Use of Graphics Processing Units for Sparse Matrix-Vector Products in Statistical Machine Learning Applications

Ahmed H. El Zein

April 2009

A thesis submitted for the degree of Master of Philosophy
at the Australian National University

بِسمِ اللهِ الرَّحمٰنِ الرَّحِيم

اللَّهُمَّ اجعَل هَذَا العَمَل لوَجهِكَ الكَرِيم

اللَّهُمَّ تَقَبَّل مِني إِنَكَ أَنتَ السَمِيعُ العَلِيم

Declaration

The work in this thesis is my own except where otherwise stated.

Ahmed H. El Zein

Parts of the work presented in this thesis were published at the International Conference on Computational Science (ICCS) in 2008 under the title: "Performance evaluation of the NVIDIA GeForce 8800 GTX GPU for machine learning." and presented as a poster at Super Computing (SC) in 2007 with the title "The sony playstation 3 and the NVIDIA 8800 GPU: Performance and programmability evaluation for machine learning."

Acknowledgements

I am indebted to many people without whom I would have not completed this project. At the top of the list is my supervisor, Alistair Rendell. I would not have been able to achieve anything without his mentoring, support and obvious care for my well-being. For all this, I am deeply grateful.

I would like to thank Alex Smola for his help and support. My thanks also go out to Muhammad Atif and Danny Robson for making coffee breaks fun, to Jin Wong for putting up with me in a single office for two years, to Warren Armstrong and Josh Milthorpe for proof reading my thesis and to Peter Strazdins, Eric McCreath, Pete Janes, Jie Cai, Arrin Daley and Joseph Anthony who make up the rest of the HPC research group for being my friends.

To Bob Edwards, Deanne Drummond, Suzanne Van Haeften and Julie Arnold, thank you for making my journey as smooth as possible.

I would also like to thank my wife, Eman who put up with long nights at Uni, my daughter, Mariam who went to sleep many nights calling out "Daddy kiss!" without getting her good-night kiss, my sister who kept my wife company while I was writing this, my mother and father who made me what I am, and the rest of my family, thousands of miles away whose support I depend on.

Thank you all.

Abstract

Graphics Processing Units (GPUs) offer orders of magnitude more floating-point performance than conventional processors. Traditionally, however, accessing this performance for general purpose programming has required the user to cast their problem into a graphical framework of nodes and vertices. In 2007 this situation changed dramatically when NVIDIA released its CUDA programming model for GPUs.

The objective of this thesis is to assess the viability of using an NVIDIA GeForce 8800 GTX GPU and the CUDA programming model for statistical machine learning (SML) applications.

At the heart of the SML method is the iterative solution of a set of equations. Each iteration involves two matrix-vector products, where the matrix is generally sparse and does not change between iterations. Key issues considered in this work are what fraction of the SML application should be migrated to the GPU, the cost of moving data to and from the GPU, the efficient implementation of Sparse Matrix-Vector products (SpMV) on the GPU, and the relative merits of using sparse versus dense matrix routines.

In implementing the SpMV routine on the GPU a range of different CUDA options were considered, including the type of memory used to store different data quantities, the use of float, float2 and float4 data types, the number of threads per block, the use of coalesced memory reads etc. Following a preliminary performance characterisation of the 8800 GTX, 335 different SpMV implementations were constructed and their performance tested using 735 matrices from the Florida sparse matrix collection. From this a small number of best performing implementations were identified and an attempt made to create a blackbox implementation that would correctly select the optimal implementation for a given sparse matrix type.

The blackbox SpMV routine along with dense matrix counterparts were then integrated with the SML application. The times to complete a variety of problems

were compared when using the CPU only or CPU and GPU, and a detailed breakdown of the various parts of the computation given.

Contents

Chapter 1

Introduction

For the past few years, the demand for more realistic games in an ever growing market has led GPU manufacturers to produce high end graphics cards that are able to render complex scenes at very fast frame rates. As graphics problems are highly parallel in nature, GPUs have been designed as massively parallel architectures. Furthermore, to deal with the developments in graphics programming and the increasingly complex processing, GPUs have gradually made a transition from fixed-function pipeline devices that are only able to perform fixed operations, to general purpose processors with some special graphics oriented units. The large commodity GPU market and ruthless competition have realised relatively low cost devices with very high Floating-Point Operations per Second (FLOP/s) ratings.

GPUs use Single Instruction Multiple Data (SIMD) architectures to minimise control logic and power requirements, providing a greater number of FLOP/s per watt than CPUs. CPUs attempt to mask memory latencies with large amounts of cache comprising a significant portion of the CPU die space [25, 30]. In contrast, GPUs have minimal amounts of cache, relying on the ability to execute thousands of threads in parallel, masking memory latencies by switching between the large number of threads [42]. The die space saved by these methods can be invested in Arithmetic Logic Units (ALUs). The overall result is a less versatile general purpose processor that has a much lower cost per FLOP (both in terms of price and power requirements) and a much higher peak FLOP/s rating. For the past few years, the gap between CPUs and GPUs has been growing in this regard [46].

Throughout the evolution of GPUs, the difficulty of programming these devices has deterred wide adoption by the scientific community. Until recently, GPUs only supported domain specific, graphics oriented languages. Describing

scientific problems in terms of graphics primitives is a daunting task that requires a strong understanding of graphics programming. Higher level but still domain specific languages were designed to make programming GPUs easier [35, 27, 47]. Yet, while General Purpose computation on GPUs (GPGPU) became easier, the hardware itself hampered GPGPU by not supporting many of the features used in general purpose programming such as scatter operations, thread synchronisation and shared memory [13].

ATI and NVIDIA (two of the largest GPU manufacturers) eventually released GPUs with completely programmable processors [45]. The new unified architectures also provided the missing hardware features (scatter operations, shared memory, synchronisation) that were required for GPGPU. To facilitate GPGPU programming for these GPUs, ATI AND NVIDIA released programming toolkits for them [42, 4]. However while ATI released an assembly like language, NVIDIA released a C/C++ syntax compatible language named CUDA, which allowed the programmer to mix CPU and GPU code in the same module. NVIDIA and to a lesser extent ATI, finally presented the scientific community with easy to program, massively parallel multi-threaded devices with many of the hardware capabilities needed to efficiently execute scientific applications.

In 2007, GPU hardware lacked double-precision floating-point support which held back its adoption by many within the scientific community [23]. Current GPUs from both manufacturers offer double-precision but with lower FLOP/s the for single precision. ECC memory has not been announced for any upcoming products by either of the manufacturers.

For many scientific applications, the lack of ECC memory and double-precision are not important. The application can be robust enough to deal with bit flips and not all applications require double-precision. For many other applications, numerical methods can be used to produce high precision results [44] and still perform better on the GPU than the CPU [23].

Machine learning (ML) is a branch of artificial intelligence, that attempts to develop algorithms that will allow applications to modify themselves based on some analysis of data. The term "statistical" in Statistical Machine Learning (SML) reflects the emphasis on statistical analysis and methodology, which is a widely used approach in modern machine learning. Applications for SML include natural language processing [3], syntactic pattern recognition, search engines [51], medical diagnosis [2], bioinformatics [16], stock market analysis [26], classifying DNA sequences [58], speech and handwriting recognition [5], object recognition in computer vision [32, 34], game playing and robot locomotion [29].

Internet searching is a high profile SML application. Companies like Google, Yahoo and Microsoft all spend enormous amounts of money on server hardware and power costs of running the servers. Solutions that lower the cost of computations either by increasing the throughput of systems with the same power requirements or decreasing the power requirements of a system that maintains the same throughput are of huge benefit in terms of running costs. GPUs offer potential solutions in this area.

The Bundle Methods for Regularised Risk Minimisation (BMRM) application is an open source, modular and scalable convex solver for many machine learning problems [54]. A portion of these problems are not affected by the use of single precision. This SML application's computation is dominated by matrix-vector products which depending on the nature of the datasets can be either dense and sparse. Dense matrix-vector products are easily parallelised and have been shown to perform well on GPUs [21, 33, 31]. Sparse matrix-vector products on GPUs have not been as successful as their dense counterparts, but the new unified architectures are potentially flexible enough to provide performance improvements for Sparse Matrix-Vector products (SpMV) on the GPU. SpMV on GPUs will be discussed in more detail in the next chapter.

CUDA provides a Basic Linear Algebra Subprograms (BLAS) library (CUBLAS [40]) that is utilised to evaluate matrix-vector products on the GPU. NVIDIA did not provide SpMV routines at the start of this work and so SpMV routines were developed and evaluated on the GPU as part of this work. This work is based on the GeForce 8800 GTX as it was the best performing product from NVIDIA at the start of this work.

Chapter 2 presents background material on GPGPU, CUDA, hardware specification of the GeForce 8800 GTX GPU, statistical machine learning and sparse matrices. The bandwidth between the GPU and host along with the internal GPU memory bandwidth were measured and are presented in chapter 3. Chapter 3 also presents a detailed investigation of dense matrix-vector products on the GPU. Chapter 4 outlines the design process of the SpMV implementations and provides detailed analysis of various SpMV implementations. An attempt to select the best implementation based on matrix characteristics is also investigated. Chapter 5 presents the results from integrating GPU code with the BMRM application and finally chapter 6 presents conclusions and future directions.

Chapter 2

Background

This section presents background information on General-Purpose computation on GPUs (GPGPU), details of the GeForce 8800 GTX GPU and the CUDA programming environment used in this work. This section also discusses sparse matrices and provides more details on the SML algorithm used in the BMRM application.

2.1 Programming GPUs

GPUs were first introduced as non-programmable, fixed-function pipelines with specific functions. The GPU starts with a scene defined by a list of coordinates call vertices. Each vertex undergoes a series of transformations with the end being a "pixel" that will then be displayed on the screen. Vertices that will not end up on the screen (they could be hidden by other vertices for example) are ignored. Each pixel is then given a color based on attributes such as texture and lighting and the the end result is a frame to be displayed on the screen.

Later, products were released [46] that contained embedded programmable components. The series of transformations applied to the vertices as well as the pixel colouring could be programmed in the form an assembly language. These programs are referred to as vertex shaders and pixel or fragment shaders.

As the instructions sets of these embedded components grew, so did the complexity and length of the shader programs. It soon became unrealistic to write shaders in the assembly level languages provided by the vendors. This gave rise to higher level languages that varied in their specification yet all attempted to make shader programming easier [45].

Higher level languages made GPGPU a lot easier, but they were still de-

signed from a graphics perspective so were not easily accessible to scientists with a non-graphics background. Indeed GPGPU with these languages required a considerable amount of graphics knowledge in order to map a general problem into a graphics problem solvable with graphics APIs. To simplify the process of writing general purpose code for GPUs, GPGPU languages came into existence. The following subsections consider some of the languages that are available for GPGPU.

A much wider survey of available options for programming GPUs has been published by Owens et al. [46] in 2007. This survey summarises and analyses the latest research in GPGPU.

2.1.1 Shader Languages

This class of languages facilitates the writing of shaders but with added portability and programmer productivity. A separate shader program must be written for each of the vertex and fragment processors. These languages also differ in many aspects that will be illustrated in the relevant sections.

Cg

Cg (or C for graphics) [35] differs from the other languages in this section in that it has a clear separation between code meant to run on the CPU and code intended for the GPU. Cg attempts to find a balance between providing the whole feature set of C and providing a maximum shader feature set. For example it omits many high level shader-specific facilities yet provides the same operators as C (but ones that accept and return vectors as well as scalars). Cg is compiled into an assembly level language for either OpenGL [53] or Direct3D [11].

OpenGL Shading Language

OpenGL, an open standards group, released the OpenGL Shading Language (GLSL) [27] also know as GLslang as part of the OpenGL 2.0 Specification. This language enables direct compilation of C-like programs to graphics hardware machine code. Unlike Cg and HLSL, there is no assembly level language involved. The compiler is embedded into the graphics driver. OpenGL is supported on a wide spectrum of operating systems and graphics cards and is therefore a more open and compatible language than either Cg or HLSL [27].

The Direct 3D High-Level Shading Language

The Direct 3D High-Level Shading Language (HLSL) [47] was developed in close partnership with NVIDIA and is similar in many aspects to Cg. As part of Direct3DX, it only compiles to Direct3D and is only supported on Microsoft operating systems [47].

2.1.2 Languages for General Purpose Computing

These languages differ from the previous languages in that they are more suited to general purpose computing. However they offer the programmer less control over the graphics pipeline.

Shader Metaprogramming Language

Sh is both a shading language and a runtime API to use the Sh shaders [37]. It is embedded in C++ as a domain specific language and defines special tuple and matrix types that are used extensively in shader code. Sh can be used to write vertex or fragment shaders for a GPU in C/C++. The code is then compiled at runtime to the target device [37]. Sh can also treat shaders as first class objects and by combining connection and combination features in Sh allow the creating of complex stream programs for GPGPU computing [36]. In Sh there is no clear distinction between GPU and CPU code, nor is there any explicit mechanism to move data to and from the GPU memory.

Brook and BrookGPU

Brook is an extension of standard ANSI C and is designed to incorporate the ideas of data parallel computing and arithmetic intensity into a familiar and efficient language. BrookGPU, a GPU targeted version of Brook was developed by Buck et al. [15] based on the idea that a GPU can be viewed as a stream processor.

Brook differs from all previous languages in that separate shaders for the vertex and fragment processor are not needed. Instead a kernel is written that operates on every element in a stream. BrookGPU provides a level of abstraction that eliminates the need to view computations on the GPU in terms of graphics operations.BrookGPU also virtualises two aspects which are critical to stream computing, the number of kernel outputs and stream dimensions and size. If the shader program requires more outputs than what the hardware supports for example, BrookGPU will compile the program into several smaller programs

that run in turn on the GPU to accommodate the extra outputs needed. This virtualisation can also be used to provide complex data types not supported by the hardware [15].

C for CUDA

CUDA is an architecture and programming model for parallel computing developed by NVIDIA. NVIDIA provides a C like API called C for CUDA. The CUDA architecture, programming model and development environment are expanded upon in section 2.2.

2.2 Target Hardware, Language, Execution and Programming Model

The GeForce 8 Series GPUs were the first NVIDIA GPUs to be based on the new "unified architecture". Figure 2.1 illustrates the architecture of the GeForce 8800 GTX used in this work. At the heart of the device lies the Streaming Processor Array (SPA) consisting of eight Texture Processor Cluster (TPC) units. Each TPC contains two Streaming Multiprocessor (SM) units and a texture unit. The SM in turn consists of eight Stream Processors (SP), a special function unit, a 8192 wide register file and 16KB of shared memory. When running CUDA applications each SP (clocked at a default of 1.35 GHz) is able to issue one multiply-add (MAD) instruction per cycle. This gives each SM a peak performance of $1.35^9 \times 8 \times 2 \div 2^{30} =$ 20.1 GFLOP/s, and the GeForce 8800 GTX with 16 SMs an aggregate performance of 321.6 GFLOP/s. The GeForce 8800 GTX has 768 MB of global, frame buffer memory that can be read from or written to by the host CPU as well as the GPU. The 768MB of GDDR3 memory is clocked at 900MHz and is accessed via a 384-bit (48 byte) wide interface in 128-bit wide words. This gives a theoretical peak bandwidth of $(48\ bytes \times 900 \times 10^6\ MHz \times 2\ (DDR)) \div 2^{30} = 80\ GB/s$.

NVIDIA provides CUDA, a software programming model and a programming environment that enables the creation of parallel applications for these new unified GPUs. The CUDA runtime library exposes parts of the GPU and hides others, with the overall result of presenting a massively parallel co-processor to the programmer. The CUDA toolkit provides the programmer with methods to manage memory transfers between the GPU and the host, synchronisation barriers and methods to control the invocation of code on the GPU. Provided with CUDA are Basic Linear Algebra Subprograms (BLAS) and Fast Fourier

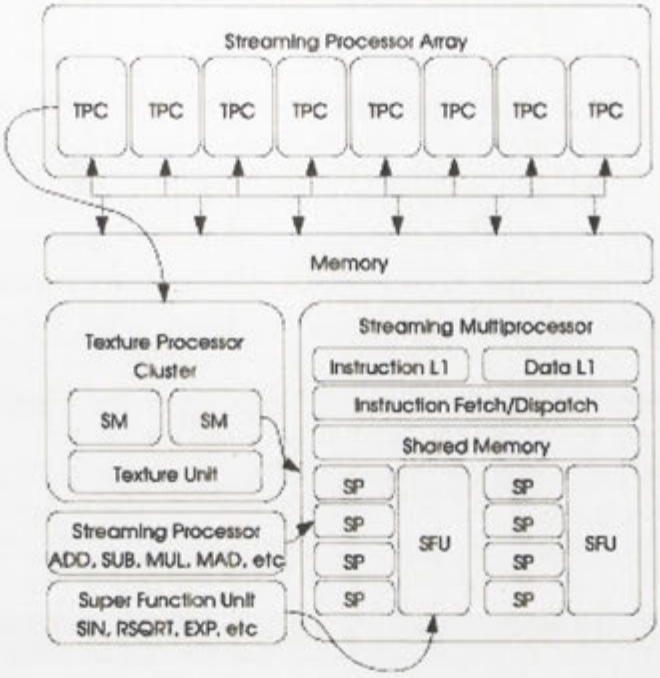


Figure 2.1: GeForce 8800 GTX architecture

Transform (FFT) implementations [40, 41] . The programming model, memory hierarchy and execution model defined by CUDA are expanded upon in the next sections.

2.2.1 The CUDA Programming Model

Writing applications for CUDA enabled GPUs involves copying data from the host to the GPU memory, invoking the GPU code (in the form of one or more kernels) and copying the results of the computation back to the host. CUDA provides methods to allocate memory on the GPU as well as methods to copy data between GPU memory and host memory. CUDA also provides functions to allocate page-locked memory. Bandwidth to and from page-locked memory is faster, as DMA transfers must be done from page-locked memory and having the data in page-locked memory saves the driver from having to copy it to page-locked memory before initiating the DMA transfer.

CUDA also provides the mechanisms to execute a kernel on the device. A kernel is a function call that is executed by all the threads launched on the GPU. Thousand of threads can be launched on the device. Threads are clustered into

blocks of between 1 and 512 threads. The GeForce 8800 GTX can accommodate up to 64K × 64K blocks in what CUDA labels a grid.

Threads are automatically assigned an index so that different threads can fetch data from different memory locations. Each thread is then able to retrieve the dimensions of the grid and block as well as its thread index within its block and its block index within the grid. As an aid to the programmer, CUDA offers 1, 2 or 3 dimensional indexing of the blocks if it would better suit the data. Only 1 or 2 dimensional indexing of the grid is supported.

Listing 2.1: Example kernel and host invocation methods with use of 2D indexing

```
__global__ void matAdd( float A[N][N], float B[N][N], float C[N][N] )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    /* set the blocks to contain 16x16 threads */
    dim3 dimBlock(16, 16);
    /* total number of blocks depends N */
    dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x, (N + dimBlock.y - 1) / dimBlock.y);
    /* Kernel invocation */
    matAdd <<< dimGrid, dimBlock >>>(A, B, C);
    ...
}
```

Listing 2.1 shows an example of a kernel using 2 dimensional indexing along with the code to invoke the kernel. Each block is executed on a single Streaming Multiprocessor (SM). Within the SM each thread executes on an SP. The large register file allows the SM to create more threads than available SPs. The number of threads that can be resident on the SM depends on the threads resource usage in terms of registers and shared memory. In fact an SM can accommodate multiple blocks if there exists enough resources for all the threads in the multiple blocks.

2.2.2 GeForce 8800 GTX Memory Hierarchy

The GeForce 8800 GTX offers many different memory types, each with its advantages and disadvantages. These memories are summarised in table 2.1.

Each SM has 8192 on-chip registers that are shared between the multiple threads running on the SM. If the number of registers needed is not enough to support all the threads within a block, variables are stored in local memory which despite its name is off-chip. Local memory is local in scope only. Variables that spill out into local memory are physically stored in global memory (discussed in the next paragraph). In addition to the registers, each SM has 16KB of on-

Table 2.1: Different Memory types available on the GeForce 8800 GTX GPU.

Memory	Location	Cached	Host Access	GPU Access	Scope	Lifetime
Register	On-chip	N/A	N/A	R/W	One thread	Thread
Local	Off-chip	No	N/A	R/W	One thread	Thread
Shared	On-chip	N/A	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	R/W	All threads + host	Application
Constant	Off-chip	Yes	R/W	R	All threads + host	Application
Texture	Off-chip	Yes	R/W	R	All threads + host	Application

chip shared memory divided into 16 banks such that successive 32-bit words are assigned to successive banks. Bank conflicts will incur performance hits. In the absence of conflicts shared memory reads are as fast as register reads [42]. Registers and local and shared memory are accessible from the GPU only.

Global memory is the main memory type of the GPU. It is the largest memory space and provides read and write access from both the host via DMA and the SMs. Global memory is not cached, relying instead on the thousands of threads running on the GPU to mask latency. Global memory has a latency of 400 - 600 cycles while a floating-point arithmetic instruction (ADD, MUL, SUB) has an issue latency of 4 cycles and a throughput of 8 operations per cycle [42]. The CUDA programming guide [39] states that in order to achieve optimal memory bandwidth, memory reads must be coalesced. This occurs when all 16 threads read from aligned, consecutive, memory addresses, and the hardware is able to transform the individual memory accesses into a number of 64-byte memory transactions. Coalesced 32-bit reads result in one 64-byte transaction, coalesced 64-bit reads result in a single 128-byte transaction and coalesced 128-bit reads result in two 128-byte transactions. Non-coalesced 32-bit reads are an order of magnitude slower than coalesced 32-bit reads. The coalescing rules described here are for the 8800 GTX GPU architecture. Latest generation GPUs have different (less stringent) rules. More information can be found in the CUDA programming manual, which has an extensive section on memory coalescing [42].

Another available memory type is texture memory. Texture memory is not physically separate from global memory. However CUDA allows global memory to be accessed via the texture units in the SM. This enables the texture cache, a read-only cache shared by all SPs in the SM. The use of the texture cache speeds up reads to memory. On the other hand the texture cache is not kept coherent so

changes to the memory after it has been cached are not reflected in the cache. This restricts the texture memory to a read only memory from the GPUs view-point. Texture cache is optimised for 2D spatial locality so accessing memory references that are closer together will result in better performance. Texture cache locality applies to accesses across threads first (since threads run in parallel), and accesses within the same thread last. With texture references a cache hit will lower the pressure on DRAM but will not lower fetch latency. The cache working set is between 6KB - 8KB per SM.

The GeForce 8800 GTX has 64KB of cached, constant memory. The cache of the constant memory is optimised for many threads accessing the same memory location. If all the threads executing concurrently on the SM (ie. within a half-warp) read the same memory location, the cost after the original fetch is that of a register access.

Figure 2.2 illustrates the memory hierarchy described above. More details of the NVIDIA hardware can be found in [39].

