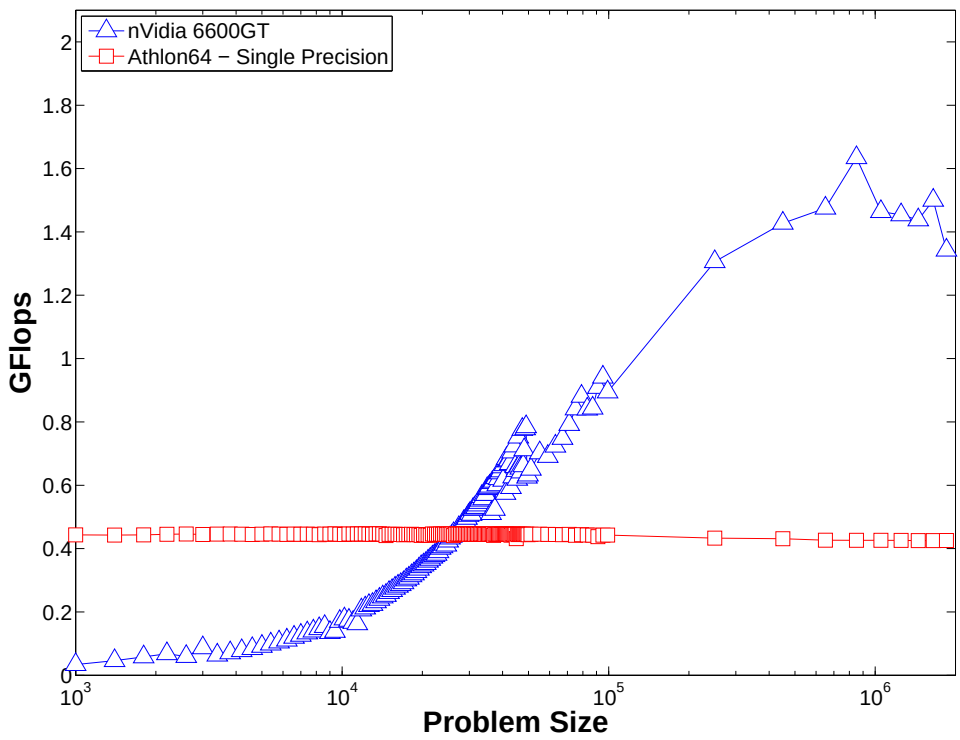


Figure 4.4. Performance of the sum operation

How the layout value of R_{min} affects the performance is shown in Fig. 4.5. The case with $R_{min} = 128$ shows clear plateaus. With this case, no quad reductions are performed up to $0.75(128^2) \approx 13k$ data items. The array is simply read back to the CPU to be summed. After that, the 1 quad (up to 49k) and 2 quad reduction levels (up to 197k) are easily seen. The excessive data transfer to the CPU makes $R_{min} = 128$ inefficient for the smaller vector lengths. The optimum R_{min} probably lies around

64. The $R_{min} = 32$ case is actually reducing too much and not sending enough data to the CPU, which is why its performance actually drops as it goes to the next level of reduction.

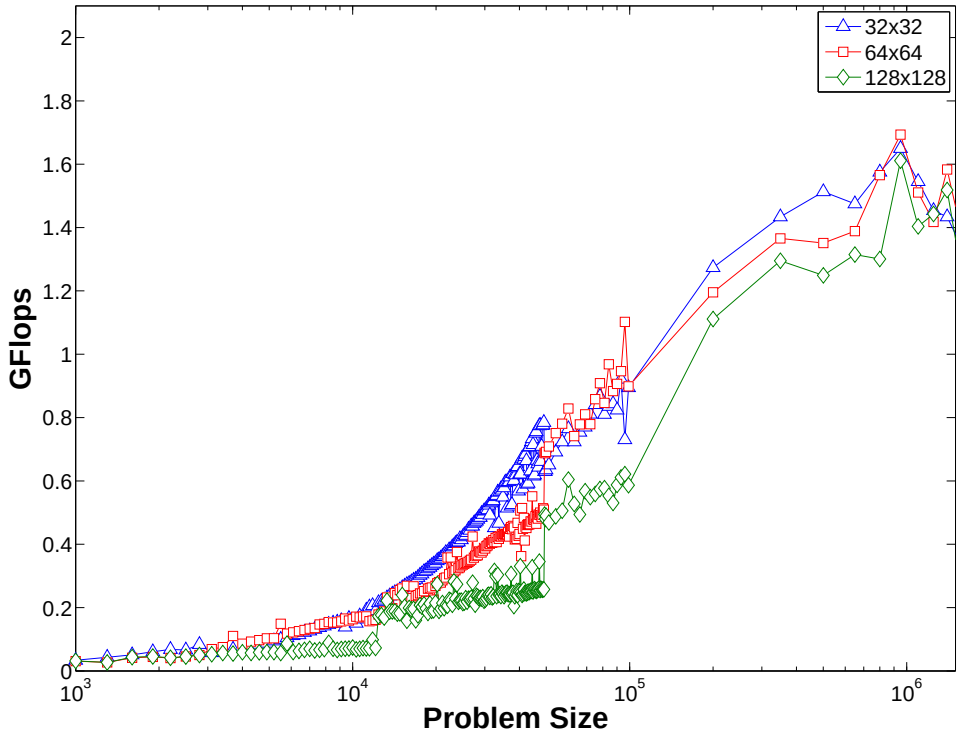


Figure 4.5. Performance of the sum operation for various cases of R_{min}

The dot product reduction can take place in two steps, array multiplication and then a sum reduction, or within a single program that reads 8 values and produces an array of a quarter the size, which is then summed. The latter approach avoids an additional array read and write of intermediate values, and since the computations on the GPU (and CPU) are memory bound, this increases the speed by 50%. Fig. 4.6 shows the dot product performance. An optimal performance of about 1.75 Gigaflops is obtained on the GPU for large vector lengths. The CPU obtains 0.55 GFlops for large single precision vectors.

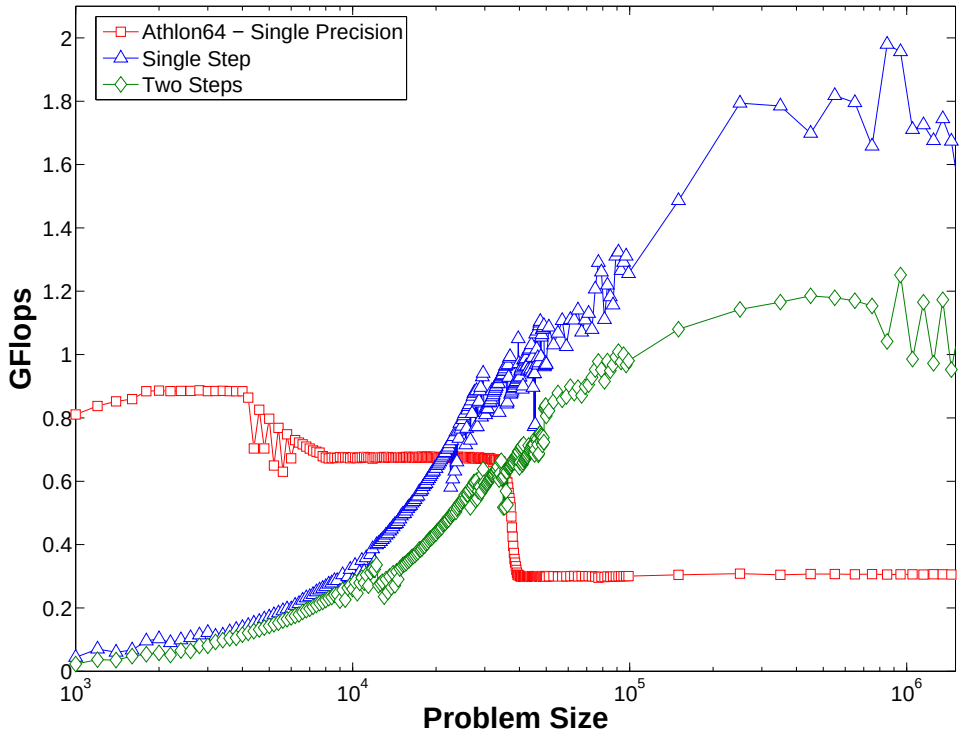


Figure 4.6. Performance of the dot-product operation with different approaches

4.4 Comments

Because of the partial reducibility of the 2D layout scheme, the first stages of reductions of this sort produce reduced array sizes that are still integer dimensions (of very predictable size). In addition, after these reduction stages, it can also be guaranteed that the data has been reduced to an array of size less than R_{min}^2. For a typical $R_{min} = 32$, this is less than 1024 data items. For this small vector length, it is inefficient to further reduce the data on the GPU and the data is read to the CPU where is summed using a traditional CPU summation algorithm.

Summation using this algorithm is numerically far less prone to round-off errors. Using a naive summation of a million single precision data items can frequently lead to errors in the summation of the order of 1%. One hundred million single precision items naively summed can have no precision at all. While more accurate and prob-

ably necessary even on a CPU, the quad reduction approach to summation is also slightly slower than naive summation because intermediate values must be stored and retrieved between each stage of the reduction. Since the speed of a summation is dictated on both the CPU and GPU by sequential memory access times, the staged reduction approach takes from 25% more time for one level of reduction to $\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + ... = \frac{1}{3} = 33\%$, for a large number of reductions.

In reduction operations, the padded data must be treated appropriately. For a sum operation or dot product, the padded data is set to zero, so it has no effect. For max and min operations the padded data is set to the first data item of the array, so that that 0 never mistakenly is reported as the maximum or minimum value in the array.

CHAPTER 5

SPARSE MATRIX OPERATORS

A vector multiply by a sparse matrix is usually the essence of scientific calculations. These operations dominate solution times in iterative solvers, since they frequently require random memory accesses. Sparse matrices are never actually stored entirely in memory, and several strategies exploit their sparsity to store the non-zero entries in a row or column-compressed format to save on space. In this implementation, the matrix-vector multiply is implicit, and the sparse matrix is cast as successive operations on the input vector, p, thereby yielding a result vector, w. Many of these sparse-matrices represent a discrete version of calculus operations, like a divergence or a gradient. Others represent interpolation or integration operations.

5.1 Mesh data-structures

When solving a PDE on a domain, the geometry must first be discretized. Cartesian block-structured meshes do not require explicit mesh-connectivity information and often suffice for simplified cases, but they prove to be inadequate in the case of complicated geometric topology. Unstructured meshes involving tetrahedra, for instance, capture complex geometries well and are popular choices for the vast majority of practical simulations. This versatility comes at the cost of having to construct mesh-connectivity information explicitly, which can consume a sizeable amount of memory in large cases. These systems also give rise to sparse-matrices that lack a coherent structure, thereby forfeiting the use of several specialized matrix-solvers.

With regard to sparse-matrix operators, the mesh connectivity information is useful in determining the data-elements in the vector that the operator is meant to work on. For instance, a gradient operator which seeks to evaluate the flux of a particular quantity at the faces of the elements in the mesh, must first obtain the values of that quantity existing at the two cells that lie on either side of the face. This information is stored in a connectivity list that maintains the indices of the two adjacent cells for every face in the mesh. Thus, by performing a loop over all the faces, the gradients at faces can be obtained. In some sense, this approach implicitly represents the non-zero entries of each row in the matrix representing an operator. A combination of such matrix operations can eventually be used to represent the discrete form of the partial differential equation that is to be solved.

