Hybrid Algorithms for Efficient Cholesky Decomposition and Matrix Inverse using Multicore CPUs with GPU Accelerators

Gary Macindoe

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

UCL.

2013

I, Gary Macindoe, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Signature : __

Abstract

The use of linear algebra routines is fundamental to many areas of computational science, yet their implementation in software still forms the main computational bottleneck in many widely used algorithms. In machine learning and computational statistics, for example, the use of Gaussian distributions is ubiquitous, and routines for calculating the Cholesky decomposition, matrix inverse and matrix determinant must often be called many thousands of times for common algorithms, such as Markov chain Monte Carlo. These linear algebra routines consume most of the total computational time of a wide range of statistical methods, and any improvements in this area will therefore greatly increase the overall efficiency of algorithms used in many scientific application areas.

The importance of linear algebra algorithms is clear from the substantial effort that has been invested over the last 25 years in producing low-level software libraries such as LAPACK, which generally optimise these linear algebra routines by breaking up a large problem into smaller problems that may be computed independently. The performance of such libraries is however strongly dependent on the specific hardware available. LAPACK was originally developed for single core processors with a memory hierarchy, whereas modern day computers often consist of mixed architectures, with large numbers of parallel cores and graphics processing units (GPU) being used alongside traditional CPUs. The challenge lies in making optimal use of these different types of computing units, which generally have very different processor speeds and types of memory.

In this thesis we develop novel low-level algorithms that may be generally employed in blocked linear algebra routines, which automatically optimise themselves to take full advantage of the variety of heterogenous architectures that may be available. We present a comparison of our methods with MAGMA, the state of the art open source implementation of LAPACK designed specifically for hybrid architectures, and demonstrate up to 400% increase in speed that may be obtained using our novel algorithms, specifically when running commonly used Cholesky matrix decomposition, matrix inverse and matrix determinant routines.

Original Contributions

The main contributions in this thesis are a collection of optimised algorithms that may be applied to general blocked linear algebra routines on hybrid and heterogenous architectures, such as systems with a multicore CPU and GPU accelerator, and systems consisting of multiple GPU accelerators. The first contribution is a new automated approach for blocked linear algebra routines that allows a new level of dynamic blocking to balance the workload more efficiently between heterogenous CPU and GPU computing devices, which have varying clock speeds and very different memory capacities. The second contribution considers the problem of transferring diagonal submatrices between CPU memory and GPU memory, which is an essential operation in many blocked linear algebra routines. We develop a novel algorithm for achieving this transfer efficiently and demonstrate the resulting improvement in speed. The third contribution is an original method for running multiple GPU kernel functions simultaneously on GPUs that do not have inherent hardware support for this capability. In these cases, a large number of GPU processors may often be left idle, waiting for a single kernel to complete. We demonstrate our algorithm using an example whereby a Cholesky decomposition kernel may be run concurrently with a matrix multiply kernel, achieving much higher performance and efficiency than previously possible on the same hardware using existing state of the art linear algebra libraries. We employ the Cholesky decomposition, matrix inverse and determinant operations as motivating examples, and demonstrate up to a 400% increase in speed that may be obtained using combinations of the novel approaches presented.

Acknowledgements

This thesis was funded by the EPSRC grant, EP/E052029/1.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

Since the 18th century, the use of mathematical models has been successfully employed to make scientific predictions about the world, and indeed the wider universe, in which we live. Nowadays, many of the mathematical models of interest are too complex to be tackled by pen and paper and require high speed computing to realise their utility fully. High performance computing has developed into a vital component of modern scientific inquiry, and this has become particularly noticeable over the last 20 years with the advent of low-cost processing power [98, 24].

Computational modelling enables us to analyse and make sense of larger amounts of data, and allows us to make testable predictions, with computer simulations often taking the place of expensive or sometimes even impossible physical experiments [110]. Perhaps most significantly, this increase in computational power gives us the capability of employing powerful statistical methodology that was previously beyond our reach in many cases; Bayesian approaches are a prominent example [128]. Bayesian methodology uses the language of probability theory and provides scientists with a means of reasoning in a consistent manner about the sources of uncertainty that often strongly affect the scientific questions of interest [67, 96]. Such methods allow us to quantify the uncertainty associated with both the measurements from any given experimental setup and as well as our understanding of the underlying structure of the physical system. In terms of mathematical modelling, they allow us to quantify uncertainty in the unknown parameters of a model as well as the uncertainty in the mathematical form of the model itself [16, 96].

Bayesian computing consists mainly of estimating integrals that are often of high dimension. Such problems can be solved using Monte Carlo methods [52, 104], however these are usually computationally intensive procedures that involve a large number of random simulations from the probability distribution of interest; it is no coincidence that the recent rise in the use of Bayesian methodology has occurred at the same time as wider availability of low-cost

high speed computer hardware [108, 128]. In the rare cases when this distribution is known, samples may be drawn with relative efficiency. Usually however, calculation of the distribution is only available to us by solving a complex mathematical model and employing a more sophisticated, computationally intensive simulation method [106].

As a motivating example, Markov chain Monte Carlo (MCMC) [20, 45, 16] algorithms may be used to generate samples from arbitrary posterior probability distributions given by Bayes' Theorem:

$$p(\boldsymbol{\theta}|\mathbf{y}) \propto p(\mathbf{y}|\boldsymbol{\theta})p(\theta) \tag{1.1}$$

In Bayes' theorem, $p(\mathbf{y}|\boldsymbol{\theta})$ is the likelihood and offers a measure of mismatch between the observed data $\mathbf{y}$ and the mathematical model evaluated with model parameters $\boldsymbol{\theta}$. The prior $p(\boldsymbol{\theta})$ characterises our prior knowledge regarding the parameter values that might be plausibly correct [105]. The posterior distribution $p(\boldsymbol{\theta}|\mathbf{y})$ characterises our final estimate of uncertainty, and provides a way of consistently and automatically taking into account information from both the prior and the observed data.

A Markov chain [45] provides a sequence of random variables, in which each value in the sequence depends only on the previous value. Such a Markov chain can be constructed so that it converges to a specified stationary distribution of interest [120]; the posterior distribution in the Bayesian context [105]. The chain is started at a random value, usually a random sample from the prior probability distribution, and a "burn-in" period [20, 33, 57, 128] is necessary to allow the chain time to explore the parameter space and converge to the regions of highest probability. A Markov chain generates correlated samples from the chosen target distribution and may be run to collect as many samples as required to obtain a particular level of statistical accuracy.

The most popular MCMC algorithm is the Metropolis-Hastings algorithm [83, 54] and most other MCMC algorithms can be shown to be special cases of it [16]. Metropolis-Hastings is part of a subclass of MCMC methods called accept-reject algorithms, since they generate samples according to some proposal distribution and then either accept or reject the sample according to some acceptance criteria. In Metropolis-Hastings a new sample is generated according to some proposal probability distribution. The Metropolis-Hastings ratio is then calculated as:

$$\alpha(x, x^*) = \frac{p(x^*)q(x|x^*)}{p(x)q(x^*|x)},$$

where x is the current sample, x^* is the proposed sample and p and q are the target and proposal probability density functions, respectively. The sample x^* is accepted as the next sample in the chain with probability α. The Metropolis-Hastings algorithm is simple to implement but its performance is highly dependent on the choice of proposal distribution [16]. A Markov Chain is said to "mix" well if explores all regions of the distribution and makes large steps that are accepted with high probability [16].

For certain problems, Monte Carlo methods require a great amount of processing time, sometimes even weeks [128]. There is therefore a great demand for high performance numerical algorithms for simulation that harness the potential benefits of parallel processing and reduce computation time [108, 128]. Up until the end of last century numerical algorithms were designed primarily to run efficiently on a single core CPU. These algorithms enjoyed an automatic performance increase with every new generation of CPUs according to Moore's Law [85], which states that the transistor count on a silicon chip will double approximately every 18 months as newer manufacturing processes shrink the size of the transistor and in turn allow an increase in CPU clock speeds. Other components in a computer system however are not governed by Moore's Law and consequently, as CPU speeds were increasing, RAM speeds and latencies did not keep up [39]. Recently there has been a shift in the CPU industry to increase transistor count in CPUs by adding more CPU cores that operate independently of one another while keeping the CPU speed the same [46]. Numerical algorithms are therefore having to be redesigned to exploit the performance available by running operations in parallel on multi-core CPU architectures [13]. In addition, with the advent of general purpose computation on graphics processing units (GPGPU), numerical algorithms are being redesigned to include the use of incredibly powerful parallel processing available from GPUs.

BLAS [69] and LAPACK [15] are two numerical libraries that are currently being redesigned to exploit parallel processing. The BLAS library is split into three sections depending on the algorithmic complexity. BLAS 1 algorithms are $O(N)$ complexity, BLAS 2 algorithms are $O(N^2)$ and BLAS 3 algorithms are $O(N^3)$. GEMM (general matrix multiplication) is a BLAS 3 operation and one of the most basic routines from which all other BLAS 3 operations can be derived [66]. The performance of BLAS 3 algorithms may be more easily improved on parallel architectures than BLAS 1 or BLAS 2 algorithms, as they are more amenable to being split into multiple small independent workloads that can be processed in parallel by separate processing units [37]. They also have a higher ratio of floating point operations (Flops) to memory bandwidth requirements, which is also known as computational intensity [25, 31]. The LAPACK library provides more complicated algorithms than BLAS, including linear system

solvers and matrix decomposition routines. LAPACK uses BLAS operations as integral parts of its algorithms, and therefore its performance is highly dependent on an optimised BLAS library being available. Many hardware vendors supply optimised BLAS libraries for particular architectures, such as the AMD Core Math Library [2] and the Intel Math Kernel Library [4].

1.1 Computer Simulations

Performing a numerical simulation on a computer can bring accuracy problems due to the way floating point numbers are handled [48]. Floating point numbers are represented on a computer as an integer amount (the mantissa) and an integer exponent. This provides a fixed number of significant digits to quantify real numbers and accuracy can be lost when performing mathematical operations on values with large differences in magnitude.

Higham [55] presents a comparison of the theoretical error bounds of five different methods of recursively summing a list of floating point numbers. These include first sorting the list in increasing or decreasing order of value or absolute value; summing in pairs of values of similar magnitude; summing the negative values and positive values separately; and keeping track of a low magnitude error and applying it to each addition in order to correct any error from the previous addition [61]. Each method in the order listed increases the number of operations required to sum the list and it is up to the programmer implementing the summation to choose an appropriate algorithm to balance complexity and speed with numerical error.

1.1.1 Generating Random Numbers on a Computer

Monte Carlo simulations require a large amount of random numbers. This presents a problem as computers are deterministic by nature and need special algorithms, known as pseudo-random number generators, to provide streams of numbers that statistically have the same properties as those produced randomly [72]. Pseudo-random number generators consist of a state, a function to initialise the state to some value dependent on a seed, a function to update the state every time a random number is drawn and a function to generate a random number from the current state. Given the deterministic nature of computations, the pseudo-random number generator will end up repeating its stream of numbers after a certain amount have been generated. This amount is known as the period of the generator. Pseudo-random number generator algorithms fall into several categories including Linear and Non-linear Congruential Generators, Lagged Fibonacci Generators, Linear Shift Feedback Registers and Tausworth Generators [117, 70]. Good random number generators have a small state, efficient functions, a large period and generate numbers with good randomness properties [117, 71].

1.2 Approaches to Parallel Simulation

Since the advent of computers there have been efforts to use multiple machines to work on a problem simultaneously in an attempt to reduce the time taken to solve large problems. All that is needed to have computers work together is a means of communication between the processors doing the calculations. Examples range from local computers with more than one processor installed to multiple computers connected across the internet [121]. Any task can be divided into subtasks and, if they do not depend on one another, can be performed concurrently. Since there is more than one processor with possibly different capabilities involved in the simulation it is more efficient to distribute the workload among processors so that more capable processors do more work. Failure is also an issue when multiple processors are involved and techniques have been developed to cope with this, such as simply having the subtask repeated on a different processor when the original one fails to return a result.

Certain Monte Carlo algorithms such as computing the expected value of a function of a random variable are trivial to implement in parallel as they can be easily broken down into smaller subtasks and the results combined once each subtask has been completed. Parallelising Monte Carlo algorithms can however affect the bias and variance of the resulting estimate unless certain rules are followed when designing the simulation [108]. These rules are discussed in more detail in Section 2.2.2.

Markov chain Monte Carlo methods can also be implemented in parallel although this type of algorithm is less obvious to implement in parallel as the simulation of the next value in the chain is dependent on calculating the current value first. This is known as the Markov property. There are two approaches to completely parallelising MCMC methods: running multiple independent chains in parallel (similar to parallel Monte Carlo) and parallelising the generation of a single chain. Many researchers have investigated ways to speed up the generation of a single Markov chain using parallel processing [63, 24, 26, 119], including the use of parallel libraries for expensive, time-consuming operations, however the options are limited by the inherently sequential nature of this algorithm [128]. We note that MCMC algorithms often make use of Gaussian random variables, which require repeated use of Cholesky decomposition, matrix inverse and matrix determinant routines. For large matrices, the key idea of splitting these expensive operations into independent subtasks that may be computed in parallel becomes vital for achieving efficient running times.

1.2.1 Communication and Synchronisation

When a program is to be implemented on a parallel computer it is critical that the size of each parallel subtask is large enough to outweigh any costs incurred performing interprocess communication [24]. For problems that can be split into large independent subtasks that each take a long time to execute, the speed of communication between processing nodes is not much of an issue. These coarse-grained parallel problems are thus suited to distributed memory clusters, where the speed of communication between nodes is on the order of milliseconds. Programs that are split into smaller parallel subtasks are suited to shared memory processors, such as multicore CPUs, which have faster interprocess communication. These are termed fine-grained parallel problems.

1.2.2 Parallel Random Number Generators

In addition to the previously mentioned properties, pseudo-random number generators for parallel environments must also generate a sequence of random numbers for each processor that appears to be independent of the sequences being generated on the other processors. This means that the sequences cannot overlap at any point and that the numbers being generated on the processors cannot be used to guess the next number in any of the sequences. Methods of parallelising serial pseudo-random number generators include Leapfrog, sequential splitting and independent sequences [32], and shuffling Leapfrog [131]. Each method (apart from independent sequences) relies on being able to efficiently calculate an arbitrary element in the sequence [117, 32].