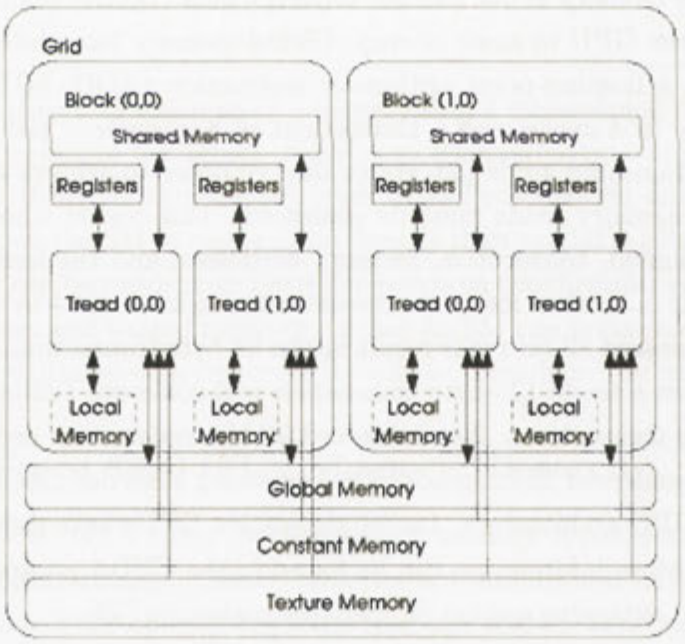


Figure 2.2: CUDA memory model

2.2.3 The CUDA Execution Model

When a kernel is launched on the device, the number of threads, blocks and grids are all specified at launch time. These values may be calculated based on the problem size, passed as parameters at run time or set as constant values. The thread scheduler will schedule blocks to available SMs. Each SM may be allocated multiple blocks depending on the resource usage of the block. As previously indicated, SMs can accommodate more threads than SPs. The SM will then execute all the threads in batches of 32 threads. These 32 threads are called a warp. Each warp is free to diverge from the others with no penalty. The warp is actually executed in two sets of 16 threads each. The SM will continue to execute all threads until they have all terminated at which time any remaining blocks can be scheduled to it. There are no guarantees on the order in which threads are executed or which blocks are scheduled before others. The CUDA programming guide recommends launching a large number of blocks on the device to ensure that memory latencies can be masked.

Threads in the same block can synchronise or shared data via the on-chip shared memory. However, this is not possible between threads of different blocks as only threads within the same block can be guaranteed to be resident on the SM at the same time.

2.3 Statistical Machine Learning

One of the key objectives in Machine Learning (ML) is classification: given some patterns x_i, such as pictures of apples and oranges, and corresponding labels y_i, such as the information whether x_i *is* an apple or an orange, to find some function f which allows us to estimate y from x automatically. Statistical Machine learning (SML) attempts to solve such problems with statistical methods. In this quest, convex optimisation [1] is a key enabling technology for many problems. For instance, Teo et al. [54] proposed a scalable convex solver for such problems. It is an iterative algorithm that involves guessing a solution vector w, using this to evaluate a loss function $l(x, y, w)$ (that calculates a penalty based on the amount of error in the solution) and its derivative $g = \partial_w l(x, y, w)$, and then updating w accordingly. This process is repeated until a desired level of convergence is achieved (see Fig. 2.3). The majority of time is spent evaluating the matrix-

[1] A convex function has a single minima. Convex optimisation attempts to minimise a convex function.

vector products, and the elements of matrix (X) do not change between iterations. This characteristic makes this application a good candidate for computing on the GPU as the cost of moving the matrix to the GPU can be amortised by successive matrix-vector products.

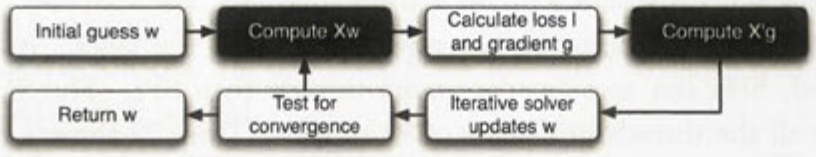


Figure 2.3: Iterative solver algorithm. The black boxes refer to matrix-vector operations which are likely to speed up the application the most if accelerated by a GPU

Many ML datasets are very sparse, as shown in Table 2.2. Exploiting the sparsity decreases the memory footprint of the matrix as well as the number of floating-point operations required for the matrix-vector product. Unfortunately it also introduces random memory access patterns and indirect addressing, which is likely to result in less efficient utilisation of a GPU's hardware.

Table 2.2: Statistics for some typical ML datasets

Domain	Dataset	Rows	Columns	Nonzero Elements	Density
Intrusion Detection	KDDCup99	3,398,431	127	55,503,855	12.86%
Ranking	NetFlix	480,189	17,770	100,480,507	1.17%
Text Categorization	Reuters C11	804,414	47,236	60,795,680	0.16%
Text Categorization	Arxiv astro-ph	62,369	99,757	4,977,395	0.08%

2.4 Sparse Matrices

A matrix is considered to be sparse if many of its coefficients are zero and there exists an advantage to exploiting its zero coefficients. Whether exploiting the zero coefficients would lead to an advantage or not, is dependent on the number of zeros, their patterns and the underlying architecture of the machine used [19]. Many applications involve the use of sparse matrices. Conjugate gradient and

multigrid solvers are often based on sparse matrix-vector products when used in fields such as computational fluid dynamics and mechanical engineering. Sparse matrices are also used in graph theory.

The exploitation of sparsity is achieved by discarding zero elements from the sparse matrix. By doing so the memory requirements and number of arithmetic operations needed for the matrix-vector product are greatly reduced. However, indirect random memory references are introduced as the index of each element must be explicitly stored in the sparse data structure and the memory reads from the vector will no longer be consecutive. Many different formats for storing these matrices have been designed to take advantage of the structure of the sparse matrix or the specificity of the problem from which they arise [50].

The Compressed Sparse Row (CSR) format is the most widely used of these formats [50, 10, 19]. The CSR format (also named Compressed Row Storage) stores non-zero elements in a dense vector `val`. For each value in `val`, its column index from the original matrix is stored in a dense vector of the same size `ind` at the same offset. A third array (`ptr`) carries the offset of the first element in every row. This is illustrated by figure 2.4. A Sparse Matrix-Vector products (SpMV) with a matrix in the CSR format is straightforward as shown by figure. 2.5.

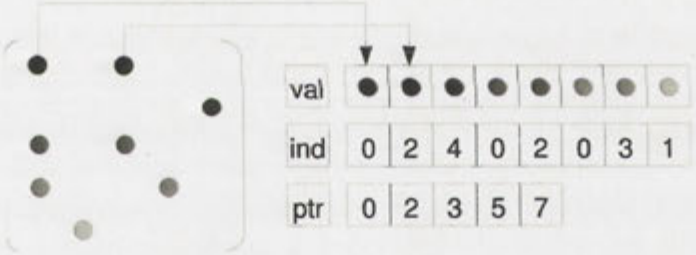


Figure 2.4: CSR format

```
for each row i do
  for l=ptr[i] to ptr[i+1]-1 do
    res[i] = res[i] + val[l] * vec[ind[l]]
```

Figure 2.5: pseudo-code for Sparse Matrix-Vector product (SpMV).

The most extensive evaluation of SpMV on current architectures is the work by Williams et al. [60], which examined the performance of SpMV kernels across

a broad range of multicore architectures including a dual core AMD Opteron 2214, a quad core Intel Clovertown processor as well as a dual socket STI Cell Blade system and the 8 core Sun Niagara2 processor. The work evaluated a range of optimisations for each of the architectures and analysed the performance bottlenecks for all the architectures showing that the memory bandwidth is the common limiting factor in SpMV performance. Only 14 datasets were evaluated and results showed a median performance of 2 GFLOP/s for both the AMD and Intel CPUs, a median performance of 3 GFLOP/s for the Niagara2 processor and 6 GFLOP/s for the Cell Blade system.

2.5 Sparse Matrices on GPUs: Previous Work

SpMV has not been a popular candidate for implementation on GPUs. This is due to the irregular nature of the problem, characterised by indirect addressing (array1[array2[index]]) [56]. Two GPU implementations of SpMV were published at SIGGRAPH'03. Bolz et al. [13] defined sparse matrices in the Modified Sparse Row (MSR) format and were able to perform 120 SpMV operations per second with a matrix containing 37k nonzero elements on a 500MHz GeForce FX GPU. That is equal to roughly 9 MFLOP/s. The second implementation was by Krüger et al. [31] targeting banded matrices and achieving a performance of about 110 MFLOP/s on an ATI Radeon 9800 GPU. In 2005 Ujaldon et al. [56] published SpMV results on a GeForce 6800. They achieved 222 MFLOP/s with the BCSSTK30 matrix (1036208 nonzero elements, stored in CSR format) from the Harwell-Boeing collection [20]. The Harwell-Boeing Sparse Matrix Collection is a set of standard test matrices arising from problems in linear systems, least squares, and eigenvalue calculations from a wide variety of scientific and engineering disciplines. The majority of the matrices are less than 1000×1000 and the collection contains 292 matrices.

More recently in Graphics Hardware 2007, Sengupta et al. [52] published SpMV results based on an efficient segmented scan using CUDA. This work is most relevant to the present work as the authors used the same hardware and programming language. Sengupta et al. [52] reported SpMV performance of 215 MFLOP/s for a 294,267 nonzero element matrix. These results are comparable to those published by Ujaldon et al. [56] almost 2 years prior. They are also just below that of CPU implementations at the time [52, 24].

Square matrices of size $n \times n$ and e nonzero elements in CSR format (figure 2.4)

were used for the experiments performed by Sengupta et al. [52]. The algorithm used requires additional `flag` and `product` temporary data structures with e entries each to be created. Matrix multiplication then proceeds in four steps.

1. The first kernel runs over all entries. For each entry, it sets the corresponding `flag` to 0 and performs a multiplication on each entry: `product`[i] = `val`[i] * `vec`[`ind`[i]] .

2. The next kernel runs over all elements in `ptr` and sets the head flag to 1 for each `flag`[`ptr`[i]] through a scatter. This creates one segment per row.

3. A backward segmented inclusive sum scan is then performed on the e elements in `product` with head flags in `flag`.

4. To finish, a final kernel is run over all rows, adding the gathered value from `product`[i] to the result array.

While the implementation provided by Sengupta et al. [52] is very efficient in that it doesn't waste any instructions on zero elements and is not dependent on matrix structure, it has a large overhead in terms of extra memory operations. Since the SpMV is a memory bound operation it suggests that optimising for less arithmetic operations by introducing more memory operations would not achieve favourable results. One memory fetch is the equivalent of 100 to 150 floating-point adds or multiplies. In addition, the extra work that results from operating on zero elements might be an issue for lockstep SIMD architectures but is less relevant to the GeForce 8800 GTX since the warp architecture limits the effect of one warp on the other in terms of divergence.

An interesting question not considered in the paper is: If non-coalesced 32-bit memory reads are an order of magnitude slower than coalesced 32-bit and 4 times slower than non-coalesced 128-bit reads, what are acceptable software designs that while increasing the number of memory references, still show a better overall memory bandwidth performance? [2]

To the end of this thesis, two notable contributions in the area of sparse matrix-vector products on CUDA enabled GPUs were published. The first by Bell and Garland [9] investigated a variety of sparse matrix formats. Each of these formats requires an SpMV kernel and in the case of CSR format both a sequential and coalesced CSR implementation were created. The authors also

[2] We are grateful to the examiner for bringing to our attention, work by Satish et al published in "Designing efficient sorting algorithms for manycore GPUs", IPDPS 2009

investigated the use of texture memory and found a performance gain through its use. Both structured and unstructured matrices were considered. The structured matrices were composed of standard discretizations of Laplacian operations in 1, 2 and 3 dimensions. The Unstructured matrices were represented by a set of 14 matrices taken from previous work by Williams et al [60]. In comparison, the work presented here is focused on a single sparse matrix format type (CSR), exhaustively studies the performance of all possible implementation options, uses a significantly larger number of sparse matrices, and makes an attempt to map directly from matrix attribute to optimal implementation. A more detailed comparison between this thesis and the work of Bell and Garland is presented in section 4.7.

The second contribution by Baskaran and Bordawekar [8] focuses solely on the CSR storage format. They identify four optimisations, i) exploiting synchronisation free parallelism, ii) optimised thread mapping, iii) optimised off-chip memory access, iv) exploiting data reuse. They evaluate their implementation using 19 sparse matrices taken from the Florida sparse matrix collection [17]. They compare their performance with that of Bell and Garland [9] and the NVIDIA CUDPP library [1] which has an SpMV implementation based on the segmented scan approach of Sengupta et al. [52]. Although more similar to the work presented here in that they focus exclusively on CSR format, it represents a more traditional approach to program optimisation that is less amenable to automation, has many fewer implementations, and uses many fewer test matrices.

Chapter 3

Dense Matrix-Vector Performance

At the heart of an SML application is a matrix representation of the learning datasets where each row of the matrix represents the values of all the attributes for a particular dataset. This matrix remains constant throughout the lifetime of the application. Each iteration of the application involves a normal and transpose product of the matrix (see fig. 2.3).

The objective is to offload the matrix-vector products to the GPU. The matrix will first be transferred to the GPU at the start of the application. Each matrix-vector computation will then consist of:

1. Copying the vector to the GPU.
2. Computing the matrix-vector product on the GPU.
3. Copying the result back to the host.

The above discussion, suggests two targets for our evaluations. The first is to measure and evaluate bandwidth between GPU and host as this will quantify the cost of copying the matrix to the GPU as well as the cost of copying the vector and result between the GPU and host for each iteration. The second is to measure and evaluate the performance of matrix-vector products, normal and transpose for both dense and sparse formats. The evaluation of the matrix-vector products will involve comparisons with the performance on the host to determine first if the GPU offers performance advantages and if so, what are the characteristics of the matrices where such advantages are observed. Likewise, sparse and dense products on the GPU are compared to identify what matrix characteristics identify performance advantages in using sparse over dense products.

Measuring the bandwidth between the GPU and CPU is achieved by measuring the time to copy different sized data blocks in each direction. Measuring the performance of the matrix-vector products is more complex. There are several parameters that could affect the performance of matrix-vector products, such as the size and the shape (ratio of rows to columns) of the matrices. This chapter will only analyse the performance of dense matrix-vector products. Chapter 4 will detail the implementation and performance of sparse matrix-vector products on the GPU.

3.1 Hardware and Software Setup

Benchmarks were performed on two systems. The first system, that also hosts the GeForce 8800 GTX GPU, is an AMD system containing a 2GHz dual core AMD Athlon64 3800+ processor with 2GB of PC3200 DDR memory. The processor has 128KB of L1 cache, 1 MB of L2 cache and a theoretical peak performance of 8 GFLOP/s. The GPU is installed in a PCIe 1.0 slot with a peak theoretical bandwidth of 4GB/s.

While the processor in this system was acceptable at the start of this work, processors have improved markedly in the last 18 months. A new Intel system was added for comparison. This system is a Sun Ultra^{TM} 24 Workstation with a Q6600 2.4 GHz Core2 Quad CPU (4 core, 8MB L2 cache, 1066MHz FSB) and 4 GB of DDR2-667 memory. Table 3.1 provides a comparison between the hardware of the two systems as well as the maximum memory bandwidth and single and double-precision FLOP/s achieved on both CPUs using matrix multiply routines.

Memory bandwidth evaluations between the host and the GPU were performed with host-side functions provided by the CUDA toolkit version 2.0. Dense matrix-vector products on the GPU were performed with the CUDA BLAS library version 2.0 (CUBLAS). ATLAS[1] [59] version 3.6.0 and 3.8.2 in addition to Intel's Math Kernel Library (MKL) version 10.0.1.014 were used for the dense matrix-vector products on the two CPUs. Both systems were dedicated for benchmarking and the wall clock time was used to measure the time for all the benchmarks. All benchmarks were run 100 times and results were averaged. FLOP/s were calculated as $((2 \times M \times N) \div time)$ where M and N are the dimensions of the matrix.

[1] Automatically Tuned Linear Algebra Software, http://math-atlas.sourceforge.net

Table 3.1: Hardware Comparison between the two systems with results from benchmarking memory bandwidth and matrix multiply performance.

System	AMD System	Intel System
Processor	AMD Athlon 64 X2 3800+	Intel Core2 Quad Q6600
Clock Rate	2.0 GHz	2.4 GHz
Cores	2	4
L1 Cache Size	128 KB per core	64KB per core
L2 Cache Size	512 KB shared	8MB (4MB shared/2 cores)
Memory	2x 1GB DDR-400	4x 1GB DDR2-667
lmbench* benchmark (1 copy)	3402 MB/s	5278 MB/s
lmbench* benchmark (2 copies)	5087 MB/s	5863 MB/s
lmbench* benchmark (3 copies)	N/A	4953 MB/s
lmbench* benchmark (4 copies)	N/A	6238 MB/s
SGEMM◇ (GFLOP/s, single core)	6.8	atlas: 13.3, mkl: 18.1
SGEMM◇ (GFLOP/s, all cores)	11.9	atlas: 44.9, mkl: 66.1
DGEMM◇ (GFLOP/s, single core)	3.5	atlas: 7.6, mkl: 9.0
DGEMM◇ (GFLOP/s, all cores)	6.5	atlas: 26.8, mkl: 31.3

* lmbench was compiled with gcc on both platforms. The Intel compiler on the Q6600 gave similar results.

◇ Using ATLAS on the AMD system and MKL on the Intel system with 2 square matrices with dimensions of 2000×2000.

3.2 Bandwidth between the Host and GPU

GPU memory bandwidth benchmarks were performed on the AMD system that hosts the GPU. Table 3.2 shows the results for the bandwidth benchmarks between the host and the GPU with pageable and page-locked memory (see section 2.2.1) as well as the internal GPU memory bandwidth. Latency reported is the time to copy 1 byte.

Table 3.2: Host initiated memory transfer rates (GB/s)

Benchmark	Latency in μs	Bandwidth in GB/s		
		1KB	1MB	100MB
Main Memory (pageable) to GPU	22	0.03	0.80	1.10
GPU to Main Memory (pageable)	18	0.04	0.40	0.50
Main Memory (page-locked) to GPU	18	0.04	2.70	3.10
GPU to Main Memory (page-locked)	15	0.05	2.80	3.00
GPU Memory to GPU Memory*	12	0.25	53.95	65.12

* Host initiated memory copies. GPU initiated memory copies would have lower latency.

Latency measurements show that when using pageable memory, the latencies of copying data to and from the GPU are $22\mu s$ and $18\mu s$ respectively. When using page-locked memory latencies are $18\mu s$ and $15\mu s$. Internal GPU memory transfers have a latency of $12\mu s$. This is due to the fact that the GPU to GPU memory copies in table 3.2 are initiated from the host. The GPU memory fetch time is 400-600 cycles and memory transfers initiated from the GPU would have a smaller latency.

Moving to the bandwidth results from pageable memory, host to GPU transfers of 1KB perform poorly at 30MB/s. Bandwidth increases dramatically to 800 MB/s for 1MB transfers and reaches a maximum of 1.1 GB/s for 100MB transfers. GPU to host results are very similar at 40MB/s for 1KB transfers. They are however about 2× slower for transfers of 1MB and 100MB. When using page-locked memory, there is not a large difference in bandwidth for transfers of 1KB, however transfers of 1MB and 100MB perform about 3× and 6× faster for transfers to and from the GPU respectively. Internal GPU to GPU memory transfers performed poorly with 1KB transfers. Transfers of 1MB showed improved

bandwidth and a maximum of 65GB/s was achieved with 100MB transfers; this is about 81% of peak theoretical performance.

Results from table 3.2 suggest that the page-locked memory should be used for faster memory transfers. When using page-lock memory, data was able to move at a maximum rate of 3GB/s between the host and the GPU with more than 85% of this bandwidth being achieved with only 1MB transfers. This is about half the bandwidth of main memory as presented in table 3.1. This limitation is a result of the PCIe bus which has a theoretical limit of 4GB/s in each direction. All newer GPUs support the newer PCIe 2.0 bus with a theoretical limit of 8GB/s in each direction and the bandwidth between the GPU and host would be expected to double. The low bandwidth between the GPU and host clarifies the large cost of moving data to the GPU. In terms of the SML application, it indicates that many iterations of matrix-vector products will be needed to offset the cost of the initial matrix transfer. The cost of copying the vector and result between the GPU and host for each matrix-vector product will need to be identified as well. The large internal GPU bandwidth is about 31× the internal bandwidth of the host. This indicates an advantage for the GPU in memory bound computations such as matrix-vector products.

Having quantified the cost of memory transfers between the host and the GPU, the performance of dense matrix-vector products on the GPU are evaluated.

3.3 Dense Matrix-Vector Performance

This section, will discuss the results from evaluating the performance of single precision, dense matrix-vector products using both dedicated matrix-vector routines (SGEMV) and matrix matrix routines (SGEMM) on the AMD and Intel CPUs as well as the GPU. Both the SGEMV and SGEMM routines are used to calculate the product of a NxM matrix with a Mx1 vector.

Given a matrix of dimensions M×N, listing 3.1, illustrates a simple algorithm for computing the matrix-vector products.

Listing 3.1: Structure of Dense Matrix-Vector Products

```
for(i=0;i<M;i++)
    for(j=0;j<N;j++)
        res[i] += A[i][j] * vec[j] ;
```

Listing 3.1 shows that the multiplication has a cost of $O(N \times M)$ and both the matrix and vector are streamed into the processor with a unit stride. Accessing memory with unit stride maximises the performance benefits of cache and is the

optimal access method for both the GPU and CPU systems. On both AMD and Intel CPUs, the matrix-vector products would be able to perform two operations per one element of the matrix. This would allow us to achieve a peak performance of $(bandwidth \div 4\ bytes) \times 2\ operations$ in GFLOP/s. From table 3.1, memory bandwidth is seen to reach 6GB/s on both CPU systems. It is interesting to see that both these processors, while generations apart give the same predictable amount of performance for dense matrix-vector products due the similar memory bandwidth performance.

The SML application includes a normal matrix-vector multiplication as well as a transpose matrix-vector multiplication and therefore, both normal (N) and transpose (T) options in the matrix-vector routines are evaluated. The evaluations cover a range of different sizes and ratios between rows and columns.

In the performance analysis of matrix-vector products, the effect of the size of the matrix on performance is observed as well as the shape of the matrix (ie. the ratio of rows to columns). To observe the effect of these parameters, two experiments are performed. In the first, matrix-vector products are evaluated with square matrices of ascending sizes and in the second, the size of a matrix is kept constant and the ratio between the number of rows to the number of columns is modified. In addition, to factor in the per iteration overheads of data transfers between host and GPU, the results for the GPU include the time required to transfer the vector to the GPU and the resulting product vector back to the host.

Performance for SGEMM and SGEMV calls performing matrix-vector products will be given for the CPUs and the GPU. A range of matrix sizes from 1024 to 10240 with increments of 128 are tested. The results are provided for both normal and transpose multiplications. ATLAS and MKL permit the matrix to be stored in either row or column major format, while CUDA only supports matrices in column major format. All evaluations are therefore conducted on matrices stored in columns major format.

3.3.1 Effect of Size on Dense Matrix-Vector Performance

The first set of performance results for SGEMM and SGEMV are for the AMD CPU and are presented in figure 3.1 and partially replicated in table 3.3. A variety of matrix sizes from 1024 to 10240 with increments of 128 are shown. Since ATLAS 3.6.0 performed best on the AMD system, only those results are presented.