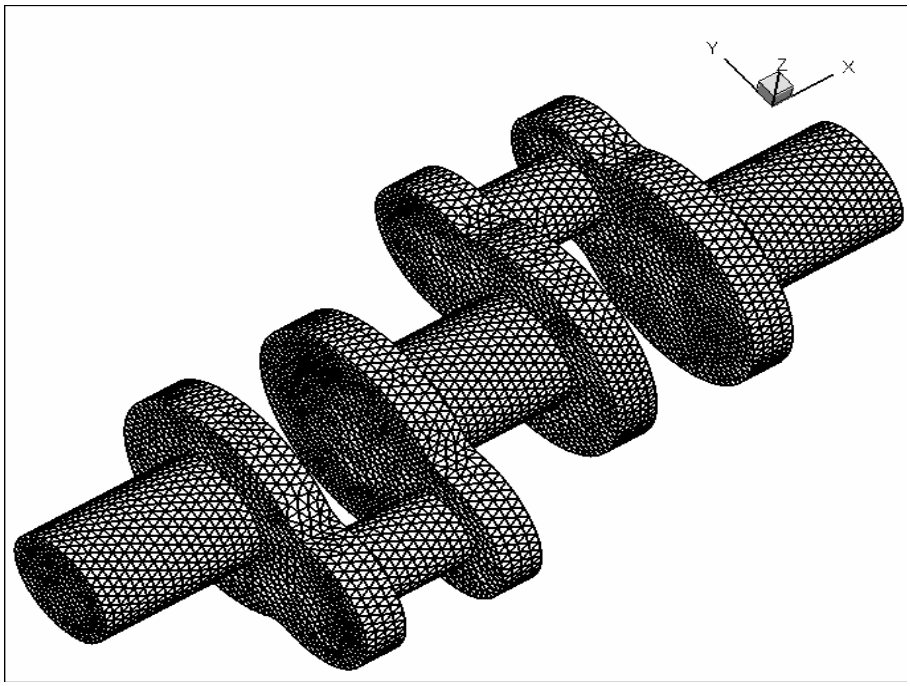


Figure 5.1. Unstructured tetrahedral mesh of a crankshaft (from NetGen). This particular mesh consists of 37151 nodes, 178486 cells, 370319 faces and 228983 edges.

This paradigm allows a large amount of flexibility in the way the equations are constructed, and with the use of approporiate polymorphism, operators can be designed

to work on several hardware platforms. The object-oriented paradigm in place therefore allows switching between GPU-operators, IBM Cell-operators and conventional CPU-based operators with very minimal effort. For a GPU-based implementation of the operator, the integer-based array indexing strategy can no longer suffice, as they need to be recomputed as texture-coordinate locations. This is simple to do, as any index `i` can be converted to a two-dimensional coordinate `[x,y]` by the operation `[x,y] = [mod(i,width)+0.5,floor(i/width)+0.5]`, where `width` is the width of the two-dimensional GPU array (The 0.5 is added since addressing is done at the center of each fragment). This compute is done first on the CPU during the preprocessing stage after the connectivity structures are read-in from the mesh file, and then uploaded to texture memory, so the appropriate connectivity structures are readily available to the relevant operator during the matrix-multiplication stage.

Certain restrictions also exist in this paradigm. For certain connectivity structures like Edge-to-Face (which contains face indices for every edge in the mesh), an additional level of indirection is required since the number of faces touching an edge is not constant. The connectivity structures store the number of faces for each edge, in addition to the locations of the faces themselves. Thereafter, loops would have to be invoked within the fragment program to provide a second level of indirection in order to obtain a final value.

As mentioned earlier, there are two broad categories of indirect memory-access patterns - scatter and gather. A gather operation is an indirect read from memory, of the form: `x=a[i]`, where `i` denotes the array-index. A gather operation maps naturally to a texture-fetch operation, where each fragment value can be the result of computations involving data from several locations in the texture memory. The Gradient operator is a good example of this category, which is evident from its associated fragment program:

```
// Gradient operator: Cell->Face

uniform sampler2DRect F2C;
uniform sampler2DRect Cell;

void main(void)
{
    vec4 CellCoord = texture2DRect(F2C,gl_TexCoord[0].xy);

    // Gradient(f) = Q(cell[2]) - Q(cell[1])
    gl_FragColor.r =   texture2DRect(Cell,CellCoord.ba).r
                     - texture2DRect(Cell,CellCoord.rg).r;
}
```

In the example shown above, `F2C` represents a face-to-cell connectivity structure that contains two indices, `cell[1]` and `cell[2]`, for every face in the mesh. Obviously, boundary faces only have the `cell[1]` index, while `cell[2]` is null. On a CPU, these indices point to different locations on a single-dimensional array. On the graphics processor, the F2C structure is a two-dimensional `vec4` array which contains the 2D indices for `cell[1]` in its '`r`' and '`g`' components; and those for `cell[2]` in its '`b`' and '`a`' components. The program merely fetches the appropriate values from `Cell`, subtracts them and writes the result to the output array.

A scatter operation, on the other hand, is an indirect write to memory, of the form: `x[i]=a`. This form of memory access is not natively supported on the GPU, since a fragment is specifically mapped to a specific coordinate location on the screen, as determined by the rasterizer, and deviation from this location is not possible. This type of operation is frequently required in several circumstances, like a divergence operation, for instance. The discrete divergence operator for a finite-volume paradigm determines the algebraic sum of values located on faces of the polygon at the cell-center, as dictated by the Gauss theorem. (A detailed discussion of these discrete-calculus operators is provided in [31])

A conventional approach to this operator on the CPU involves a loop that visits all faces in the mesh, adding the face-value for each cell located on one side of the face,

and subtracting the same value from the cell located on the other side - essentially a scatter. Such algorithms must be reformulated as a gather operation, as a workaround for the hardware limitation. The reformulated algorithm would now be a loop over all cells in the mesh, adding the values located at the faces, multiplied by the appropriate sign determined by the outward-facing normals. This would mean that a new cell-to-face (C2F) connectivity structure would now have to be constructed. If polyhedral cells are to be accounted for, an supplemental indexing array would be required as well. Put together, the resulting fragment program for the divergence operator is given:

```
// Divergence operator: Face->Cell
uniform sampler2DRect C2F;
uniform sampler2DRect C2F_SIGN;
uniform sampler2DRect index_SE;
uniform sampler2DRect Face;
uniform float texWidth;
uniform float nfWidth;

void main(void)
{
   float off, TINY = 1e-5, c2f_coord, sum = 0.0f, signval;
   vec2 Coord, face_coord;

   // Obtain the start location
   vec3 index = texture2DRect(index_SE,gl_TexCoord[0].xy).rgb;

   for (off = 0.0; off < 6.0; off++) {
     if (off == index.r) break;
     // Compute the Cell2Face coordinate location
     Coord = vec2(mod(index.g+off+TINY,texWidth) + 0.5,
                  floor((index.g+off+TINY)/texWidth) + 0.5);
     // Compute the face-coordinate
     c2f_coord  = texture2DRect(C2F,Coord).r;
     signval    = texture2DRect(C2F_SIGN,Coord).r;
     face_coord = vec2(mod(c2f_coord+TINY,nfWidth) + 0.5,
                       floor((c2f_coord+TINY)/nfWidth) + 0.5);
     // Use the coordinate to fetch the Face value, and accumulate
     sum += signval*texture2DRect(Face,face_coord).r;
   }
   gl_FragColor.r = sum;
}
```

The program, when executed for every element of the cell-array, first determines the number of faces (`index.r`), starts a loop for that number, dynamically computes the appropriate indices in `C2F` (`face_coord`), and uses that to accumulate to a variable (`sum`), which is then finally written out.

There are some aspects worth mentioning here. Firstly, `texWidth` is the width of the `C2F` array, which is required for the dynamic index-computation. Another oddity is the fact that a conditional is used to break out of the loop which runs a fixed number of times. This is a hardware limitation, since the graphics processor was not designed for loop constructs, and therefore the compiler must manually unroll the loop. Since loop unrolling is a compile-time process, variable loop limits are forbidden. The number 6.0 is chosen based on the fact that the common 3D element, the hexahedron, has six faces for each cell (other elements like the tetrahedron have only four). This value would have to be changed if the code is to account for polyhedra with more facets; but that change is trivial since shaders can be modified and compiled at run-time. And finally, a small value `TINY` is always included in the coordinate computation to account for rounding artifacts, since graphics processors deviate from the IEEE standard by rounding-to-zero rather than rounding-to-nearest.

5.2 Results

The performance results of various sparse-matrix operators are shown below. Each data-point represents the time taken by the operator to evaluate one element (or 'unknown') of the given mesh. For instance, in a gradient operator which evaluates quantities that reside on faces in the mesh, the cost-per-unknown is defined by dividing the operator-time with the number of faces. To avoid noise in the performance timings, these ratios are averaged over several hundred iterations for consistency. For larger problem-sizes, this cost would be expected to decrease as any fixed costs become amortized - a trend that is observed in general for the graphics processor with

increasing mesh-sizes, whereas the cache-based CPU tends to show the opposite behaviour, i.e., an increase in cost. This can be explained by the fact that there is an increase in the likelihood of memory-fetches falling out of the cache boundaries with larger mesh sizes.

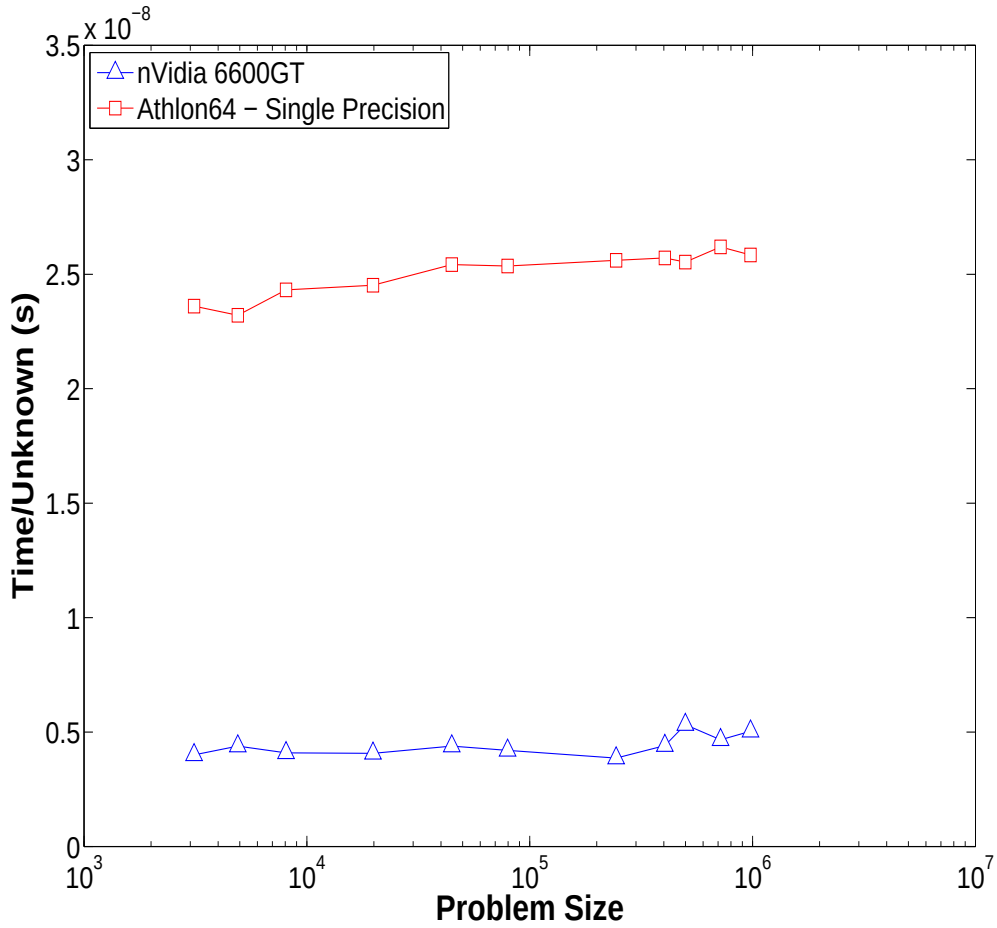


Figure 5.2. Performance of the gradient operator. Problem Size denotes the number of faces in the mesh.