1.3 Hardware Accelerators

A hardware accelerator is a specialised piece of hardware that performs a specific group of operations faster than it would take a more general purpose piece of hardware. Modern examples of hardware accelerators include graphics cards, which accelerate the numerical operations required to render 3D graphics. Unlike a traditional CPU, GPUs dedicate more space on each silicon chip to processing units, rather than memory caches and flow control, making them able to process data in a highly parallel fashion [94]. Recently there have been developments to allow graphics cards to be used for other applications that would benefit from the parallel processing power available using General Purpose Graphics Processing Unit (GPGPU) computing [74, 25, 73, 97].

1.3.1 Hybrid Multicore Parallel Programming

HMPP [36] is a programming environment consisting of a compiler wrapper and a software library. Special commands are inserted into the application code to tag sections (“codelets”)

that are to be executed on a hardware accelerator. These commands are parsed by the compiler wrapper to produce a version of the codelet that targets the hardware accelerator when compiled. A single codelet can be compiled for multiple targets and the required executable code can be determined at runtime depending on the hardware accelerators available. The original unaccelerated version of the program can be easily reproduced by recompiling with the original compiler, since the HMPP commands are inserted into regions of the code that are otherwise ignored (e.g. comments). This provides a way for software companies to experiment with accelerated versions of their products without altering the original code. HMPP currently supports Fortran and C with Java currently being developed. It can target OpenCL [12], CUDA [5] and OpenMP [34].

1.3.2 GPGPU

Early GPUs had separate types of processor for each stage of the 3D graphics rendering pipeline with the output from one processor being fed directly into the next [84, 75]. Triangles defining a 3D scene would have their orientation calculated by dedicated vertex processors. Fragment processors would then take the triangles and work out which of them are obscured by others in the 3D scene, resulting in fragments of the final image. The vertex and fragment processing stages of 3D graphics rendering have a lot of inherent parallelism [84, 97] and GPUs would take advantage of this by having several vertex and fragment processors to compute each vertex and fragment independently. As human vision is slow in comparison to the speed of GPUs [97], latency can be high with graphics rendering and many pixels can be at different stages of the processing pipeline at once. This is shown in Figure 1.1; in particular, having a fixed number of each type of processor implemented in hardware caused load balancing problems for scenes that have more vertices than fragments and vice versa, which cause one type of processor to have the majority of the processing workload.

As the complexity of 3D graphics increased programmers demanded more functionality from vertex and fragment processors [84]. As a result the processors gained increased functionality and were able to run vertex or pixel shader programs written in languages such as HLSL [99] or GLSL [113]. These languages allowed programs to be written that describe how each pixel of the final scene is to be shaded [84]. The first such graphics card to feature programmable vertex processors was the nVidia GeForce 3 in 2001 [74]. This was followed by the GeForce 6800 in 2005 which additionally featured programmable fragment processors [84]. Toolkits also emerged that enabled these processors to perform general purpose computation on the GPU giving rise to the field of GPGPU [97]. These toolkits allowed programmers to write code to be executed in parallel across the programmable shader cores [78, 25]. As the graph-

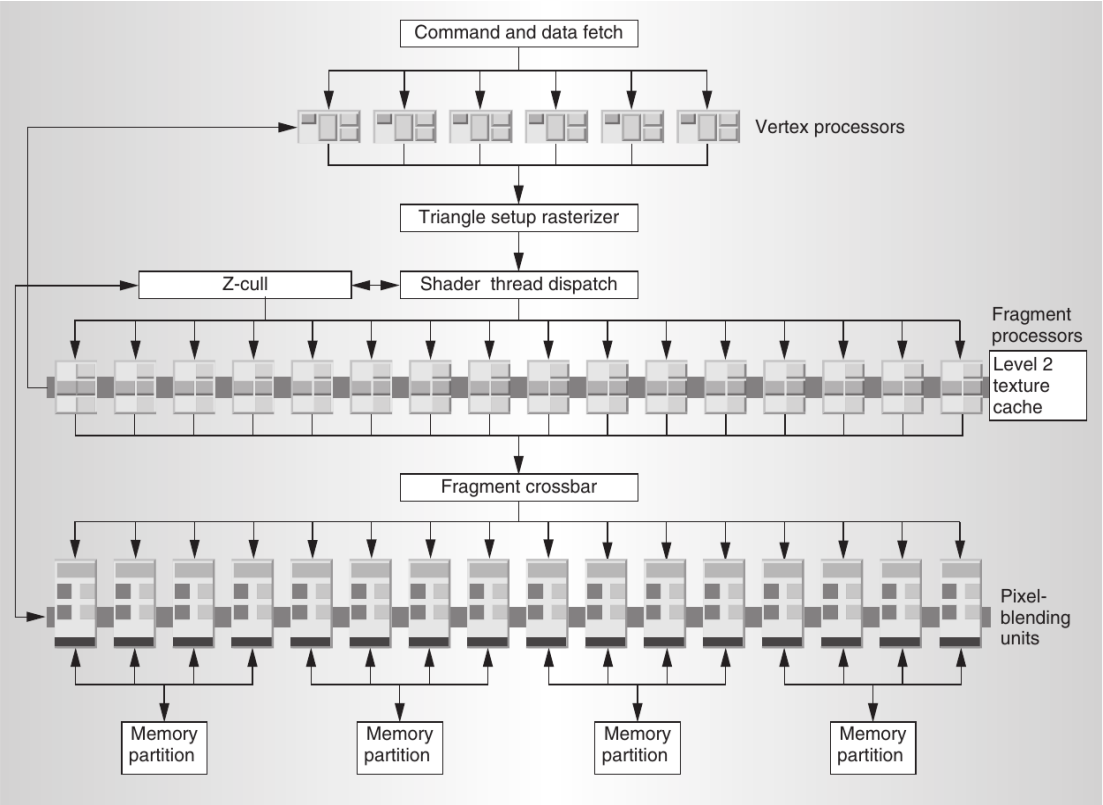


Figure 1.1: This is a block diagram of the graphics processing pipeline in the nVidia GeForce 6800 GPU [84]. Data flows into the six vertex processors at the top of the diagram and then into the 16 fragment processors and finally the 16 pixel blending units along the bottom of the diagram. Vertex processors calculate the geometry and orientation of the triangles defining a 3D scene while fragment processors work out which triangles are obscured by others in the final scene. Pixel blending units apply the colour information to the pixels in the final scene. Having a fixed function graphics pipeline creates load balancing issues as certain 3D scenes such as those with a large number of small triangles will use the vertex processors more than the fragment processors while other 3D scenes with a small number of large triangles will use the fragment processors more than the vertex processors.

ics card drivers only exposed Direct X or OpenGL APIs these toolkits provided a layer over the graphics APIs to allow data to be transferred into the vertex and pixel buffers, to execute arbitrary shader programs and download results from the framebuffer [97].

The generic operations added to both the vertex and pixel processors caused their functionality to overlap. In 2005 nVidia introduced the GeForce 6800 with unified graphics processing cores [84]. These are a single type of processor with generic functionality for running vertex or pixel shader programs. With the introduction of the GeForce 8800 nVidia also provides their own toolkit and API for GPGPU computing, named CUDA [89, 73]. AMD has also produced unified architecture GPUs along with several toolkits for GPU computing including CTM [6]. Both companies now contribute towards the OpenCL standard for GPU computing [115].

1.4 Summary

The advent of multicore processors has meant that individual workstations are now inherently parallel computers. This has brought parallel computing into the mainstream, whereas previously it was reserved for those with large enough budgets for several computers and a network to connect them all. An advantage of a multicore processor is that connections between processing cores are many orders of magnitude faster than the connections in a computer network. In addition, current multicore processors have cores that are identical to each other and therefore algorithms may be more easily optimised and the workload more easily balanced across the available cores. In contrast, computer clusters may have processors of varying speed and differing specifications. A final advantage of multicore processing lies with its shared memory, which further simplifies the development of parallel code and allows efficient parallelism of algorithms that otherwise would not benefit from being run in a distributed parallel environment.

With the development of GPGPU computing and hardware accelerated computing however, parallel architectures have moved towards a heterogeneous, distributed memory environment. Although this type of system results in a far larger number of processing units at the programmer's disposal, it also introduces a number of significant challenges that need to be addressed. Firstly, there is greater communication overhead between the CPU and the accelerator, and this must be taken into account during algorithmic design to ensure efficiency. Secondly, we must now deal with multiple processors with differing specifications, which makes it harder to evenly and efficiently balance the workload.

In this thesis we consider these challenges and present novel approaches for redesigning blocked linear algebra operations to benefit from the multitude of parallel architectures currently available through GPUs. We begin in the next chapter by summarising existing technologies

for general parallelisation of computer instructions on CPUs and GPUs. We then investigate the Markov chain Monte Carlo method in greater detail. This is a motivating example from Statistics that, due to the Markov property, would appear to have limited scope for parallelism unless one considers lower-level approaches such as the parallelism available in the underlying linear algebra routines that it utilises in order to further increase its computational efficiency and scalability. We discuss high-level approaches to parallelising Markov chain Monte Carlo algorithms, before delving deeper into the underlying linear algebra libraries that are employed within such statistical methods. We give an overview of current BLAS and LAPACK implementations available for different types of processing units, and review existing libraries developed for hybrid architectures, in particular the MAGMA library, which we use for benchmarking the contributions presented in the later chapters.

Chapter 2

Related Work

There are many approaches to writing and extending computer codes for parallel distributed and shared memory architectures, and in this chapter we give a brief overview. We begin by giving a summary of technologies available for parallelising existing computer codes. We then consider the example of Markov chain Monte Carlo and review possible approaches to parallelisation of this useful class of algorithms, in particular focussing on parallelised random number generation and examining how the intrinsic structure of these methods limit the extent to which they may be parallelised. We conclude that further improvements in performance are likely to come only from more efficient parallelisation of the underlying linear algebra routines upon which MCMC methods strongly depend. Finally, we give an overview of the parallelised numerical libraries based on BLAS and LAPACK that are currently available for single processor, GPU and hybrid architectures.

2.1 Technologies to Parallelise Existing Code

2.1.1 MPI

MPI [125] is a message passing library interface specification for programs using the message passing parallel programming model. The message passing parallel programming model moves data from the address space of one process into the address space of another. It is used primarily on distributed memory multiprocessor machines where the memories are connected by a communications network although it can also be used on shared memory machines. The MPI standard is a specification for a library that implements message passing [50, 43]. It does not specify which programming language it should be implemented in although language bindings for Fortran, C and C++ form part of the standard. Version 1.0 of the MPI standard was released in 1992 and adopted the best features of existing message passing systems [101, 27]. It was created by researchers from academic, government and industrial backgrounds mainly from the EU and US. The advantages of creating a standard for message passing are portability and ab-

straction as the standard does not specify how the library specification is to be implemented allowing vendors to implement part of it using specialised hardware to improve performance. The aim of the MPI Forum [42] is to create a practical, portable, efficient and flexible standard for message passing. The library API that forms part of the standard should allow for efficient and reliable communication in heterogeneous processing environments and also be thread-safe.

MPI has also been considered as the API of choice to implement message-passing between hardware accelerators [116]. Currently GPUs are under explicit control of a CPU and have no peer-to-peer message passing capabilities despite this being one of the features of the PCI-Express bus to which they are connected. MPI also currently only considers CPUs as sources or sinks of messages. Stuart *et al.* [116] compare three attempts at extending MPI to support GPUs, all of which have been made obsolete in some way or another due to improvements in GPU vendor libraries. They introduce their own extensions and discuss the modifications needed to the MPI standard. Firstly GPUs need to be able to communicate directly with each other over the PCI-Express bus and also with the network card in the host machine. Each accelerator will need to be assigned an MPI rank so that it can be the source or sink of any communications and an MPI library consisting of GPU functions would need to be written. MPI will also need new communicators to broadcast messages to all GPUs, all CPUs or all CPUs and GPUs within a particular machine.

2.1.2 OpenMP

OpenMP [34] is an API standard designed for shared memory parallel programming much in the same way that MPI is a standard for message passing. In order to have a scalable parallel application both scalable hardware and software are needed. Distributed memory systems provide scalable hardware for message passing so their scalability relies on software built on the message passing programming model. With the introduction of shared memory multiprocessing a message passing programming model became too elaborate and complex for software to scale well.

The OpenMP standard builds on existing standards for shared memory multiprocessing including MPI, POSIX Threads (PThreads) [86] and the unfinished X3H5 standard [109]. X3H5 was a project to develop an ANSI standard for shared memory multiprocessing but only got as far as implementing parallel loops when interest was lost due to the popularity of distributed memory systems. Using MPI for shared memory multiprocessing requires a lot of effort by the programmer to explicitly partition data structures across processors [34] and as a result the entire program must be rewritten to use the parallel data structures. PThreads [86] is a threading library for POSIX-compliant operating systems. It provides an interface to directly control op-

erating system threads and is very low-level. It is only available on POSIX-compliant systems so is not entirely portable and is not available for Fortran in which most scientific code is written. It uses a task-based programming model where each thread is assigned a task to perform rather than a data-parallel model which suits scientific code better.

OpenMP is a standard set of compiler directives to allow programmers to express parallelism and an API specification for an accompanying runtime library [29]. The directives are split into three sections to cover control structures, data sharing and synchronisation. An OpenMP aware compiler will process the directives to execute sections of code in parallel whereas a compiler that does not support OpenMP will simply ignore them. The standard is designed to be language agnostic however it specifies a set of compiler directives that are to be available in Fortran, C and C++. The standard builds on X3H5 to include support for coarse-grained parallelism as loop-level parallelism has limited scaling on shared memory architectures due to Amdahl's Law [51]. How well a parallel algorithm scales depends on how well it fits into the parallel programming model being used. OpenMP therefore has an additional set of directives allowing different parallel programming models such as task-based programming [17] and directives that can be nested allowing each sub-thread to further generate more sub-threads.

OpenMP implements parallelism using a "fork-join" model where a team of threads is created when a parallel directive is encountered [28]. It allows the programmer to specify which variables from the data environment are to be shared among threads and which are to be private. The compiler does all the work to efficiently execute sections of code in parallel and the programmer need not know the number of threads being created. This is in contrast to MPI, where data sharing must be done explicitly by the programmer, and in contrast to PThreads, where shared variables must have mutexes created to control accesses by multiple threads.