The first observation from figure 3.1 is that the results from the use of one

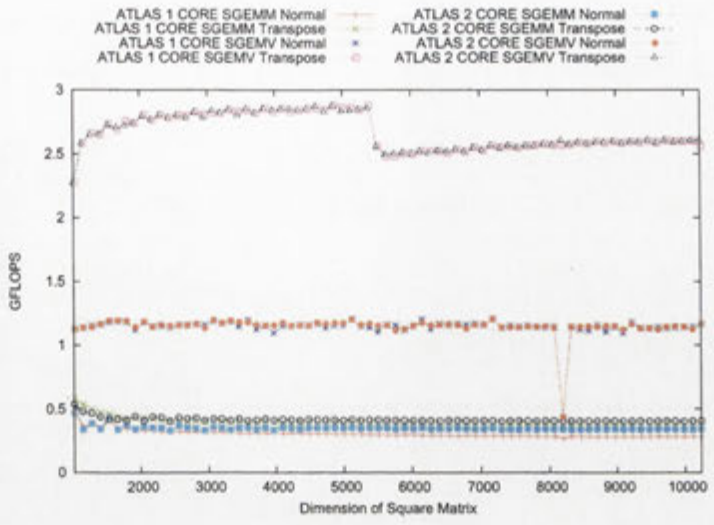


Figure 3.1: Performance (GFLOP/s) for square matrix-vector products on the AMD System using ATLAS 3.6.0.

core are essentially the same as those resulting from the use of 2 cores. This is due to the fact that a single core of the AMD CPU is able to saturate the memory bandwidth. Given that the limiting factor of matrix-vector products is the memory bandwidth, the addition of a core doesn't provide an advantage. While the memory benchmark results in table 3.1 show low bandwidth for a single core, the memory controller in the AMD CPU is shared between the two cores and there is no reason why a single core cannot achieve the same bandwidth achieved by two cores [48, 61]. The SGEMM/V results were further validated by disabling one of the 2 cores and re-running the benchmarks, producing the same results.

The second observation is that the SGEMV routines perform better than SGEMM routines for matrix-vector products as would be expected. SGEMV routines performed at about 1.2 GFLOP/s for normal products and about 2.5 GFLOP/s for transpose products. SGEMM routines performed at about 0.4 GFLOP/s for both normal and transpose products.

The third observation is that for the SGEMV routine, the transpose products outperformed the normal products by more than a GFLOP. This is due to the fact that the matrix is stored in column major format and when performing the transpose product, the matrix is accessed in unit stride while a normal product would access the matrix in strides of M where M is the numbers of rows in the matrix. (Results of benchmarking SGEMV routines with the matrix stored in

row major format produced the the same results only with the normal products performing in the 2.5 GFLOP/s range and the transpose products performing in the 1.2 GFLOP/s range.)

The final observation is that the performance of ATLAS routines are very consistent across a variety of different sized matrices.

Table 3.3: Performance in GFLOP/s for square matrix-vector products on the AMD System using ATLAS v3.6.0 (see figure 3.1).

Dimension	Performance in GFLOP/s							
	1 Thread				2 threads			
	SGEMM		SGEMV		SGEMM		SGEMV	
	N	T	N	T	N	T	N	T
1024	0.47	0.58	1.13	2.28	0.46	0.54	1.12	2.27
2048	0.33	0.43	1.18	2.80	0.36	0.41	1.18	2.80
3072	0.32	0.40	1.19	2.82	0.36	0.42	1.20	2.84
4096	0.31	0.39	1.15	2.86	0.35	0.42	1.18	2.85
5120	0.30	0.39	1.20	2.86	0.35	0.42	1.20	2.84
6144	0.29	0.38	1.20	2.52	0.35	0.41	1.17	2.52
7168	0.29	0.37	1.21	2.56	0.35	0.41	1.20	2.57
8192	0.27	0.37	0.43	2.57	0.34	0.40	0.43	2.61
9216	0.28	0.38	1.18	2.60	0.34	0.41	1.17	2.59
10240	0.28	0.37	1.16	2.56	0.34	0.41	1.17	2.61

The next set of results provided are for the Intel system, where the same experiments conducted on the AMD system are repeated. The Intel MKL library performed best on this system so only the MKL results are shown. The Intel system has four cores and so results are provided from the utilisation of one, two and four cores.

The first observation in figure 3.2 is that again, there is no noticeable performance increase due to the involvement of multiple cores in the matrix-vector product. For the Intel system, this was indicated by the memory benchmark results in table 3.1 where the results for one core were over 80% of what was achieved with four cores.

The second observation is that unlike the results for the AMD system the performance grows gradually until matrix sizes of 5000×5000, where the performance start to stabilise.

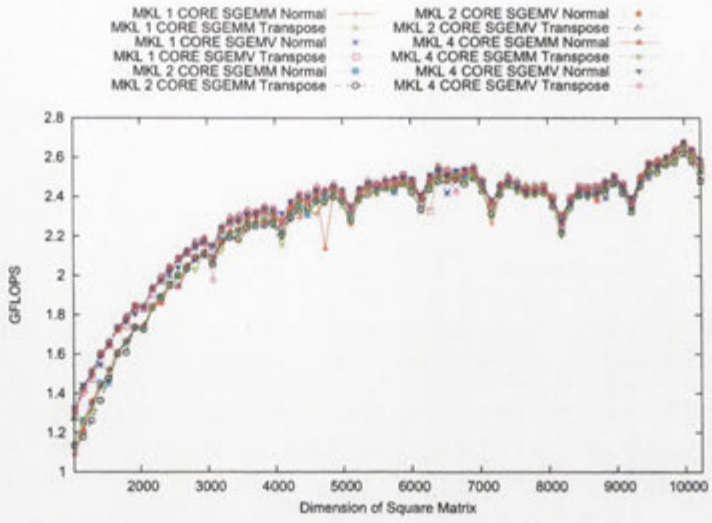


Figure 3.2: Performance (GFLOP/s) for square matrix-vector products on the Intel System using MKL 10.0.1.014.

The third observation is that the normal and transpose performance are essentially identical in performance. The fourth and final observation is that SGEMV and SGEMM routines are very close in performance and virtually identical for matrix sizes over 5000×5000. As before, some of the results presented in figure 3.2 have been replicated in table 3.4 for added clarity.

The final set of results from the evaluation of how size affects performance are from the GPU and are presented in figure 3.3.

Looking at figure 3.3, it is observed that the performance of the SGEMV routines both normal and transpose is better than the SGEMM routines as was the case for both AMD and Intel results. The performance of the SGEMV routines for normal products gradually increases until matrix sizes of 8192, where the performances drops dramatically. This behaviour is most likely due to the internal blocking mechanisms and their affect on the Translation Lookaside Buffer (TLB).

The SGEMV normal multiply performs generally much better than the transpose multiplies. The transpose multiply does however perform better in the case of matrices smaller than 2000×2000. The performance of the transpose products was erratic. Again some of the GPU matrix-vector results have been replicated in part in table 3.5.

Figure 3.4, plots the best performing SGEMV results from both of the CPU as well as the GPU. The GPU achieves between $2\times$ to $10\times$ better than the host depending on the size of the matrix and whether the product is normal or

Table 3.4: Performance (GFLOP/s) for square matrix-vector products on the Intel System using MKL v10.0.1.014 (see figure 3.2).

Dimension	Performance in GFLOP/s											
	1 Thread				2 threads				4 threads			
	SGEMM		SGEMV		SGEMM		SGEMV		SGEMM		SGEMV	
	N	T	N	T	N	T	N	T	N	T	N	T
1024	1.10	1.13	1.28	1.32	1.13	1.13	1.28	1.27	1.09	1.14	1.32	1.31
2048	1.74	1.72	1.83	1.83	1.74	1.73	1.84	1.84	1.75	1.73	1.85	1.84
3072	2.09	2.06	2.15	2.14	2.09	2.06	2.15	2.15	2.09	2.06	2.15	1.97
4096	2.23	2.15	2.26	2.22	2.21	2.20	2.27	2.25	2.27	2.19	2.32	2.28
5120	2.25	2.27	2.33	2.31	2.31	2.28	2.35	2.33	2.33	2.29	2.35	2.33
6144	2.38	2.32	2.39	2.37	2.39	2.34	2.40	2.38	2.41	2.36	2.41	2.39
7168	2.26	2.30	2.37	2.36	2.37	2.31	2.37	2.36	2.37	2.32	2.39	2.36
8192	2.28	2.21	2.29	2.25	2.30	2.22	2.28	2.25	2.31	2.20	2.28	2.24
9216	2.36	2.32	2.35	2.35	2.38	2.32	2.38	2.36	2.39	2.33	2.39	2.37
10240	2.53	2.48	2.54	2.51	2.56	2.48	2.56	2.52	2.56	2.53	2.59	2.57

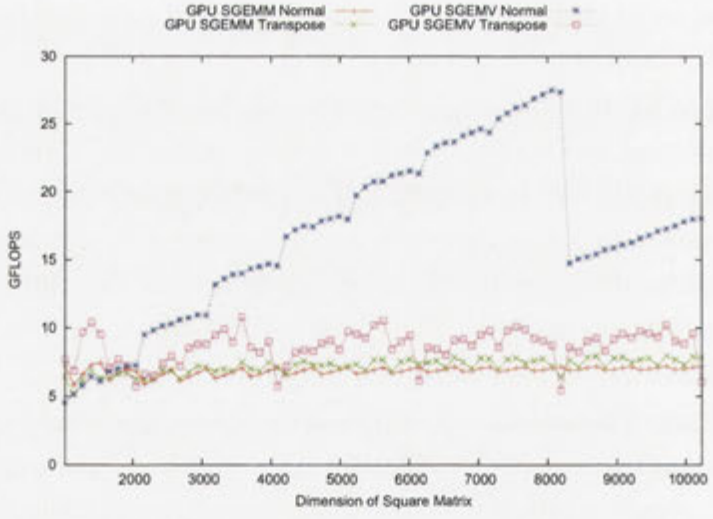


Figure 3.3: Performance (GFLOP/s) for square matrix-vector products on the GPU with CUBLAS 2.0.

Table 3.5: Performance (GFLOP/s) for square matrix-vector products on the GPU with CUBLAS 2.0 (see figure 3.3).

Dimension	Performance in GFLOP/s			
	SGEMM		SGEMV	
	N	T	N	T
1024	7.23	6.39	4.54	7.71
2048	6.88	6.30	7.26	5.75
3072	7.01	7.11	10.92	8.85
4096	7.11	7.20	14.54	5.77
5120	7.13	7.23	17.94	9.78
6144	7.12	7.32	21.31	6.19
7168	7.17	7.73	24.34	9.80
8192	7.17	6.33	27.32	5.44
9216	7.16	7.45	16.24	9.28
10240	7.20	7.84	18.00	6.05

transpose.

Many programs and architectures favour specific problem sizes due to architecture or coding design. To investigate the presence of any such issues in the BLAS libraries used or the systems evaluated, experiments were repeated on matrices of sizes 4160×4160 to 4224×4224 in increments of one. The results of these evaluations are presented in figure 3.5.

Figure 3.5 shows that the SGEMV routines from the CUBLAS library are the most affected by the problem size. Performance for normal products increased $4\times$ when the dimensions of the matrix were a multiple of 16 while the transpose product performance increased $2\times$ for the same cases. This is due to the fact that the matrices are stored in column major format and the numbers of rows must be multiples of 16 for the memory to be aligned properly for the hardware to coalesce memory reads (see section 2.2.2). ATLAS SGEMV routines on the AMD system show an improvement of 15% to 50% when the dimension of the matrix is a multiple of two. The performance of the MKL SGEMV routines on the Intel system remained consistent regardless of the matrix dimensions.

Matrix-vector products have an algorithmic complexity of $O(N \times M)$ where N = number of rows and M = number of columns of the matrix. The CUBLAS library implementation of SGEMV launches 8,192 threads (64 blocks, 128 threads)

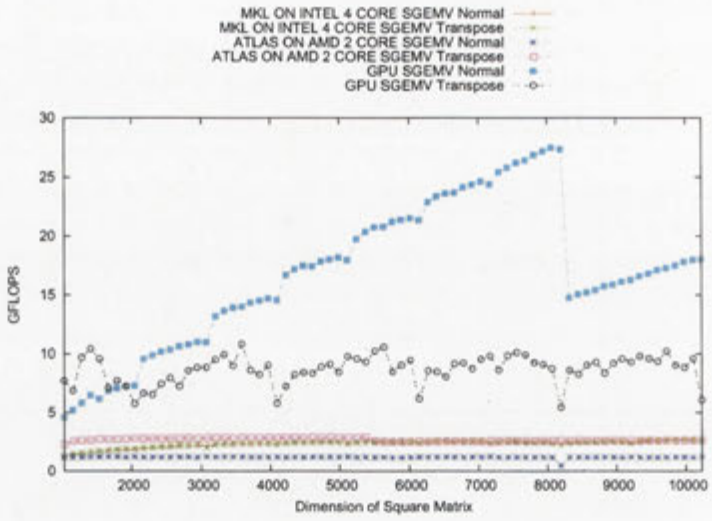


Figure 3.4: CPU vs GPU (ascending sizes).

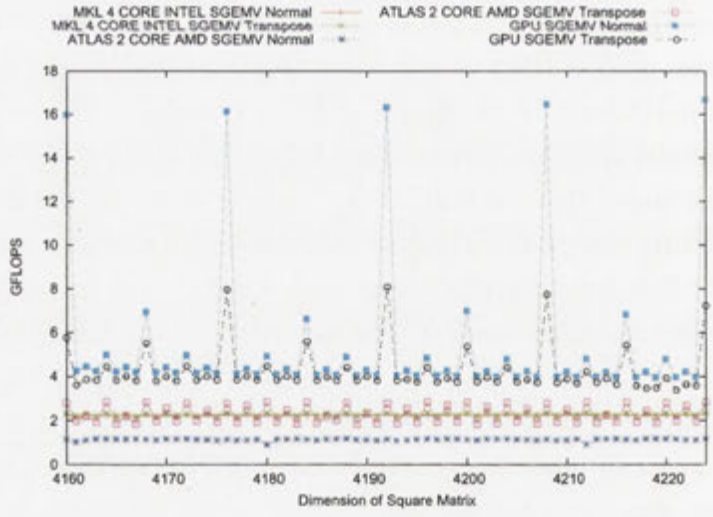


Figure 3.5: Evaluating matrices of unit increments exposes the preference of programs and hardware for specific problem sizes.

each looping over as many rows as needed to compute all rows. For given M we would expect the performance of the SGEMV routine to peak when N is a multiple of 8,192 due to maximum utilisation of all threads. The performance should increase linearly as N approaches a multiple of 8,192. Figure 3.3 shows that the performance of the SGEMV routine does indeed scale linearly with respect to N and peaks at $N = 8,192$. The performance drops drastically after that due to poor utilisation as a total of 16,384 threads would have been created but only 8,193 of them would contribute work. As N increases, the performance starts to ascend linearly to another maximum (expected) at $N = 16,384$.

3.3.2 Effect of Shape on Performance

The effect of matrix shape on performance of the SGEMV routines from the ATLAS, MKL and CUBLAS libraries, was evaluated using a matrix containing 26,214,400 elements (approximately 100MB in size). Keeping the number of elements of the matrix constant, the number of rows and columns are varied from a 128×204800 matrix to 204800×128. Figure 3.6 presents the results of the evaluations of the two CPUs. Only results for the the SGEMV routines are presented for both CPUs as they consistently outperform the SGEMM routines. Figure 3.7 presents the results of both SGEMV and SGEMM GPU routines as neither of them consistently outperforms the other. Results are also partially replicated in table 3.6 for further clarification.

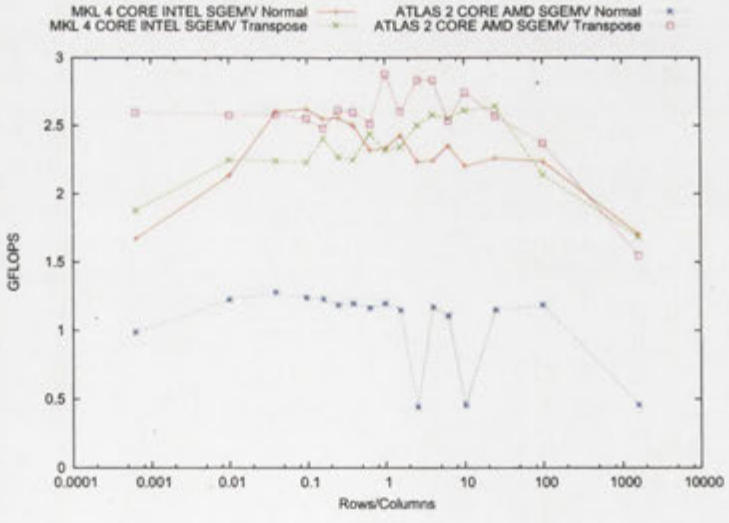


Figure 3.6: Performance (GFLOP/s) as function of shape for matrix-vector products on the CPU.

Starting with the results of the MKL SGEMV routine on the Intel system presented in figure 3.6, both normal and transpose products showed a small drop in performance when the number of rows or columns dropped to 128. The normal products saw an increase in performance form small number of columns, while the transpose products saw an increase for small number of rows. The ATLAS normal SGEMV results lagged the transpose SGEMV results by almost 1.5 GFLOP/s for most of the test cases. A small decrease in performance is observed for small number of rows and a much higher drop of 50% is observed at matrices of sizes 8192×3200, 16384×1600 and 204800×128. The ATLAS transpose SGEMV results were not affected by a small number of rows but show a significant drop in performance for a small number of columns.

Generally the results indicate that the performance of the MKL implementation on the Intel CPU is more stable than the ATLAS results on the AMD CPU. However, the ATLAS SGEMV results on the AMD CPU produce better performance than the MKL implementation on the Intel CPU when the matrix is read in unit stride (row major + normal products or column major + transpose products) and much worse than the MKL implementation on the Intel CPU when they are not.

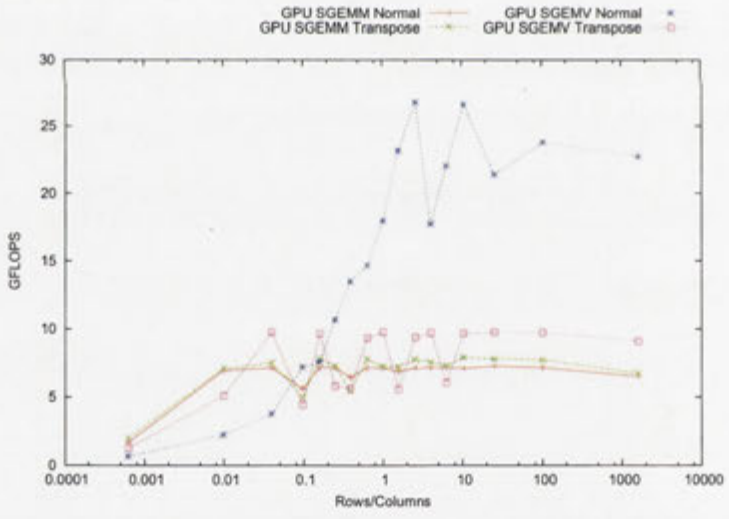


Figure 3.7: Performance (GFLOP/s) as function of shape for matrix-vector product on the GPU.

The SGEMM results on the GPU shown in figure 3.7 show the normal and transpose products producing similar results. Performance of the SGEMM routines are quite stable but decline in performance for very small number of rows.

The SGEMV transpose results are not as stable with a drop of 50% in performance showing for some matrices. The SGEMV transpose products also drops in performance for small number of rows. The SGEMV normal product performs much worse than the others for number of rows under 2048 after which performance increases rapidly. There is a sharp drop in performance for the matrix of size 10240×2560 but performance levels out as the number of rows continues to grow. An important observation is that for small number of rows, the use of SGEMM routines will perform better than SGEMV routines.

Table 3.6: Performance (GFLOP/s) as function of shape for matrix-vector products on the CPU (see figures 3.6 and 3.7).

Rows	Columns	AMD		Intel		GPU			
		SGEMV		SGEMV		SGEMM		SGEMV	
		N	T	N	T	N	T	N	T
128	204800	0.99	2.59	1.67	1.87	1.70	1.95	0.65	1.29
512	51200	1.22	2.58	2.13	2.24	6.97	7.06	2.23	5.11
1024	25600	1.28	2.58	2.60	2.23	7.13	7.49	3.76	9.77
1600	16384	1.24	2.55	2.62	2.23	5.60	4.93	7.18	4.44
2048	12800	1.23	2.48	2.55	2.40	7.16	7.79	7.54	9.66
2560	10240	1.18	2.62	2.56	2.26	7.15	7.23	10.65	5.79
3200	8192	1.19	2.60	2.50	2.24	6.43	5.41	13.48	5.61
4096	6400	1.16	2.51	2.32	2.43	7.14	7.75	14.64	9.33
5120	5120	1.20	2.88	2.33	2.31	7.13	7.20	17.94	9.78
6400	4096	1.15	2.60	2.43	2.34	6.87	7.18	23.14	5.57
8192	3200	0.44	2.84	2.23	2.50	7.12	7.73	26.75	9.40
10240	2560	1.17	2.83	2.24	2.58	7.13	7.51	17.71	9.71
12800	2048	1.11	2.54	2.35	2.55	7.15	7.25	21.99	6.03
16384	1600	0.46	2.74	2.20	2.61	7.09	7.87	26.60	9.68
25600	1024	1.15	2.57	2.26	2.64	7.25	7.74	21.37	9.75
51200	512	1.19	2.37	2.24	2.13	7.14	7.68	23.74	9.71
204800	128	0.46	1.55	1.70	1.68	6.51	6.72	22.70	9.09

Finally, figure 3.8 combines SGEMV performance results for the AMD and Intel CPU with the results from the GPU.

Figure 3.8 shows that matrices must have a minimum of 1000 rows to achieve perform gains on the GPU. When the number of rows are below 2560, the

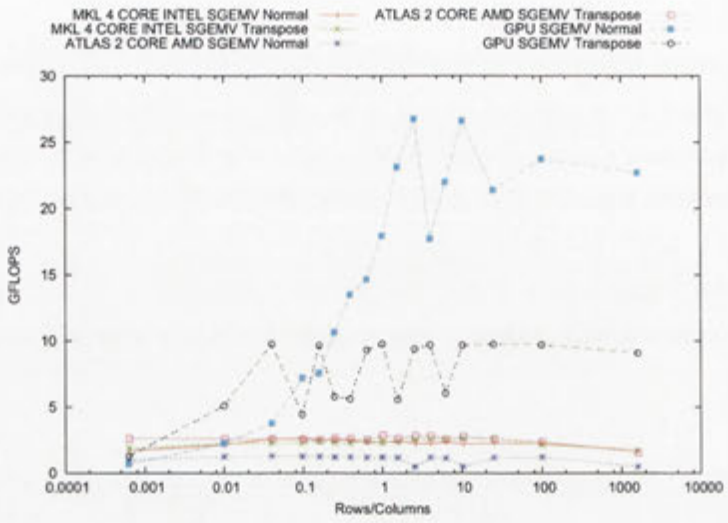


Figure 3.8: CPU vs GPU (Changing Shapes).

SGEMM routines provide better performance then the SGEMV routines.

3.3.3 Conclusion

Figure 3.4 showed that for square matrices, the GPU was at least 2× as fast as the CPUs. Figure 3.8 showed that a minimum of 1000 rows was needed to achieve even the slightest performance gain.

The main decision that needs to be made at this point is when to use the GPU over the CPU for a matrix-vector product. The results in this chapter indicate that the benefits of performing matrix-vector products on GPUs are realised for larger matrices with at least 2048 rows. The exact performance gains will depend on the number of matrix-vector products conducted.