The GPU implementation of the gradient operator (Fig. 5.2), shows a consistent 5x improvement over the CPU for all meshes. Like all scientific operations this is clearly bound by the memory bandwidth of the hardware and, to a smaller extent, the matrix bandwidth.

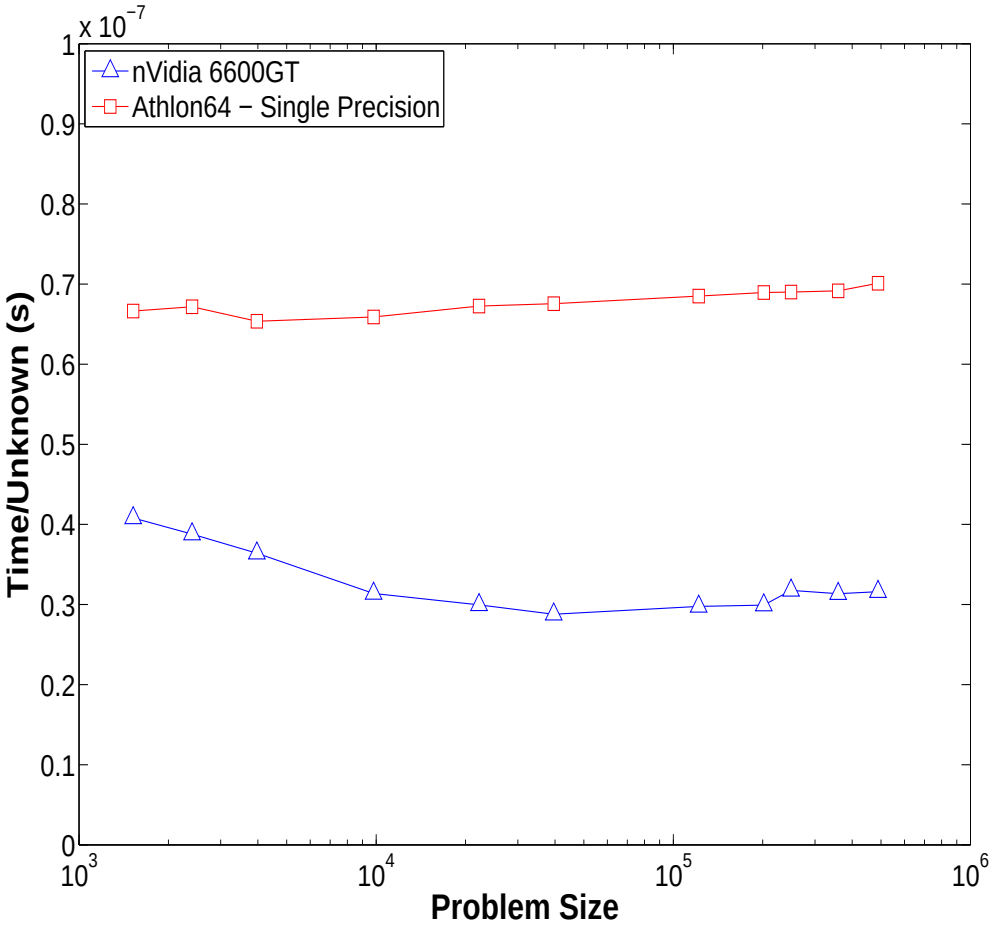


Figure 5.3. Performance of the divergence operator. Problem Size denotes the number of cells in the mesh.

The divergence operator (Fig. 5.3) also shows a similar trend, but the difference in performance is less pronounced - the graphics processor outperforms the CPU by a factor of about 2.5x for relevant problem sizes. It is worthwhile to note the deterioration in performance of the divergence operator when compared to the gradient is due to more memory accesses per result.

The curl operator (Fig. 5.4) is also similar to the divergence, and can be classified as a scatter-type operation. This operator typically operates on a scalar value which is located at faces (like a face-normal fluid velocity component, for instance) to obtain a quantity at edges in the mesh (like the stream-function). It also performs similarly,

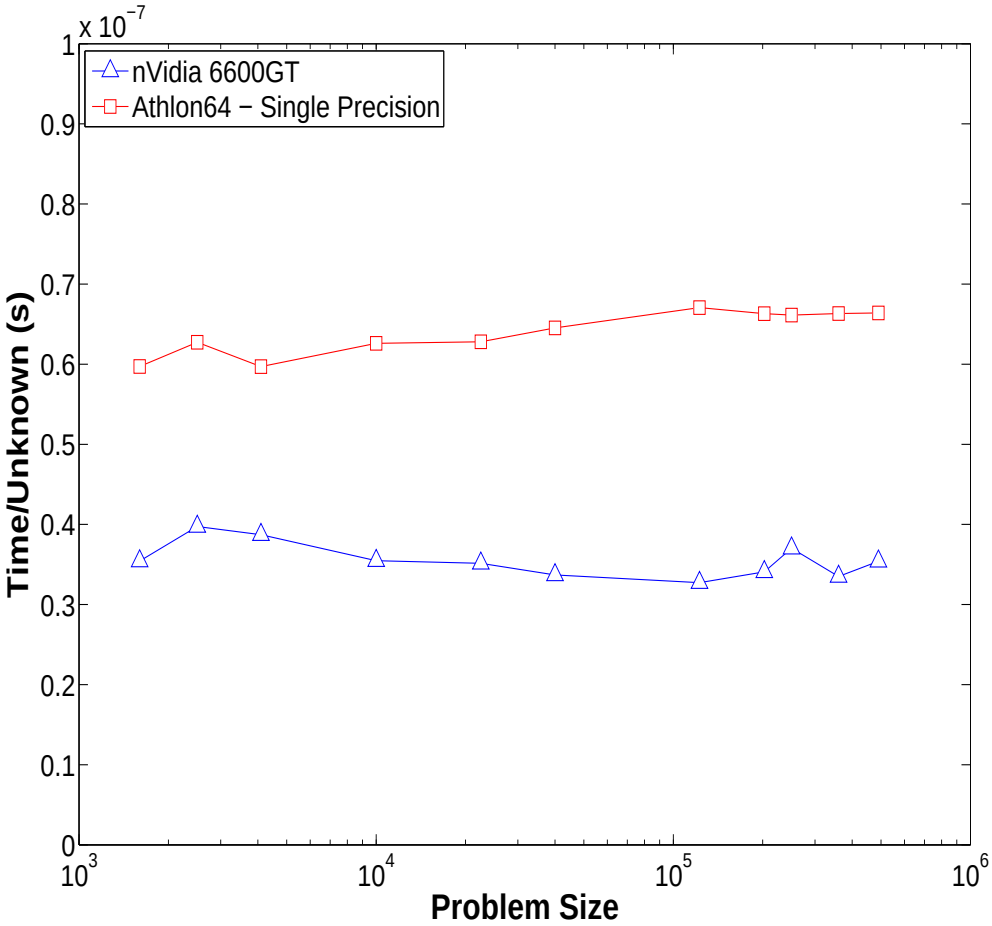


Figure 5.4. Performance of the curl operator. Problem Size denotes the number of edges in the mesh.

showing an improvement of about 2x for relevant problem sizes. A complement to the curl operator is the rotation operator which operates on edge-quantities to obtain values at faces in the mesh. Both operators are described later in the context of the Exact Fractional Step method.

The interpolation operator (Fig. 5.5) is a lower order reconstruction technique that is used to obtain vector quantities at the cell centroid (like a cell-centered velocity) from scalar quantities, like a face-normal fluid velocity component. This is given by the discrete interpolation formula as described in [26]:

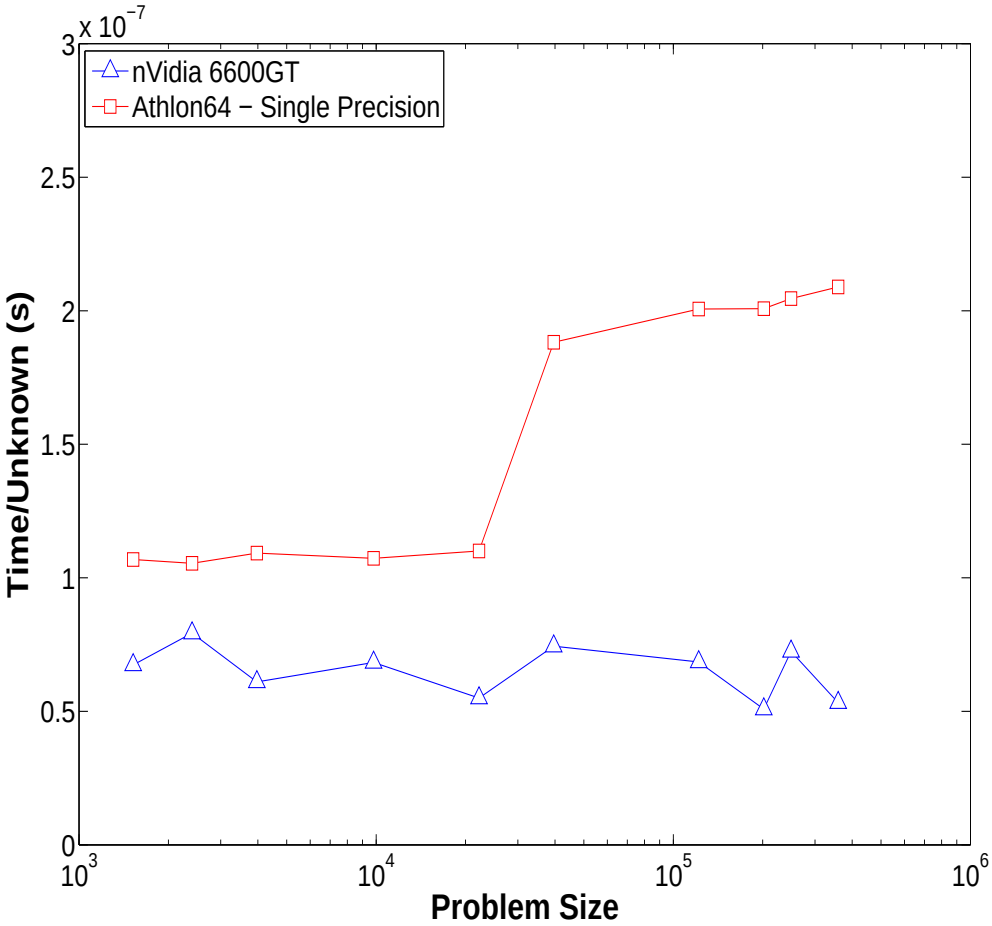


Figure 5.5. Performance of the interpolation operator. Problem Size denotes the number of cells in the mesh.

$$\mathbf{v}^{CG} = \frac{1}{Vol_C} \sum \pm u_f A_f (\mathbf{x}_f - \mathbf{x}_c) \tag{5.1}$$

where CG stands for the (cell or face) center of gravity and the $\pm$ is to account for the fact that u_f should point out of the cell in question. A_f is defined as the face-area, $\mathbf{x}_f$ is the face-position vector, and $\mathbf{x}_c$ is the cell-position vector. This also qualifies as a scatter-type operation and involves more computational work per cell than a divergence operator.