2.1.3 SSE

Internet Streaming SIMD Extensions (SSE) is an Instruction Set Architecture developed by Intel to improve the performance of 3D graphics rendering on its 32-bit CPUs [118]. Intel's CPUs required a $1.5 - 2\times$ increase in floating-point arithmetic performance to produce a noticeable improvement in 3D graphics quality. Graphics operations are SIMD parallel and adding SIMD units is a cost-effective way to improve floating-point performance on a general purpose CPU. This had already been done in earlier Intel CPUs with the MMX instruction set which performed SIMD operations on integers [100, 21]. When designing the new instructions Intel also studied how the CPU uses data and decided to introduce instructions that allow the programmer to differentiate between data that is reused and data that is only used once [118]. Data

that is reused should be loaded into the cache while data that is only used once should not be cached (potentially requiring some reusable data to be removed) and instead should be streamed through the processor. They also included a large amount of customer feedback from software developers when designing the new instructions in order to make them as general purpose as possible so that they would also provide a speed increase to other applications such as speech recognition and multimedia encoding.

SSE implements SIMD parallelism using a new set of 128-bit registers that can accommodate four single-precision floating point variables (or 32-bit integers, or two double-precision floating point variables or 64-bit integers) [21]. This provided the needed increase in floating point performance for 3D graphics while requiring the least increase in processor die area and complexity. Internally, Intel's CPUs already performed floating-point arithmetic in 80-bit floating-point units so the increase from 80-bits to 128-bits was less of an implementation challenge than an increase to 256-bits or wider. For problem sizes that are not a multiple of the SIMD width the remaining elements must be processed separately therefore an increase in SIMD width gives diminishing returns as well as requiring increased memory bandwidth to keep the SIMD units supplied with data. The MMX instruction set reused the 80-bit x87 floating point registers to perform two-way SIMD operations on two 32-bit integers in the lower 64-bits of the register. This required no special operating system support [100, 21] although it meant that floating point arithmetic could not be performed at the same time as MMX integer operations and was more complex for the programmer to manage. The decision to use a new set of registers for SSE required explicit operating system support to save and restore the extra registers between context switches but means that scalar integer operations can be performed in parallel to SIMD floating point arithmetic [102] and improves general purpose performance due to an increased number of registers. SSE also moves scalar floating-point arithmetic to the lower bits of the new SSE registers, rather than keeping them in the existing MMX/x87 registers. This keeps results between SIMD and scalar arithmetic consistent as they now both performed in 32-bit precision rather than scalar arithmetic being performed in the extended 80-bit precision x87 registers as with MMX.

Memory operations on vector computers typically require accesses to be aligned on a multiple of the SIMD width as hardware to support misaligned loads and stores is complex [118, 100, 102]. The SSE instruction set provides aligned memory and computational instructions along with load and store instructions that correct misaligned memory accesses. In addition instructions may have memory "hints" applied to them that cause data that will be needed soon to be loaded into the cache early so that by the time it is needed it is already cached. Shuf-

fling memory instructions that dynamically reorganise non-contiguous data such as that stored in an "array of structures" format are also included in SSE although they incur a 25% performance penalty to use over data that is organised in a contiguous "structure of arrays" format [118].

Being motivated primarily by 3D graphics performance SSE contains special instructions such as reciprocal division and square root which are commonly used to calculate surface normals for lighting 3D scenes. These operations are implemented to less than IEEE standard precision in order to be fast although they can be combined with Newton-Raphson iterations to improve accuracy and still be faster than ordinary division or square root instructions [118].

SSE has been implemented in every Intel-compatible processor since the Pentium III in 1996 [102].

2.1.4 Compiler Autovectorisation

Traditional vector computers require programs to be rewritten as a sequence of vector operations on whole arrays at once [14]. This may be impossible in cases where there are data dependencies between elements in the array. SIMD processors are relatively recent and perform the same operation on a small number of contiguous elements contained in a vector register. To use these registers loops within a program need to strip-mined to the SIMD vector length and a separate loop added to operate on remaining scalar elements. Compiler autovectorisation [87] seeks to have these loop transformations applied automatically by the compiler replacing the strip-mined loop with vector instructions that use the SIMD registers within the CPU.

Automatic vectorisation for vector computers is a mature research area and the work is now being applied to modern SIMD processors [87, 21, 41]. The main areas of research are detecting data dependencies and loop analysis, finding loops that are viable targets for transformations that increase parallelism [87]. Most of the theory for automatic vectorisation on vector computers applies to Fortran arrays and so does not take into account pointer aliasing that can be performed in programming languages such as C. SIMD architectures also usually have stricter memory alignment and access requirements and limited, domain-specific mathematical instructions [87].

In GCC vectorisation is applied as part of other loop optimisations to code in GCC's intermediate GIMPLE representation [87]. A number of tests are iteratively applied to loops in this form before a vectorising loop transform is applied. The tests involve checking that a loop has a countable number of iterations, has no loop dependencies and that the operations performed in the loop have corresponding vector instructions on the target architecture. This causes a problem in GCC where the optimisations applied need to be generic and not specific to any par-

ticular platform. If some operations do not have equivalent vector instructions they are left as a sequence of scalar operations. Data dependencies can also be ignored when the vector length is known and the dependency span is greater. Support for non-contiguous or misaligned memory accesses may not be available on the target platform. For misaligned accesses GCC employs loop versioning to test at runtime if the accesses are aligned or not.

2.1.5 CUDA

Driven by the increasing demand for realistic, real-time 3D graphics rendering, GPUs have become highly parallel computing devices with a large number of processing cores, very high instruction throughput and high memory bandwidth. Graphics rendering is SIMD parallel and has a high ratio of arithmetic operations to memory operations so there is less of a requirement for cache and control logic. GPUs consequently have more die area dedicated to data processing and memory latency is hidden by arithmetic operations instead of large caches. CUDA [94] is a GPU architecture, instruction set and programming model for nVidia graphics cards that enables general purpose computing on the GPU (GPGPU). It is distributed as a software environment comprising a compiler, developer tools and a runtime library. The runtime library is callable from Fortran, C and C++ and contains DirectCompute, OpenCL and OpenAAC APIs.

GPU kernel functions are written in "CUDA-C" which is a language similar to C with extensions for synchronisation barriers, thread indices and explicit access to shared cache memory. This provides a familiar programming environment for programmers keeping the learning curve low. A complete maths library is available along with intrinsic GPU functions. Thread indices are organised as a 2 or 3 dimensional grid of 2 or 3 dimensional block of threads. This provides a thread hierarchy allowing both coarse and fine grained parallelism. A block of threads is executed on a single core in a GPU. Each core executes as many thread blocks as will fit in the cache and registers. Because each thread block that is currently being run is resident on a processing core switching between them is fast. Remaining thread blocks are run when other blocks finish and free up resources on a processing core. This allows code written with many blocks to scale well to future GPUs with more cores and more cache or registers per core.

Threads within a thread block can communicate with each other via a shared cache memory that is present on the processing core. Communication in this manner requires that all threads in a block synchronise to ensure that writes by threads are visible by others in the thread block. Communication between blocks on different cores has to be performed via atomic operations in global graphics memory.

GPU code is compiled by the nVidia compiler into generic GPU assembly or a binary object targeting a particular class of GPU. The runtime library provides functions to allocate

memory for function parameters, upload and download data and executable code to the GPU and launch kernels using a particular configuration of thread blocks. Execution is asynchronous with respect to CPU and the library provides an explicit synchronisation function that blocks until the GPU is finished. The library also provides functions to query the "Compute Capability" of a GPU and load either binary images that are compatible or generic GPU assembly which will be just-in-time compiled for the GPU.

CUDA-capable GPUs are implemented as a number of identical multi-threaded "Scalar Multiprocessors". Each thread is pipelined to improve instruction level parallelism although there is no branch prediction or out of order execution in order to keep each SM simple. When a kernel is launched on the GPU, thread blocks are distributed over the SMs until they are full. Any remaining blocks are placed in a queue and scheduled on an SM when others finish. Each SM is capable of running hundreds of software threads concurrently in groups of consecutive software threads called warps. Threads within a warp share the same instruction counter and run similarly to SIMD vector threads except that threads are allowed to follow different code branches. However for best performance it is not advisable to have threads within a warp follow different code paths as the SM has to follow all branches even if some of the threads are suspended. As each warp is multi-tasked nVidia refers to this hybrid multi-threaded SIMD paradigm as SIMT (Single Instruction across Multiple Threads).

2.1.6 OpenCL

As the number of CPU cores increases and GPUs become more general purpose computing devices there is an emerging overlap in technologies and features. OpenCL [9] is a standard for heterogeneous computing resources that targets this overlap. It was created by Khronos - a non-profit industry consortium of hardware and software vendors and academics that creates open standards for parallel computing. nVidia is the chair of the consortium and Apple is in charge of editing the specifications. It took the consortium six months to publish the first version of the OpenCL standard and there are several conforming implementations from companies such as AMD, Intel and nVidia across several operating systems and hardware platforms. The next version of the standard took a further 18 months to publish and is backwards compatible with the first. The consortium manages a developer community and makes the specifications free to obtain. Several books on OpenCL programming have also been published.

OpenCL aims to fully utilise the computational power available in CPUs, GPUs and other hardware accelerators to accelerate parallel computational intense code portably across different platforms. It provides a runtime library to query and set up devices, manage memory and execute code. In OpenCL terminology an N-dimensional compute domain is defined and ker-

nels are run across the domain. The OpenCL platform model consists of one host and many compute devices, each of which has one or more compute units containing many processing elements. An OpenCL application runs on the host and submits work items to the compute devices via an OpenCL context. An OpenCL context groups compute devices and creates and manages work queues and memory.

Kernels are written in a subset of ISO C99 with extensions for intrinsic functions, vector types and a corresponding vector maths library which supports IEEE 754 compliant floating-point error bounds. Being designed to run on GPUs as well as other accelerators thread indexes and memories have a similar hierarchy to that used in CUDA. Kernels can also be compiled to an intermediate representation to be loaded and compiled to machine code by the runtime. For GPUs the intermediate representation is CUDA-compatible GPU assembly.

2.1.7 HMPP

While the high computational power of GPUs makes them attractive for hardware accelerated parallel computing there are many companies with a significant amount of code that do not have the resources required to rewrite their applications from the ground up to use a GPU. Although technologies such as CUDA make it easier to write code targeting GPUs even experienced developers may introduce bugs into an otherwise stable software product and companies cannot risk this.

HMPP [36] aims to simplify the process of converting an existing code base to use hardware accelerators while maintaining application portability. The product documentation compares this to integrating the GPU into an existing application rather than porting the application to use the GPU. It consists of a C and Fortran compiler preprocessor, development tools and a runtime library for a heterogeneous multi-core environment on Unix-like operating systems. In a similar manner to OpenMP, HMPP contains compiler directives that mark functions as candidates for hardware acceleration. These directives are processed by the HMPP preprocessor to automatically convert the existing code to call a GPU version of the function if a compatible GPU accelerator is available. The GPU version of the function is developed using tools from the GPU vendor and the preprocessor automatically handles the transfer of any function arguments onto the GPU. HMPP compiler directives are similar to OpenMP directives in that they are ignored by compilers that don't recognise them. Recompiling code with HMPP directives with the original application compiler produces the original application executable removing any risk that may be inherent in exploring a new technology.

The HMPP programming model transfers arguments onto the GPU, executes the function and downloads the results when the function is complete. More advanced usage of the direc-

tives allows results to remain on the GPU if they are to be used in subsequent GPU functions. Directives also allow a specific accelerator to be chosen for execution and FPGAs will also be targeted by the product. The runtime library is able to detect which accelerators are available on the platform at runtime and run the original function if no accelerators are available. It also handles any exceptions raised by the accelerator hardware and can be used with other multiprocessing technologies such as OpenMP and MPI.

2.2 Parallel MCMC Implementations

We now give a summary of the ways in which Markov chain Monte Carlo algorithms may be parallelised, and discuss the challenges and limitations that result.

2.2.1 Parallel Pseudo-Random Number Generation

Computers are entirely deterministic in that when repeatedly given the same inputs they will generate the same output. This causes problems for algorithms that require a source of randomness, such as Monte Carlo simulations, but which also need to be repeatable. Physical sources of randomness can be connected to a computer, for example, counting the number of ionised particles emitted from a radioactive sample every second using a Geiger counter [126]. However these are often a poor source of randomness as the underlying distribution of samples is often unknown, unbounded, and is not repeatable. A class of algorithms known as Pseudo-Random Number Generators, or PRNGs [71], output a stream of numbers that give the appearance of being distributed randomly while being computed deterministically. PRNGs consist of an initialisation function that initialises the generators' internal state according to a seed value and a generation function that updates the state and outputs a random number. Initialising a PRNG multiple times with the same seed will result in the same stream of numbers being produced which allows simulations based on them to be repeatable exactly and also debugged more easily. Since the state of the PRNG is finite the stream of numbers will begin to repeat at some point. The amount of numbers output before the stream starts to repeat is known as the period of the PRNG. A large period is desirable but often results in a larger state having to be stored. PRNGs are categorised according to the form of their generation functions which may be combined in order to increase the quality of the generated output at the cost of state size and complexity [71].

Linear congruential generators (LCGs) have a generation function of the form

$$x_i = (ax_{i-1} + c) \bmod m \tag{2.1}$$

The state of an LCG consists of solely the previous value generated and so is very small. The parameters a, c and m define the period of the generator and the statistical quality of the stream

of numbers generated. The generation function is short and consists of few operations so should be fast.

Lagged Fibonnaci Generators (LFGs) have a generation function based on a generalisation of the Fibonnaci sequence where the next number in the stream is based on two previous numbers. The general form is

$$x_i = x_{i-j} \oplus x_{i-k}, \quad 0 < j < k \tag{2.2}$$

The operator $\oplus$ may be an arithmetic or bitwise operator. Since a generator of this type requires the k previous numbers to be available the state is of size k. The parameters j, k and the operator $\oplus$ decide the period of the generator. The quality of the output is dependent on the parameters but also on the initialisation of the state which has to be performed with another PRNG.