Chapter 4

SpMV Construction and Evaluation

As noted in chapter 2, exploiting the sparsity of a matrix can decrease the number of instructions needed to compute a matrix-vector product as well as the memory footprint of the matrix. This exploitation also introduces a slightly more complex data structure and random memory accesses. The point at which the sparsity of a matrix becomes large enough to benefit from its exploitation is system and implementation dependent. This chapter presents a range of implementation options on the NVIDIA GPU system, assesses their performance as a function of matrix attributes, and by doing so establishes a set of implementations such that for any matrix the best performing implementation is part of that set. This chapter also presents efforts to identify the level of sparsity that warrants the use of sparse matrix-vector products on the GPU over their dense counterparts.

There are many storage formats for sparse matrices as discussed in section 2.4. This work is based on the Compressed Sparse Row (CSR) format as it is widely used in the scientific community [52, 56, 13].

Listing 4.1: Structure of CSR Sparse matrix-vector products

```
for(i=0;i<M;i++)
    for(j=ptr[i];j<ptr[i+1];j++)
        res[i] += val[j] * vec[ind[j]] ;
```

Listing 4.1 shows the kernel of a Sparse Matrix-Vector (SpMV) product when using the CSR format. In comparison to the dense matrix-vector product shown in listing 3.1, the number of iterations of the second loop is not constant, but can vary for each iteration of the outer loop. In addition the `vec` array is referenced indirectly, not in unit stride as is the case for dense matrix-vector products. As a consequence, predicting the access pattern of the `vec` array is not possible, causing

the performance of the SpMV products to be closely related to the structure of the sparse matrix. In short, the SpMV performance achievable on both the CPU and GPU is not as clear as in the case of dense matrix-vector products.

Parallelisation of the code in listing 4.1 can proceed by assigning different iterations of the outer i loop to different execution units. Each iteration of the second loop then involves fetching 3 elements from memory and performing two floating-point instructions. The low FLOP to byte ratio indicates that memory bandwidth is critical for SpMV performance.

Both GPU and CPU systems require high arithmetic intensity (ratio of floating-point operations to memory loads) to reach peak performance. Consider single precision SpMV on the GPU which has a theoretical peak memory bandwidth of 80GB/s or 20×2^{30} 32-bit `float` elements per second and a peak performance of 321.6 GFLOP/s. For a kernel to achieve peak performance, each 32-bit `float` element read from memory must contribute to $321.6 \div 20 \approx 16$ floating-point instructions. SpMV products fall short of such ratios, performing only two floating-point operations per three 32-bit `float` elements and therefore SpMV implementations are better evaluated by memory bandwidth efficiency than by FLOP/s.

However, while the efficiency of the implementation can indeed be evaluated by percentage of maximum bandwidth achieved, evaluation of the work as whole will involve a broader context such as the comparative performance of sparse matrix-vector products on the CPU and dense products on the GPU.

In table 3.2, the GPU memory bandwidth was measured to be 65 GB/s. This was achieved by the use of a high level CUDA function, which raises the following questions:

- How do different configurations of threads per block and blocks per grid affect performance?
- What is the effect of using texture memory instead of global memory? [1]
- What is the difference (in terms of bandwidth) between coalesced and non-coalesced access?
- How does using `float`, `float2` and `float4` vector data types affect memory bandwidth?

[1] Constant memory and shared memory are not evaluated as they are small, specific memories not suitable for general usage.

In order to design an efficient SpMV implementation for the GPU, a more thorough understanding is needed of how the internal GPU memory bandwidth is affected by the above factors.

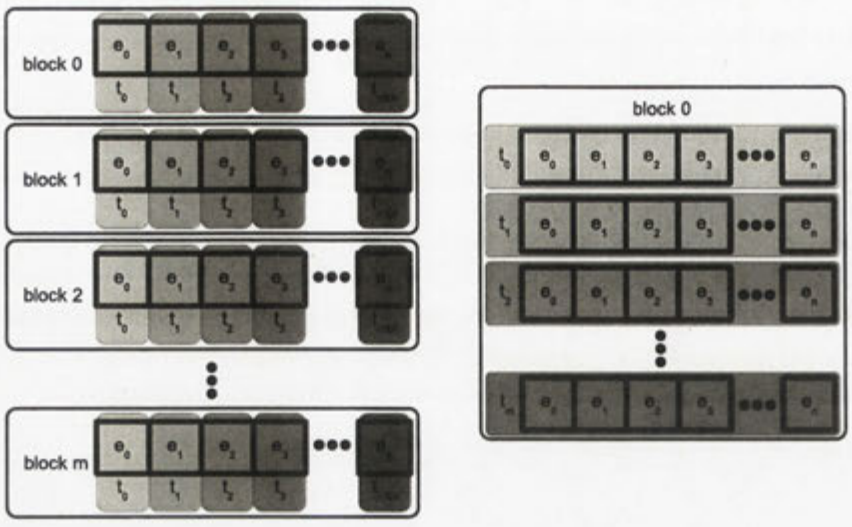
The beginning of this chapter will elaborate on the needed memory bandwidth analysis for the GPU and present the results from the more thorough investigation. The rest of the chapter will focus on the SpMV implementations on the GPU. This will start by discussing the methodology used for evaluation of the implementations followed by a detailed view of the various implementation options available. Evaluations of the various options will follow. The final section of this chapter deals with the selection of one or more of the resulting implementations based on the matrix attributes to produce the best performance.

4.1 Memory Bandwidth Analysis

As discussed above, memory bandwidth is critical to the overall performance of SpMV products. In section 3.2, it was shown that internal GPU memory transfer had a maximum bandwidth of 65 GB/s. This was achieved with the `cudaMemcpy` routine for large (100MB) data transfers. In this section bandwidth is measured in a loop similar to listing 4.1. Two memory bandwidth benchmark frameworks were written to explore coalesced and sequential access methods from global and texture memory, the use of `float`, `float2` and `float4` data types and the effect of the number of threads per block and the number of blocks per grid on performance is considered.

Coalesced and sequential methods of reading in elements from memory are illustrated in figure 4.1. Data is organised in m rows, each row containing n elements. Figure 4.1(a) illustrates how the benchmarks would access memory using coalesced reads. Specifically, each row is assigned to a separate block, and within that block each thread reads in consecutive elements (see section 2.2.1 for a further details of coalesced reads). Figure 4.1(b) illustrates how the sequential benchmark would read the elements from memory. In this case each row is assigned to a separate thread and each thread reads in the entire row.

The benchmarks read a predefined number of `float` elements from memory. The kernel is launched on the GPU with a predefined number of threads per block and blocks per grid. The total number of elements to be read from memory is divided amongst all the threads on the GPU so that all threads have an equal workload to read from memory. In the coalesced benchmark, the total workload



(a) Coalesced reads of m rows using m blocks. In each block, k threads cooperate in reading 1 row of n elements.

(b) Sequential reads of m rows. Each thread reads in 1 row of n elements.

Figure 4.1: Coalesced and sequential memory reads.

for the block is coalesced. In the sequential benchmark, each thread reads its prescribed workload independently from any other thread. The coalesced and sequential memory benchmarks have the same memory access pattern that a coalesced or sequential SpMV implementation would have for the `val` and `ind` arrays if every row of the hypothetical matrix had an equal number of nonzero elements per row.

Three different values for the total amount of data read from memory were considered (1.5MB, 55MB, 390MB), but as there was minimal variance between them, only results from the medium (55MB) data size will be shown here (the full set of results are available in appendix A). The benchmarks were conducted with 16, 32, 64, 128, 256 and 512 threads per block, and with `float`, `float2` and `float4` data types. Each benchmark was run three times with a selected number of blocks so the resulting workloads would be 20, 100 and 1000 float elements per thread. Both coalesced and sequential memory benchmarks were applied to global and texture memory.

It is important to note that while the following sections compare between "coalesced" memory reads from global and texture memory, the hardware does

not coalesce texture memory reads. What is meant here is that threads in a block perform texture reads from consecutive memory locations in the exact same manner that is used for coalescing global memory reads.

4.1.1 Coalesced Memory Benchmark Results

In the case of the coalesced benchmark (see figure 4.1(a)), each block of threads cooperate in reading a single row. The number of elements in the row of the hypothetical matrix is the product of the number of threads multiplied by the workload size. To illustrate this, a configuration of 32 threads and a workload of 20, relates to a row with $32 \times 20 = 640$ elements. A configuration of 128 threads and a workload of 100, relates to a row with $128 \times 100 = 12800$ elements.

Table 4.1 presents coalesced memory bandwidth for reading 55MB from both texture and global memory. Rows are grouped according to workload (the number of `float` elements each thread will read from memory). Results for 32, 64 and 128 threads per block are presented as they were always found to give the highest bandwidth. For each row in the table the results of reads from global memory with `float`, `float2` and `float4` units are given first, followed by similar results from texture memory.

Focusing on the results (table 4.1) from global memory and starting with a workload of 20 `float` elements; at 32 threads per block the bandwidth almost doubles when moving from `float` to `float2` data types, but decreases slightly on moving to `float4`. With 64 threads per block and the same workload a similar trend is observed, although the increase from `float` to `float2` is less and the drop from `float2` to `float4` is larger. For `float` and `float2` data types 64 threads is much better than 32, but for `float4` the results are almost identical. For 128 threads the results of `float` and `float2` are comparable and greater than the results for 32 or 64 threads, while the result for `float4` remains consistent with the results from 32 and 64 threads.

If the workload is increased from 20 to 100 `float` elements very similar results are found, both in terms of trends and absolute values. Increasing the workload to 1000 `float` elements results in a considerable drop in bandwidth for `float` and `float2` data types and essentially identical results for `float4`.

Overall, from the global memory results given in table 4.1, `float2` types consistently performed better in all scenarios except for a workload of 100 `float` elements and a block size of 128, where they are outperformed by `float` data types by a small margin of 5%.

Table 4.1: Benchmark results in GB/s for reading 55MB of data from global or texture memory via coalesced reads for `float`, `float2` and `float4` data types.

Threads /Block	Blocks	Bandwidth in GB/s					
		Global Memory			Texture Memory		
		float	float2	float4	float	float2	float4
Workload of 20 floats per thread							
32	22400	25	40	36	24	36	44
64	11200	46	58	37	36	45	48
128	5600	62	63	37	43	50	52
Workload of 100 floats per thread							
32	4480	27	44	38	25	39	50
64	2240	46	61	37	40	50	52
128	1120	64	61	37	45	52	54
Workload of 1000 floats per thread							
32	448	26	42	38	24	39	51
64	224	39	55	38	38	51	53
128	112	48	54	37	36	45	51

For texture memory and with a workload of 20 `float` elements and using 32 threads per block memory bandwidth increases when moving from `float` to `float2` to `float4` data types. The same trend, but with a slight increase in magnitude is observed when the number of threads per block is 64 or 128. There is not much variation in texture memory performance when moving to different workload sizes. Performance of texture memory tends to drop somewhat at workloads of 1000 `float` elements. Overall, `float4` consistently outperforms `float` and `float2` data types.

Comparing texture and global memory bandwidth at 32 threads per block; texture memory with `float4` units outperforms global memory with `float2` units by 10%, 13% and 21% for workloads of 20, 100 and 1000 respectively. At 64 and 128 threads per block, global memory reads with `float2` units provides better performance than texture memory. From these benchmarks it appears that the best performance is achieved for coalesced reads from global memory by using `float2` data types with 64 or 128 threads per block. At 32 threads per block, coalesced reads from texture memory with `float4` data types is the best option.

4.1.2 Sequential Memory Benchmark Results

In contrast to the coalesced benchmark, the sequential benchmark assigns each row to a separate thread (see figure 4.1(b)). As a consequence the workload size directly corresponds to the number of non zero elements in the hypothetical sparse matrix. Results for sequential benchmarks are presented in table 4.2. Best performance was found between 16 and 64 threads per block, thus results are given for those thread per block sizes rather than 32, 64 and 128 threads per block, as was used in the coalesced memory read benchmarks.

Table 4.2: Benchmark results in GB/s for reading 55MB of data from global or texture memory via sequential reads for `float`, `float2` and `float4` data types.

Threads /Block	Blocks	Bandwidth in GB/s					
		Global Memory			Texture Memory		
		float	float2	float4	float	float2	float4
Workload of 20 floats per thread							
16	44800	9	16	19	6	12	25
32	22400	9	17	19	8	14	25
64	11200	9	17	19	8	16	29
Workload of 100 floats per thread							
16	8960	8	16	19	5	10	18
32	4480	8	17	19	5	11	20
64	2240	8	15	20	10	11	20
Workload of 1000 floats per thread							
16	896	2	4	8	2	4	8
32	448	<1	1	2	<1	1	2
64	224	<1	1	2	<1	1	1

At small workloads of 20 `float` elements per threads, the bandwidth achieved while reading from global memory increases as the width of the data type is increased. The results are essentially identical for all three configurations of threads per block, with 9GB/s for `float` data types rising to 19GB/s for `float4`.

Increasing the workloads from 20 to 100 `float` elements has negligible effect. At workloads of 1000 `float` elements bandwidth is however, much lower than previous workloads, often only 1GB/s. This large drop in performance at 1000 elements is most likely cause by TLB thrashing as the stride between blocks (for `float` types) becomes $1000 \times 4bytes = 4KB$ across each SM or $4KB \times 16 = 64KB$

across the whole GPU.

Texture memory results follow the trends of global memory for all workloads with the highest bandwidth resulting from use of `float4` data types. Workloads of 20 `float` elements result in better performance than larger workloads.

In summary, both global and texture memory benchmarks performed best with the use of `float4` data types. Comparing `float4` performance of global and texture memory types show very similar results except for workloads of 20 `float` elements, where texture memory shows a slight advantage. For workloads of 20 or 100 `float` elements 64 threads per block offers best performance, while workloads of 1000 `float` elements performed best with 16 threads per block.

4.1.3 Coalesced vs Sequential Memory for SpMV

As noted in section 4.1 the workloads in the coalesced and sequential benchmarks correspond to a different number of elements in the hypothetical sparse matrix. To determine when it is best to use coalesced or sequential reads for a sparse matrix it is necessary to normalise the results. The normalisation is presented in figure 4.2, where the best performing data types and memory configurations are used from each benchmark. Specifically, `float2` results from global memory were used for the coalesced reads, while `float4` results from texture memory were used for the sequential reads. Due to the use of `float2`, the minimal number of `float` elements that can be read in with coalesced reads is $2 \times 16 = 32$ `float` elements. Effective bandwidth for smaller numbers of elements per row is calculated as $(\textit{bandwidth} \div 32) \times \textit{elements per row}$. The same method is used to calculate the effective bandwidth from sequential reads when the number of elements is less than four.

The results presented in figure 4.2 show that for small numbers of elements per row sequential reads provide better effective bandwidth than coalesced reads. This changes when the number of elements per row grows larger than 64. This result indicates that both coalesced and sequential implementations of SpMV should be considered.

4.2 SpMV Implementations

The results from the previous section suggest that SpMV implementation that utilise coalesced and sequential memory reads must be evaluated and that the number of nonzero elements per row will determine at which point one imple-

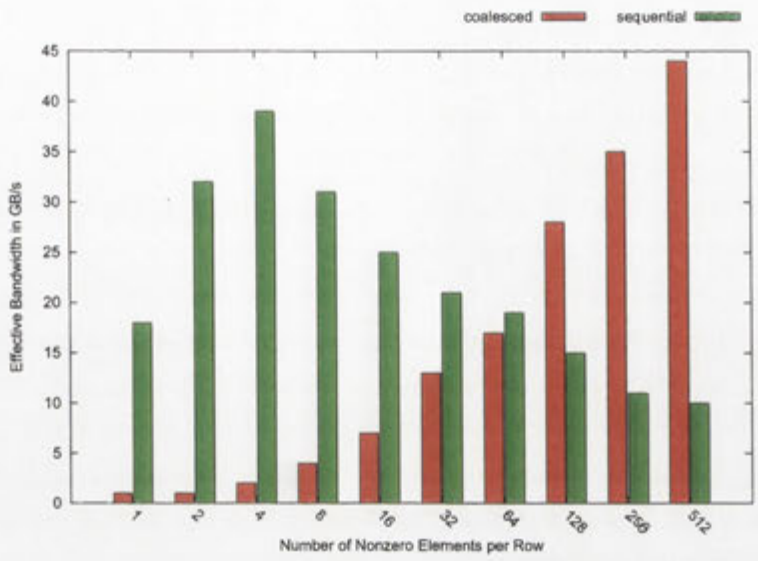


Figure 4.2: Normalisation of memory benchmark results for coalesced and sequential benchmarks.

mentation will outperform the other. From this point on, the term "coalesced implementation" refers to an implementation where the matrix data is read with coalesced memory reads while the term "sequential implementation" refers to an implementation where the matrix data is read with sequential memory reads. The benchmarks were conducted under the assumption that the average number of nonzero elements per rows is representative of the matrix as a whole.

The SpMV routine use five different arrays. The matrix is represented by three arrays (`val`, `ind` and `ptr`) while the vector and result are stored in two other arrays (`vec` and `res`). Both coalesced and sequential implementations have a number of implementation options in terms of which memory to use for each of these five arrays. As described in in section 2.2.1, the GPU offers three different memory storage options. These are global, texture and constant memory.

Global and texture memory both utilise the 768MB of physical memory. Constant memory is limited to 65,536 bytes (16,384 floats/ints or 8,192 doubles) and is therefore not a realistic candidate for storage of large arrays. Constant memory could however, still be utilised for the `ptr` or `vec` arrays, though they would restrict the size of the matrix to 16,384 rows or 16,384 columns respectively (even less if used jointly). The use of constant memory for either of those arrays would be desirable if it offered a large performance gain over other memory types. Shared memory can also be used to cache memory fetches from global memory if

that data can be used by many threads.

The `val` and `ind` arrays are the largest arrays in the SpMV operation. These arrays are both accessed in unit stride and are the only arrays that should be accessed via wide data types (`float4`, `int2`, ... ect) as they have logically consecutive elements. As such, the memory bandwidth analysis in section 4.1 can be applied directly to the `val` and `ind` arrays.

The `ptr` array is accessed twice per row with consecutive elements. Optimising `ptr` access will be important when the number of elements per row is low. The use of memory with cache should also be beneficial. Sequential implementations could coalesce `ptr` elements to shared memory, then access them at fast speeds from shared memory. The use of shared memory for caching will introduce synchronisation overheads and so will be benchmarked to identify whether it would perform better than texture or constant memory.

The `res` array has only one available storage option as only global memory is writeable from the GPU. In addition there is no need to explicitly coalesce writes to the `res` array in global memory as the results from consecutive executing units are consecutive elements in the `res` array.

Table 4.3: Memory options for the different arrays.

Array	Global Memory		Texture Memory		Constant Memory	Shared Memory	Wide Data Types
	coalesced	sequential	coalesced	sequential			
`val`	✓	✓	✓	✓	×	×	✓
`ind`	✓	✓	✓	✓	×	×	✓
`ptr`	×	✓	×	✓	✓	✓	×
`vec`	×	✓	×	✓	✓	×	×
`res`	✓	✓	×	×	×	×	×

Table 4.3 summarises the different options available for each array and indicates that large number of implementations can be constructed based upon the memory options for these arrays.

This section analyses the different implementation options and identifies the best performing implementations to be constructed and benchmarked. Each implementation, is named as follows:

`[coa/seq]-[f1,f2,f4]_[val,ind storage]-[ptr storage]-[vector storage]`

To illustrate, `coa-f2_memm-cnst-text` indicates a coalesced implementation using `float2` units in global memory for the `val` and `ind` arrays, constant memory

for the `ptr` array and texture memory for the `vec` array. `seq-f4_memm-shrm-cnst` indicates a sequential implementation using `float4` units for the `val` and `ind` arrays, shared memory for the `ptr` array and constant memory for the `vec` array.

Table 4.4 summarises the options available to the arrays involved in the SpMV operation and the factors that affect the read performance of each array.

For each attempt to evaluate a storage option, both coalesced and sequential implementations will be benchmarked to make sure that the results from the evaluation of a storage option is consistent across both implementations.

4.3 Performance Evaluation Methodology

This section describes the hardware setup used for the evaluations and provides details on how each storage option is evaluated.

4.3.1 Evaluation Platform

The hardware use to perform the following evaluations has been previously described in section 3.1 but is briefly summarised here. It contains a 2 GHz dual core AMD Athlon64 3800+ processor with 2GB of PC3200 DDR memory. The processor has 128KB of L1 cache and 1 MB L2 cache and a theoretical peak performance of 8 GFLOP/s. The GeForce 8800 GTX GPU was installed in a PCIe 1.0 slot that has a peak theoretical transfer rate of 4GB/s.

4.3.2 Test Matrices

In order to evaluate the SpMV implementations, test datasets needed to be obtained. The small number of test matrices used in previous works [60, 14] motivated the search for large amounts of real world test cases and so the Florida Sparse Matrix Collection was used to obtain many real world examples. Datasets that were used by Teo et al [54] for evaluation of the original application code were also used. However in order to observe a specific aspect of the implementations performance, it was desirable to be able to generate matrices with specific properties. A simple sparse matrix generator was therefore written. This generator allows the specification of the dimensions of the matrix, the sparsity and a technique that determines the placement of non zero elements. Figure 4.3 illustrates the three techniques used for matrix generation.

Table 4.4: Possible optimal implementations as deduced from previous benchmarks and analysis.

Array	Storage Options	Factors Affecting Performance
`val` & `ind`	• A coalesced implementation using `float2` data types in global memory. • A sequential implementation using `float4` data types in texture memory	1. The number of elements per row. 2. The width of the data types.
`ptr`	• Global memory. • Texture memory. • Constant memory. • Global memory with shared memory caching.	1. The number of rows of the matrix as the `ptr` array is accessed twice per row. 2. The average number of elements per row. The importance of optimising `ptr` access is emphasised when the number of elements per row is low.
`vec`	• Global memory. • Texture memory. • Constant memory.	1. The size of the `vec` array. This relates to the size of the matrix columns. 2. The number of memory fetches to the vector array. This relates to the sparsity of the matrix. The more sparse the matrix, the less fetches to the `vec` array. 3. The pattern of memory access. This relates to the fill type of the matrix. These are the whole, row and block fill types.

(a) Random (b) Constant Row (c) Structured

Figure 4.3: Different techniques to generate test matrices.

Random

This technique makes use of a uniform random number generator to determine the nonzero elements in the matrix. For each (row,column) combination, a random number is produced ranging from 1 to 100. If that number is larger than the required sparsity that location is assigned a value, else it is set to zero. The minimum number of elements in a row is set to one. This results in a very unstructured matrix where the number of elements per row can vary greatly between consecutive rows. Figure 4.3(a) illustrates the result of this technique.

Constant Row

This technique imposes more structure than the Random technique. The number of elements per row is set to $(the\ number\ of\ columns \times the\ required\ density)$ for all rows in the matrix. The positions of the elements for each row are randomly generated with a uniform random generator. Comparing results from using this matrix with the Random matrix allows us to assess any overhead associated with variances between the number of elements in consecutive rows. Figure 4.3(b) illustrates this matrix structure.

Structured

This technique is similar to the Constant Row technique in that the number of elements per rows is consistent for all rows in the matrix. It differs in that all the rows have the same column distribution. Using this matrix in benchmarks results in the same elements of the `vec` array being read for each row of the matrix, maximising the effects of cache in the read performance of the `vec` array. Figure 4.3(c) illustrates the result of this generation technique.

4.3.3 Performance Measurements

When taking into consideration the nature of the application discussed in section 2.3, it can only be a significant comparison if the GPU's FLOP/s results take into consideration all overheads resulting from the use of the GPU indication conversion from `double` to `float` and copying data between host and GPU memories. Therefore the time used in calculating FLOP/s for both sparse and dense matrix-vector products include the time taken to:

1. Convert the vector from `double` to `float`.
2. Copy the vector from main memory to the GPU.
3. Perform the matrix-vector product.
4. Copy the result from the GPU to main memory.