An integration operator (Fig. 5.6) is a complementary operator to interpolation, yielding scalar values at faces from cell-centered vector quantities and represents

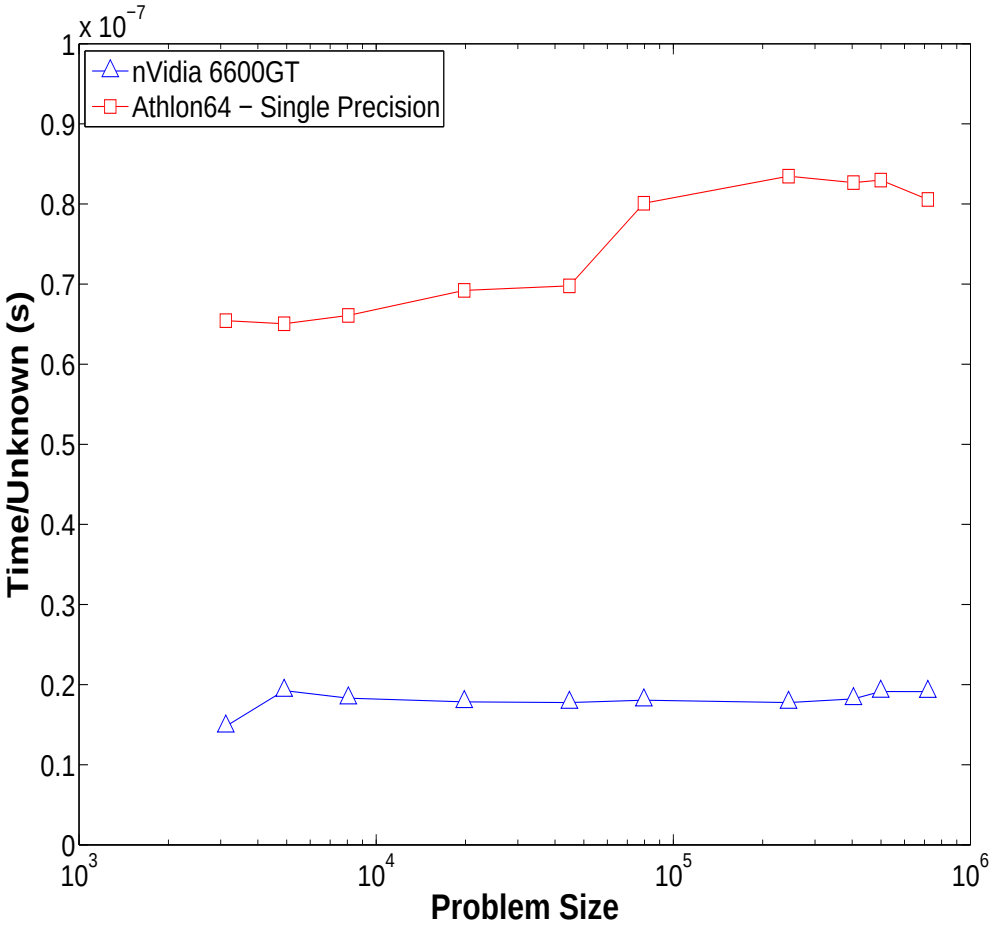


Figure 5.6. Performance of the integration operator. Problem Size denotes the number of faces in the mesh.

the integration along the median-dual edge connecting the two cell-centroids. The operator is first-order accurate and is described by the following formula:

$$u_f = \sum \pm \mathbf{v}^{CG} \cdot (\mathbf{x}_f - \mathbf{x}_c) \tag{5.2}$$

The performance improvement is by a factor of 4x for both operators.

5.3 Handling of Boundary Conditions

The description of any PDE system is complete only when the boundary conditions are defined. For a discrete system, this process involves the specification of values on boundary entities such as nodes, edges or faces in the mesh. Boundary conditions fall into two broad categories - Dirichlet and Neumann. A Dirichlet condition is always specified directly for the variable itself, such as a specified constant temperature on all inlet faces in a heat-diffusion problem for instance. Neumann conditions, on the other hand, are applied to derivatives of the variable in the system of PDEs. This condition becomes useful in situations where the gradient of velocity (shear for Navier-Stokes), or the gradient of temperature (heat-flux, by the Fourier Law) is specified at the boundaries.

When boundary conditions are viewed from the perspective of implementation, it falls under the scatter category of operators, since the process involves the specification of values at a sub-set of entities in the mesh (`x[i] = bc`). One possible approach is to reformulate it as a gather - using a method that is very similar to the other operators seen so far. This technique involves the use of a boolean 'flag' field which specifies whether a given entity (such as a boundary node/edge/face) lies on a boundary or not. If it does, then a reference must be provided to another field which specifies the actual boundary-condition value. The actual application process involves looping through all entities in the mesh and then referencing the 'flag' field to determine the boundaries, each of which involves a conditional statement. This tends to be slightly problematic, since it is a considerable waste of memory resources in addition to the sheer inefficiency of the approach. All faces must be visited to apply boundary conditions to a very small number of them.

A cost-efficient alternative is to use the point-sprite feature in OpenGL. This feature is intended for rendering small bitmaps (known as sprites) at arbitrary locations on-screen. In this scenario, this ability is used to write to a single fragment loca-

tion, which provides the effect of specifying values for boundary entities located at arbitrary points in the solution field.

Although this technique is straightforward in intent, making it efficient is less trivial. In a conventional OpenGL implementation, this involves a `glVertex` call for every boundary entity in the mesh - achieved by placing a loop within the `glBegin...glEnd` construct and using `GL_POINTS` instead of `GL_QUADS` as the primitive type. However, since this is an API library call, it involves the CPU and a heavy transfer of information across the system's front-side-bus, which can be quite ineffcient. For a large number of boundary conditions, this could well be in the thousands. An interesting work-around is to use the Vertex Buffer Object extension in OpenGL, which places this information on a buffer in high-performance memory on-board the GPU. Thereafter, only a single call to OpenGL is required to render all points in the buffer.

The following code-segment shows the generic approach to the implementation of point-sprites:

```
// Generate a buffer ID
glGenBuffers( 1, &bufferID );
// Bind the vertex buffer
glBindBuffer( GL_ARRAY_BUFFER, bufferID );
// Store in the vertex-buffer...
glBufferData( GL_ARRAY_BUFFER, 2*NumPoints*sizeof(float),
              point_coord, GL_STATIC_DRAW );
// Enable Point-sprites
glEnable( GL_POINT_SPRITE_NV );
// Don't replace texture-coordinates for each point...
// Use vertex coordinates instead
glTexEnvi( GL_POINT_SPRITE_NV, GL_COORD_REPLACE_NV, GL_FALSE );
// Hardware-acceleration while rendering point-sprites to FBOs requires
// this parameter to be set explicitly.
// Otherwise, a software-fallback is triggered.
glPointParameterfEXT( GL_POINT_SPRITE_COORD_ORIGIN, GL_LOWER_LEFT );
// Define the vertex-buffer pointer
glVertexPointer( 2, GL_FLOAT, 0, NULL );
// Enable the client-state and render points
glEnableClientState( GL_VERTEX_ARRAY );
glDrawArrays( GL_POINTS, 0, NumPoints );
// Now that we're done with the state, disable it.
glDisableClientState( GL_VERTEX_ARRAY );
```

```
// Release the buffer
glBindBuffer( GL_ARRAY_BUFFER, 0 );
// Disable point-sprites
glDisable( GL_POINT_SPRITE_NV );
// Delete the buffer
glDeleteBuffers( 1, &bufferID );
```

Note that this segment assumes the following:

- Appropriate viewport settings have been made.
- A texture is currently bound to the Framebuffer Object.
- A boundary-condition fragment program that sets the appropriate values has been compiled, linked and attached to the fragment processor.

The segment is fairly self-explanatory, and several online resources for VBOs also exist. A brief look at the main-content:

- After generating a buffer ID and binding it to the vertex buffer, `glBufferData` transfers data from a location in main memory to the graphics card. This data contains the two-dimensional coordinates for the locations of individual fragments that represent boundary entities.
- The `glVertexPointer` call defines the stride of the data (assuming that the data is packed tightly in groups of 2 in the array). The `NULL` pointer is an indication to the driver that the data being referenced points to the start of the buffer that is currently bound to `GL_ARRAY_BUFFER`.
- Finally, the `glDrawArrays` call starts rendering points by referencing data in the buffer. During the rendering pass, the fragment program only recieves texture-coordinates for the boundary fragments, and values can be set according to the program.

This approach yields a hardware-accelerated rendering path for point-sprites to a texture attached to the Framebuffer Object. An nVidia 6600GT using this approach consistently renders about 60 Million vertices (or points) per second. This will be inefficient in cases where the mesh has a large surface-to-volume ratio, but these situations are rare in practice. In real problems, the surface mesh is less than 2 percent of the interior mesh. In a parallel-processing configuration involving multiple graphics cards, this technique is also used to update the solution on entities that lie on CPU domain boundaries.

CHAPTER 6

THE CONJUGATE GRADIENT ALGORITHM

The Conjugate Gradient (CG) algorithm is a popular iterative method for solving systems of the form Ax=b, where the matrix A is symmetric and positive-definite. Direct solvers like Gaussian elimination and LU decomposition techniques have the advantage of reusability, since the matrix A has to be factored only once in the solution process and is then applicable for multiple cases of b. They are also less prone to round-off issues, as opposed to iterative techniques which gradually accumulate errors with increasing iterations. However, direct methods usually require the entire matrix to be stored in memory, and this becomes impossible for even moderately sized problems.