Tausworthe Generators have a generation function of the form

$$x_i = (a_1 x_{i-1} + a_2 x_{i-2} + \cdots + a_n x_{i-n}) \bmod 2 \tag{2.3}$$

where the $a_0, a_1 \ldots a_n$ are the parameters of the generator and define the period and statistical quality of the output stream. The size of the state is n since a combination of the n previous numbers determine the next. The $\mathrm{mod}2$ at the end of the generation function means that Tausworthe generators produce bit values that are either 0 or 1 hence they are slow, requiring 32 or 64 operations to produce a 32-bit or 64-bit random number respectively.

Linear Shift Feedback Generators (LSFR) are based on a linear shift feedback register which is updated by shifting the bits to the right, inserting a new bit in the most significant position that is the result of a bitwise operation on one or more less significant bits in the register. The operation is commonly XOR giving rise to XORSHIFT PRNGs. The recursion formula can be represented by a characteristic polynomial.

A Weyl PRNG has the simple form

$$x_i = x_{i-1} + c \bmod m \tag{2.4}$$

with c being an odd constant and m being the maximum value to output.

The output of a PRNG is usually a stream of uniformly distributed unsigned integers of the native word size of the computer. The numbers are uniformly distributed over the whole range of the type i.e. $[0, 2^{32} - 1]$ on a 32-bit computer. There are several tests which can be performed to assess the statistical quality of the random stream produced by a PRNG. An example of a simple test is to take a large number of samples from the output of a PRNG and calculate the mean. The closer to the middle of the output range the calculated mean is the more

uniformly the output is distributed. There are a large number of more complicated tests that can be performed and most of them are available as part of the DIEHARD [79] and TestU01 [72] batteries of tests. A good PRNG will have a small state, efficient update function and pass all tests related to the statistical quality of the random numbers generated.

When performing a random simulation on a parallel computer extra steps need to be taken to ensure that the streams of numbers output by the PRNGs on each processor are independent of each other. Using the same seed and PRNG on each processor will result in exactly the same stream being reproduced on each processor, essentially replicating the simulation verbatim. It is possible to use a serial PRNG in a parallel context by controlling access to the PRNG using traditional parallel computing techniques such as shared locks and mutexes. With a parallel simulation however this can quickly become the performance bottleneck making it preferable to use a PRNG specifically tailored for use on parallel computers. Methods of converting an existing serial PRNG into a parallel PRNG commonly involve splitting the sequential stream of random numbers into several streams with a much shorter period that can be generated in parallel. In order to do this it needs to be possible to efficiently calculate random numbers at arbitrary positions in the stream, which a lot of PRNGs are incapable of. Even where this can be done there is a non-zero probability that at some point the random number streams will overlap, given that they are all being generated by the same deterministic algorithm. An alternative is to design a parallel PRNG algorithm that can generate multiple independent sequences concurrently. The class of parallel PRNG algorithms, or PPRNGs, is a subclass of the class of PRNGs and so they share the same properties such as state size and period but also have an additional property which is the number of streams that can be generated in parallel.

There are several PPRNGs available for different parallel computing platforms. A PPRNG suited to coarse-grained parallel computing environments, such as those used in Monte Carlo simulations, is presented by C. K. Tan [117]. Their algorithm, PLFG, is based on a 32-bit LFG using parameters suggested by Knuth [65]. The number of independent streams is limited by the period of the generator used to initialise the lag tables, which in this case is a serial Mersenne Twister PRNG with a period of $2^{19937} - 1$. The choice of using a PRNG with a large period for initialisation, coupled with the specific lag values used, means the probability of multiple sequences overlapping is minimal. Implementing a parallel PRNG in this way from a serial PRNG is a form of sequence splitting. The authors demonstrate their PLFG algorithm using a 2D Ising Model Monte Carlo simulation where they find the performance, both in statistical quality of the output streams and in random numbers produced per second, is better than a PPRNG from the SPRNG library [80], which combines two LFGs using different operators.

The advantage in performance disappears however in their second example which is solving systems of linear equations using Relaxed Monte Carlo methods.

The de facto standard PRNG for Monte Carlo simulation is the Mersenne Twister [81] so called because it has a period of $2^{19937}-1$ which is a Mersenne Prime. It is a LFSR PRNG based on a recursion over the 32-bit binary field $\mathbb{F}_2^{32}$ outputting uniformly distributed unsigned 32-bit integers. It has gained popularity for its high statistical quality, or degree of equidistribution, of random numbers while having a high speed and relatively small state consisting of 624 32-bit integers. Its speed is due to the use of bitwise operations which on CPUs of the time were faster than integer or floating point arithmetic used in LCGs. Nowadays, however, integer and floating point arithmetic are almost as fast as bitwise operations so the speed advantage of the Mersenne Twister over other PRNGs is minimal. This has led the authors to redesign their algorithm to use SSE and Altivec vector instructions on Intel and PowerPC CPUs, respectively, in order to regain the performance advantage, creating a SIMD-oriented Fast Mersenne Twister [111], or SFMT. Instead of just being based on a recursion over $\mathbb{F}_2^{32}$, the SFMT is simultaneously based on recursions over $\mathbb{F}_2^{64}$ in order to output 64-bit integers and $\mathbb{F}_2^{128}$ to use 128-bit SIMD. The original MT recursion

$$g(\mathbf{w}_{32}^{624}) = (\mathbf{w}_0|\mathbf{w}_1)\mathbf{A} \oplus \mathbf{w}_m \tag{2.5}$$

involving a state vector $\mathbf{w}$ consisting of 624 32-bit integers, is adapted for SFMT to

$$g(\mathbf{w}_{128}^{156}) = \mathbf{w}_0\mathbf{A} \oplus \mathbf{w}_m\mathbf{B} \oplus \mathbf{w}_{n-2}\mathbf{C} \oplus \mathbf{w}_{n-1}\mathbf{D} \tag{2.6}$$

$\mathbf{w}$ is the state vector consisting of 156 128-bit integers and $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ are sparse 128-bit matrices chosen so that bitwise SIMD operations can be used for the matrix multiplications. The $|$ character represents the bitwise or of two integers and $\oplus$ represents bitwise exclusive-or. The indices $n-1$ and $n-2$ are chosen for speed as the values are likely to still be stored in CPU registers. This means that the generation function will be fast as it only involves loading the values from the state vector $\mathbf{w}$ at 0 and m, which can be overlapped with the other operations in the CPU pipeline. The output from the recursion is multiplied by a tempering matrix in order to increase the degree of equidistribution of the numbers generated. The SFMT PRNG also includes a block generation function. The idea behind this is that as the generation function becomes more efficient the function call overhead becomes more significant. The block generation function copies the state into an output vector and iterates over the vector length before copying the new state back, generating an entire vector of random numbers for each function call. The SFMT is implemented in ISO C99 which has support for fixed width portable integer types. Proprietary vector extensions to access SSE and Altivec instructions C are used. The

authors' study found that the SFMT is $2.1\times$ faster than the original Mersenne Twister when using the traditional generation function and $3.77\times$ faster when using the new block generation function. Furthermore when compared to four other PRNGs from the GNU Scientific Library [44] the SFMT was found to be faster on most platforms tested and for those it on which it is slower it exhibits a higher statistical quality. They note that while the WELL PRNG theoretically has a higher quality output it may not be observable in practice and is almost as fast as the SFMT.

The Mersenne Twister is highly configurable as the parameters used in the recursion can be changed to alter the properties of the PRNG. The Mersenne Twister Dynamic Creator (MTDC [82]) has been created primarily to allow multiple instances of the same Mersenne Twister to be used in parallel computing environments but can also change the bit-width of the random numbers generated or create a Mersenne Twister with a specified period. The generation function of a random number generator can be represented as a characteristic polynomial. Under the hypothesis that two RNGs are independent if their characteristic polynomials are co-prime, the Dynamic Creator embeds an integer ID into the least significant bits of the vector parameter of the characteristic polynomial and performs a search for the rest of the bits and tempering parameters that produce a PRNG that has the required period and output. The creation algorithm scales exponentially with the period and number of independent PRNGs required and is completely deterministic and therefore repeatable.

The Mersenne Twister for Graphics Processors (MTGP [112]) is a class of PRNGs based on the Mersenne Twister for use on GPUs implemented in CUDA but portable to OpenCL. The Mersenne Twister Dynamic Creator has also been adapted (MTGPDC) in order to generate parameter sets with an embedded CUDA SM ID in order to generate multiple independent streams across all SMs on a GPU. Each block of threads on an SM operate on the same PRNG which is cached in shared memory to take advantage of the memory hierarchy. If the size of the array holding the state is greater than or equal to $2n - m$ where n is the size of the state and m is the "middle" parameter from the original Mersenne Twister algorithm, then $n - m - 1$ random numbers can be computed in parallel. The state is therefore stored in a larger array than necessary to increase the parallelism. The number of threads per block is chosen to be the largest power of two less than or equal to $n - 2$ while, to keep parallelism high, m is chosen to be small but greater than or equal to 2. Due to the cost of integer conversion on nVidia GPUs an additional tempering matrix is used to directly generate random floating point numbers in IEEE754 format. The tempering matrices are stored in texture memory along with a lookup table used to speed up multiplications in the recursion formula. The use of texture memory

keeps register usage low and prevents shared memory bank conflicts. The MTGPDC was used to find parameters and thread block sizes for 128 independent PRNGs with the three Mersenne exponents 11213, 23203 and 44497 in order to benchmark performance. Again the authors compare their PRNG to the WELL PRNG which has a better quality output but in this case exhibits less parallelism. The parallel implementation of the Mersenne Twister algorithm that is distributed with the nVidia CUDA SDK consists of 4096 individual instances of the original Mersenne Twister PRNG, each with a period of 2^{607}. Each one is assigned to a thread in 32 blocks of 128 threads. The small period keeps the shared memory use of the algorithm small however it is still larger than the MTGP and the degree of parallelism and quality of the output is degraded. The performance of the MTGP is compared with the CUDA SDK implementation, a Hybrid Taus PRNG from GPU Gems [56], a Warp PRNG and nVidia's CURAND library [95]. The implementations were judged by timing how long they took to generate 5×10^7 uniformly distributed 32-bit floating point numbers. The Warp generator was found to be fastest and pass all quality tests despite having a smaller period than the MTGP. The MTGP has a larger period but fails a quality test relating to $\mathbb{F}_2$-linearity, which is common for all LFSR PRNGS. In Monte Carlo simulations, however, this failure is not considered to be a problem.

Another PRNG for CUDA GPUs has been developed by W B Langdon [68] based on a Park and Miller LCG previously implemented in C++ using the Rapidmind GPGPU toolkit. The algorithm is simple but requires at least 46-bits of integer precision. Their CUDA implementation runs 1 PRNG per thread and uses double precision floating point throughout in order to get the required precision. The authors opt to replace the algorithm in Equation 2.1 with the more complex

$$x_i = (x_{i-1} \times a) - m \times \lfloor (x_{i-1} \times a) \times m^{-1} \rfloor \tag{2.7}$$

which uses less expensive GPU instructions including the $\lfloor x \rfloor$ which represents the rounding of a floating-point number x down to the nearest integer. The m^{-1} is evaluated at compile time and replaced with a constant and a temporary variable is used to store the result of the $x_{i-1} \times a$ to save it being evaluated twice. The implementation is run on a pre-production nVidia Tesla GPU using 24 blocks of 64 threads for maximum performance and occupancy. They report the performance of their algorithm in terms of instruction throughput rather than random numbers per second by examining the PTX assembly code, counting the instructions needed to generate one random variable, then multiplying by the number generated per second. In doing this they also include the memory operations and integer conversions which are not incorporated in instruction throughput so their reported result of 35 GFlops/s and a $130\times$ speed increase over a CPU implementation is inaccurate.

The XORGENS family of PRNGs are based on XORSHIFT PRNGs with different periods and parameters. They overcome the limitation of other LFSR PRNGs that fail the $\mathbb{F}_2$-linearity test by combining the output stream with that of a non-linear Weyl PRNG. XorgensGP [88] implements an XORGENS PRNG for CUDA GPUs. In common with other parallel LFSR PRNGs, including MTGP, the state is traversed recursively using the formula

$$x_i = x_{i-r}A + x_{i-s}B \tag{2.8}$$

generating $\mathbf{min}(s, r-s)$ elements in parallel for each iteration. Each CUDA thread has its own state cached in shared memory and therefore there is one independent random stream per thread. In terms of state size and period XorgensGP lies in between PRNGs from the CURAND library and the MTGP while for speed in random numbers per second they are all roughly equal. The TestU01 battery of statistical tests was applied to the output of the three generators. As expected MTGP failed the tests relating to $\mathbb{F}_2$-linearity while the XorgensGP passed all the tests. The CURAND library also failed one of the tests which the XorgensGP passed which is unexpected given that they are both based on a combination of an XORSHIFT and Weyl PRNG.