There exists a one time cost of converting and copying the matrix to the GPU. This is not included in the FLOP/s calculation. FLOP/s were calculated as $(2 \times number\ of\ nonzero\ elements \div total\ time)$. Each evaluation was run 100 times on a dedicated machine with the average wall clock time used to calculating a performance result.

4.4 SpMV Implementation Assessment

In section 4.2 numerous implementation options for an SpMV implementation on the GPU were discussed. The memory bandwidth benchmark results in tables 4.1 and 4.2 suggest that the implementation options for the `val` and `ind` arrays can be realistically limited to:

1. Coalesced reads using `float2` units (`coa-f2_memm-memm-memm`).
2. Sequential reads using `float4` units (`seq-f4_text-memm-memm`).

This section outlines an attempt to select the best options for the `ptr` and `vec` arrays. In evaluating the various implementation options, only those parameters that are readily associated with the matrix itself were considered, namely:

- The number of rows in the matrix.
- The number of columns in the matrix. This is equal to the number of elements in the vector.

- The number of non zero elements in the matrix.

The benchmarks in this section were all performed with 32 threads per block as this is the size of a warp on the GPU. While the results from the benchmarks in tables 4.1 and 4.2 indicate that 64 threads per block results in best performance overall, this is not necessarily the optimal value for the SpMV implementation as a whole. There could exist different optimal values for accessing the vector and `ptr` array. In addition, each implementation will have overheads that may respond differently to different number of threads, such as the need to reduce elements in the coalesced implementation or that for sequential implementations each group of 32 threads execute the same number of instructions dependent on the largest number of elements in the group of rows assigned to them. The issue of threads per block will be in section 4.5.1.

Results are presented as a series of graphs. It is important to note that when analysing the graphs, any point along the x-axis represents both the size of the vector and the number of average nonzero elements per row. For example, in a graph of showing products of matrices at 90% sparsity, a characteristic observed at point 5000 on the x-axis could be due to the size of the vector reaching 5000 elements, or could equally be the result of the number of elements in the rows of the matrix reaching $5000 \times 0.1 = 500$ elements. For the purpose of this work, it is imperative to correctly relate performance characteristics of the implementation to either of these two attributes. In addition, for purposes of disambiguation, the term "elements" will refer to 32-bit data types throughout this chapter, even when discussing implementations that make use of wide data types (ie. `int4`, `float2`, ... etc). The terms "coalesced implementation" and "sequential implementation" will refer to "coalesced memory implementation" and "sequential memory implementation" respectively.

The first set of options that will be evaluated are the storage options for the `vec` array followed by the storage options of `ptr` array.

4.4.1 Evaluating `vec` Storage Options

Based on the above discussion, the following implementations were created and evaluated:

- Coalesced Implementations:
  - C1. `coa-f2_memm-memm-memm`
  - C2. `coa-f2_memm-memm-cnst`
  - C3. `coa-f2_memm-memm-text`
- Sequential Implementations:
  - S1. `seq-f4_text-memm-memm`
  - S2. `seq-f4_text-memm-cnst`
  - S3. `seq-f4_text-memm-text`

Matrices of fixed number of rows and varying number of columns from 100 to 10,000 in 100 columns increments were used for the evaluations. Four different levels of sparsity were considered. All matrices were generated with the Random technique discussed in section 4.3.2 and the number of threads per block for all implementations was set at 32 threads per block.

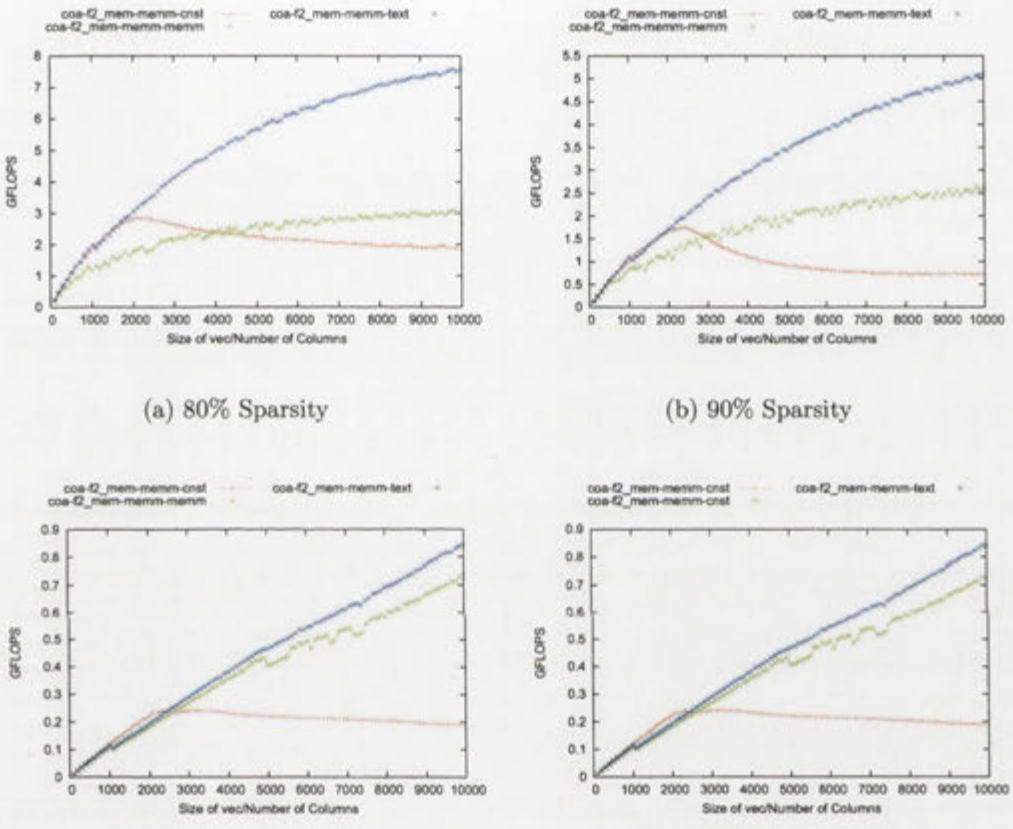
Figure 4.4 presents the results for the coalesced implementations while figure 4.5 presents the results for the sequential implementations.

Looking at the results of coalesced implementations, figure 4.4(a) shows that at 80% sparsity texture memory offers the best performance. The performance of constant memory is on par with texture memory until the x-axis reaches about 2000. However as constant and global memory results never outperform texture memory, they can be ignored for this sparsity. The same results are seen at 90% sparsity in figure 4.4(b) indicating that the size of `vec` has more effect on performance than the average number of nonzero elements per row.

Figure 4.4(c) shows results for 99% sparsity. While texture memory outperforms other options for `vec` storage for the majority of vector sizes, constant memory outperforms texture memory between 1000 and 2500. Global memory can be safely ignored. As these characteristics are repeated at 99.9% sparsity in figure 4.4(d), this again leads to the conclusion that performance is dependent on the size of `vec` rather than the number of elements per row in the sparse matrix.

These results indicate the size of `vec` is key to determining its best storage location. As a result texture and constant memory will both be further evaluated as storage options for the `vec` array depending on the size of the `vec` array.

The results for the sequential implementations at 80% sparsity are presented in figure 4.5(a). These show that constant memory offers the best performance followed by texture memory with global memory offering the worst performance. It is also observed that the performance of all implementations start to drastically decline at a `vec` size of about 1700 until they all perform equally poorly at about 2400. The `vec` size of 1700 corresponds to an average of $1700 \times 0.2 = 340$ nonzero elements per row, while a value of 2400 corresponds to an average of



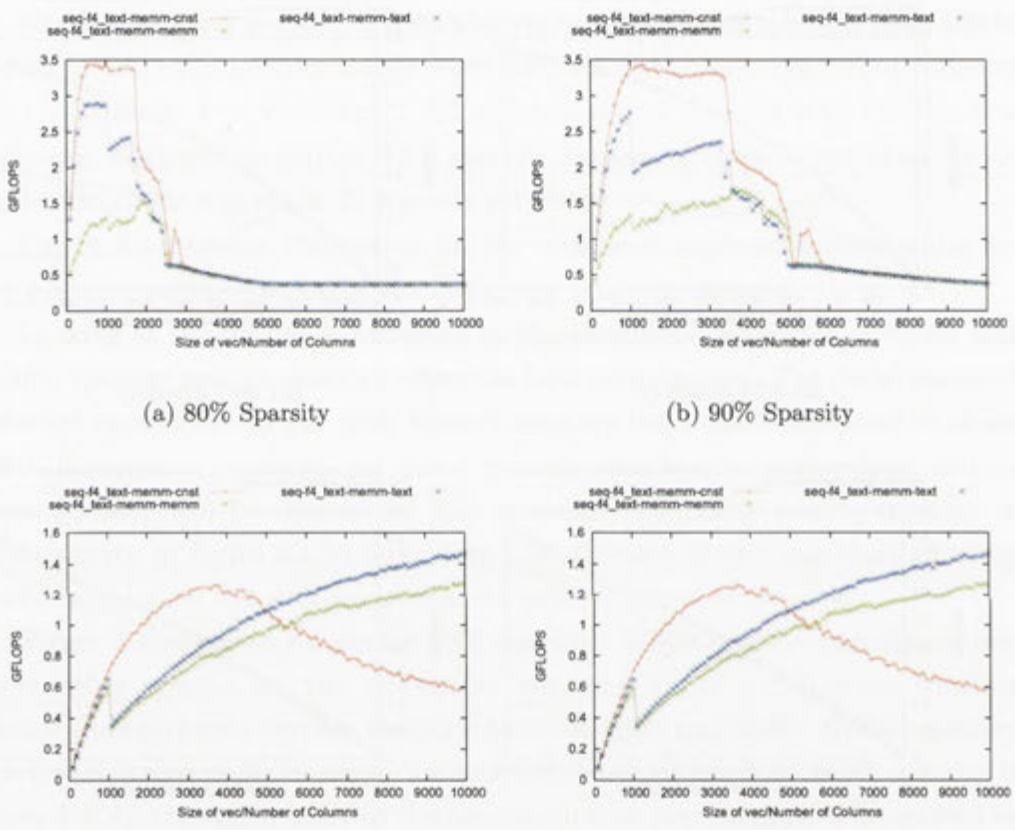
(a) 80% Sparsity

(b) 90% Sparsity

(c) 99% Sparsity

(d) 99.9% Sparsity

Figure 4.4: Evaluating different storage options of the vector for coalesced implementations at different levels of sparsity using Random matrices and 32 threads per block.



(a) 80% Sparsity

(b) 90% Sparsity

(c) 99% Sparsity

(d) 99.9% Sparsity

Figure 4.5: Evaluating different storage options of the vector for sequential implementations at different levels of sparsity.

$2500 \times 0.2 = 500$ nonzero elements per row.

At 90% sparsity the sequential SpMV results shown in figure 4.5(b) are very similar to the results at 80% sparsity, except that they appear to be scaled in the direction of the x-axis by a power of 2. This might be expected given that the average number of elements per row at 80% sparsity is double that at 90% sparsity and so suggest that the observed drop in performance is a result of the average number of elements per row reaching 340. Notice the decline in performance at 90% sparsity starts at a `vec` size of about 3400, which also corresponds to an average of $340 \times 0.1 = 340$ non zero elements per row.

At 99% sparsity the sequential SpMV results in figure 4.5(c) do not show the drastic drop in performance that was observed in the previous graphs. This is due to the fact that at 99% sparsity the largest matrix benchmarked $(10,000 \times 10,000)$ contains only 100 elements per row, on average. There is also a new characteristic that presents itself, which is that the performance of constant memory is outperformed by texture memory at point 5000 along the x-axis.

At 99.9% sparsity, the sequential SpMV results in figure 4.5(d) are almost identical to the results at 99% sparsity in figure 4.5(c). The lack of difference between the two sparsities would suggest that in the absence of sufficient nonzero elements per row, the size of the `vec` array is dominant in terms of the effect it has on performance.

In summary, the optimal storage option for the `vec` array are for coalesced implementations:

- Constant memory if the average number of elements per row is below 2400.
- Texture memory if the average number of elements per row is above 2400.

and for sequential implementations:

- Constant memory if the size of the vector is below 5000.
- Texture memory if the size of the vector is above 5000.

4.4.2 Evaluating `ptr` Storage Options

The significance of which storage option is selected for the `ptr` array is expected to be highly dependent on the elements per row of the sparse matrix since it affects how often it will be accessed. To highlight the difference in performance resulting from available `ptr` storage options, the average number of elements per

row is set as a constant, small value to increase the ratio of `ptr` to `ind`, `val` and `vec` accesses. The sparsity or matrix structure have no effect on the number of `ptr` reads and a minimal effect on the pattern of reads, and are therefore not varied within the evaluations.

The previous sections have argued that the most promising storage options for the `vec` array in terms of performance are constant and texture memory depending on the matrix characteristics. Adding the analyses for the available options for the `ptr` array enumerated in table 4.4, the following list of implementations are created and evaluated with results presented in figure 4.6.

Coalecsed Implementations:

C1. `coa-f2_memm-cnst-cnst`

C2. `coa-f2_memm-memm-cnst`

C3. `coa-f2_memm-text-cnst`

C4. `coa-f2_memm-cnst-text`

C5. `coa-f2_memm-memm-text`

C6. `coa-f2_memm-text-text`

Sequential Implementations:

S1. `seq-f4_text-cnst-cnst`

S2. `seq-f4_text-memm-cnst`

S3. `seq-f4_text-shrd-cnst`

S4. `seq-f4_text-text-cnst`

S5. `seq-f4_text-cnst-text`

S6. `seq-f4_text-memm-text`

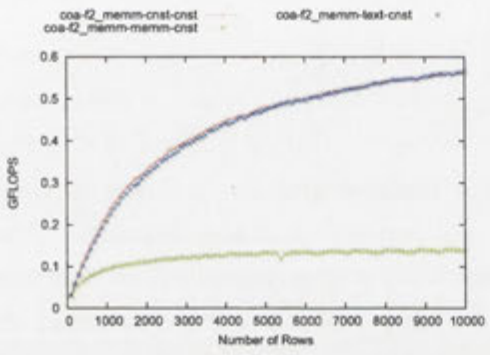
S7. `seq-f4_text-shrd-text`

S8. `seq-f4_text-text-text`

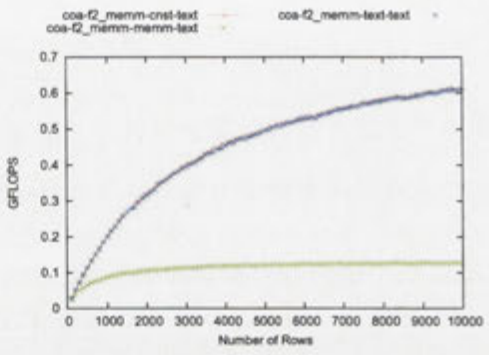
Figure 4.6(a) shows the results from coalesced implementations with the vector stored in constant memory. These results show texture and constant memory preforming equally well as storage options for the `ptr` array, while global memory is far behind. Repeating the benchmarks with the `vec` stored in texture memory gives the results presented in figure 4.6(b); these show the same characteristics as functions of the `ptr` storage type. The poor performance of the coalesced implementations in these two graphs is a result of the low number of elements per row (10 in this case).

Figure 4.6(c) presents sequential results where the `vec` array is stored in constant memory. These results show all implementations performing almost identically with the implementation storing the `ptr` array in texture memory offering a slight advantage in terms of performance. This slight advantage is not evident however, when the `vec` array is stored in texture memory (figure 4.6(d)).

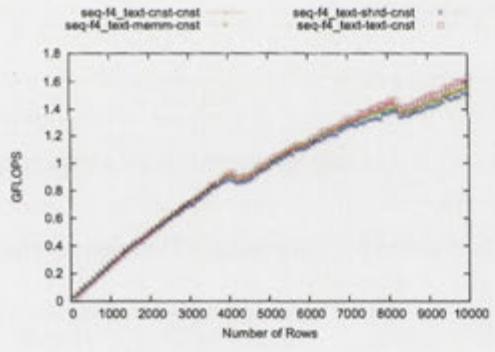
In summary, global memory is not a suitable candidate for `ptr` storage. The



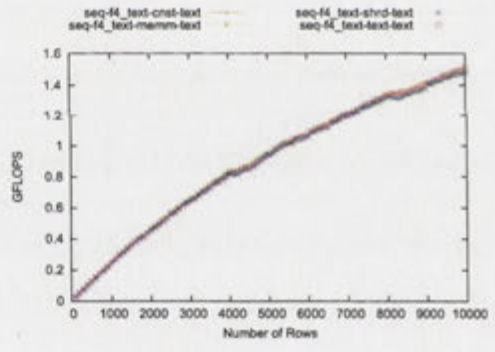
(a) Coalesced implementation. `vec` stored in constant memory.



(b) Coalesced implementation. `vec` stored in texture memory.



(c) Sequential implementation. `vec` stored in constant memory.



(d) Sequential implementation. `vec` stored in texture memory.

Figure 4.6: Evaluating different storage options of the `ptr` array. Matrices are 99% sparse and the number of columns are kept constant at 1000.

use of constant, texture and shared memory for `ptr` storage all resulted in essentially the same performance. There was a small advantage for texture memory over the other storage options in figure 4.6(c) for large number of rows. Texture memory is therefore selected as the optimal storage option for the `ptr` array.

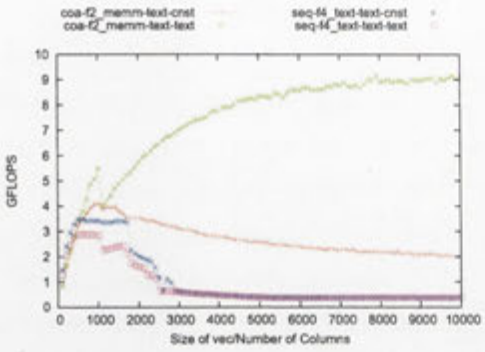
4.4.3 Coalesced v Sequential SpMV Implementations

The objective now is to create a set of heuristics that given parameters such as number of elements per row, sparsity and vector size, will produce the implementation that will deliver the highest performance. Having evaluated the storage options for the various arrays, the remaining decision to be made is when to use the coalesced implementation over the sequential implementation. The results in the previous sections suggest that this will be dependent on the number of elements per row. The `coa-f2_memm-text-text`, `coa-f2_memm-text-cnst`, `seq-f4_text-text-text` and `seq-f4_text-text-cnst` implementations will be benchmarked together to identify the overlap point between these implementations and determine the matrix characteristics that determine which implementation results in the best performance. Random matrices of the same type and dimensions used in the analysis of storage options for the `vec` array, are used here (rows set at 3000, varying columns).

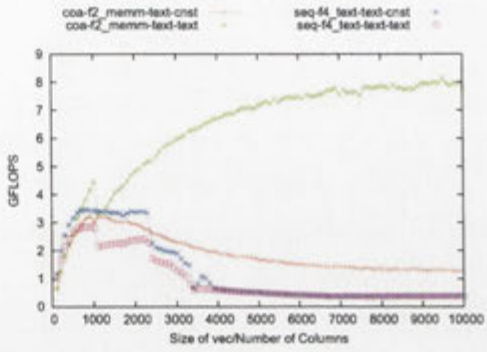
The results are shown in figure 4.7 for a variety of sparsities. Changing the sparsity of a matrix has two effects. This first is that the number of elements per row decreases at any given column size. The second is that the ratio of accessed elements to total size of the `vec` array is reduced. This reduction affects the use of texture and constant memory cache.

The first set of results at 80% sparsity (figure 4.7(a)) shows the `coa-f2_memm-text-text` implementation performing best for all `vec` sizes larger than 500. For `vec` sizes smaller than 500, the `coa-f2_memm-text-text` implementation is slightly outperformed by the `seq-f4_text-text-cnst` implementation. The same observation are made in figure 4.7(b) where the results at 85% sparsity are presented.

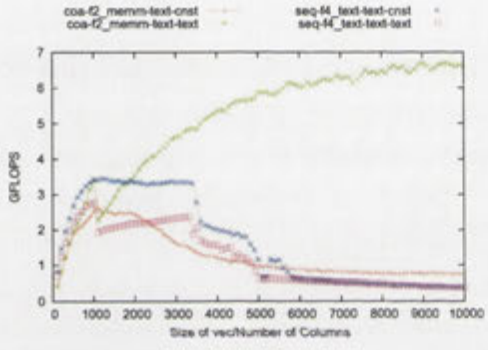
Figures 4.7(c) and 4.7(d) present results at 90% and 95% sparsity respectively. Both set of results are similar to the previous (80% and 85% sparsity) results in that smaller `vec` size perform best with the `seq-f4_text-text-cnst` implementation, while larger `vec` sizes perform better with the `coa-f2_memm-text-text` implementation. The difference is the point at which one implementation outperforms the other. At 80% and 85% sparsity the `coa-f2_memm-text-text` outper-



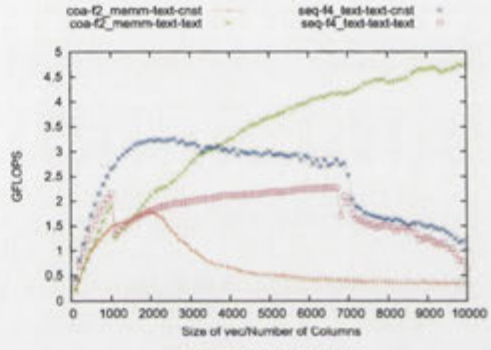
(a) 80% sparse.



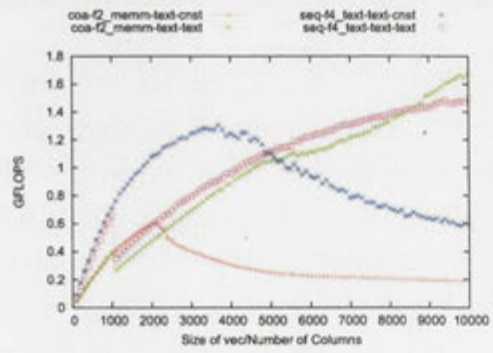
(b) 85% sparse.



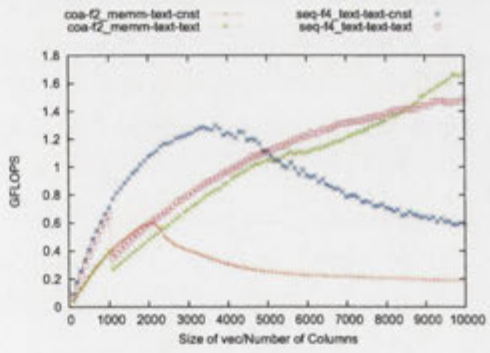
(c) 90% sparse.



(d) 95% sparse.



(e) 99% sparse.



(f) 99.9% sparse.

Figure 4.7: The four most promising implementations at various levels of sparsity.

formed the `seq-f4_text-text-cnst` implementation when the `vec` array reached 500 elements, while at 90% sparsity it was at a `vec` size of 1800 and at 95% sparsity a `vec` size of 3600.

Results at 99% and 99.9% sparsity are essentially identical. Both figures 4.7(e) and 4.7(f) show the `seq-f4_text-text-cnst` implementation performing best for `vec` sizes of under 5000, the `seq-f4_text-text-text` implementation performing best between `vec` sizes of 5000 and 8500, and the `coa-f2_memm-text-text` implementation performing best for `vec` sizes larger than 8500.