When A is sparse, factoring of such matrices generally tends to yield triangular factors that contain many more non-zero elements than the matrix A itself [29] and therefore, direct methods are no longer advantageous. Iterative techniques are generally more memory- and cost-efficient in these cases. Such systems frequently arise in the solution of discretized linear and non-linear partial differential equations such as the Poisson equation. They also form a large portion of the CPU cost of numerous incompressible flow solvers, since the solution for pressure is basically a Poisson equation to ensure continuity.

In theory, the Conjugate Gradient algorithm is guaranteed to converge in N iterations (where N is the number of unknowns in the system). However, this is never true in practice (due to round-off error), and convergence is usually achieved at a much faster rate. The algorithm (shown below) primarily consists of three operations that

must be highly efficient for the solution to be competitive in terms of computational cost - reduction operations like a vector dot-product, the axpy operation, and the sparse-matrix multiply operation.

$r_0 = b - Ax_0$
$z_0 = Pr_0$
$p_0 = z_0$
$\eta_0 = r_0 \cdot z_0$
for $i = 0, 1, 2, \ldots$ **do**
  $w_i = Ap_i$
  $\delta_i = p_i \cdot w_i$
  $\alpha = \eta_i / \delta_i$
  $x_{i+1} = x_i + \alpha p_i$
  Exit if convergence criteria is satisfied
  $r_{i+1} = r_i - \alpha w_i$
  $z_{i+1} = Pr_{i+1}$
  $\eta_{i+1} = r_{i+1} \cdot z_{i+1}$
  $\beta_{i+1} = \eta_{i+1} / \eta_i$
  $p_{i+1} = z_{i+1} + \beta_{i+1} p_i$
end

Algorithm 1: The Standard CG Algorithm

Having demonstrated the improvement in performance on a graphics processor over the CPU in these three elements, it is natural to expect a similar trend when the complete CG solver is implemented. As a practical example and a test-case for evaluation, the heat-diffusion equation is considered:

$$\frac{\partial (\rho c T)}{\partial t} = \nabla \cdot k \nabla T + S \tag{6.1}$$

Here, the temperature T is the fundamental unknown, S is any source term and the material parameters are, k the conductivity, and ρc the heat capacity. In steady-state conditions and the absence of a source, this reduces to a simple Poisson equation. In this work, the spatial discretization of this equation is implemented using a Discrete Calculus method, which is described briefly here. For an in-depth discussion, the reader is encouraged to refer to [31].

6.1 Node-based Discretization

In the node-based method of discretization, the temperature is placed at nodes in the mesh, each with a surrounding control volume defined as a dual-mesh cell. This is illustrated in Fig. 6.1, shown with normals for each of the dual-faces in 2D.

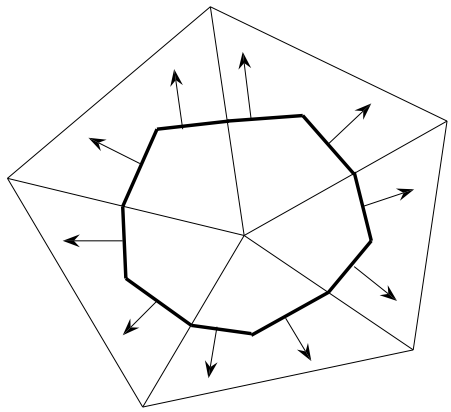


Figure 6.1. Dual mesh cell (formed by the bold lines and shown with dual-face normals) represents a nodal control volume for the enclosed node.

In 3D, the dual-faces are represented by triangles with vertices at the node, face and edge centres. Temperature is assumed to vary linearly within each cell, thereby yielding a constant gradient and consequently, when scaled with the diffusivity of the material, also yields a constant heat-flux in each cell. This gradient in a cell is defined by the relation:

$$\nabla T = \frac{1}{V_c} \sum T_f \mathbf{n}_f \tag{6.2}$$

A subsequent integration of the heat-flux for each cell yields the flux through the dual-faces (represented by a tilde), given by the equation:

$$Q_{\tilde{f}} = \sum_{cells} -k_c \nabla T \cdot \mathbf{n}_{\tilde{f}e} \tag{6.3}$$

The divergence of these fluxes then yield the temperatures at nodes. The resulting matrix system is symmetric, positive-definite and therefore, a good candidate for a solution using the CG solver.

6.2 Performance Results

Tests for performance were conducted using several discretization approaches, using meshes that gradually increase in resolution. As with the sparse-matrix operators, performance statistics were averaged over several iterations of the CG solver, and then subsequently divided by the number of unknowns to determine the computational cost per unknown. As a fair comparison, 2D triangular meshes (such as the one shown in Fig. 6.2) were used to solve the Poisson equation for temperature using the following boundary conditions:

$$\begin{aligned} x = 0 \quad & T = 0 \\ x = 1 \quad & T = 1 \\ y = 0 \quad & \frac{\partial T}{\partial y} = 0 \\ y = 1 \quad & \frac{\partial T}{\partial y} = 0 \end{aligned} \tag{6.4}$$

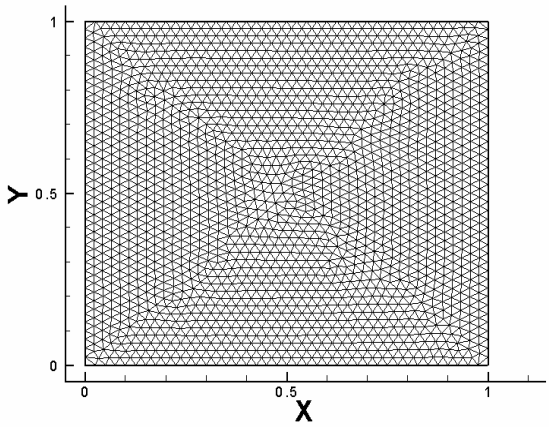


Figure 6.2. Typical mesh used for performance evaluation

The performace comparison between processors for the node-based discretization approach (Fig. 6.3) shows that the graphics processor outperforms the CPU by a factor of roughly 2.5x, with a lower computational cost as the mesh resolution increases.

Similar tests were also performed on three-dimensional meshes with more complicated geometry; such as heat-diffusion through a crankshaft mesh shown in the

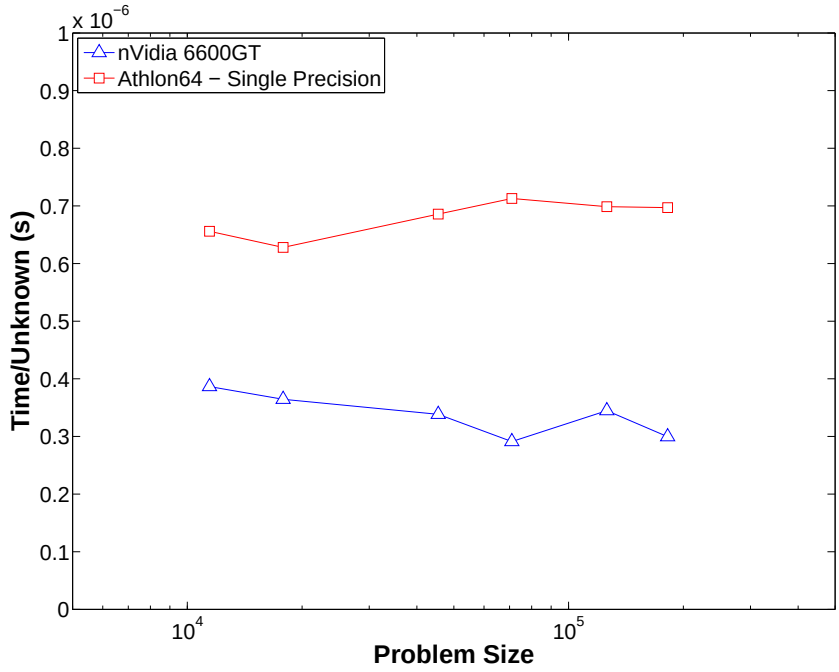


Figure 6.3. Performace comparison of the Conjugate Gradient solver using the node-based discretization of the Poisson equation. Problem size denotes the number of nodes in the mesh.

previous chapter. In this particular case, Dirichlet conditions for temperature were specified at the ends of the crankshaft, while Neumann conditions (for insulated walls) were specified on the other boundaries. Contour plots for the temperature distribution is shown in Fig. 6.4.

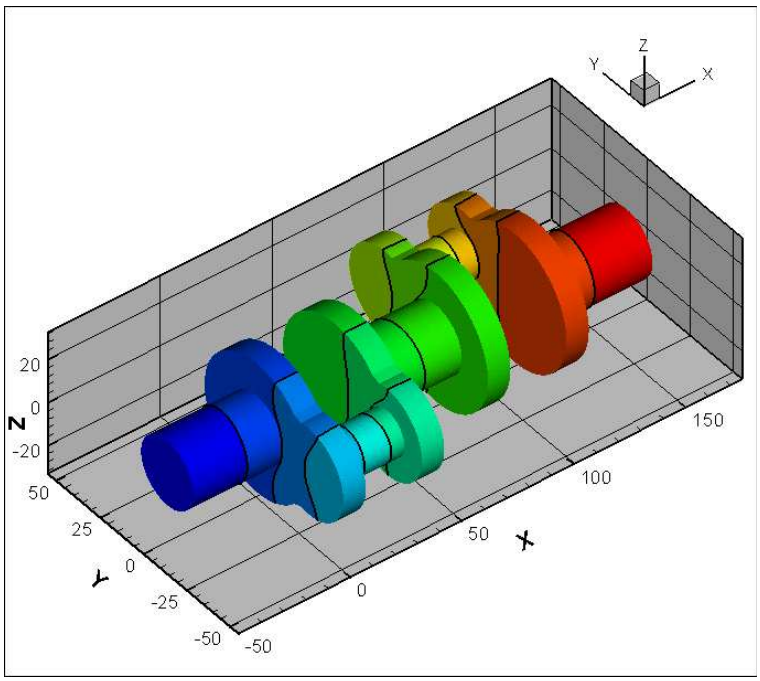


Figure 6.4. Contour plot for temperature along the Crankshaft

CHAPTER 7

IMPLEMENTATION OF THE NAVIER STOKES EQUATIONS ON GRAPHICS PROCESSORS

Fluid flows play an important role in several physical processes used in the industry today. In general, information about the structure of the flow in a process can be obtained from experimental measurements or from flow visualization studies, but a full picture of the flow field is often hard to obtain using this approach. Computational Fluid Dynamics, commonly abbreviated as CFD, is a technique to model fluid flow using computer simulations, and has proven to be a valuable tool to complement experimental findings in flow structure studies. The flow structure is computed by solving the mathematical equations that govern fluid dynamics. The result is a complete description of the three-dimensional flow in the entire flow domain in terms of the velocity field, pressure distribution and other related physical quantities.

7.1 Equations

The incompressible Navier-Stokes equations for fluid-flow (also assuming constant density), is given as:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \nabla \cdot (\nu \nabla \mathbf{u}) \tag{7.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{7.2}$$

Here, $\mathbf{u}$ is the velocity vector, p is the kinematic pressure (divided by density), and ν is the kinematic viscosity. For incompressible flow, the divergence of velocity is zero - a physical assumption which dictates that the pressure responds instantaneously

with changes in the velocity. This assumption generally simplifies the equations, but it also makes the task of solving them numerically challenging.