An analysis of several common PRNGs used for Monte Carlo simulation has been carried out by Vladim Demchik [35]. Demchik took the PRNGs from popular high energy physics simulation software packages for CPUs and implemented them for ATI GPUs using the ATI Intermediate Language which is included in the ATI Stream Computing SDK. The ATI architecture has 4 32-bit components per register and memory reference presented to the programmer as x, y, z and w components of variables. This translates into 128-bit SIMD. As with nVidia GPUs single precision floating point performance is highest followed by integer performance while double precision floating performance is slowest. Demchik extracted each PRNG from the software packages and decomposed them into initialisation, generation and finalisation functions all of which were implemented for the GPU. Each PRNG was written to produce 4 32-bit random numbers per call using the 4-element vector ATI architecture. PRNG parameters were stored in global graphics memory with a separate lag table used for each PRNG instance. Each PRNG was converted to a PPRNG using the sequence splitting method. In order to compare performance between ATI and nVidia, the first PRNG implemented was the Park and Miller LCG PRNG presented by Langdon. In addition to Langdon's findings with respect to performance, Demchik notes that the short period of the PRNG will be exhausted by a modern GPU using 1024 threads in approximately 0.002 seconds making it unsuitable for large simulations. An XOR128 PRNG was next to be implemented. Being based on a 128-bit XORSHIFT PRNG it is simplest to implement on ATI GPUs using the 4 32-bit components as one 128-bit variable. It

is also fast on a GPU using only bitwise operations on integers and integer to float type conversions. It has a 64KB state when being run using 4096 threads and will exhaust the PRNG period in 10^{17} years on current hardware. RANECU, RANMAR and RANLUX PRNGs were also included in the study. RANECU uses slower integer operations but having a simpler algorithm compensates for this. It has 128KB of state when run with 4096 threads and will exhaust the period in 31 years. Both RANMAR and RANLUX are able to generate uniformly distributed floating point numbers directly without needing conversion instructions and in addition RANMAR can potentially produce 900 million independent sequences. When run with 4096 threads RANMAR consumes 6MB of memory to store its state. RANLUX requires to discard a large number of initial random numbers. The original algorithm was substantially rewritten to take into account of the memory hierarchy by rearranging the layout of the state and temporary array used to discard the initial numbers. It requires 448KB of memory to store PRNG state which is small enough to fit in the GPU cache. Finally, the implementation of the Mersenne Twister from the CUDA SDK was also ported to ATI GPUs. It requires 19 4-component vectors in memory to store its state consuming roughly 10MB. On ATI the CUDA implementation uses too many registers so has to store temporary variables in much slower graphics memory. Performance was compared against the CPU implementations of the PRNGs. On the CPU the XOR128 and RANMAR PRNGs were found to be fastest. However, the PRNGs were run in single threads and multiplied by the number of available on the CPU to obtain a figure for multi-threaded performance. A simple application of Amdahl's Law [3] shows why this is a theoretical best-case performance figure that will never be obtained in practice. The ATI GPUs used in the performance benchmark were simultaneously being used to drive graphics displays. They were timed for 1000 iterations generating 4×10^7 random numbers each iteration storing the results in graphics memory. XOR128 and Langdon's PPRNGs were found to be fastest on the GPU as they use the smallest number of memory operations per random number generated indicating that graphics memory bandwidth is the main bottleneck in all PPRNGs studied.

2.2.2 General Solutions for Parallelising Monte Carlo Algorithms

An introductory overview of the main challenges faced when performing Bayesian Inference in a parallel computing context is given in [128]. In a continuous setting the aim is to infer the parameters of a probability density model constructed using observed data from a system of interest. The parameters themselves are uncertain and so are assigned a prior probability density. Bayes' Theorem is then used to evaluate the posterior density given the likelihood of the data for a set of parameters. The parameters being inferred may contain other hidden unobserved parameters and also measurement error so this must be integrated out of the likelihood.

This integration is commonly intractable so computationally intense numerical integration using Monte Carlo methods is used. To decompose this into a set of independent tasks that can be performed on a parallel computer, the underlying conditional independence structure of the statistical model can be analysed. The structure is represented as a directed acyclic graph with nodes representing parameters and edges representing dependencies between the parameters. There is more than one possible graph for any statistical model and the more sparse a graph is, the more independent the parameters are and it becomes possible to perform many independent local computations in parallel before combining results [128].

Monte Carlo integration is computationally intense but also "embarrassingly parallel". Calculating the expectation of a particular probability distribution for example involves computing an average of a large number of independent samples from the distribution. By the Law of Large Numbers this is guaranteed to converge to the true value of the expectation with the error decreasing as the number of samples increases. Samples can be generated from the distribution and a partial average computed on separate computational nodes before being combined into a single expectation value by a master node. With Monte Carlo algorithms being trivial to implement in parallel the main obstacle to overcome is generating large quantities of independent random samples across multiple computers using a suitable PPRNG.

When the dimensionality of the posterior becomes large it generally becomes impractical to implement a Monte Carlo integration scheme. Markov Chain Monte Carlo methods are used instead, which as described in the first chapter start at an arbitrary position within the distribution and generate a new sample based on the previous according to some acceptance ratio that ensures convergence to the stationary distribution. MCMC methods converge after a number of samples have been generated during the transient phase of the chain, called the "burn-in" period. When several chains are being run in parallel each must be burned in and a large number of samples are wasted, although of course a much larger number of samples can be generated in the same amount of time compared to running one much longer chain. In order to minimise the number of samples wasted due to burn-in it is advantageous to explore techniques to improve the convergence of a single chain before attempting to run several in parallel. It may also be worthwhile investigating the generation of a single chain in parallel although this does not scale as well as running several serial chains in parallel. When chains need to be burned in there are diminishing returns when adding more chains in parallel therefore the design of an MCMC algorithm and its parallel implementation should be considered together. Exact sampling methods are able to generate a sample from a distribution without burn-in, however these are not widely applicable and are expensive to compute. Each parallel Markov Chain

could be initialised with an exact sample to completely remove the issue of burn-in. Indeed, even one exact sample could be used to initialise one chain and used to generate further samples to initialise the rest of the parallel chains. If exact sampling is not available then the chains' starting positions are usually chosen to be over-dispersed with respect to the prior distribution and the chains are burned in until they converge. If a Markov Chain requires a long burn-in then it may be more effective to find ways to generate a single chain in parallel. The Markov property prevents a single chain from being fully parallel, therefore any parallelism will be confined to sample generation. Markov chains generate correlated samples from the distribution of interest, and so the number of "effectively independent samples" generated per second is often used as an appropriate metric to assess the performance of an MCMC algorithm.

Rosenthal [108] presents an outline of issues that become apparent when using parallel algorithms for Monte Carlo methods and suggestions on how to overcome them. There are a wide range of parallel computing systems ranging from Cray supercomputers to individual desktop PCs communicating over the internet. In between these two extremes are small local networks of ordinary PCs that can be used as a cluster. This is becoming an increasingly common parallel computing environment.

Computing an expectation of a function of a random variable distributed according to some probability distribution using Monte Carlo is easy to run in parallel. Each PC generates a number of samples from the distribution and computes its own local mean before a master PC collects all the results and computes the overall average. The result has the same mean as a sequential algorithm and the variance is reduced linearly with the number of PCs used therefore it is more important to start a parallel Monte Carlo simulation with a unbiased estimator than a low variance one. The number of samples generated by each PC should be dependent on the CPU speed and the resulting average weighted accordingly. For a small independent probability of any PC failing to return a result the master PC can just ignore it. If the probability of failure is dependent on the number of samples generated but bounded by an upper value then the bias introduced by ignoring the result is at most Mp where M is the upper bound of the samples generated and p is the probability that the processor will fail. Repeating experiments that fail on another PC increases the runtime to approximately double. When PC speed is unknown then a time can be specified for each simulation to stop. When this time approaches all PCs should stop the generation of the current sample unless they are still working on the first. This is known as the Unbiased Stopping Rule.

Parallelising Markov Chain Monte Carlo by running multiple chains in parallel is also trivial and gives a slightly less than linear speedup with the number of PCs used due to having

to burn in a Markov Chain on each one. To determine burn-in time for each chain convergence diagnostics can be used but this may introduce bias into the samples if the method used leads to a burn in time that is too short. Output from the chains can also be used to diagnose convergence as well as theoretical burn in bounds and simply using a fixed predetermined burn in time. When perfect sampling algorithms are available for MCMC, they should be used even if they are expensive to compute, as each Markov Chain can be initialised with a sample from the target distribution then apply the Markov kernel each iteration to produce a new sample. This removes the need for any burn in and subsequent bias that may be introduced by using convergence diagnostics. When a group of homogeneous PCs is connected together reliably Metropolis-Coupled MCMC can be performed which is similar to population MCMC [108]. Metropolis-Coupled MCMC involves updating a group of Markov Chains in parallel then proposing swaps between chains with acceptance governed by the Metropolis-Hastings ratio [47].

Brockwell [24] presents a novel algorithm for parallel MCMC when other methods are infeasible. He observes that Monte Carlo algorithms are trivial to implement in parallel, however Markov Chain Monte Carlo algorithms are not nearly as trivial. MCMC is easiest to parallelise by running multiple chains in parallel and combining their results. If burn in is long however it is better to parallelise the generation of a single chain. This could be done by partitioning the state space into blocks and having a chain explore each block, however this requires analysis of the target distribution and so is not always possible. He proposes pre-fetching as a solution to this problem by calculating multiple likelihoods at once in parallel ahead of time avoiding having to analyse the target distribution. Conceptually any task that can be divided into multiple independent subtasks can be parallelised by performing those tasks in parallel on separate CPUs. It is critical that the size of each subtask is large enough to outweigh any overhead in transferring the task to another CPU. When running parallel simulations involving random number generators it is important that they generate streams of random numbers that are independent of one another. To do this separate seeds can be used or, better, a library such as SPRNG (http://sprng.cs.fsu.edu/) which provides parallel random number generator implementations. CPU speed may also be important in performing a parallel simulation therefore he also suggests using load balancing or a queue to ensure slower CPUs get less work than faster ones. Queues are only effective for smaller subtasks with a low communication overhead compared to the workload.

Brockwell's goal is to use parallel processing to speed up the generation of a single Markov Chain. It is significant that he suggests it is more worthwhile to use high performance libraries in an attempt to increase speed before investigating other approaches to parallel processing. He

then outlines two existing methods of performing MCMC in parallel: regeneration and blocking. Regeneration picks a point in a discrete state space and then runs multiple chains starting from that state in parallel until they end on that state. The values of the chains can then be concatenated into one larger chain. He mentions that this approach can be modified for continuous distributions but in general is not suited to distributions with a high number of dimensions as the probability of returning to the starting state reduces as the number of dimensions increases. Blocking involves splitting the state space into partitions with a chain exploring each one. The partitions have to be chosen carefully to create a valid chain and updates have to be able to be carried out in parallel. He then introduces his own method, pre-fetching. It is a viable alternative to other methods when the likelihood calculation is the rate limiting step. It generates a tree of states from the bottom-up by calculating one likelihood in parallel on each CPU. Each parent node has the same likelihood as the rejection child. The master CPU takes steps down the tree from the top evaluating the Metropolis-Hastings ratio at each step and deciding which state to visit next. The algorithm requires $2^{\text{height\_of\_tree}}$ homogeneous CPUs for best performance. The speed increase is $\log(\text{no\_of\_CPUs})$ and is not very efficient as $2^{\text{height\_of\_tree}} - 1$ calculated paths are wasted. However it is straightforward to implement and provides a useful alternative parallel algorithm when no other is appropriate.

His experiments were performed using Bayesian analysis of an ARFIMA process with an uninformative, high-variance prior on the process parameters. ARFIMA (Auto-Regressive Fractionally Integrated Moving Average) models are used to model time series that have long term memory. Each Metropolis-Hastings step picks one of three parameters and generates a random walk proposal. This was performed on a cluster of 32 Dual CPU 1.6GHz Athlon workstations running Linux connected together with gigabit ethernet. The algorithm was implemented using C++, GSL and MPICH and run three times for ten thousand iterations. Each likelihood calculation took on average 8ms to complete. A drastic decrease in performance was noticed when CPU speeds were different.

Brockwell mentions that further algorithmic refinements are needed to reduce the effect of different processor speeds on the performance of the method. One such approach he suggests is distributing the same likelihood calculation to multiple processors and using the first answer returned, although this would likely make the algorithm very inefficient as the same calculations would be being carried out multiple times. Other approaches mentioned include performing some online analysis of processor speeds or having the sequential part of the algorithm travel down the tree as likelihoods are being calculated and cancel those that are no longer needed. It has also been suggested that the tree could be analysed and a guess could be made where the

chain may end up. This would allow the tree to be made deeper in those directions. The use of hardware accelerators or multi-core processors has not been explored in this context, and would likely provide further speed increases.

Motivated by the use of MCMC for Bayesian Inference of images in computational biology, Byrd [26] has developed a general method for running any MCMC algorithm in parallel called "Speculative Moves". The idea behind Speculative Moves is to generate a number of proposals in parallel across each processing core in effect performing multiple expensive likelihood evaluations in parallel. If the first sample generated is not accepted into the Markov Chain then the second will be evaluated and so on until one is accepted and the rest discarded. If all proposed samples are rejected the state of the chain remains the same. In this respect the algorithm is inefficient as at most one sample will be accepted into the Markov Chain for all samples generated, wasting processing power for those that are discarded. The increase in speed is therefore dependent on the acceptance rate of the MCMC algorithm and there are diminishing returns when adding more processing cores.

The high dimensionality of medical imaging problems coupled with having to burn-in a Markov Chain means that hours of computation are needed for each image. Speculative Moves is targeted towards shared memory architectures comprising multiple processor cores either in a single processor or multiple processors connected to the same computer. It can also be run on small-scale clusters and may be combined with other MCMC optimisation and parallel techniques. Obtaining samples after burn-in is embarrassingly parallel so this method is primarily concerned with speeding up the generation of samples during burn-in by optimising the implementation of any MCMC algorithm rather than speeding up convergence as is common with other MCMC algorithms such as Metropolis-Coupled MCMC.

Byrd demonstrates the utility of his method by applying it recognising artefacts in medical imaging with between a $1.5\times$ and $15\times$ speed up over a sequential CPU program. When the rejection rate is as high as 75%, as is common in MCMC algorithms, computational savings of between 40% and 60% can be had when using dual core or quad core CPUs. Using multi-core CPUs results in a greater saving as the communication between cores is faster than using multiple CPUs.

2.2.3 Specific Parallel Monte Carlo Algorithms

In conventional Metropolis-Hastings the current state of the Markov Chain is required to be fully realised before a proposal sample can be generated. It is this Markovian property that makes most MCMC algorithms non-trivial to implement efficiently in parallel. Such MCMC algorithms may be run in parallel using multiple independent Markov Chains and combining

them in some way such as waiting for each chain to reach the starting state of another chain then concatenating them, creating a single large chain from multiple smaller tours. These methods do not speed up the burn-in period however, due to each chain having to be burned-in separately. In addition, combining chains using tours can only feasibly be applied to discrete distributions. In Independent Metropolis-Hastings, as its name suggests, the proposed sample can be generated independently of the current state of the Markov Chain and provides a means of implementing an efficient parallel MCMC algorithm.

Jacob *et al.* [59] propose that all samples in a Markov Chain and any part they contribute to the acceptance criteria may be computed in parallel when using Independent Metropolis-Hastings. This leaves a simple draw from a uniform distribution along with a multiply to complete the acceptance calculation and a comparison operation to be carried out sequentially when all samples have been generated. Since sample generation and likelihood calculations are the most computationally expensive parts of any MCMC sampler, this converts an inherently sequential algorithm into one that is easily parallelisable and enjoys almost linear speedup with respect to the number of processing cores employed.