Observing all six graphs in figure 4.7, only three implementations performed best in any of the evaluations. These are the `coa-f2_memm-text-text`, `seq-f4_text-text-cnst` and `seq-f4_text-text-text` implementations.

The `coa-f2_memm-text-text` implementation in all of the graphs shows a sharp drop in performance after the number of columns reaches 1000 elements, after which performance gradually increases until at 2000 columns it reaches the same level of performance as it did at 1000 columns. The fact that this occurs at the same `vec` size in all of the graphs indicates that this is related to the size of the `vec` array rather than the average number of elements per row of the sparse matrix. In other words, the `coa-f2_memm-text-text` has two performance curves; one for vector sizes of less than 1000 elements and another for vector sizes greater than 1000 elements.

The first performance curve of the `coa-f2_memm-text-text` implementation (below 1000 columns) crosses with that of the `seq-f4_text-text-cnst` implementation in figure 4.7(a) just after 500 columns and in figure 4.7(b) just after 700 columns. These number of columns both correspond to about 100 elements per row of the sparse matrix.

The second performance curve of the `coa-f2_memm-text-text` implementation (over 1000 columns) consistently surpasses that of the `seq-f4_text-text-cnst` implementation when the number of elements is about 180 elements. This can be observed in figure 4.7(b) at 1200 columns ($1200 \times 0.15 = 180$ elements), in figure 4.7(c) at 1800 columns ($1800 \times 0.10 = 180$ elements) and in figure 4.7(d) at 3600 columns ($3600 \times 0.05 = 180$ elements).

In the absence of large numbers of nonzeros per rows, the size of the `vec` array dominates as the indicator of the best performing implementation as is shown in figures 4.7(e) and 4.7(f) for 99% and 99.9% sparsity.

In summary, the above analysis of the results indicates that the following three implementations make most efficient use of the GPU's capabilities, with each performing best for a different set of matrix characteristics.

1. `coa-f2_memm-text-text`

2. `seq-f4_text-text-cnst`

3. `seq-f4_text-text-text`

4.4.4 Multiple Row Implementations

The implementations discussed in the previous sections all process a maximum of one row per thread or block (depending on whether the implementation is coalesced or sequential). In order to evaluate the effect of processing multiple rows each thread or block (depending on the implementation) was assigned multiple rows. Rows can be assigned in a block or cyclic pattern. A block pattern would assign consecutive rows to each thread or block while a cyclic pattern would assign consecutive rows to different threads or blocks. To account for multiple rows, the names of the implementations were modified to:

`[coa/seq]-[sr,mr_c,mr_b]-[f1,f2,f4]_[val,ind storage]-[ptr storage]-[vector storage]`

where `sr` represents a single row implementation, `mr_c` represents a cyclic, multi-row implementation and `mr_b` represents a blocked, multi-row implementation.

Both blocked and cyclic options were implemented and evaluated for the `coa-f2_memm-text-text` and `seq-f4_text-text-text` implementations. The coalesced implementation was evaluated with 60,000 × 2,000 matrices at 95% sparsity resulting in an average of 100 elements per row. The sequential implementation was evaluated with 60,000 × 500 matrices at 95% sparsity resulting in an average of 25 elements per row. All three generation methods described in section 4.3.2 were used with 2, 5, 10 and 20 rows per thread or block .

Coalesced Implementation Results

Table 4.5 presents the results for the blocked (`coa-mr_b-f2_memm-text-text`) and cyclic (`coa-mr_c-f2_memm-text-text`) coalesced multi-row implementations.

When only one row is assigned per block, both blocked and cyclic implementations perform at 4.8 GFLOP/s regardless of the matrix structure. Increasing the assigned number of rows per block results in an increase in performance for both implementations with a slight advantage to the blocked implementation. As a result, the blocked option is identified as the better option for the coalesced implementations.

Table 4.5: Comparison between performance (in GFLOP/s) of consecutive and strided multiple row coalesced memory implementations. Evaluation matrix is 60,000×2,000 @ 95% sparsity.

fill	Implementation	Rows/Block				
		1	2	5	10	20
Random	cyclic	4.8	5.0	5.0	5.1	5.1
	blocked	4.8	5.0	5.1	5.3	5.3
Constant row	cyclic	4.8	5.0	5.1	5.1	5.2
	blocked	4.8	4.9	5.2	5.3	5.3
Structured	cyclic	4.8	5.0	5.1	5.2	5.2
	blocked	4.8	5.0	5.3	5.3	5.3

Sequential Implementation Results

Table 4.6 presents the results for the blocked (`seq-mr_b-f4_text-text-text`) and cyclic (`seq-mr_c-f4_text-text-text`) sequential multi-row implementations.

Table 4.6: Comparison between performance (in GFLOP/s) of consecutive and strided multiple row sequential memory implementations. Evaluation matrix is 60,000×500 @ 95% sparsity.

fill	Implementation	Rows/Thread				
		1	2	5	10	20
Random	cyclic	3.8	3.8	3.9	3.8	3.8
	blocked	3.9	3.6	2.9	2.0	0.8
Constant row	cyclic	3.9	3.9	4.0	4.0	4.0
	blocked	4.0	3.6	2.9	1.7	0.8
Structured	cyclic	3.7	3.7	3.7	3.7	3.8
	blocked	3.8	3.4	2.8	1.6	0.8

When assigning one row per thread, both cyclic and blocked sequential implementations result in similar levels of performance. As the number of assigned rows per thread increases the cyclic implementation results in negligible variances in performance, while the blocked implementation shows drastic performance degradation. This is true regardless of matrix structure. As a result the cyclic option is identified as the best option for sequential multi-row implementations.

4.4.5 Evaluation of Selected Implementations

The previous analysis attempted to realise a set of implementations S so that for any given matrix, the best performing SpMV implementation would belong to S. For simplicity, this set will be referred to as the Best Performing Set (BPS). The implementations that were chosen from the series of synthetic matrix benchmarks are:

- `coa-mr_b-f2_memm-text-text`
- `seq-mr_c-f4_text-text-text`
- `seq-mr_c-f4_text-text-cnst`

To evaluate this BPS, a large number of other implementations were created and compared against. To create the large pool of implementations all options outlined in section 4.2 were implemented and tested, with the exception of the use of shared memory for the `ptr` array as the previous results did not show any performance gain resulting from its use (see figure 4.6). This results in a total of 335 implementations (including single and multi-row implementations).

Rather than evaluating these implementations with synthetically generated random matrices, the Florida sparse matrix collection [17] was used as it provides real world matrices from various scientific fields. When this evaluation was undertaken (November 2008), the Florida collection contained over 2200 matrices, however only matrices containing 10^5 to 10^7 nonzero elements were selected as candidates for computing on the GPU ($< 10^5$ elements were considered too small). This resulted in 747 matrices which were further reduced to 735 matrices by removing the matrices that would not fit on the GPU after padding the matrix to accommodate the `coa-mr_c-f4_memm-text-text` implementation as it requires the largest amount of padding.

The BPS presented above was then be evaluated by comparing the difference between the performance of the optimal implementation (from all of the 335 implementations) and the best performing implementation in the BPS. This was repeated for each matrix in the set of 735 selected matrices. For any given BPS this is an ideal performance result since it assumes that ever sparse matrix can be correctly mapped to the best implementation in the BPS.

Table 4.7 presents the results of that evaluation in the form of the average difference in performance along with the standard deviation, median and maximum difference in MFLOP/s.

Table 4.7: Difference in performance (in MFLOP/s) between the set of selected implementations from the previous analysis and the optimal implementations.

Set of Implementations	Difference in Performance			
	average	stdev.	median	maximum
original set	61	130	19	1383

The average difference between the performance of the optimal implementation and the BPS over all the 735 datasets is 61 MFLOP/s, with a median of 19 MFLOP/s, a standard deviation of 130 MFLOP/s and a maximum difference of 1383 MFLOP/s.

It is important to recognise that while the results in table 4.7 are for a specific BPS of size 3, it says nothing about whether this selection is better or worse than any other set of three implementations. It is possible that an other selection of three implementations have better performance characteristics than the three implementations listed above. This issue is addressed in table 4.8 and elaborated below.

The maximum possible performance for a set of three implementations can be obtained by creating all the combinations of size 3 from the pool of implementations mentioned above. The full set of 335 implementations cannot be used to create these combinations, as the resulting combinations will quickly become too numerous, with a BPS of three giving $\binom{335}{3} = 6,209,895$ possible combinations. To reduce this size a smaller pool of possible implementations for the combinations was created. This pool contains all the implementations that were optimal for any dataset at least twice. From this list, all single row implementations are removed and substituted with the two complementing multi-row implementations (see section 4.4.4). This results in a list of 34 implementations, which for a BPS of three gives $\binom{34}{3} = 5,984$ combinations. All the generated sets are then evaluated in the same manner as the original selection. The average, standard deviation, median and maximum difference between the performance of the optimal implementation and the best performing implementation in the set are calculated. The best preforming combination where then defined as that which either has:

1. The smallest average, or

2. The smallest maximum.

To avoid confusion the process of selecting the optimal implementation on the basis of the smallest average difference in performance will be referred to as the "first criteria", while the process of selecting the optimal implementation on the basis of the smallest maximum difference will be referred to as the "second criteria".

This approach was then extended to include the BPS from sets of size one to seven with the results given in table 4.8. As before, the optimal performance for each set is presented in the form of the average, standard deviation, median and maximum difference in the performance between the optimal implementation in the set, and the overall optimal implementation. Table 4.8 presents results using both criteria. Also included as row one are the results of evaluating the original proposed BPS with the same values repeated for both criteria.

Table 4.8: Difference in performance (in MFLOP/s) between each BPS from sets of sizes one to seven and the optimal implementations.

Number of Implementations	Criteria							
	"first criteria"				"second criteria"			
	average	stdev.	median	maximum	average	stdev.	median	maximum
original selection	61	130	19	1383	61	130	19	1383
1	630	1397	384	10103	1603	1106	401	5649
2	163	307	22	1498	163	307	22	1498
3	53	126	15	1383	136	256	22	1240
4	43	122	10	1383	44	84	15	725
5	34	77	10	725	42	78	15	615
6	31	77	5	725	32	70	9	615
7	28	74	5	725	29	67	9	615

When selecting the best implementation on the basis of the "first criteria", the implementation that produces the best results is the `seq-mr_c-f4_text-memm-text` implementation with an average performance difference of 630 MFLOP/s, a standard deviation of 1397 MFLOP/s, a median of 384 MFLOP/s and a maximum difference of 10,103 MFLOP/s. The results of adding a single implementation (limiting the set to two implementation) is an average performance difference of 163 MFLOP/s, a standard deviation of 307 MFLOP/s, a median of 22 MFLOP/s and a maximum difference of 1,498 MFLOP/s. As the numbers of implementations in a set increases the difference in performance between the implementations in the set and the optimal implementations continues to decrease. The decrease in average, median and maximum are very noticeable

when moving from one to three implementations per set. The move from three to four implementations creates a less noticeable decrease in average and median and the maximum remains constant. Increasing the number of implementations from four to five shows another large decrease in maximum, although the average doesn't change much. Increasing the size of the sets from six upwards results in minimal small changes.

Selecting the BPS on the basis of the "second criteria" shows similar trends in that the difference in performance decreases as the number of implementations increase. The best single implementation is the `coa-mr_b-f1_memm-text-text` implementation with an average performance difference of 1,603 MFLOP/s, a standard deviation of 1,106 MFLOP/s, a median of 401 MFLOP/s and a maximum difference of 5,649 MFLOP/s. Increasing the number of implementations from one to two results in a drastic reduction in the difference between the BPS and optimal implementations. Further increasing the number of implementations continues to produce noticeable reductions up until four implementations where the magnitude of the difference stabilises to an extent. Sets of four implementations or more are not affected much by the addition of an extra implementation.

Comparing the two selection criteria shows the difference decreases as the number of implementations increase. As expected the first criteria results in smaller average differences but with much larger maximum differences.

When comparing the original selection of three implementations to the BPS of three implementations (resulting from selecting the lowest average) in table 4.8 show the results of the original selection to be very similar. Table 4.9 highlights the difference in the implementations used in the original selection and the implementations in the optimal set of three implementations.

Table 4.9: Comparison between original selection and optimal set of size 3.

Original selection	BPS of size 3	comment
`coa-mr_b-f2_memm-text-text`	`coa-mr_c-f2_memm-text-text`	different
`seq-mr_c-f4_text-text-text`	`seq-mr_b-f4_text-text-text`	different
`seq-mr_c-f4_text-text-cnst`	`seq-mr_c-f4_text-text-cnst`	identical

Both sets in table 4.9 contain one coalesced implementation and two sequential implementations. Comparing the implementations in the two sets show that they are identical in their basic structure, the only difference being the preference for cyclic or blocked multirow distribution patterns. The small differences in

performance between the two sets are most likely due to the results of the wide range of matrix structures used in the evaluations. The difference between these two sets are only eight MFLOP/s in average performance difference.

When comparing the best possible set of three implementations to sets of other sizes, it is important to note that each added implementation will increase the challenge of mapping matrix attributes to the best performing implementation. Given that the performance difference decreases very slowly when increasing the number of implementation beyond four, suggests that four implementations provides a balance between performance and number of implementations.

4.5 Mapping Matrices to their Optimal Implementation

From the above analysis, the optimal set of four implementations was:

- `coa-mr_b-f2_memm-cnst-cnst`
- `coa-mr_c-f2_memm-text-text`
- `seq-mr_c-f4_text-text-cnst`
- `seq-mr_c-f4_text-text-text`

The objective is now to create a decision tree that identifies which implementation to use based on the matrix attributes (rows, columns and number of nonzero elements). Weka [22], a collection of machine learning algorithms for data mining tasks containing tools for data classification was evaluated to build a variety of possible decision trees. Random Forests and Fast decision tree learners were applied to the data resulting from benchmarking the 335 to create decision trees. The problem with these trees was that they were over fitted with up to 100 levels and up to 1000 leaves. Limiting the number of implementations to the 34 implementation previously discussed, still resulted in over fitted trees with 10 levels and up to 300 leaves. Conversely, limiting the number of levels in the tree resulted in trees with low percentages of correctly classified instances. In addition, the software only considers whether the classification was correct or not and therefore considers a classification resulting in a difference of 10 MFLOP/s to be equal to an incorrect decision resulting in a difference of 1000 MFLOP/s.

For these reasons a simpler decision tree based on empirical analysis was attempted that took into consideration the magnitude of difference rather than the number of correct classifications.

The analysis conducted in section 4.2, showed that certain matrix attributes affected different implementation options in terms of which performed best for a matrix. The analysis identified the average number of nonzeros per row as the parameter that distinguishes whether to use coalesced or sequential implementations. The number of columns was shown to have direct effect on which memory type to use for the `vec` array. The `ptr` array was shown to perform equally well for texture and constant memory. Building upon this knowledge a simple tree was created. This tree is presented in figure 4.8.

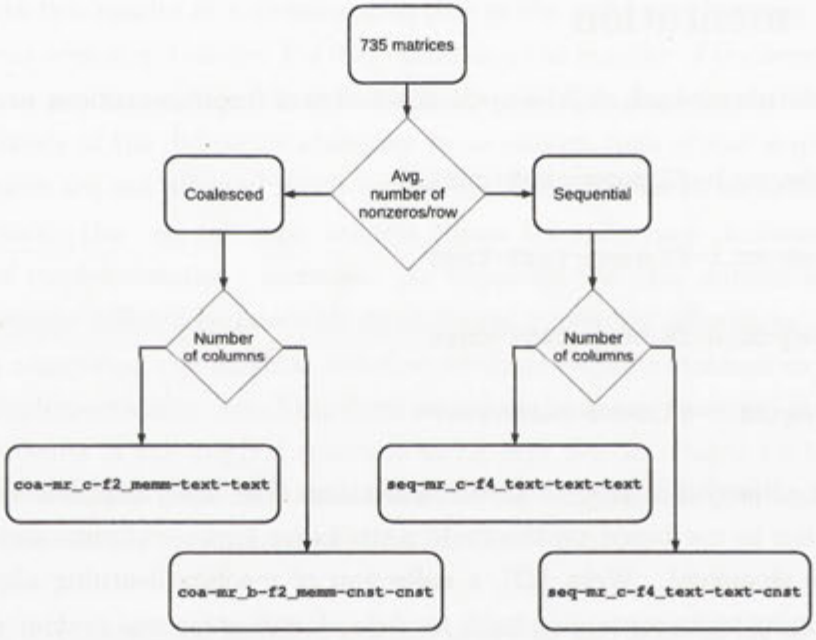
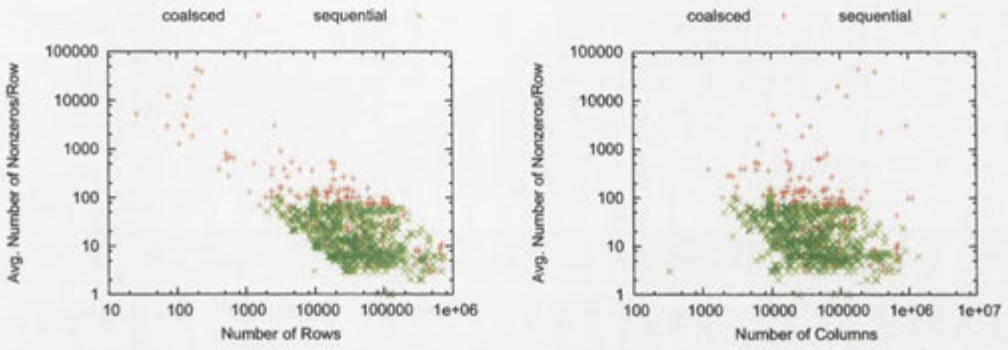


Figure 4.8: A simple decision tree to distinguish the best performing implementations based on the matrix parameters.

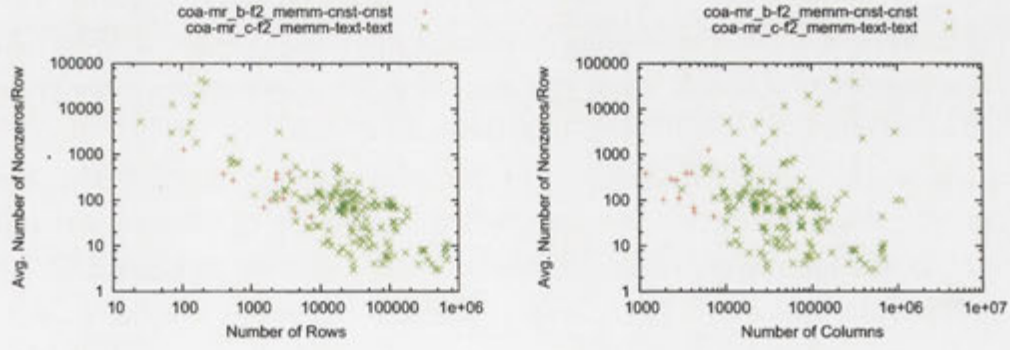
The tree includes parameters to determine whether to go down a potential branch but doesn't state the actual values. A search for the best values for these attributes can be done by comprehensively searching all the possible combinations of these attributes and selecting the combination that offers the best overall performance. A comprehensive search is possible since the use of constant memory imposes an upper limit of 16000 elements on the size of the `vec` array.

Rather the blindly search all possible values, the performance of these implementations are analysed to find likely starting values or ranges. Figure 4.9 presents plots of all the 735 matrices used in the evaluation of the implementations. Each matrix is plotted by its attributes with different shaped points representing whether the best performing implementation within the BPS is coalesced or sequential. Both graphs (figure 4.9) indicate that the value for the average number of nonzeros per row should be between 30 and 100.



(a) Rows X Avg. nonzers/row. (b) Columns X Avg. nonzers/row.

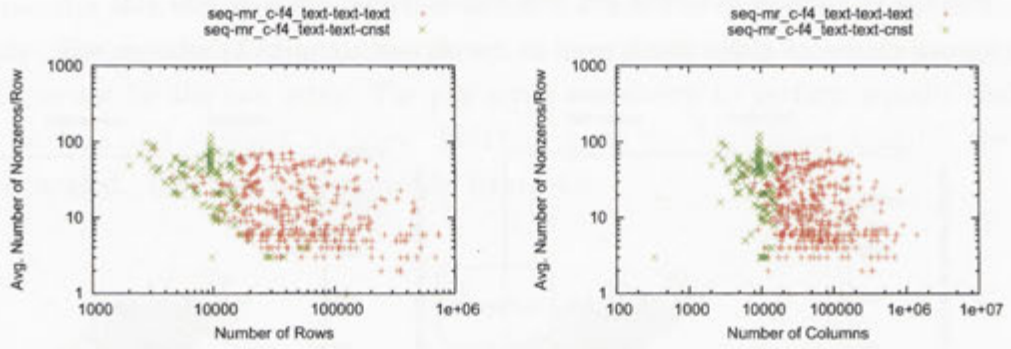
Figure 4.9: Looking for patterns to distinguish between the coalesced and sequential implementations in the BPS of size 4.



(a) Rows X Avg. nonzers/row. (b) Columns X Avg. nonzers/row.

Figure 4.10: Looking for patterns to distinguish between the different coalesced implementations in the BPS of size 4.

Figure 4.10 attempts to distinguish between the two coalesced implementations. Figure 4.10(a) doesn't indicate any statistical significance for the number of rows in this decision. Figure 4.10(b) indicates that the number of columns plays a more significant role. The number of columns at where one implementation outperforms the other is somewhere between 2000 and 6000 columns.



(a) Rows X Avg. nonzers/row.

(b) Columns X Avg. nonzers/row.

Figure 4.11: Looking for patterns to distinguish between the different sequential implementations in the BPS of size 4.

Finally, figure 4.11 investigates the patterns that differentiate between the two sequential implementations. Figure 4.11(a) shows a weak relation between the number of rows and the better implementation. This weak relation is most likely a result of the fact that most matrices used for evaluation are square, rather than there being any real link between the attribute and the implementations. The distinction between the two implementations is some what clearer in figure 4.11(b). The number of columns where one sequential implementation outperforms the other lies between 10,000 and 11,000 columns.

From the previous analysis each combination of these values were then used in an evaluation process and the combination that results in the best performance was chosen and used to create the final decision tree given in figure 4.12.

Using this decision tree the actual performance results are collected and presented in table 4.10 along with the ideal performance if all matrices were perfectly mapped to the best implementation in the BPS.

Table 4.10 shows that the results achieved with the decision tree are quite far from the ideal case. The average difference and standard deviation are 3× the ideal case, while the median and maximum difference are 2× the ideal case.