7.2 Discretization

The Navier-Stokes equations can be discretized into a convenient block LU decomposition [24] of the form:

$$\begin{bmatrix} A & G \\ D & 0 \end{bmatrix} \begin{bmatrix} U^{n+1} \\ p^{n+1} \end{bmatrix} = \begin{bmatrix} r^n \\ 0 \end{bmatrix} + \begin{bmatrix} bc\text{'s} \\ bc\text{'s} \end{bmatrix} \tag{7.3}$$

where G and D are the discrete gradient and divergence operators mentioned earlier, and A is a sub-matrix whose structure depends on the form of temporal and spatial discretization. The pressure p must always be solved implicitly when the equations are incompressible to enforce the incompressibility constraint (which must be true at the next time level n + 1). This is defined by the bottom row of the matrix in Eq. 7.3. The vector r^n is the explicit right-hand side of the momentum equations, and *bc's* are the boundary conditions for the momentum and pressure equations.

The discretization method in this case is using a staggered-mesh approach, where the discrete velocity unknowns are the face normal velocity components (which are located at the primary mesh faces) and the pressures (which are located at the cell centroids) as shown in Fig. 7.1. This is opposed to a collocated arrangement that involves both the velocity and pressure variables at cell-centers.

The normal velocity component U_f is defined by the equation:

$$U_f = \int_f \mathbf{u} \cdot \mathbf{n} dA_f \tag{7.4}$$

where $\mathbf{u}$ is the velocity vector, $\mathbf{n}$ is the outward pointing face normal vector and A_f is the face area. Note that the unknown U_f includes the face area in this definition.

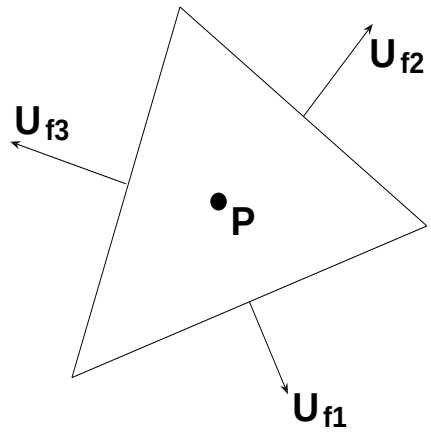


Figure 7.1. Unstructured staggered mesh scheme for the incompressible Navier Stokes equations

If the face normal velocity U_f is assumed as constant along the cell face, then the cell center velocity vector can be reconstructed (to first-order) using the relation:

$$\mathbf{u}_c = \frac{1}{V_c} \sum_{faces} U_f (\mathbf{x}_f - \mathbf{x}_c) \tag{7.5}$$

The convective fluxes at the cells are computed using the relation:

$$\mathbf{C}_c = \frac{1}{V_c} \sum_{faces} U_f \mathbf{u}_f \tag{7.6}$$

where $\mathbf{u}_f$ is either the upwind cell-velocity or the average of the two cell-velocities on either side of the face if a central-differencing scheme is used.

The diffusive fluxes at the cells are computed using the relation:

$$\mathbf{D}_c = \frac{1}{V_c} \sum_{faces} \left[\left(\nu \frac{A_f}{L} \alpha \right) G\mathbf{u}_c + \nu \mathbf{q}_f \cdot \mathbf{n}_f - \nu \mathbf{q}_f \cdot (\mathbf{r}_1 - \mathbf{r}_2) \left(\frac{A_f}{L} \alpha \right) \right] \tag{7.7}$$

where the various terms are given as :

$$G\mathbf{u}_c = \mathbf{u}_{c2} - \mathbf{u}_{c1} \tag{7.8a}$$

$$\mathbf{q}_f = 0.5(\nabla\mathbf{u}_{c1} + \nabla\mathbf{u}_{c2}) \tag{7.8b}$$

$$\nabla\mathbf{u}_c = \frac{1}{V_c}\sum_{faces}\mathbf{u}_f\mathbf{n}_f \tag{7.8c}$$

$$\mathbf{u}_f = 0.5(\mathbf{u}_{c1} + \mathbf{u}_{c2}) \tag{7.8d}$$

$$\mathbf{r}_1 = \mathbf{x}_f - \mathbf{x}_{c1} \tag{7.8e}$$

$$\mathbf{r}_2 = \mathbf{x}_f - \mathbf{x}_{c2} \tag{7.8f}$$

$$\alpha = \frac{(\mathbf{r}_1 - \mathbf{r}_2)\cdot\mathbf{n}_f}{L} \tag{7.8g}$$

The first term in the square brackets is the velocity-gradient tensor along the line connecting the two cell-centers. The second and third terms make corrections to the first term to account for its skewness with respect to the primary face - usually applicable to triangular and tetrahedral meshes. The velocity-gradient tensor at the face, $\mathbf{q}_f$ is computed for this purpose. For Cartesian meshes, the correction terms are zero. The orthogonality correction terms also make the system unsymmetric and therefore, for use with the CG solver, these terms must be treated explicitly (at time 'n'). A typical structure for A is to treat diffusion implicitly for stability and an explicit advection term along with a temporal term if the flow is unsteady. Both the convection and diffusion fluxes can then be integrated to the faces to obtain an equation-system for the face-normal velocites.

Although symmetric, the matrix system in Eq. 7.3 has both positive and negative eigenvalues and is therefore, not easy to invert. If an iterative method is used to solve this system then it must be converged to nearly machine precision, as any errors in the iterative solution mean that the incompressibility constraint is not exactly satisfied. These iteration errors show up effectively as local mass creation and destruction, and are highly detrimental to the overall solution accuracy.

7.3 The Classical Fractional Step Method

This system described in Eq. 7.3 can be decomposed further to yield the classical Fractional Step method:

$$\begin{bmatrix} A & 0 \\ D & -DA^{-1}G \end{bmatrix} \begin{bmatrix} I & A^{-1}G \\ 0 & I \end{bmatrix} \begin{bmatrix} U^{n+1} \\ p \end{bmatrix} = \begin{bmatrix} r^n \\ 0 \end{bmatrix} + \begin{bmatrix} bc\text{'}s \\ bc\text{'}s \end{bmatrix} \tag{7.9}$$

This method was first introduced independently by Chorin [4] and Temam [33] as a practical approach to the solution of incompressible fluid-flow. The matrix form of this approach was later described by Perot [25]. It introduces an intermediate velocity U^* and an appoximate inverse $\tilde{A}^{-1}$ to yield the following system:

$$\begin{bmatrix} A & 0 \\ D & -D\tilde{A}^{-1}G \end{bmatrix} \begin{bmatrix} U^* \\ p \end{bmatrix} = \begin{bmatrix} r^n \\ 0 \end{bmatrix} + \begin{bmatrix} bc\text{'}s \\ bc\text{'}s \end{bmatrix} \tag{7.10}$$

When written out explicitly:

$$AU^* = r^n + bc's \tag{7.11a}$$

$$DU^* = D\tilde{A}^{-1}Gp \tag{7.11b}$$

The simplest approximate inverse is:

$$\tilde{A}^{-1} = \frac{A_f}{L} I, \tag{7.12}$$

where I is the identity matrix, A_f is the face-area, and L is the distance between cell-centers. This approximation is exact if diffusion and convection are fully explicit. In doing so, the pressure becomes completely decoupled from the momentum equations and can therefore be solved as a Poisson equation; following which corrections

can be made to U^* to attain a divergence-free velocity field U at time n+1 using the relation:

$$U^{n+1} = U^* - \tilde{A}^{-1}Gp \tag{7.13}$$

If all terms in A are treated explicitly, only the Poisson equation has to be solved at each time-step to ensure incompressibility. It is at this step that all incompressible fluid-flow solvers spend the most time, and therefore justification for the need to perform this step efficiently, till machine precision.

This approach has a few major drawbacks, including the fact that it exhibits poor temporal accuracy (first order accurate) due to the approximation of A^{-1}. Also, due to the existence of iteration errors, the velocity field is never truly divergence-free at any time. This difficulty can be overcome using a variation known as the Exact Fractional Step approach [3], which is discussed next.

7.4 The Exact Fractional Step Method

The classical Fractional Step method never yields an exact solution to Eq. 7.3 due to the fact that the momentum and pressure equations are solved sequentially, thereby leading to a temporal splitting error. If they were solved simultaneously, this shortfall can be circumvented. This is achieved by the introduction of the streamfunction vector, s, the curl of which yields the face-normal velocity. In the discrete sense, this can be represented by:

$$U_f = Cs \tag{7.14}$$

By mathematical property, the divergence of the curl of any function yields zero. For Discrete Calculus methods, all calculus identities still hold algebraically (so, DC=0). Therefore, the application of the Discrete Calculus divergence operator to the curl of the streamfunction vector automatically ensures discrete incompressibility:

$$DU_f = DCs = 0 \tag{7.15}$$

In discrete terms, the streamfunction variable, s, is defined by the integrating the streamfunction vector along the edge of a polyhedral cell ($s = \int_{edge} \psi \cdot d\mathbf{l}$). Thus, the discrete curl operator, C, transfers values located at edges in the mesh to the primary faces.

Another objective of the Exact Fractional Step approach is to obviate the need for pressure in the Navier-Stokes equations. Again by mathematical property, the discrete divergence (which is defined as the negative transpose of the gradient, D = -G), and an additional rotation operator, R, is defined such that R = C^T (The rotation operator transfers variables located at primary faces in the mesh to edges). Owing to this relationship, it is evident that $0^T = (DC)^T = C^T D^T = -RG$. Thus, by applying this to Eq. 7.3 and invoking the definition $U_f = Cs$:

$$\begin{bmatrix} RAC & RG \\ D & 0 \end{bmatrix} \begin{bmatrix} s^{n+1} \\ p^{n+1} \end{bmatrix} = \begin{bmatrix} Rr^n \\ 0 \end{bmatrix} + \begin{bmatrix} Rbc\text{'}s \\ bc\text{'}s \end{bmatrix} \tag{7.16}$$

It is obvious that the off-diagonal terms in the matrix are zero. When written out explicitly:

$$RACs^{n+1} = R(r^n + bc's) \tag{7.17}$$

Note that in this case, the rotation operator, R, is defined as the transpose of the curl, C. Thus, if the Conjugate Gradient algorithm is used to solve the implicit terms in A, it can also be used to solve for RAC as well, since this system is guaranteed to also be symmetric and positive definite.