Samples from a Markov Chain are usually used in the same manner as most other Monte Carlo simulations to compute the expectation of some function of the target distribution. Jacob *et al.* also introduce a new algorithm called Block Independent Metropolis-Hastings that decreases the variance of their estimator. Block IMH splits the length of the Markov Chain into b blocks of length r, where p is the number of processing cores. Each core starts at the same sample but includes the other precomputed samples in a different order. This makes most effective use of the samples and likelihood calculations as they are time-consuming to compute. After r iterations, a chain is selected at random, either uniformly or using weighted sampling to improve convergence, and the final value of the chain is distributed to the rest of the chains to continue generation of the next block of samples. The algorithm outputs a Markov Chain of length $b \times r$ and an array of $b \times r$ samples for each of the p processing cores. If generating uniform variables is assumed to be negligible then Block IMH has the same computational cost as non-blocked IMH. Block IMH is most efficient when using blocks of size p (i.e when $r = p$) because p cores are used to generate the r samples needed for each chain. However, if only the output chain is needed, selecting $r < p$ could be used to save memory used to store each block chain.

In order to demonstrate the variance reduction of Block IMH, an example MCMC simulation was run using a standard normal target distribution, with zero mean and unit variance, and a Cauchy proposal distribution for $10,000$ iterations with an acceptance rate of 70%. The esti-

mator used was the expectation of the target distribution and the variance was compared against a standard Independent Metropolis-Hastings sequential, single-chain estimator. The choice of sample permutation method was found to have an effect on the variance reduction. Averaging over all permutations would be best although this is infeasible as the number of permutations of a set of p samples is $p!$. With the number of blocks, b, fixed at 1, the number of parallel chains, p, and samples, r, was chosen to be 4, 10, 50 and 100. For each p, a number of permutation schemes were used in order to find which is best by comparing the variance of the Block IMH estimator with that of the standard estimator. The permutation schemes tested were a "fixed order" scheme where each chain processes the same samples in the same order; a "circular" scheme where the sample order is fixed among chains but each consecutive chain starts with a consecutive sample; and, a three random permutation schemes. The random schemes were truely "random", "half-random" where the first half of the samples are permuted randomly then the following half a reversed copy of the first, and "stratified" random where an ordering is imposed upon the samples and they are then chosen so that consecutive chains have values that are "far" from one another. As expected, the fixed scheme was worst and the three random schemes roughly equivalent and better. The true random permutation scheme was chosen to be the best for its simplicity. Another, more complex example was also run using a probit regression model on a real-life RPIMA dataset resulting in a 60% reduction in estimator variance. The decrease in variance was found to be linked to the acceptance rate of the sampler in that as the acceptance rate increases, the reduction in estimator variance decreases.

Tibbits *et al.* [119] advocate the use of a Slice Sampler to avoid the slow random walk behaviour of standard Metropolis-Hastings or Gibb's samplers. Univariate Slice Samplers can be used to explore multivariate distributions. However they are less effective than if used with a univariate target distribution. Multivariate Slice Samplers are better for multivariate targets but are difficult to construct and computationally expensive due to the large number of likelihood evaluations needed at each iteration. They therefore explore the use of GPUs to construct a multivariate slice sampler that performs the likelihood evaluations in parallel comparing two implementations using OpenMP on CPUs and CUDA on GPUs.

A univariate Slice Sampler takes two steps to generate a sample, x, from a distribution, π. First some probability h is sampled uniformly between 0 and the current probability $\pi(x)$, i.e. $h \sim \mathcal{U}(0, \pi(x))$. A sample is then drawn uniformly from the slice through the target distribution, consisting of all points with probability greater than h, i.e. $x \sim \mathcal{U}(x : \pi(x) \geq h)$. It is usually required to change the size of the slice to properly encapsulate a slice of the distribution to sample from, which may be challenging to do. If the parameters are highly correlated the

sampler can perform badly. This is illustrated in [119] with a simple linear regression model where a Metropolis-Hastings sampler is compared to univariate and multivariate slice samplers with the multivariate slice sampler performing best. A multivariate Slice Sampler is similar to a univariate one, except that instead of a 2D slice being constructed a multidimensional hypercube is. This requires a large number of computationally intense likelihood evaluations which grows exponentially with the dimensionality of the target distribution. Also as the dimensionality increases the number of samples rejected increases lowering the performance of the sampler. To assess the performance of the univariate and multivariate Slice Samplers when non-trivial likelihood calculations are involved, a Gaussian process model was constructed using a synthetic dataset of spacial data. Five datasets with different numbers of locations were generated and the samplers run on them for $10,000$ iterations which was enough to guarantee $1,000$ effective samples. All chains were started on actual values to avoid the choice of starting value affecting the performance of the sampler. The multivariate Slice Sampler was found to perform better on all datasets with the univariate sampler performing progressively worse as the dataset complexity increased.

In both the OpenMP and CUDA based parallel implementations of the multivariate Slice Sampler the proposal and likelihood evaluations and slice construction are all performed in parallel. Even in 3 dimensions the hypercube slice construction requires 8 likelihood evaluations, all of which are independent. If the hypercube is required to be resized to properly encapsulated a complete slice of the distribution then the resizing and re-evaluation of the likelihoods are also performed in parallel. The rejection sampling step is also performed in parallel batches stopping with the first accepted proposal. For OpenMP the batch size is equal to the number of threads used while in CUDA it is equal to the number of SMs available. The multivariate CUDA Slice Sampler also parallelises the matrix operations within the likelihood function. The OpenMP sampler requires tuning to find the optimum number of threads to use. It was found to be dependent on the complexity of the dataset being used requiring 3 threads for the datasets with 300 and less locations and 4 threads for those with 400 or 500 locations. A 10% performance penalty was observed when moving from 4 threads to 5 due to 4 threads being able to fit on one quad core CPU and hence communicate faster than 5 threads which would be split across both CPUs in the test system. A special multithread-aware memory allocation library was used to further speed up the parallel CPU implementation. The upper bound on dataset size of 500 locations was imposed by hardware constraints on the GPU.

The CUDA implementation of the parallel multivariate Slice Sampler involves 5 steps to evaluate each likelihood. These are performed by a block of 512 threads with one dataset loca-

tion assigned to each thread. The steps are combined in such a way as to minimise the amount of intermediate results needing to be stored and to perform as much computation ahead of where it is needed. Much of the computational expense of the likelihood evaluation is consumed in the Cholesky decomposition which forms part of the likelihood. The CUDA Occupancy Calculator was used to find the optimum number of threads to use and a small search was used to find the best 2D thread block arrangement for the Cholesky decomposition. As more capable GPU hardware becomes available that can support more threads per block the authors hope to be able to increase the number of simultaneous likelihood evaluations however they fail to mention whether or not they investigated processing more than one dataset location per thread. A version of the sampler that also scales across multiple GPUs would increase the number of simultaneous evaluations to 960 with current technology.

Comparing the sequential univariate and multivariate Slice Samplers to the OpenMP and CUDA multivariate samplers the OpenMP multivariate sampler was found to have a 40% performance increase over serial CPU code. The GPU implementation was found to perform $15\times$ faster than the univariate sampler and $5.6\times$ faster than the serial CPU multivariate sampler. The Effective Samples per second (ES/s) of the each sampler decreases as the number of locations in the dataset increased due to the strong dependence between parameters. The univariate sampler was not implemented in parallel as there is thought to be little benefit. The performance results obtained with test data were similar to those obtained when the samplers were applied to a real life example analysing surface temperature data from the US.

2.3 Parallel Numerical Libraries

As noted by Brockwell [24], possibly the easiest way to improve performance of MCMC algorithms is simply to use a more efficient numerical library for the required linear algebra routines. In this section, we consider the numerical libraries available for a variety of architectures.

2.3.1 LAPACK

The LAPACK project [15] aims to provide a linear algebra library that is efficient on a wide range of high performance computers. It is developed by a group of academic and private researchers from the US and UK and extends earlier EISPACK and LINPACK projects. LAPACK specifies a standard library interface with routines for solving systems of linear equations, performing least squares regressions, calculating eigenvalues and performing matrix decompositions. It supports dense and banded matrices but not those stored in any sparse matrix format. The functions are implemented for real and complex numerical types in single and double precision floating point arithmetic. A reference Fortran implementation of LAPACK is available

through the Netlib website although this is a generic implementation and better performance can be had by using an optimised library available from CPU manufacturers.

The EISPACK and LINPACK projects ignored the cost of accessing data elements in computer memory which leads to poor performance on modern computers where the floating point performance is much faster than memory access. Modern computers have a memory hierarchy with multiple levels of fast cache memory to store frequently used data to overcome the cost of accessing slower main memory. LAPACK is therefore designed to reuse data as much as possible to run at the speed of the floating point units rather than at the speed of the memory. Recent CPUs also have multiple processing cores and LAPACK is written to expose any available parallelism to the scheduler.

LAPACK relies on an optimised BLAS implementation for best performance on any computing platform. BLAS is a similar library specification to LAPACK that contains simpler linear algebra functions operating on vectors and matrices. The BLAS are organised into three levels. Level 1 of the BLAS was first to be proposed and performs operations on vectors [69]. It is efficient on scalar CPUs but not vector or parallel CPUs. Levels 2 and 3 of the BLAS were proposed later involving vector-matrix and matrix-matrix operations respectively [38, 37]. Level 3 of the BLAS has the highest ratio of floating point operations to data elements needed (FLOP to word ratio) of the 3 BLAS levels and so level 3 routines have more opportunities for data reuse and can benefit most from CPUs with a memory hierarchy. Level 2 BLAS operations present less opportunities for data reuse than level 3 operations but more than level 1. As with LAPACK a reference implementation of the BLAS written in Fortran is available from the Netlib website however it is not optimised for any particular computer architecture.

LAPACK and BLAS routines follow a standard naming convention based on the type of matrix they operate on. The names consist of four, five or six letters. The meaning of each letter is explained in Appendix A of the LAPACK Installation Guide [22] and the proposals for the level 2 and 3 BLAS [38, 37]. The first letter of any BLAS or LAPACK routine specifies how each each data element in the matrix is stored and will be “S” or “D” for single or double precision floating point, respectively, or “C” or “Z” for consecutive pairs of single or double precision floating point numbers representing the real and imaginary parts of a complex number. The following two letters in the routine name represent the form of the matrix and include “GE” for general matrices, “SY” for symmetric matrices, “TR” for upper or lower triangular matrices and “PO” for symmetric positive-definite matrices. The remaining letters indicate the operation the routine performs. As an example the BLAS routine that performs single-precision general matrix-matrix multiplication is named “SGEMM” while the LAPACK routine that performs

Single precision	Double precision	Explanation
SDOT	DDOT	Dot product of two vectors
SSCAL	DSCAL	Multiplication of each element in a vector by a scalar value
SGEMV	DGEMV	General matrix-vector multiplication
SGEMM	DGEMM	General matrix-matrix multiplication
SSYRK	DSYRK	Symmetric rank-K update
STRMM	DTRMM	Triangular matrix multiplication
STRSM	DTRSM	Triangular matrix solve
STRTRI	DTRTRI	Triangular matrix inverse
SLAUUM	DLAUUM	Multiplication of an upper or lower triangular matrix with itself
SPOTRF	DPOTRF	Positive-definite triangular matrix factorisation or Cholesky decomposit
SPOTRI	DPOTRI	Calculate the inverse of a matrix from its Cholesky decomposition

Table 2.1: BLAS and LAPACK acronyms used throughout this thesis

double-precision positive-definite triangular factorisation is named “DPOTRF”. The LAPACK acronyms frequently used in this thesis are summarised in Table 2.1.

Some level 2 and 3 BLAS and LAPACK routines also specify “option arguments” [38, 37, 22]. These define miscellaneous options for each subroutine and are implemented as character arguments in Fortran. There are four option arguments named “trans”, “uplo”, “side” and “diag”. “trans” is set to “N” when a matrix argument is not to be transposed by the routine, “T” when the transpose is to be used and “C” when the conjugate transpose is to be used. “trans” appears in BLAS 3 routines as “transA” and “transB” when a routine performs an operation on two matrices. For routines that operate on only the upper or lower half of a matrix, “uplo” can be set to “U” or “L” respectively. “side” is used exclusively in BLAS 3 operations to specify whether a triangular matrix appears on the left, “L”, or right, “R” of an equation to solve. “diag” is also used for triangular matrices and is set to “U” when it is assumed that the diagonal is all ones and “N” when it is not.

All the algorithms used in LAPACK were rewritten as a sequence of operations on matrix blocks in order to use computationally intense routines from level 3 of the BLAS. Each algorithm has multiple ways of being rewritten to use block operations and the block algorithm chosen is the one that is expected to give the best average performance across different architectures. Blocking each algorithm also introduces a parameter, the block size, that can be tuned for each architecture so that the entire matrix being operated on fits in the CPU cache. Writes to lo-

calised areas of memory containing the block are also fast if the CPU cache is a "write-through" cache. LAPACK also contains unblocked versions of blocked routines which form part of the blocked algorithm. The unblocked versions of the LAPACK routines follow the same naming conventions but end with a "2" and may miss out one of the last three characters in the name to remain within the six-character limit.

LAPACK is designed to be efficient on computers with less than 100 vector CPUs while on single serial CPUs it should be no worse than any existing EISPACK or LINPACK implementations. BLAS performance is critical to the efficiency of the algorithms on shared memory systems while on distributed memory systems exploring parallelism within each block algorithm is also possible. Using an existing shared memory LAPACK implementation as a starting point for a distributed memory version is desirable as reducing memory accesses is also an aim in distributed memory systems where the cost of data access is far higher. Each routine in LAPACK is modular and self-contained and some contain the possibility of exploiting more than simple loop level parallelism. Therefore each routine would have to be analysed separately to produce a parallel distributed memory LAPACK implementation.

The first reference implementation of LAPACK was written in Fortran 77 using non-standard extensions for double precision complex data types. Routines that are available in multiple precisions are automatically generated from a code template as far as possible. Experiments conducted show that on a single CPU Cray system, 90% of the peak theoretical arithmetic performance was achieved and, on a multi-CPU Cray system, 70-80% of peak performance was achieved. This performance is similar to the matrix-vector and matrix-matrix mulpliy routines from the BLAS library in use which are the limiting routines of the LAPACK operations benchmarked.