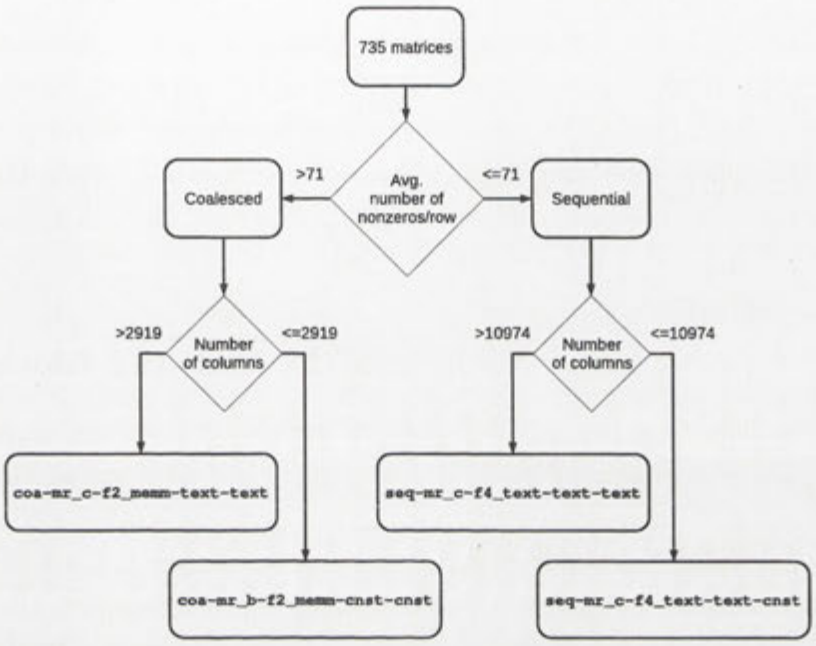


Figure 4.12: Final decision tree.

Table 4.10: Difference in performance (in MFLOP/s) the ideal BPS of size four and the result of using the decision tree.

Context	Average	Stdev.	Median	Maximum
Ideal	44	84	15	725
From Tree	119	242	28	1664

While there must exist other sets, attribute values (such as the median and standard deviation of the number of nonzero elements per row) and decision trees or other machine learning methods that would have produced better results, the realisation of such optimal cases is beyond the scope of this work.

In 2008, Ryoo et al published results from using pareto-optimisation techniques to search the optimisation space for four kernels based on efficiency and utilisation metrics [49]. The optimisation space include the number of threads per block, tiling sizes, loop unrolling and many other aspects. While the approach used by Ryoo et al is quite different to that taken here, there is similarity in the task being solved. [2]

4.5.1 Selecting the Number of Threads per Block

The previous sections identified the optimal memory options and access methods for the SpMV implementations. The best size for the BPS and the implementations to use for it were also identified. In the previous section a `blackbox` implementation was created that would select an implementation based on matrix attributes using a decision tree. The remaining implementation option identified in section 4.4 that has not been evaluated is the number of threads per block which were set to 32 for all previous benchmarks.

To select the best number of threads per block for each of the four implementation that make up the *blackbox* implementation, it is evaluated with the set of 735 sparse matrices with the number of threads per block set at 16, 32, 64, 128, 256 and 512. For each dataset the internal implementation selected by the *blackbox* implementation was noted. This data was compiled into table 4.11 where for each of the internal implementations, the number of datasets that performed best with a particular number of threads per block is noted.

Table 4.11 shows that the `coa-mr_b-f2_memm-cnst-cnst` was selected 7 out of 735 times. Of these half performed best with 32 threads per block while the other half performed best with 64 threads per blocks. The `coa-mr_c-f2_memm-text-text` was chosen 95 times with 32 and 64 also being the number of threads that performed best the majority of the time. The `seq-mr_c-f4_text-text-cnst` implementation was selected 101 times and the majority of cases saw either 64 or 128 threads per block as the optimal value. Finally the `seq-mr_c-f4_text-text-text` implementation performed best 532 times with the best results appearing at 256 threads per block followed closely

[2] We are grateful to the examiner for bring this work to our attention.

Table 4.11: Evaluating the 4 implementation as to the effect of using different numbers of threads per block. Each cell contains the number of matrices that performed best with the specified number of threads per row.

Implementation	Times selected						
	total	number of threads per block					
		16	32	64	128	256	512
`coa-mr_b-f2_memm-cnst-cnst`	7	0	4	3	0	0	0
`coa-mr_c-f2_memm-text-text`	95	3	48	30	10	3	1
`seq-mr_c-f4_text-text-cnst`	101	7	9	34	32	10	9
`seq-mr_c-f4_text-text-text`	532	13	89	91	124	140	75

by 128 threads per block. However a large number of cases perform best at 32 and 64 threads per block.

Searching through the results for the combination of settings that produced the best average performance, determines the optimal number of threads to use for each of the four implementations. It was found that all the implementations should use 64 threads per row except the `seq-mr_c-f4_text-text-cnst` implementation which should use 128 threads per block. Overall, while the performance of individual datasets are affected quite substantially by varying the number of threads per block, the overall performance is not greatly affected.

4.6 CPU v `blackbox` Performance

The minimal, maximum, median and average overall effective performance achieved with the CPU and *blackbox* implementation across all the 735 matrices used for evaluations are presented in table 4.12. The CPU results were obtained with PETSc [3] [6, 7], the same library that is used by the BMRM application to compute matrix-vector products. (PETSc is further expanded upon in the next chapter.) The standalone performance of the internal implementations that make up the *blackbox* implementation are also presented.

The results from table 4.12 show the *blackbox* implementation performing on average 7.85× the CPU and at maximum of about 20× the CPU. The structure of matrices that resulted in the poor performance in table 4.12 had huge variations in the number of nonzero elements in each row. An example of this is the

[3] Portable, Extensible Toolkit for Scientific Computation

Table 4.12: Minimal, maximum, median and average performance of the GPU blackbox SpMV implementation in GFLOP/s.

Implementation	Minimum	Maximum	Median	Average
CPU	0.04	0.43	0.18	0.2
blackbox implementation	0.06	8.59	1.24	1.57
`coa-mr_b-f2_memm-cnst-cnst*`	0.67	7.03	1.26	1.69
`coa-mr_c-f2_memm-text-text`	0.07	8.59	0.62	1.09
`seq-mr_c-f4_text-text-cnst*`	0.06	4.37	1.93	1.92
`seq-mr_c-f4_text-text-text`	0.06	3.85	1.04	1.23

*These figures are calculated from the smaller subset of datasets that satisfy the constraints of constant memory for the `vec` array.

ASIC_320k matrix where 2 rows contain about 200,000 elements, yet the average number of nonzero elements per row is only 8. Table 4.13 presents a sample of 4 matrices where the maximum number of elements per row are about $40,000\times$ the average number of nonzeros per row. These are followed by 4 matrices where the maximum is between $2\times$ to $25\times$ the average number of nonzeros per row.

Table 4.13: The performance of pathological matrices compared to the performance of of matrices where the difference between the maximum and average number of elements per row is not as great.

Dataset			Nonzeros per Row			Performance
name	rows	columns	min.	max.	avg.	(GFLOP/s)
lp1	534388	534388	2	249643	3	0.06
ASIC_680k	682862	682862	1	395259	6	0.06
rajat29	643994	643994	1	454521	8	0.96
rajat30	643994	643994	1	454746	10	0.11
inline_1	503712	503712	18	843	73	2.99
ramage02	16830	16830	45	270	170	4.71
psmigr_2	3140	3140	3	2293	172	2.51
gupta3	16783	16783	33	14672	556	8.56

These results show that the structure of the matrix is critical to the performance of the SpMV product on the GPU. This is mainly since each row is assigned to either a block or thread on the GPU. Large number of elements per

row lead to a single thread or block executing for large amounts of time while all the other threads/blocks are idle. Performance of such matrices could be improved by investing in customised matrix formats and implementations.

4.7 Recent Related Work

Very recently, Bell and Garland [9] published their efforts to design efficient SpMV kernels for the GPU. The GPU used was an NVIDIA GeForce GTX 280 GPU. This system is compared to the GeForce 8800 GTX used in the work in table 4.14.

Table 4.14: Comparison between the theoretical peak attributes of the GeForce 8800 GTX and the GTX 280 GPUs.

GPU	Performance*	Internal Bandwidth	Number of SPs
GeForce 8800 GTX	321 GFLOP/s	80 GB/s	128
GeForce GTX 280	581 GFLOP/s	131 GB/s	240
Improvement	181%	164%	188%

* This number does not take into consideration texture related capabilities as they are rarely used in general purpose programs.

From the comparison in table 4.14, the newer GTX 280 is capable of roughly 180% of the performance of a 8800 GTX in terms of FLOP/s and 164% in terms of the internal memory bandwidth. In addition to the raw performance figures, the GTX 280 also contains many other improvements. For example NVIDIA have increased the number of registers in the SMs and the maximum number of threads that can reside on the SMs. They have also reduced the penalty for unaligned memory accesses. We would expect the *blackbox* implementation on the GTX 280 to achieve roughly 160% increase in performance over the 8800 GTX.

Bell and Garland [9] investigated a variety of sparse matrix formats. These include the diagonal format, the ELLPACK format, the coordinate format, the packet format, the compressed sparse row format, and a hybrid format between the coordinate and ELLPACK formats. Each of these formats requires a SpMV kernel and in the case of the CSR format, both a sequential and coalesced CSR implementation were created. The authors also investigated the use of texture memory for the vector in the SpMV products and identified a performance gain through its use.

The authors investigated the single precision performance of structured and

unstructured sparse matrices separately. For the structured sparse matrices, the test cases were composed of standard discretisation of Laplacian operations in 1, 2 and 3 dimensions. These sparse matrices were all comprised of a number of diagonals. Storing these matrices in the diagonal format reduces the memory footprint of the matrix by a factor of 2. As memory bandwidth is the bottleneck for SpMV products, the reduction in the memory foot-print translates into an increase in performance. Bell and Garland [9] reported a maximum of 36 GFLOP/s for these structured matrices with the diagonal implementation.

Unstructured matrices were represented by the set of 14 matrices that were used by Williams et al. [60] in their paper. The attributes of these datasets are presented in table 4.15. For 12 of these matrices the best performance was achieved with the hybrid implementation, while the remaining two performed best with the coalesced CSR implementation. The hybrid implementation achieved a maximum of 22.3 GFLOP/s, while the CSR implementation achieved a maximum of 16 GFLOP/s.

Table 4.15: Datasets used by Williams et al. as well as Bell and Garland.

Dataset	Rows	Columns	Avg. Nonzeros/Row
Dense	2,000	2,000	2000
Protein	36,417	36,417	119
FEM-Sphr	83,334	83,334	72
FEM-Cant	62,451	62,451	64
Tunnel	217,918	217,918	53
FEM-Har	46,835	46,835	50
QCD	49,152	49,152	39
FEM-Ship	140,874	140,874	55
Econom	206,500	206,500	6
Epidem	525,825	525,825	4
FEM-Accel	121,192	121,192	22
Circuit	170,998	170,998	6
webbase	1,000,005	1,000,005	3
LP	4,284	1,092,610	2633

In addition Bell and Garland [9] also evaluated double-precision SpMV products on the GTX 280 GPU and achieved 16 GFLOP/s for the structured, diagonal matrices and a maximum of 13.9 GFLOP/s and 14.2 GFLOP/s for the hybrid and

CSR implementations respectively. These results compare favourably with the results published by Williams et al. [60] and shows the median GPU performance to be about 10× the leading CPU performance.

While the work by Bell and Garland [9] does compare the performance of multiple storage formations (or the implementation for these formats) and the effect of utilising texture memory for the vector of these SpMV products it does not investigate many of the other implementation options that CUDA provides.

For example, this work investigated the use of multiple data types, the use of texture and constant memory for not only the vector but the three arrays that make up the CSR matrix. A large amount of analysis was provided as to how the performance of the SpMV products if affected when the various data types and memory options are used for each of these arrays. The effect changing the number of threads per block was also investigated in this work. These issues were not discussed in the work by Bell and Garland [9].

The work by Bell and Garland [9] created two CSR implementations while this work implemented 335 different implementations. The evaluation of these implementations was performed with not 14 but 735 implementations across a much wider spectrum of applications.

The cost of copying the matrix and vector to and from the GPU is only important if the bulk of the application runs on the GPU and only the matrix-vector products are performed on the GPU. In any case Bell and Garland [9] do not provide the cost of these operations which should be roughly half the equivalent cost presented in this work as the GTX 280 GPU used by Bell and Garland [9] utilises a PCIe 2.0 bus which has 2× the performance of the PCIe 1.0 bus utilised by the 8800 GTX GPU. This work also provides a large amount of analysis to identify how multiple CSR implementations can be used together to create a single *blackbox* implementation which selects the correct implementation based on the matrix attributes.

In terms of performance, figure 4.13 provides the kernel performance (for comparison with the work by Bell and Garland [9] as well as the effective performance.

One of the key differences in the CSR results in this work and the results by Bell and Garland [9] is that while the SpMV results are based on the nonzero elements in the unmodified matrices, the matrices were padded with zeros to a multiple dependent on the implementation. On the 8800 GTX GPU this is essential to achieve coalesced memory performance. The results produced by by Bell and Garland [9] as this is not required with the GTX 280 GPU.

Much of the future work identified by Bell and Garland [9] is consistent with

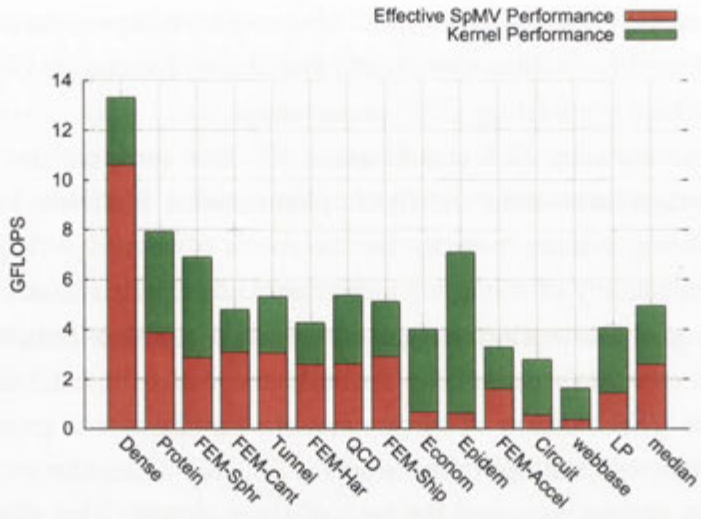


Figure 4.13: The GPU Performance for the 14 datasets used by Williams et al. For each dataset the time for the SpMV kernel as well as the effective performance is provided.

our vision. The use of multiple formats for some of the matrices that performance badly on the GPU, the use of blocked formats and the results of multiplying the sparse matrix by multiple vectors at the same time are all of interest.

4.8 Summary and Conclusion

SpMV results on the GPU show an average speedup of 7.85× and a maximum speedup of about 20× over the CPU when factoring in the overheads associated with the use of the GPU. Applications that are completely ported to the GPU would not incur these penalties and would therefore see an speedup of around 15.85× (maximum speedup would be about 31.24×) and would be expected to double if performed on a 2009 model GPU like the GeForce GTX 285. SpMV products can also be computed jointly between the GPU and the CPU which would result in even better overall performance. Offloading SpMV products to the GPU can be a very viable option depending on the structure of the sparse matrix and the number of SpMV iterations.

The results presented in this chapter compare favourably with the results presented by Bell and Garland [9] on a GTX 280 GPU and Williams et al. [60] on high end CPUs and other novel architectures. Even if the GPU and CPU performed equally well in terms of SpMV performance, the cost of adding three

GPUs for example to a system is much less then the cost of 3 new systems.

4.9 Results on a GTX 295 GPU

Without changing or updating any of the code, the complete set of benchmarks were applied to a newer GTX 295 GPU. Table 4.16 shows the difference between the 8800 GTX and GTX 295 in terms of theoretical performance. The GTX 295 actually consists of 2 GPUs on a single PCIe card. The numbers given are for a single GPU of the GTX 295 rather then the aggregate theoretical performance of the entire card.

Table 4.16: Comparison between the theoretical peak attributes of the GeForce 8800 GTX and a single GPU of the GTX 295.

GPU	Performance	Internal Bandwidth	Number of SPs
GeForce 8800 GTX	321 GFLOP/s	80 GB/s	128
GeForce GTX 295 (single GPU)	555 GFLOP/s	104 GB/s	240
Improvement	173%	130%	188%

Table 4.17: Improvement of the GTX 295 over the 8800 GTX in terms of the *blackbox* SpMV implementation performance. The numbers here can be compared to the hardware specification to deduce the scalability of the *blackbox* implementation

Implementation	Maximum	Average
blackbox implementation on the 8800 GTX	8.59	1.24
blackbox implementation on the GTX 295 (single GPU)	11.0	2.3
Improvement	128%	150%

The results for the maximum and average performance achieved on both the 8800 GTX and the GTX 295 (using a single GPU) are presented in table 4.17. Since the CSR SpMV algorithm is bandwidth bound the fact that an increase in internal memory bandwidth of the GPU of 130% leads to a 128% increase in the maximum performance in not surprising. The increase in the average performance of 150% (a 20% increase over memory bandwidth improvement) is

most likely due to the improvement in the coalescing capabilities of the newer card. Based on these results the same code would be expected to scale to future GPUs relative to the increase in internal bandwidth.

Taking into account the 164% bandwidth improvement (as reported in table 4.14) the GTX 280 used by Bell and Garland has over the 8800 GTX GPU and these new results, the performance of the CSR SpMV *blackbox* implementation can safely be predicted to increase by at least 160%. This suggests that the *blackbox* implementation presented in this thesis would outperform the CSR implementation presented by Bell and Garland [9]. However further benchmarking on the same hardware would be needed for a completely fair comparison.

Chapter 5

SML Application

The Bundle Methods for Regularised Risk Minimisation (BMRM) application described in section 2.3 is an open source, modular and scalable convex solver for many machine learning problems cast in the form of regularised risk minimisation problem [54]. BMRM utilises PETSc [7] for the matrix-vector operations.

The current objective is to only perform the matrix-vector products on the GPU and perform all other computations on the host. The PETSc objects are therefore intercepted and copied to the GPU. The matrix-vector products are then performed on the GPU and the result copied back to the CPU and packed into a PETSc object.

As previously noted, each iteration of the computation requires both a normal and transpose matrix-vector product. For dense matrix-vector products, the transpose only alters the order of reading the matrix (see section 3.3.1). In the case of sparse formats, generation of the transpose matrix is a non-trivial operation so it is pre-formed and stored on the GPU along with the original matrix.

The steps taken to perform the first matrix-vector product are as follows:

1. Step 1: Copy the matrix to the GPU.
   (a) Convert matrix elements from `double` to `float`.
   (b) Pad the nonzero elements of each row in the matrix to a multiple of a given number depending on the implementation.
   (c) copy the matrix to GPU memory.
2. Step 2: Copy the vector to the GPU.
   (a) convert the vector elements from `double` to `float`.

(b) copy the vector to GPU memory.

3. Step 3: Perform the matrix-vector product.

4. Step 4: Copy the result to main memory.

   (a) copy the result from GPU memory to main memory.

   (b) convert the result elements from `float` to `double`.

Steps 2 to 4 are then repeated for each matrix-vector product utilising the same matrix.

The *blackbox* SpMV implementation discussed in section 4.5 was integrated into the SML application along with the CUBLAS library for dense matrix-vector products. Table 5.1 presents the datasets that were used by Teo et al. in his evaluation of the BMRM algorithm [54].

Table 5.1: Datasets used in the evaluation of the BMRM application.

Dataset	Rows	Columns	Sparsity	Avg. Nonzeros/Row
adult9	32,561	123	88.72%	14
astro-ph	62,369	99,757	99.92%	77
aut-avn	56,862	20,707	99.75%	51
covertype	522,911	54	77.78%	12
kdd99	4,898,431	127	87.14%	16
news20	15,960	1,355,181	99.97%	455
real-sim	57,763	20,958	99.76%	51
reuters-c11	781,265	47,236	99.84%	76
reuters-ccat	781,265	47,236	99.84%	76
web8	45,546	300	95.76%	13

The datasets vary greatly in terms of numbers of rows, number of columns and sparsity. However 70% of the datasets are over 99.7% sparse with the remaining 30% over 75% sparse. The average number of elements per row for these datasets vary between 12 and 455 nonzeros per row.

The SML application was benchmarked three times. Once running the original unmodified code, once utilising the *blackbox* implementation for the matrix-vector products, and once forcing the use of dense formats.

The application determines the stopping criteria based upon the regularisation constant (λ) and the termination criteria (ϵ). These parameters were set to

$\lambda = 10^{-5}$ and $\epsilon = 10^{-4}$ for all the benchmarks. The results of these benchmarks are presented in table 5.2. The table presents the number of iterations and total run time for each of the three runs. In the case where the amount of memory on the GPU was insufficient to accommodate either the matrix and its transpose in the case of the sparse products, or the dense matrix in the case of dense products, a N/A is indicated.

Table 5.2: The iterations to convergence along with the total run time (in seconds) for the unmodified application and the GPU with both sparse and dense. The number of iterations reflect the effect of precision the application.

Dataset	CPU		GPU Sparse		GPU Dense	
	iterations	run time	iterations	run time	iterations	run time
adult9	2057	215.99	2112	184.07	2086	216.83
astro-ph	129	46.92	130	33.57	N/A	N/A
aut-avn	134	26.29	134	17.55	N/A	N/A
covertype	296	89.24	285	41.50	310	78.99
kdd99	159	595.21	N/A	N/A	N/A	N/A
news20	284	485.45	286	377.68	N/A	N/A
real-sim	135	26.44	137	17.48	N/A	N/A
reuters-c11	145	21.36	163	16.45	N/A	N/A
reuters-ccat	214	29.35	206	19.82	N/A	N/A
web8	549	14.90	580	8.14	979	29.30

From the ten datasets that were used to evaluate the SML application, only the "kdd99" dataset was too large for evaluation on the GPU. While the "kdd99" dataset can fit on the GPU alone, the added size of the transpose is beyond the capabilities of the GeForce 8800 GTX. Forcing the sparse matrices to use dense formats increases the memory footprint drastically. Dense representations of these matrices are so large that only three of the ten datasets could fit into the 768MB of GPU memory.

The number of iterations taken for the solution to satisfy the termination criteria is roughly equivalent among the CPU and SpMV GPU versions of the BMRM application. The only exception is the "web8" dataset when using dense matrix-vector products on the GPU where the number of iterations were double the values for CPU or SpMV on the GPU. The difference in the number of iterations is expected as the CPU and GPU are utilising different levels of precision

and finite precision floating-point arithmetic is not associative causing the order in which the operations are conducted affect the result.

Computing the SpMV products on the GPU resulted in a minimum speedup of 1.17×, a maximum of 2.15× and an average speedup of 1.5× over the CPU. These values are far below the speedups achieved for the standalone SpMV products presented in table 4.12. To determine the cause of this performance difference and therefore the total application time is broken down into its components in figure 5.1.

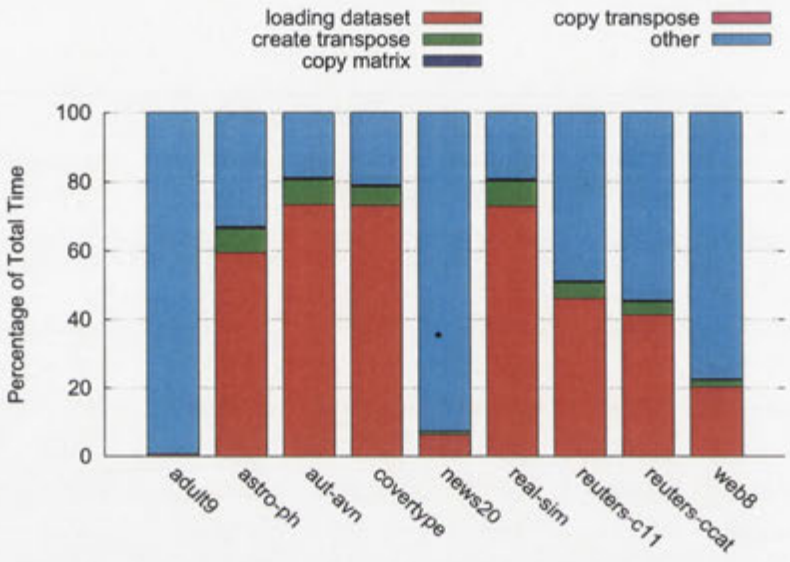


Figure 5.1: Time breakdown total application runtime.