7.5 Performance Results

Tests for performance were conducted using both the Classical and Exact approaches, using meshes that gradually increase in resolution. As with the sparse-matrix operators, performance statistics were averaged over several iterations of the

CG solver, and then subsequently divided by the number of unknowns to determine the computational cost per unknown. As a fair comparison, 2D triangular meshes (such as the one shown in Fig. 7.2) were used to solve the driven-cavity problem using the following boundary conditions specified in Eq. 7.18:

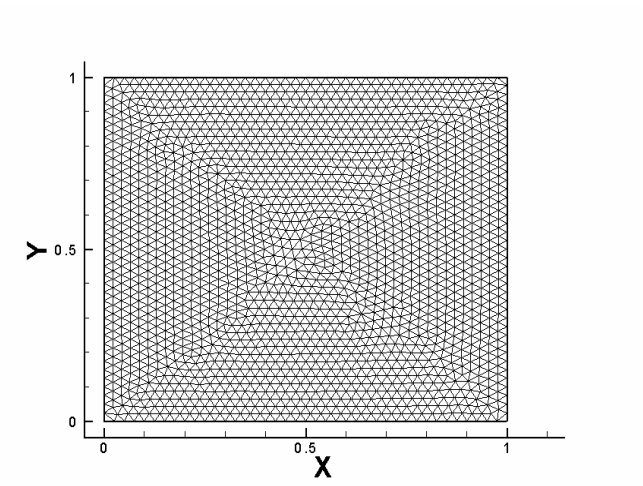


Figure 7.2. Typical mesh used for performance evaluation

$$x = 0; \quad U = 0, V = 0, \frac{\partial p}{\partial n} = 0 \tag{7.18a}$$

$$x = 1; \quad U = 0, V = 0, \frac{\partial p}{\partial n} = 0 \tag{7.18b}$$

$$y = 0; \quad U = 0, V = 0, \frac{\partial p}{\partial n} = 0 \tag{7.18c}$$

$$y = 1; \quad U = 1, V = 0, \frac{\partial p}{\partial n} = 0 \tag{7.18d}$$

In the Classical Fractional Step method, diffusion was treated implicitly (without the orthogonality-correction in the CG solver), while convection was explicit. This leads to two stages per time-step - momentum and pressure. The performace comparison for the Classical Fractional Step approach (Momentum Solution - Fig. 7.3, and Pressure Solution - Fig. 7.4) shows that the graphics processor outperforms the CPU

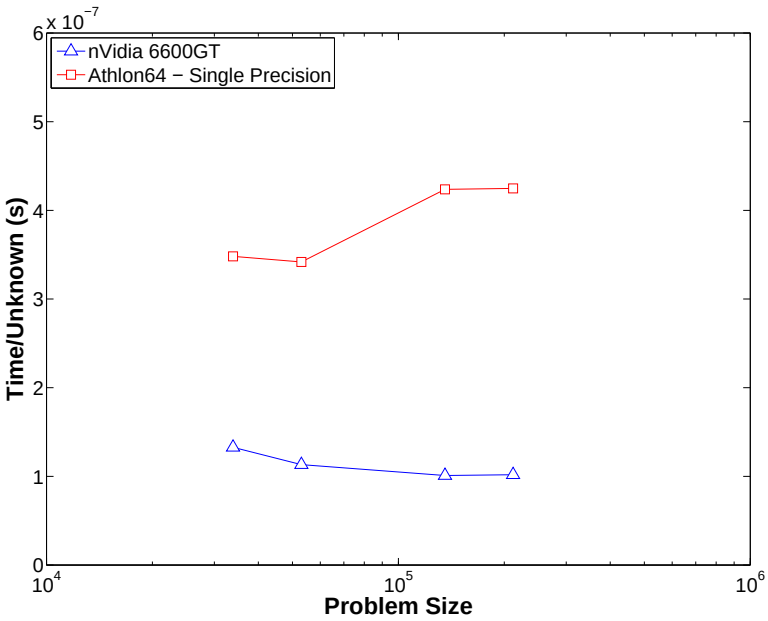


Figure 7.3. Classical Fractional Step - Performance comparison of the Conjugate Gradient solver for the Momentum equation. Problem size denotes the number of faces in the mesh.

by a factor of roughly 3x, with a lower computational cost as the mesh resolution increases.

The performance results of the Exact Fractional Step approach also exhibit a similar trend, with an improvement of roughly 3x.

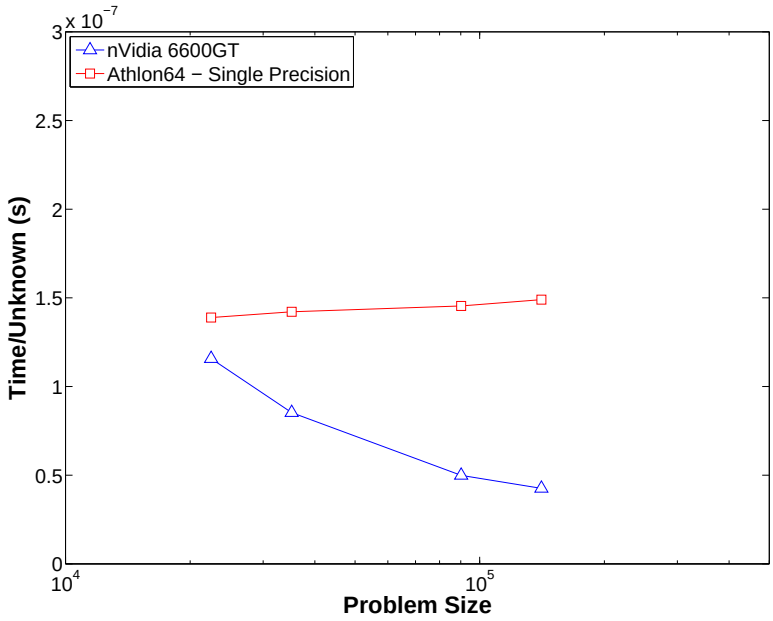


Figure 7.4. Classical Fractional Step - Performance comparison of the Conjugate Gradient solver for the Pressure equation. Problem size denotes the number of cells in the mesh.

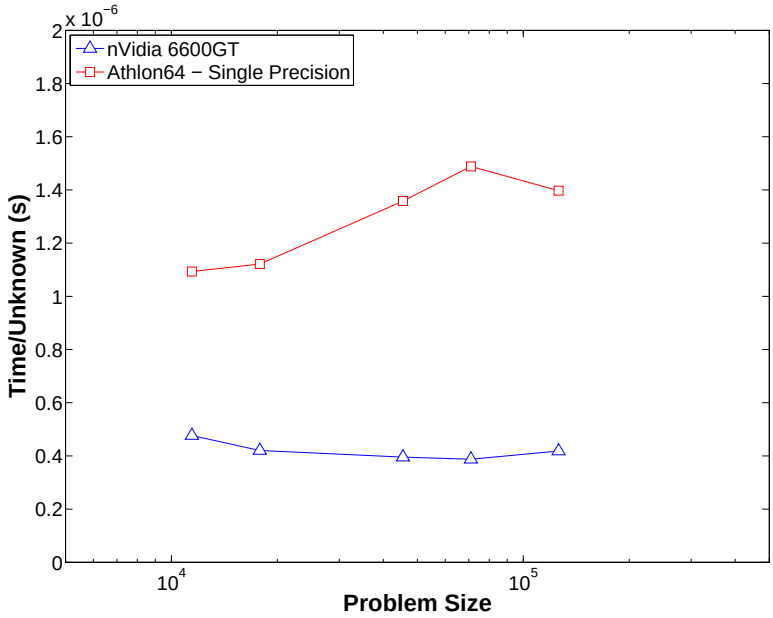


Figure 7.5. Exact Fractional Step - Performance comparison of the Conjugate Gradient solver for the Streamfunction equation. Problem size denotes the number of edges in the mesh.

CHAPTER 8

FUTURE WORK

The primary objective behind incorporating stream-processing hardware for scientific computing is to solve larger problems in a shorter time-frame. While this need is partially fulfilled with the use of Beowulf clusters, the cache-based processors in these systems are not very effective. An attractive approach would be the use of graphics hardware in a cluster-like configuration, which would provide the advantage of efficiency along with scalability as well.

Incorporating parallel-processing capabilities into the object-oriented C++ code would involve the development of parallel operators using an MPI implementation. The nagging hurdle is the task of hiding the communication latency between the MPI processes. This is dictated primarily by two factors - the communication bandwidth between processors; and the amount of memory available on each processor.

The first of these can be alleviated to a certain extent using a relatively new addition to commodity graphics processors, known as SLI (Scalable Link Interface). This feature was introduced to allow rendering to take place in parallel, using two or more GPUs installed on a single motherboard. Parallel performance is not expected from the SLI capability itself (which transfers only video data across GPUs), but rather from the fact that SLI-capable motherboards can handle multiple graphics cards. CPU-GPU communication is also improved by means of the PCI-Express x16 protocol, which provides a peak bandwidth of about 4GB/s per direction. Communication across motherboard elements offers much better bandwidth capabilities as opposed to ethernet interconnects, so effective latency hiding is expected. The memory capac-

ities on commodity GPUs also seems to be rising (current processors have almost a gigabyte of video-memory on board), so that bottleneck is now no longer significant.

Put together, this configuration should provide efficient scalability for large-scale fluid simulations.

APPENDIX A

MEMORY HANDLING ON THE GPU

A.1 Creating Arrays

Although the actual computations are done on the GPU, the task of allocating GPU arrays and initializing them with data has to be done on the CPU, by means of OpenGL library calls. This is exemplified by the following code:

```
// Dynamically allocate temporary arrays on the CPU
float* Y = (float*)malloc(N*sizeof(float));
float* X = (float*)malloc(N*sizeof(float));
float a;
```

Prior to the actual allocation routine, a unique reference must be created for each array. This is done using the OpenGL `glGenTextures` function, which also ensures that generated references are not currently in use. The generated reference is then bound to an appropriate memory layout using the `glBindTexture` function:

```
// Generate a new ID for the GPU array
// [integer 1 signifies a request for only one object]
int FieldID;
glGenTextures (1, &FieldID);

// Bind the generated reference to a Rectangular layout
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,FieldID);
```

Arrays can be represented either in a rectangular layout with arbitrary dimensions, or in a square layout with dimensions that are strictly power-of-two. This choice can be made by specifying either `GL_TEXTURE_2D` or `GL_TEXTURE_RECTANGLE_ARB` for the memory layout, respectively. Certain properties are also required to be set when an array is instantiated. For details on these statements, refer [7, 22]. But the following settings work in general for computational purposes.

```
// Turn off filtering and set proper wrap mode for the active texture...
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,GL_TEXTURE_WRAP_S,GL_CLAMP);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,GL_TEXTURE_WRAP_T,GL_CLAMP);
```

Now that a reference has been created, actual allocation of the array in graphics memory is done using the OpenGL `glTexImage2D` function. This statement explicitly requires the dimensions of the array to be specified. This can be obtained by the algorithm described in A.4. For the moment, assuming that the array size is given by FieldWidth and FieldHeight:

```
// Allocate the memory on the GPU
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_FLOAT_R32_NV,
             FieldWidth, FieldHeight, 0, GL_RED, GL_FLOAT, data_on_cpu);
```

The function specified above can be considered to be the GPU equivalent of a malloc statement in C (or a new statement in C++), which is used for dynamic allocation.

- The first argument to the function is the memory layout of the array to be created. Recall that the generated reference was bound to this layout in the `glBindTexture` statement; so, all subsequent OpenGL calls will applied to the array that is currently bound to `GL_TEXTURE_RECTANGLE_ARB` unless it is replaced by another `glBindTexture` call.

- The second argument defines a mipmap level which is irrelevant in this context.