One of the drawbacks of Fortran 77 is that it does not provide routines for dynamic memory allocation. This means that any LAPACK routine which requires a temporary working space in memory has to have an appropriately sized workspace argument passed in. Fortran 90 does not have this restriction and also has operations on arrays which are more suited to linear algebra. Fortran 90 and C implementations of LAPACK are intended also using automatic code translation as far as possible. In addition it is planned to add more routines to the LAPACK specification and add more tuning parameters other than the block size. A distributed memory version and one that takes advantage of more specific features of certain CPUs is also planned.

2.3.2 Optimised BLAS

Several hardware vendors currently produce optimised BLAS libraries for their products, usually as part of a larger library of numerical routines.

Intel develops the Math Kernel Library (MKL) [4] which is highly optimised for its range of CPU products. The MKL contains complete BLAS and LAPACK implementations as well as ScaLAPACK, FFT and LinPACK libraries. Also included are a range of vector PRNGs and a vector math library. The product brief boasts significant increases in performance over alternative libraries for the Intel computing platform. It is available in 32-bit and 64-bit sequential and multi-threaded versions (using OpenMP) and is free to individuals and academics. Any company that wishes to incorporate the MKL into their product has to pay for a commercial licence. Intel currently provides the MKL as part of its parallel studio suite of applications which includes its highly optimised C and Fortran compilers.

The AMD Core Math Library (ACML) [2] is a free set of numerical routines optimised for AMD's range of Opteron processors written and maintained by AMD. It also contains a complete Level 1, 2 and 3 BLAS implementation as well as a LAPACK implementation that is further optimised on top of the BLAS, and FFT and RNG routines. It is freely distributed and available in 32-bit and 64-bit single and multi-threaded versions using OpenMP.

nVidia distributes a BLAS library with its CUDA GPGPU Toolkit. CUBLAS [91] is implemented using CUDA to run on a single GPU and provides additional functions to upload and download vector and matrix data from the GPU. Although it is supplied with a C/C++ interface primarily it uses column-major memory layout for matrices in order to be compatible with traditional Fortran BLAS. A Fortran interface is also available. CUBLAS now implements the entire range of level 1, 2 and 3 routines in the BLAS specification although the interface for each function has been changed in order to supply a GPU "context" for the kernel execution to make the library thread-safe and asynchronous. Each CUBLAS function also returns an error status to indicate whether the GPU execution was successful which is another extension to the BLAS specification necessary for GPU BLAS.

For AMD GPUs the Accelerated Parallel Processing Math Library [1] is available using OpenCL to implement some FFT functions and level 2 and 3 BLAS routines.

2.3.3 ATLAS

LAPACK requires an optimised BLAS library to be available for a computing platform in order to offer fast performance. Each computing platform has a different number of registers, number and size of caches and processing pipeline which makes producing an optimised library for any given platform require a significant effort in producing hand written instructions to get the best performance. For computing architectures that do not have a large market share this investment is not economically viable.

ATLAS (Automatically Tuned Linear Algebra Software) [127] is a project which aims

to automatically produce BLAS and LAPACK libraries optimised for a given platform. The requirements for ATLAS are a CPU with cache and a software environment that includes a C compiler to compile code for the CPU. If the C compiler turns out to be inadequate a Fortran version of ATLAS will be used instead. Since LAPACK depends heavily on BLAS and most BLAS operations can be expressed in terms of matrix-multiply, the problem becomes one of producing an optimised GEMM routine.

Like LAPACK, ATLAS' GEMM routine is blocked so that the matrix blocks being worked upon fit into the CPU caches. ATLAS isolates the architecture-specific features into several small kernel functions that perform a fixed-size matrix-multiply. A larger function combines the smaller kernels into a complete GEMM routine and is largely unchanged on different architectures.

The architecture-specific GEMM kernels implement matrix multiply of the form $A^T B + C$ as this has the largest ratio of floating point operations to cache misses so presents the best opportunity for cache reuse. A block of A is loaded into the L1 cache and the columns of B are traversed to produce a block of C. Cache reuse is optimised when an entire block of A fits in the cache along with two columns of B and a cache line for the element of C being calculated. The inner k dimension is unrolled to fill, but not overflow, the instruction cache as well as remove loop overhead. The loops over m and n are not completely unrolled as this would overflow the instruction cache and also likely change the memory access pattern to one less optimal. Two versions of the unrolled kernel are produced for architectures that have a fused multiply-add instruction and one for those that do not, using separate multiply and add instructions. Some architectures also have multiple floating point units so to expose the parallelism to the compiler the m and/or n loops are unrolled. It is theoretically possible to control the exact number of cache misses in the GEMM kernel but in practice it is hard to achieve. Unrolling the m and n loops gives some control over this. At each step, the code generator creates a number of GEMM kernels and uses a timer to select the fastest one to modify in the following step. The result is an optimised GEMM kernel unrolled for square blocks and several other kernels with rolled loops to handle cases where the dimension is not a multiple of the unrolling factor. The process is repeated to create optimised kernels for the $A^T B + C$, $A^T B - C$ and $A^T B + \beta C$ cases.

The larger complete GEMM routine combines the smaller kernels for fixed size square matrices with the kernels handling the odd-sized cases and handles loading the matrices into the cache, transposing and multiplying by α. The first check performed is whether the optimisations will be worth performing for the problem size. If not, or if the memory allocations required to execute the optimised routine fail, a smaller GEMM routine with three simple mod-

erately unrolled loops is called. The point at which the optimised routine becomes beneficial is dependent on how fast the architecture can execute functions and multiple layers of loops and is determined when ATLAS is installed. In the larger GEMM algorithm the k dimension is also the innermost and two algorithms are written with m or n as the outermost dimension. A heuristic on k determines whether a temporary area of memory is allocated for writing to the block of C. Writing to temporary memory requires copying the temporary results back to C although the memory allocated can be aligned to allow efficient loads and stores. A panel each of A and B are copied to temporary memory and transposed if needed. If possible, when m is the outermost dimension it is advantageous to copy B entirely in one go and similarly copy A when n is outermost. Whether to use m or n as the outermost dimension is decided by which will give the best L2 cache use. L2 and higher caches vary widely in their implementation and behaviour across different computing architectures therefore no explicit blocking is performed in L2 unless the user specifies the L2 cache size to the ATLAS installer.

All the optimisations performed by ATLAS could be done automatically by a C compiler however such a compiler would be require a significant effort to write and the effort would not be worth it for most architectures. When timing the routines ATLAS flushes the CPU caches between benchmark runs, ensuring that the timings recorded are more likely to be worst case.

2.3.4 Linear Algebra on GPUs

Before the advent of GPGPU using CUDA and OpenCL, developers used various languages to develop GPU kernels including C for graphics (Cg), High Level Shading Language (HLSL) and OpenGL Shading Language (GLSL) which replaced hand coding kernels in proprietary GPU assembly instructions. Kernels would use GPU vertex processors more than fragment processors as the GPUs contained more vertex processors. 3D graphics APIs such as OpenGL and DirectX were used to load data into graphics memory as textures and trigger the kernel execution by drawing polygons. The results would be rendered into the frame buffer and could be read from there.

Jung [60] uses the BrookeGPU library to hide the complexity of transferring data and launching kernels on the GPU when developing a Cholesky decomposition for a nVidia GF6800 GPU with 16 4-way SIMD fragment processors. At each iteration the Cholesky decomposition performs three steps: a square root of the diagonal element and normalising and updating the submatrix. A kernel is implemented for each step with extra temporary memory allocated in order to allow instruction streams to overlap without producing undefined results. BrookeGPU has no support for triangular matrices unlike OpenGL which makes their Cholesky decomposition slower than a similar LU decomposition which is uncommon. Their algorithm also does

not take into account any GPU caches which makes the outer product form of the Cholesky decomposition perform better than the inner product form implemented even though it exhibits less parallelism. The rate limiting step of the Cholesky decomposition is the square root which cannot be performed in parallel. This means that for large matrices the speed of their algorithm is bound by memory bandwidth rather than instruction throughput.

BLAS implementations are also available for FPGAs. While GPUs have a high theoretical throughput only a fraction of peak performance is available. FPGAs have a peak performance that is easier to attain and they are also more power efficient than more general purpose CPUs and GPUs. Kestur *et al.* [64] have carried out a comparison of BLAS implementations on an FPGA, GPU and CPU in terms of power efficiency as well as throughput.

They start by implementing an IEEE754 compliant double precision dot product and scalar-vector multiply-add from level 1 of the BLAS (DDOT and DAXPY respectively) and use them to produce a double precision matrix-vector multiplication BLAS level 2 DGEMV function. They use a new method of reduction for FPGAs in the DDOT and a new way of storing vectors and matrices in FPGA memory in order to improve parallel computation in the DGEMV. To perform the sum reduction in the dot product the authors start with a single accumulator which feeds the running total back into the input until the list of input elements is exhausted. The input is processed in batches producing several partial sums which are then coalesced into one result. The single accumulator is improved by first adding another to create a double accumulator and then using multiple feed-forward adders to perform the coalesced sum in $log_2(n)$ steps. The feed-forward adders reduce latency in producing the final sum and the dual-stage adder reduces the RAM bottleneck further speeding up the reduction.

In order to produce a DGEMV, Kestur *et al.* [64] perform multiple independent dot products across the rows of a matrix in parallel using a DAXPY kernel. In order to improve memory bandwidth when multiple sequential accesses are performed, they introduce bank interleaved memory in a similar manner to shared memory on a GPU. Sequential elements are stored in sequential banks, all of which can be accessed simultaneously at full bandwidth. Vectors are stored in bank interleaved memory while the idea is extended to two dimensions to store matrix elements.

The experiments were conducted using a PC with a 3.16GHz Intel Core 2 Duo and 4GB RAM running the Intel MKL. An nVidia 9500GT was added to the system using CUBLAS 2.2 to benchmark GPU performance but was removed when not in use so as not to effect power consumption measured. A BEE3 FPGA was used running at 100MHz with a maximum of 16GB memory. The FPGA was found to have much better instruction throughput than the PC

and was only slightly slower than the PC. However, when measuring the number of iterations performed per joule of power using an AC power meter the FPGA was most efficient, followed by the PC then the graphics card.

With the introduction of CUDA Barrachina *et al.* [19] repeated earlier work by Jung and others and extended it to a comparison of algorithmic variants of the Cholesky and LU decompositions using a G80-based GPU. Their hybrid code was developed with the sole aim of outperforming traditional CPU-only implementations.

There are three variants each of the blocked Cholesky and LU decomposition algorithms. They all involve the same operations but executed them in a different order. Each algorithm also executes in-place overwriting the input matrix with its output. The three variants of Cholesky decomposition are shown in Table 2.2 with the rate-determining steps in each highlighted in bold. Each variant requires the use of the symmetric rank-K update and triangular matrix solve routines implemented as the SSYRK and STRSM routines in single-precision in the BLAS specification. Each variant also requires an unblocked Cholesky decomposition routine named SPOTF2 in LAPACk while variant three additionally requires the use of the BLAS SGEMM operation to perform general matrix-matrix multiplication. The three variants of the blocked LU decomposition algorithm are shown in Table 2.3 but the authors neglect to determine the rate determining step in each algorithm apart from noting that the STRSM routine in CUBLAS 1.0 is not as optimised as the SGEMM routine. Studies into the performance of the first release of the CUBLAS library found it to perform best when the memory being operated on is aligned on a 128-byte boundary so the block sizes used in the algorithms were chosen to be multiples of 32 elements. The variants were implemented as hybrid algorithms by performing the unblocked Cholesky decomposition and the LU column factorisation on the CPU. Recursion was used to divide the matrix into four blocks at each step with hybrid processing being used at the deepest level. Increasing or decreasing the level of recursion was found not to have an effect on the performance of the algorithm.

The matrix decompositions were used to calculate the solution to a linear system on the GPU. In order to obtain a double precision solution from a single precision decomposition an iterative refinement algorithm was used that had originally been developed for the Cell CPU found in the Playstation 3. The iterative refinement algorithm is executed on the CPU in single precision apart from a matrix-vector multiplication which is performed in double precision and manages to achieve equivalent accuracy to a full double precision solution.

A system with an Intel Core 2 Duo running at 1.86GHz and fitted with an nVidia 8800 Ultra graphics card was used to benchmark performance. The algorithms were implemented using

Variant 1	Variant 2	Variant 3
1. SPOTF2	1. **STRSM**	1. SSYRK
2. STRSM	2. **SSYRK**	2. SPOTF2
3. **SSYRK**	3. SPOTF2	3. **SGEMM**
		4. STRSM

Table 2.2: The three variants of the blocked Cholesky decomposition with the rate determining step of each algorithm in bold. Each variant requires the SSYRK and STRSM routines from the BLAS to perform symmetric rank-K update and triangular matrix solve operations. Each variant also requires an unblocked Cholesky decomposition routine named SPOTF2 in LAPACK. Variant three additionally requires a single precision matrix multiplication routine implemented as SGEMM in the BLAS.

Variant 1	Variant 2	Variant 3
1. STRSM	1. SGEMM	1. STRSM
2. SGEMM	2. SGEMM	2. SGEMM
3. SGEMM	3. SGEMM	
	4. STRSM	

Table 2.3: The three variants of the blocked LU decomposition with the rate determining step of each algorithm in bold. Each variant requires triangular matrix solve and general matrix multiplication operations implemented as the STRSM and SGEMM routines in the BLAS.

Fortran 77 with CUDA and CUBLAS versions 1.0. The CPU implementation used GotoBLAS with the reference LAPACK built on top. The blocked unpadded GPU implementations were found to outperform blocked CPU code for 3000-square matrices and up using the Cholesky decomposition and 1500-square matrices and up using the LU decomposition. The SGEMM routine in CUBLAS 1.0 is optimised better than the SSYRK or STRSM routines so variant 3 of the blocked Cholesky algorithm performs best. Performing an STRSM on a large matrix performs particularly poorly therefore variant 1 is slowest. Padding the matrices to 32 elements results in a small performance improvement for the SGEMM routine with the smallest increase in performance in variant 2 of the algorithm which relies on the SSYRK routine. With the LU decomposition variant 1 performs worst as it relies on the STRSM routine heavily.

The authors found that although GPUs have poor double precision performance, single precision can be used along with iterative refinement on the CPU and still yield an overall faster routine than using double precision throughout.