Figure 5.1 shows that loading the dataset from file into memory is a large bottleneck, dominating the total application time for four of the nine runs. Three of the nine runs spent between 25% to 50% of the total time loading the dataset and only two of the runs spent less than 10% of the execution time loading the dataset. Creating the transpose consumes between 1% and 10% the total run time. The copying of both normal and transpose sparse matrices to the GPU accounts for just 1% to 2% of the total execution time.

The large amount of time spent in loading the datasets, suggest that large reductions in total runtime could be achieved by optimising the dataset storage format. For example utilising binary formats for the dataset representation rather than Matrix Market text format [12] would decrease the total execution time.

In Section 4.3.3, the performance results were noted to contain the time penal-

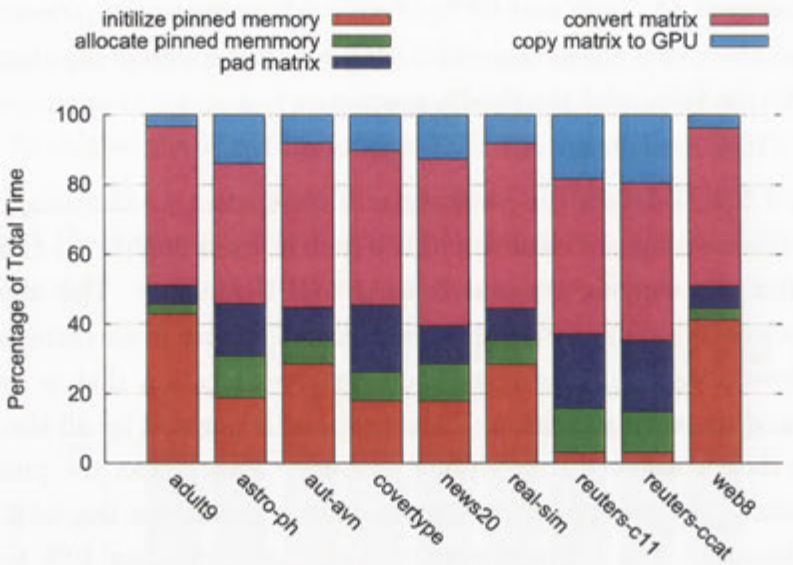
ties inured in converting `double` elements to `float` and copying the `vec` and `res` arrays between the host and GPU. Figure 5.2 presents the percentage of total time that these operations represent, both for the process of copying the original matrix to the GPU and the SpMV product.

Figure 5.2(a) details the percentage of time spent in allocating non-pagable memory, converting the elements of the matrix from `double` to `float`, padding the matrix and copying the matrix to the GPU memory. This shows that the cost of allocating pinned memory is significant. There is an extra cost of about 40 milliseconds for initialising pinned memory allocation that is inured for the first pinned memory allocation. This overhead is negated by all the speedups in memory data transfer for the SpMV products. Other than the pinned memory initialisation, the cost of converting the matrix dominates the total time for all of the datasets. The conversion process consumes between 40% to 50% of the total time. The cost of padding the matrix is the second largest cost, consuming between 5% to 30% of total time. The actual copying of data only consumes between 5% to 20 % of the total time taken for the matrix to reach GPU memory.

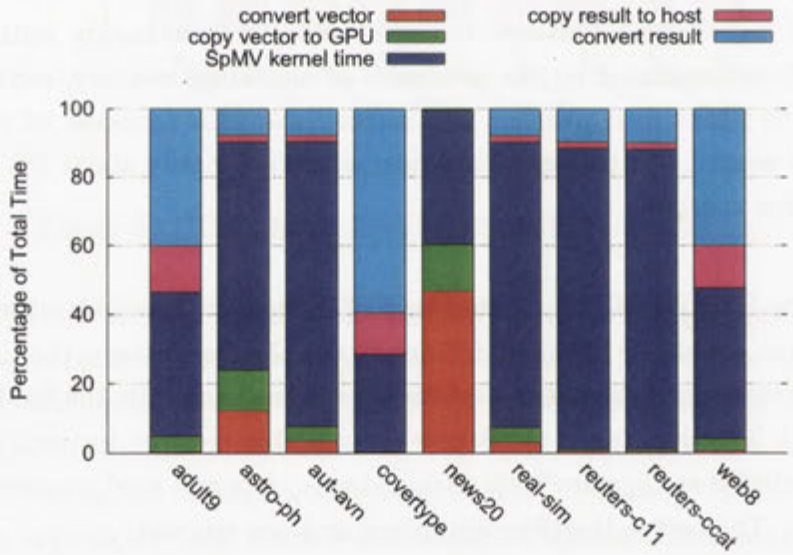
In is important to realise that while the time for matrix initialisation on the GPU is dominated by the overheads of allocating memory, converting from `double` to `float` and padding the matrix, the whole process of initialisation for both normal and transpose matrices account for only about 2% of the total application runtime.

Figure 5.2(b) details the percentage of time spent converting the vector and result from `double` to `float` and `float` to `double` respectively, the time taken to copy the vector and the result, and the time taken to execute the SpMV kernel on the GPU. The time taken to allocate non-pagable memory for vector and result is not included as they are both allocated only once and used repeatedly for each iteration. The cost is therefore amortised and not relevant.

The results of the SpMV operation breakdown in figure 5.2(b) indicates that for the majority of datasets the actual SpMV kernel time is the dominate sub operation. The collective conversion time for both vector and result consumes between 10% to 60% of the total time. The combined copying time for both vector and result consumes between 2% to 20% of the total time.



(a) Time breakdown of the process of copying a matrix from host to GPU memory.



(b) Time breakdown of a single SpMV product.

Figure 5.2: The penalties of conversion and transfers on the matrix copy process and the SpMV product.

5.1 Conclusion

These results show between 1.2× to 2.15× improvement in total runtime of the BMRM application. This improvement is very small in comparison to the capabilities of the GPU. There exists a large overhead in converting between `float` and `double` data types in addition to the overhead of moving data back and forth between the GPU and host. Part of this overhead could be eliminated by modifying the application to utilise `float` data type. The use of newer GPUs which support PCIe 2.0 would reduce the cost of transfers between the host and GPU by 50%.

The use of newer GPUs would also provide the ability to perform double-precision SpMV products. This would also remove the conversion overhead and at the same time increase the number of applications that would benefit from the use of the GPU. The use of double-precision would increase the footprint of the sparse matrix by about 30% and double the memory foot-print of both the `vec` and `arrays`. The increased memory footprint of all data is counterbalanced by the fact that newer GPUs have about 2× the bandwidth available on the 8800 GTX for both host-GPU transfers and internal GPU bandwidth. Using newer GPUs are expected to result in double the performance of single precision with a performance hit of about 30% for using double-precision due to the increased memory foot-print of the sparse matrix.

The above paragraphs discuss SpMV optimisations, however, the further optimisation of the SpMV product will not result in large reductions in total execution time as the total runtime is often dominated by the time to load the dataset to memory rather then the SpMV product. This reduces the impact of accelerating the SpMV operations. Removing the time to load each dataset into memory from the results presented in table 5.2 allows the comparison between the computation time spent by the CPU and SpMV GPU versions of the application. The SpMV GPU application is a minimum of 1.17× and maximum of 5.29× faster then the CPU in this computation.

In summary, the GeForce 8800 GTX offers benefits for the matrix-vector products performed in this SML application and newer GPUs will enhance this. Whether this performance difference is enough given improvements in CPU technology is not clear. More fundamentally the runtime performance of this application is dominated by non-computational tasks that need to be addressed first.

Chapter 6

Conclusions and Future Work

The main objective of this work was the evaluation of the GPUs in general and specifically the GeForce 8800 GTX for SML applications. Performing the SpMV products on the GPU only resulted in 1.2× to 2.15× improvement in total runtime. This is largely due to the amount of time taken to load the dataset into memory, and if this is ignored, the GPU computes the SML task 1.17× to 5.29× faster then the CPU. Based on the limited number of dataset tested with the BMRM application, SpMV on the GPU results in minor advantages to SML applications. However a wider survey of SML applications and datasets should be made to identify if this is the case across all SML applications or not.

The bulk of this work focused on the implementation and evaluation of SpMV routines for the 8800 GTX GPU as presented in chapter 4. SpMV products on the GPU achieve an average improvement of 7.85× the CPU performance of PETSc. This includes penalties resulting from the time taken to convert the `vec` array from `double` to `float` elements and the time to transfer results to and from the GPU, which is substantial as shown in figure 5.2(b). A much larger improvement in performance will be seen when the whole application is ported to the GPU, limiting the need for transfers between the GPU and host. Reductions of these overheads have already been realised with newer GPUs that use PCIe 2.0 to double the bandwidth between the host and GPU.

At the beginning of this work, the CSR sparse matrix format was identified as the preferred format as it was widely used in the scientific community and it was considered the most robust format. The recent work by Bell and Garland [9] showed that performance can be increased by utilising other different formats. This is not surprising as research has been published articulating the benefits of optimising the storage format for sparse matrices on the CPU [18]. The develop-

ment of an auto-tuning library for SpMV on the GPU would be beneficial.

While optimising for the CPU is reasonably well understood, GPU optimisation techniques are not as well established [45]. In table 3.5 the SGEMV performance on the GPU of a 2048×2048 square matrix is listed as 7.26 GFLOP/s, while in figure 4.13 the SpMV performance on the GPU for a 2000×2000 dense matrix is shown to be 10.5 GFLOP/s. The large performance gain in the SpMV product is the results of the use of texture memory to store the `vec` array in the case of the SpMV product, and the under utilisation of the GPU in the case of the dense product. This example highlights two of the most important optimisations in GPU programming, and the extent such optimisations can have on performance; Specifically sufficient utilisation of the GPU execution units and exploration of the various memory options. Volkov and Demmel [57] showed how detailed analysis of the GPU memory system can be used to optimise matrix-matrix multiplies and produce significant performance gains. Memory evaluation and benchmarking is especially important for memory bound GPU applications.

It is important to realise that GPUs are rapidly evolving and these issues could change in the next generation. Owens et al. [45] points to the Sony Playstation 3 and the Microsoft XBox 360 gaming consoles, both of which support host to GPU connections that offer much higher bandwidth than the PCIe bus. The Fusion project by AMD is also highlighted as it attempts to place both CPU and GPU on the same die removing large bottlenecks in GPU-CPU communication. As the bandwidth between the GPU and host increases the number of applications that would benefit from the use of a GPU as a coprocessor and indeed magnitude of improvement would increase. Newer Generation GPUs allow asynchronous memory transfers between the Host and GPU. This allows the GPU to overlap computation with host to GPU memory transfers which will further reduce the overhead of moving data to and from the GPU.

Another obstacle that prevents wide adoption of GPGPU is the different languages for the different GPUs. The work presented in this thesis utilised CUDA for the GPU. This prevents the code written from being used for other GPUs and therefore, applications written in CUDA suffer from vendor lock-in. The Open Computing Language (OpenCL) attempts to provide a solution for this problem. OpenCL is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs and other processors [28]. OpenCL provides a uniform programming environment for software developers to write applications that will run not only on multiple GPUs but on any architecture that implements OpenCL for their system [28]. The OpenCL

1.0 specification [38] was released in February 2009 and both NVIDA and ATI have announced support for it [55, 43].

The use of OpenCL will allow rapid evaluations of multiple GPUs as they would use the same code. However, taking into consideration the large amount of optimisations that are implemented for the different architectures, it is not clear if the advantage of a single language would be mitigated by the need for separate code paths for each architecture in order to apply architecture specific optimisations.

For that reason, rewriting the implementations in OpenCL and attempting the same process of creating a *blackbox* implementation on other hardware is part of the future work envisioned. Other plans include the use of multiple GPUs to perform matrix vector products. The CSR format and the implementations described in this thesis are all easily split across multiple processing units. Indeed, proof-of-concept code has already been written.

Other areas for more study include the use of greedy algorithms rather then exhaustive search in the selection of the BPS of a given size; Doing this would reduce the complexity of the selection process from $O(N^M)$ to $O(N \times M)$. Pruning the number of implementations to a subset that have realistic chances in performing best, in addition to the greedy selection approach, will reduce the time for the whole *blackbox* creation process dramatically. By reducing the time, automating the process becomes achievable.

Finally, it would be desirable to integrate our sparse matrix-vector implementation into a widely used library such as PETSc. Application that use PETSc to perform SpMV products could then easily utilise a GPU enabled version of the library to investigate the possibility of performance benefits without the need to rewrite or even recompile their application.

Appendix A

Memory Bandwidth Benchmarks

Table A.1: Bandwidth of various memory access methods for both Texture and Global Memory.

Size	WL	threads /blocks	Blocks	Coalesced Access						Sequential Access					
				Global Memory			Texture References			Global Memory			Texture References		
				float	float2	float4	float	float2	float4	float	float2	float4	float	float2	float4
ĩ1.5 MB	4	16	6400	12	17	23	12	19	26	8	14	21	14	18	25
		32	3200	21	31	32	22	31	41	8	16	32	18	29	40
		64	1600	36	46	28	32	42	48	8	15	28	18	31	47
		128	800	44	46	29	36	44	46	8	14	30	17	30	46
		256	400	42	44	27	35	41	45	8	14	28	17	30	45
		512	200	37	39	24	30	35	42	8	14	23	16	27.	40
	20	16	1280	13	23	31	14	24	35	9	16	18	6	12	22
		32	640	25	39	33	24	36	45	9	17	18	7	14	26
		64	320	40	50	33	33	43	46	9	17	18	8	15	28
		128	160	55	52	33	40	44	45	9	16	18	8	15	27
		256	80	55	51	31	40	43	45	8	16	18	8	15	26
		512	40	46	47	28	34	41	45	8	16	17	8	14	26
	100	16	256	13	23	32	14	24	38	8	15	18	5	9	18
		32	128	19	32	32	24	38	48	8	15	18	5	10	19
		64	64	35	55	33	26	46	49	8	15	19	6	11	19
		128	32	36	55	32	26	41	48	8	15	19	6	11	20
		256	16	35	54	31	26	41	48	8	15	19	6	11	20
		512	8	33	52	29	25	38	45	8	15	18	6	11	20
ẽ5 MB	20	16	44800	14	22	30	14	22	30	9	16	19	6	12	25
		32	22400	25	40	36	24	36	44	9	17	19	8	14	25
		64	11200	46	58	37	36	45	48	9	17	19	8	16	29
		128	5600	62	63	37	43	50	52	9	17	19	8	16	29
		256	2800	61	59	34	45	52	55	9	17	19	8	15	28
		512	1400	52	54	30	41	51	53	9	17	19	8	15	28
	100	16	8960	14	25	34	15	25	38	8	16	19	5	10	18
		32	4480	27	44	38	25	39	50	8	17	19	5	11	20
		64	2240	46	61	37	40	50	52	8	15	20	10	11	20
		128	1120	64	61	37	45	52	54	7	14	20	5	11	20
		256	560	63	59	35	45	52	54	7	14	19	6	11	20
		512	280	54	56	33	41	51	54	7	14	19	6	11	20
	1000	16	896	13	24	34	15	26	40	2	4	8	2	4	8
		32	448	26	42	38	24	39	51	0	1	2	0	1	2
		64	224	39	55	38	38	51	53	0	1	2	0	1	1
		128	112	48	54	37	36	45	51	0	1	2	0	1	2
		256	56	47	52	36	35	44	50	0	1	2	0	1	2
		512	28	51	56	32	38	48	49	0	1	1	0	1	1
ẽ390 MB	100	16	64000	14	25	34	15	25	38	8	16	19	5	10	18
		32	32000	27	43	39	25	39	51	8	17	19	5	11	20
		64	16000	47	62	38	41	51	53	7	15	20	6	11	20
		128	8000	65	62	37	46	52	54	7	14	20	5	11	20
		256	4000	63	60	35	46	53	55	10	14	20	6	11	20
		512	2000	54	57	34	40	51	54	7	14	20	6	11	20
	1000	16	6400	14	25	35	15	26	41	2	4	8	2	4	10
		32	3200	27	43	39	27	42	54	0	1	1	0	1	1
		64	1600	47	62	39	41	52	54	0	1	1	0	1	1
		128	800	63	61	38	45	52	55	0	1	1	0	1	1
		256	400	62	60	37	45	52	55	0	1	1	0	1	1
		512	200	55	59	35	42	52	55	0	1	1	0	1	1

Bibliography

[1] CUDPP: CUDA Data Parallel Primitives library. http//www.gpgpu.org/developer/cudpp/.

[2] E. Alexopoulos, G. D. Dounias, K. Vemmos, and E. Alexopoulos. Medical diagnosis of stroke using inductive machine learning. In *In Proceedings of Workshop on Machine Learning in Medical Applications, Advance Course in Artificial Intelligence-ACAI99*, pages 91–101, 1999.

[3] Y. Altun, I. Tsochantaridis, and T. Hofmann. Hidden markov support vector machines, 2003.

[4] ATI. *ATI Stream Computing User Guide*, 1.4.0 edition, March 2009. http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx.

[5] E. Atwell. Machine learning from corpus resources for speech and handwriting recognition. In *In J. Thomas & M. Short (Eds.) Using Corpora for Language Research: Studies in the Honour of Geoffrey Leech*, pages 151–166. Longman, 1996.

[6] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[7] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[8] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on gpus. *IBM Research Report*, RC24704, 2009.

[9] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[10] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors. Technical Report CMU-CS-93-173, Aug 1993.

[11] D. Blythe. The direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[12] R. F. Boisvert, R. Pozo, and K. Remington. The Matrix Market exchange formats: Initial design. Technical Report NISTIR 5935, National Institute of Standards and Technology, Gaithersburg, MD, USA, Dec. 1996.

[13] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[14] L. Buatois, G. Caumon, and B. Lvy. Concurrent number cruncher: An efficient sparse linear solver on the GPU. In *High Performance Computation Conference (HPCC), Springer Lecture Notes in Computer Science*, 2007.

[15] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat. Brook for GPUs: Stream computing on graphics hardware, 2004.

[16] J. Cheng and P. Baldi. A machine learning information retrieval approach to protein fold recognition. *Bioinformatics*, 22(12):1456–1463, 2006.

[17] T. A. Davis. University of florida sparse matrix collection. *NA Digest*, 92, 1994.

[18] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding Communication in Sparse Matrix Computations. In *IEEE International Parallel and Distributed Processing Symposium*, April 2008.

[19] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, 1986.

[20] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (Release I). Technical Report RAL 92-086, Chilton, Oxon, England, 1992.

[21] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, New York, NY, USA, 2004. ACM.

[22] E. Frank, M. A. Hall, G. Holmes, R. Kirkby, and B. Pfahringer. WEKA - A machine learning workbench for data mining. In O. Maimon and L. Rokach, editors, *The Data Mining and Knowledge Discovery Handbook*, pages 1305–1314. Springer, 2005.

[23] D. Göddeke, R. Strzodka, and S. Turek. Accelerating double precision FEM simulations with GPUs. In F. Hülsemann, M. K. Ulrich, and Rüde, editors, *18th Symposium Simulationstechnique*, volume Frontiers in Simulation, pages 139–144, 2005.

[24] J. D. Hormozd Gahvari, Mark Hoemmen and K. Yelick. Benchmarking sparse matrix-vector multiply in five minutes. SPEC Benchmark Workshop 2007, January 2007.

[25] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell. Exploring the cache design space for large scale cmps. *SIGARCH Comput. Archit. News*, 33(4):24–33, 2005.

[26] W. Huang, Y. Nakamori, and S.-Y. Wang. Forecasting stock market movement direction with support vector machine. *Computers & Operations Research*, 32(10):2513 – 2522, 2005. Applications of Neural Networks.

[27] D. B. John Kessenich and R. Rost. *The OpenGL Shading Language*. 3Dlabs, 1.10 edition, April 2004.

[28] Khronos Group. OpenCL, 2009. http://www.khronos.org/opencl/.

[29] N. Kohl and P. Stone. Machine learning for fast quadrupedal locomotion. In *in The Nineteenth National Conference on Artificial Intelligence*, pages 611–616, 2004.

[30] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: Iram. *Computer*, 30(9):75–78, Sep 1997.

[31] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 908–916, New York, NY, USA, 2003. ACM.

[32] M. Kubat, R. C. Holte, and S. Matwin. Machine learning for the detection of oil spills in satellite radar images. *Mach. Learn.*, 30(2-3):195–215, 1998.

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

[34] M. A. Maloof, P. Langley, T. O. Binford, R. Nevatia, and S. Sage. Improved rooftop detection in aerial images with machine learning. *Mach. Learn.*, 53(1-2):157–191, 2003.

[35] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM.

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

[37] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[38] A. Munshi. OpenCL 1.0 specification, February 2009.

[39] NVIDIA. *NVIDIA GeForce 8800 GPU Architecture Overview*, November 2006.

[40] NVIDIA. *CUDA CUBLAS Library*, 2.0 edition, March 2008.

[41] NVIDIA. *CUDA CUFFT Library*, 2.0 edition, March 2008.

[42] NVIDIA. *NVIDIA CUDA Programming Guide*, 2.0 edition, July 2008.

[43] NVIDIA. OpenCL for NVIDIA, 2009. http://www.nvidia.com/object/cuda_opencl.html.

[44] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product with applications. In IEEE, editor, *Proceedings of the 2004 IEEE International Symposium on Computer Aided Control Systems Design, Taipei, Taiwan, 2004*, pages 152–155, pub-IEEE:adr, 2004. IEEE Computer Society Press.

[45] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[46] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[47] C. Peeper and J. L. Mitchell. *Introduction to the DirectX 9 High Level Shading Language.*

[48] L. Peng, J.-K. Peir, T. Prakash, Y.-K. Chen, and D. Koppelman. Memory performance and scalability of intel's and amd's dual-core processors: A case study. pages 55–64, April 2007.

[49] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM.

[50] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, NASA Ames Research Center, Moffett Field, CA, 1990.

[51] F. Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, 2002.

[52] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106. ACM, Aug. 2007.

[53] SGI. OpenGL - reference manual, the official reference documantation for openGL, release 1. *Addison Wesley, ISBN 0-201-63276-4*, 1992.

[54] C. H. Teo, A. Smola, S. V. Vishwanathan, and Q. V. Le. A scalable modular convex solver for regularized risk minimization. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 727–736, New York, NY, USA, 2007. ACM.

[55] The Inquirer. AMD also comes out in support of OpenCL, 2009. http://www.theinquirer.net/inquirer/news/964/1049964/amd-also-comes-out-in-support-of-opencl.

[56] M. Ujaldon and J. Saltz. The gpu on irregular computing: Performance issues and contributions. In *CAD-CG '05: Proceedings of the Ninth International Conference on Computer Aided Design and Computer Graphics (CAD-CG'05)*, pages 442–450, Washington, DC, USA, 2005. IEEE Computer Society.

[57] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[58] J. T.-L. Wang, S. Rozen, B. A. Shapiro, D. Shasha, Z. Wang, and M. Yin. New techniques for DNA sequence classification. *Journal of Computational Biology*, 6(2):209–218, 1999.

[59] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[60] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[61] D. Ye, J. Ray, C. Harle, and D. Kaeli. Performance characterization of SPEC CPU2006 integer benchmarks on x86-64 architecture. pages 120–127, Oct. 2006.