- The third argument specifies the internal data format of the array, which is specified by the enumerant `GL_FLOAT_R32_NV`. This enumerant is defined in the extension wrangler, and is used to specify a one-component 32-bit data format for each element of the array. Other options include `GL_FLOAT_RGB32_NV` and `GL_FLOAT_RGBA32_NV` for three and four-component formats respectively.

- The fourth and fifth arguments specify the width and height of the array.

- The sixth argument specifies whether the array contains border elements, which is again irrelevant in a computational context, and is specified as null.

- The seventh argument specifies the number of components that will be used in the array. In this case, since only one component is used, `GL_RED` is specified. Other options include `GL_RGB` and `GL_RGBA` for three and four-component formats respectively.

- The eighth argument specifies the format of data which resides in *CPU* memory. This is always expected to be `GL_FLOAT` for computational purposes.

- The last argument is a pointer to the array of data residing in CPU memory which is meant to be loaded on to the GPU. Specifying a value of NULL for this argument merely allocates the data on GPU memory, but does not initialize it with any data.

A.2 Transferring data from main memory to GPU arrays

The `glTexImage2D` function serves as a means of allocating memory on the GPU as well as loading the array with initial data, as specified in the previous section. Once data is transferred on to the GPU memory, it can be freely modified in main memory without affecting anything on the GPU array. Care must be taken to ensure that the array is bound to the required shape before calling the `glTexImage2D` routine. For instance:

```
// Bind the array reference to a Rectangular layout
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,FieldID);
// Allocate the memory on the GPU and load it with initial data
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_FLOAT_RGBA32_NV,
             FieldWidth, FieldHeight, 0, GL_RGBA, GL_FLOAT, data_on_cpu);
```

A.3 Transferring data from GPU arrays to main memory

The `glGetTexImage` function is used to retrieve data from GPU arrays into main memory. This statement takes arguments that are similar to those used in `glTexImage2D`, but with data-transfer in the opposite direction. The texture must first be bound prior to making the call to `glGetTexImage`.

```
// Bind the array reference to a Rectangular layout
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,FieldID);
// Transfer data to main memory
glGetTexImage(GL_TEXTURE_RECTANGLE_ARB, 0, GL_RED, GL_FLOAT, ptr_on_cpu);
```

Since this routine doesn't perform any memory-bounds checks, it is important to ensure that the array on the CPU be allocated with sufficient memory to hold the contents of the entire GPU array, to prevent memory over-stepping and painful segmentation faults.

Another option, is the `glReadPixels` function. This function actually reads information from the framebuffer into main memory. However, since textures can be attached to the Framebuffer Object, this can be used as an alterative for texture readback. This function requires the origin of the read-location and the dimensions of the section as additional arguments and so, it can be used to read sections of the array into main memory. Prior to the `glReadPixels` call, the texture should be attached to the Framebuffer Object first:

```
// Attach texture to the framebuffer
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
                          FieldShape, FieldID, 0);
// Perform the readback to main memory
glReadPixels(ReadOrigin_x, ReadOrigin_y, SectionWidth, SectionHeight,
             GL_RGB, GL_FLOAT, ptr_on_cpu);
```

Data-transfers from GPU arrays are quite expensive, as they choke the system bus and require explicit synchronization between the CPU and the GPU; and should therefore be used sparingly.

A.4 Algorithm: Mapping arrays on memory to GPU arrays

inputs : Size of the CPU array, N; Minimum dimension, R_{min}
outputs: Array dimensions, $FieldHeight$ and $FieldWidth$; and block size, $FieldBlock$

$factor = 0.75$
$B_{dim} = 1$
$S_{min} = N$

while $S_{min} \geq factor \cdot R_{min} \cdot R_{min}$ **do**
  $B_{dim} = B_{dim} \times 2$
  $S_{min} = 1 + floor((N-1)/(B_{dim} \cdot B_{dim}))$
end

$y_{max} = 1 + floor(\sqrt{S_{min}})$
$y_{min} = 2$
$y = 0$
$x = 0$

while $y < y_{min} \parallel x > R_{min}$ **do**
  $y = y_{max}$
  while $mod(S_{min}, y) > 0$ & $y \geq y_{min}$ **do**
    $y = y - 1$
  end
  $x = S_{min}/y$
  $S_{min} = S_{min} + 1$
end

$FieldHeight = B_{dim} \times y$
$FieldWidth = B_{dim} \times x$
$FieldBlock = B_{dim} \times B_{dim}$

APPENDIX B

SOURCE CODE FOR REDUCTIONS

B.1 Sum reduction of rectangular arrays

This routine requires the reference ID of the array and the parameters generated by the algorithm in A.4 as input. It also also assumes that the necessary vertex / fragment shaders described in Chapter 4 have been compiled using the `CompileKernel` routine in Chapter 2 and stored in the variable `sum_program`; and that two temporary arrays `FieldID0` and `FieldID1` have been allocated to at least half the dimensions of the array being reduced.

```
float sum(int FieldID)
{
   int wd = FieldWidth;
   int ht = FieldHeight;
   int bl = FieldBlock;
   int src_handle;

   // Activate the Sum program
   glUseProgramObjectARB(sum_program);
   // Set the dataflow interface
   src_handle = getInput("Source");
   // Set the viewport
   setGPUview(wd,ht);
   // Assign initial fields
   InputID  = FieldID;
   OutputID = FieldID0;

   //  Block reduction in 2x2's
   while (bl > 1) {
      // Reduce block dimension...
      bl = bl/4;
      // Bind the output
      setOutput(OutputID);
```

```
      // Bind the input
      setInput(src_handle,InputID);
      // Run the GPU program
      RunProg(wd/2,ht/2,wd,ht);
      // Swap Input/Output arrays...
      if (OutputID == FieldID0)
      {
         OutputID  = FieldID1;
         InputID  = FieldID0;
      } else {
         OutputID  = FieldID0;
         InputID  = FieldID1;
      }
      // Reduce each dimension by half...
      wd = wd/2; ht = ht/2;
   }

   // Read-back approach
   // Bind the texture to target
   glBindTexture(GL_TEXTURE_RECTANGLE_ARB,OutputID);

   // Download from the target
   float *tmp = new float[wd*ht];
   glReadPixels(0,0,wd,ht,GL_RED,GL_FLOAT,tmp);
   float cpu_sum = 0.0;
   for (int i = 0; i < wd*ht; i++)
      cpu_sum += tmp[i];
   delete [] tmp;

   // Return the value
   return cpu_sum;
}
```

BIBLIOGRAPHY

[1] Bolz, J., Farmer, I., Grinspun, E., and Schrooder, P. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), ACM Press, pp. 917–924.

[2] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph. 23*, 3 (2004), 777–786.

[3] Chang, W., Giraldo, F., and Perot, J.B. Analysis of an exact fractional step method. *J. Comput. Phys. 180* (2002), 183–199.

[4] Chorin, A. J. Numerical solutions of the Navier-Stokes equations. *Mathematics of Computation 22* (1968), 745.

[5] Fernandez, A. R. Lighthouse3D-GLSL Tutorials. Tech. rep. `http://www.lighthouse3d.com/opengl/glsl/`.

[6] Galoppo, N., Govindaraju, N. K., Henson, M., and Manocha, D. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2005), IEEE Computer Society, p. 3.

[7] Göddeke, D. GPGPU–Basic Math Tutorial. Tech. rep., FB Mathematik, Universität Dortmund, Nov. 2005. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300, `http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu`.

[8] Godekke, D., Strzodka, R., and Turek, S. Accelerating double precision FEM simulations with GPUs. In *ASIM '05: 18th Symposium on Simulation Techniques* (2005).

[9] Goodnight, N., Woolley, C., Lewin, G., Luebke, D., and Humphreys, G. A multigrid solver for boundary value problems using programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 102–111.

[10] Gummaraju, J., and Rosenblum, M. Stream programming on general-purpose processors. *micro 0* (2005), 343–354.

[11] Harlow, F.H., and Welch, J.E. Numerical calculation of time dependent viscous incompressible flow of fluid with free surface. *Physics of fluids 8* (1965), 2182.

[12] Harris, M. Mapping computational concepts to GPUs. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), ACM Press, p. 50.

[13] Harris, M. J., Coombe, G., Scheuermann, T., and Lastra, A. Physically-based visual simulation on graphics hardware. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 109–118.

[14] Kapasi, U. J., Rixner, S., Dally, W. J., Khailany, B., Ahn, J. H., Mattson, P., and Owens, J. D. Programmable stream processors. *Computer 36*, 8 (2003), 54–62.

[15] Kipfer, P., Segal, M., and Westermann, R. UberFlow: a GPU-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), ACM Press, pp. 115–122.

[16] Kolb, A., Latta, L., and Rezk-Salama, C. Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), ACM Press, pp. 123–131.

[17] Krakiwsky, S. E., Turner, L. E., and Okoniewski, M. M. Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU). *Microwave Symposium Digest 2* (2004), 1033–1036.

[18] Kruger, J., and Westermann, R. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph. 22*, 3 (2003), 908–916.

[19] Larsen, E. S., and McAllister, D. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)* (New York, NY, USA, 2001), ACM Press, pp. 55–55.

[20] Lindholm, E., Kilgard, M. J., and Moreton, H. A user-programmable vertex engine. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM Press, pp. 149–158.

[21] McCool, M., Toit, S. Du, Popa, T., Chan, B., and Moule, K. Shader algebra. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), ACM Press, pp. 787–795.

[22] Molofee, J. NeHe–OpenGL Tutorials. Tech. rep., 1997. `http://nehe.gamedev.net`.

[23] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. E., and Purcell, T. J. A survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113.

[24] Perot, J. B. An analysis of the fractional step method. *J. Comput. Phys. 108*, 1 (1993), 51–58.

[25] Perot, J. B. Comments on the fractional step method. *J. Comput. Phys. 121* (1995), 190.

[26] Perot, J. B., Vidovic, D., and Wesseling, P. Mimetic Reconstruction of Vectors. *Compatible Spatial Discretizations, IMA Volumes in Mathematics and its Applications 142* (2006), 173–188.

[27] Rost, R. J. *OpenGL Shading Language*. Addison Wesley, 2004.

[28] Scheidegger, C., Comba, J., and Cunha, R. Practical CFD simulations on the GPU using SMAC. *Computer Graphics Forum 24*, 4 (2005), 715–728.

[29] Shewchuk, J. R. An introduction to the conjugate gradient method without the agonizing pain. Tech. rep., Pittsburgh, PA, USA, 1994.

[30] Stam, J. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1999), ACM Press/Addison-Wesley Publishing Co., pp. 121–128.

[31] Subramanian, V. *Discrete Calculus Methods and their Implementation*. PhD thesis, University of Massachusetts, Amherst, 2007.

[32] Tarditi, D., Puri, S., and Oglesby, J. Accelerator: using data parallelism to program GPUs for general-purpose uses. *SIGPLAN Not. 41*, 11 (2006), 325–335.

[33] Témam, R. Sur l'approximation de la solution des équations de Navier-Stokes par la méthode des pas fractionnaires (II). *Archive for Rational Mechanics and Analysis 33* (Jan. 1969), 377–385.

[34] Wei, X., Li, W., Mueller, K., and Kaufman, A. E. The lattice-Boltzmann method for simulating gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics 10* (2004), 164–176.