2.3.5 CULA

CULA [8] is a BLAS and LAPACK library utilising both the CPU and GPU in a computer. It provides two interfaces depending on whether the arguments for the numerical routine are in system memory or graphics memory to save on unneeded transfer of data. It also contains interfaces for Fortran and MATLAB and a "bridge" interface to allow code written to use the MKL, ACML or Netlib LAPACK to use CULA instead.

CULA was originally written to be an implementation of LAPACK for GPUs. However it was found that some LAPACK routines exhibit limited parallelism making their GPU implementation slower than their CPU counterpart. Transferring a matrix block into system memory and using the CPU to factorise it before transferring it back into graphics memory was found to be faster so CULA became a hybrid library targeting CPUs with a GPU connected as an accelerated math coprocessor. As the GPU and CPU operate asynchronously with respect to each other and GPUs can overlap memory transfers with computation, using the CPU for block factorisation comes at no cost if the GPU can perform a large enough task at the same time.

CUBLAS was found to be a poor choice of optimised BLAS upon which to build a GPU LAPACK implementation as it is inefficient for the problem sizes LAPACK uses most frequently. CULA includes a GEMV routine that is 25% faster than CUBLAS when the matrix is not transposed and 300% when it is, resulting in a 25% increase in LAPACK speed. GEMM is used most in LAPACK and its performance is critical to the performance of LAPACK. There are four cases that a GEMM routine can be optimised for depending on relative sizes of input matrix dimensions. CUBLAS GEMM is optimised for the case where all dimensions are

roughly equal however in LAPACK the most used cases are when one or two dimensions are much shorter than the others. CULA contains a GEMM that is 10%-30% faster than CUBLAS GEMM for the cases used in LAPACK resulting in a 10% faster LAPACK library.

The LAPACK interface was designed when memory was a scarce resource on computers and when there was only one area of system memory. Hybrid computing has two areas of memory for the CPU and GPU and also two processors. Removing workspace memory parameters from the LAPACK interface results in fewer function calls to work out the size of workspace needed for a routine. It also reduces the chance for programmer error and the number of routines that need to be implemented for different combinations of CPU and GPU memory. CULA implements a GPU memory pool to allocate memory within functions to reduce the increased overhead of allocating memory on the GPU when compared to cost incurred when allocating memory for use by the CPU. Pooling of system memory may already be performed by the operating system to decrease the cost of frequent allocations but is not implemented by the GPU driver.

Benchmarks of the initial release of CULA on an Intel Core i7 920 system with 6GB RAM and a Tesla C1060 GPU show that CULA is $2\times$ - $4\times$ faster in single precision when compared to the MKL. Double precision performance is poor on this generation of GPUs therefore CULA was only $1.5\times$ - $2\times$ faster although this is likely to change on newer generations of GPU.

2.3.6 MAGMA

MAGMA [13] is a project by the University of Tennessee to create a LAPACK-like library for hybrid multi-core CPU and GPU systems. MAGMA is motivated by the end of routine increases in CPU frequency and the consequent end of automatic performance increases for high performance computing algorithms based on CPUs and the subsequent shift to using GPU accelerators. As power consumption is related to the cube of clock frequency GPUs with many slower cores operating in parallel also have a power advantage over CPUs. Widespread adoption of GPU computing depends upon the availability of numerical libraries available for multicore and accelerator architectures easing the development time to port existing software to new environments. The aim is to have several algorithms for each LAPACK operation and select the most suitable algorithm at runtime based on the processing hardware available. MAGMA is the result of many research projects focussed on individual aspects of performing dense linear algebra on GPUs and hybrid CPU with GPU accelerator systems.

Baboulin *et al.* [18] performed preliminary research into the issues surrounding dense linear algebra on hybrid, hardware-accelerated architectures. When designing algorithms for dense linear algebra it is a common understanding to use fine grained parallelism for small

cores with limited memory and rely on asynchronous dynamic scheduling to hide memory latency. These techniques have been used successfully in algorithms for GPUs, FPGAs and the Cell processor and work well when the FLOP:word ratio of the algorithm is high. Baboulin *et al.* [18] apply these techniques to algorithms for the Cholesky, LU and QR decompositions. Blocked Cholesky decomposition algorithms already exist that split the algorithm into fine-grained subtasks that have a high ratio of FLOPs to memory bandwidth being based on level 3 BLAS. The LU and QR decompositions are currently based around level 2 BLAS and so have less arithmetic intensity. It is an ongoing task to redesign these algorithms to use level 3 BLAS exclusively.

Pivoting is used in algorithms such as the LU decomposition to ensure numerical stability by processing elements in a specific order. Gaussian elimination with partial pivoting is used synchronously in the reference implementation of LAPACK while for clusters there exists an algorithm that implements pivoting using the minimum number of inter-node communications. For multi-core architectures pairwise pivoting can be performed but is expensive in terms of computation. Running an LU decomposition on the GPU with Gaussian elimination and partial pivoting reveals that pivoting consumes 30% of the overall computation time which leads Baboulin *et al.* to research alternatives to pivoting. They found that pivoting is not needed to ensure stability when the elements of the matrix being computed are approximately distributed according to the standard normal distribution and, in cases where this is not true, that most random matrices become normally distributed after a few iterations of Gaussian elimination. They therefore try to find a method of transforming matrices into ones sufficiently random that pivoting is not needed. Two transforms have previously been used on CPUs called the "Discrete Fourier" and "Random Butterfly" transform (DFT and RBT respectively). Both require FFT and an RNG to be implemented for the GPU.

In contrast, the QR decomposition, while requiring almost twice the number of arithmetic operations than the LU decomposition, does not require pivoting to ensure stability. It also contains more level 3 BLAS operations in blocked form making it better suited to GPUs than the LU decomposition.

Baboulin *et al.* [18] performed experiments on numerical accuracy using Matlab with Higham's Matrix Computation Toolbox and sample matrices from the Matlab matrix gallery. The accuracy of solutions to systems of linear equations were compared when computed using the LU decomposition with Gaussian elimination and partial pivoting (as implemented in Netlib LAPACK), Gaussian elimination and Gaussian elimination followed by the RBT. Also included were results computed by QR decomposition. Accuracy was measured using component-wise

backward error before applying an iterative refinement procedure and counting the number of iterations required to converge to a solution of fixed accuracy. For the first three matrices used in their experiments LU decomposition with Gaussian elimination produces an accurate enough solution that a RBT is not needed. For the remainder of the matrices except the last the RBT gives a more accurate result than Gaussian elimination with partial pivoting. With the last matrix in the matrix gallery the RBT fails to produce a more accurate result than Gaussian elimination with partial pivoting as was found by a previous study that used the RBT on ill-conditioned matrices.

In addition to experiments on accuracy performance of the LU decomposition pivoting variants on the GPU was evaluated. The algorithms were implemented in the same manner as the reference LAPACK implementation replacing the BLAS calls with calls to CUBLAS. Using a RBT is negligible on the GPU therefore it has the same performance characteristics as using Gaussian elimination without pivoting. Performance of the LU decomposition without pivoting is 30% higher when the matrix is randomised beforehand. Using RBT without pivoting improves the performance of the LU decomposition by $2\times$ compared with using Gaussian elimination with partial pivoting.

Further experiments into dense linear algebra algorithms on GPUs, specifically the LU, QR and Cholesky decompositions, showed that high performance could be obtained with minimal effort although it may not be the best available. The Cholesky decomposition had the highest performance as it is all BLAS 3. Adding a hybrid step computing the diagonal unblocked factorisation on the CPU doubled the performance of the Cholesky decomposition for smaller matrices. The GPUs used in the study are not capable of overlapping memory transfers with computation. Using an unblocked BLAS 2 based factorisation on the GPU gave similar performance results as for the LU and QR decompositions which were slower than a hybrid algorithm. Replacing the CUBLAS 1.0 `SGEMM` and `SSYRK` calls with implementations from Volkov *et al.* [124] further improved the performance of their algorithms highlighting the importance of an optimised BLAS implementation for LAPACK routines.

Another of the first research projects leading to MAGMA, carried out almost simultaneously with the work of Baboulin *et al.* [18], was the implementation of hybrid LU, QR and Cholesky matrix factorisations by Volkov *et al.* [124]. The speed of their routines is attributed to a fast GPU SGEMM implementation that is 60% faster than the one distributed with CUBLAS 1.1. The code for their matrix multiply routines in single and double precision as well as the SSYRK and DSYRK routines derived from them was licensed by nVidia to be included in CUBLAS 2.0. Also included in their work is a fast synchronisation barrier for the entire GPU.

A suite of benchmarks was performed on all the GPUs used in their study to identify any bottlenecks in the CUDA architecture to avoid when writing high performance code. A form of strip-mining of loops is performed automatically on the GPU but is expensive in terms of register usage therefore for best performance short vector threads, or small thread blocks, should be used. The number of 32-bit registers on the GPU is much larger than the shared cache space in contrast to most other vector architectures. Register access is faster than shared memory access and arithmetic instructions that have both operands in registers have higher throughput than the equivalent instruction that reads an operand from shared memory.

In order to assess the overhead involved in launching a kernel on the GPU a minimal kernel was written and executed on the GPU a large number of times with a single synchronisation instruction at the end. Kernel launches were found to take $3 - 7\mu s$ on all systems. Adding a synchronisation instruction after each launch increased the time to $10 - 14\mu s$ per launch showing the added cost of synchronising. Repeating the experiment using the DirectX API gave inferior results showing that CUDA is the more efficient platform.

The PCI Express bus which connects the graphics card to the rest of the computer can theoretically transfer data at 4GB/s for version 1.1 of the specification used with 16 lanes. Repeatedly copying increasingly larger amounts of memory onto the GPU from main memory found that each transfer has a constant $11\mu s$ overhead and runs at 75% of the theoretical bandwidth. The main memory used is marked as "pinned" memory to prevent the operating system from paging it which also results in higher transfer speeds. When using multiple GPUs in the same system transfer speeds to the second GPU run at half speed. This is because marking memory as pinned only has an effect on memory transfers performed using the same CUDA context. Using multiple GPUs requires using separate contexts so pinned memory in one context will be unpinned in another and transfers will run at reduced speed.

nVidia does not publish detailed information about the memory hierarchy on its GPUs so a pointer chasing benchmark was used to find out the speed and size of various undocumented caches on the GPU. Pointer chasing involves running $k = A[k]$ in a long unrolled loop. The loop is highly dependent with the next address to load stored at the current address being fetched. The time taken to execute each fetch is therefore dominated by the latency of the memory used to store the array A. By varying the size of the array and stride of addresses being fetched various aspects of the memory hierarchy can be discovered. Pointer chasing on the GPU found a fully associated 16 entry TLB, 20-way set associative L1 cache and 24-way set associative L2 cache all of which are not mentioned in the CUDA documentation. Running the benchmark across different GPUs found that the number of L2 caches scales with the number of multi-

processors on the GPU. Uncached accesses in main memory were found to take between 470 and 720 cycles which corresponds to the documented latency of 400-600 cycles. Shared memory accesses have 36 cycle latency which is approximately the same as the arithmetic pipeline latency.

For the purpose of measuring pipeline latency a collection of kernels was written that execute an individual instruction repeatedly with dependent arguments. This means that one operation has to fully traverse the pipeline before the next can start. The instruction is repeated a large number of times in an unrolled loop and the kernel execution is timed. The instructions are chosen to utilise each one of the arithmetic processing units in the GPU multiprocessors. Loops of $a = a + b$, $a = a \times b$ and $a = a \times b + c$ were used to stress the single precision arithmetic units. Each instruction took 24 cycles to execute at best in single precision. Repeating the experiment with b in shared memory increased the latency to 28 cycles for the multiply-add and 26 cycles for the other operations. On GPUs that can perform double precision the operations were found to take 48 cycles when b is in a register and 52 when b is in shared memory. Loops of $a = log(a)$ and $a = \frac{1}{\sqrt{a}}$ were used to test the latency of the single precision special function unit which was found to be 28 cycles. The nVidia documentation recommends running at least 6 warps or 192 threads to completely hide pipeline latency which corresponds to a latency of 24 cycles.

Memory copies in GPU memory run at 86% of the theoretical bandwidth when the copy is aligned and contiguous. Misaligned or non-contiguous memory copies run at 1/10th theoretical bandwidth on older GPUs although on newer GPUs a higher proportion of the bandwidth is still attained for misaligned copies and when the stride is small.

The peak single precision instruction throughput is 98% of the theoretical throughput. This was measured by writing a kernel that contained a batch of six independent register to register FMAD instructions in an unrolled loop executed one million times. Six FMADs were used in order to hide the pipeline latency. 64 threads was found to be the minimum number needed to get 98% throughput and a similar result was obtained using 16 threads with double precision multiply-adds. Placing one of the FMAD arguments in shared memory saw the throughput drop to 66% as did introducing dependence between the instructions, stalling the pipeline, although this had less of an effect on newer GPUs where 75% throughput could still be achieved. Replacing the FMAD instruction (2 FLOPs) with an FADD instruction (1 FLOP) gave 74% of the peak theoretical throughput on older GPUs and 99% on newer ones.

The global synchronisation barrier developed by Volkov *et al.* [124] is based on the assumption that updating 32-bit words in global memory is an atomic operation on the GPU. Two

arrays of synchronisation variables are allocated in global memory. A master thread spins on the values of all the "arrival" variables and updates the corresponding "wakeup" variables when all the arrival variables have been updated by their respective threads. Each slave thread updates their arrival variable and spins on the value of their private wakeup variable. This method of global synchronisation costs $1.5\times$ - $5.4\times$ less than the alternative of a kernel launch which also involves synchronising with the CPU.

In order for an algorithm to be bound by the instruction throughput on a particular architecture it needs to have a higher FLOP:word ratio than the architecture. BLAS level 1 and 2 routines require more bandwidth than operations so will always be bound by the slower memory bandwidth on a GPU. This has implications for LAPACK's LU decomposition which relies on BLAS 2 routines. For BLAS level 3 routines such as matrix multiply which can be implemented as a blocked algorithm, the size of the blocks used reduces the bandwidth required making the performance bound by the higher throughput of the processor.

Volkov *et al.* [124] implement GPU kernels to perform matrix multiplications of the form $C = \alpha AB + \beta C$, $C = \alpha AB^T + \beta C$ and $C = \alpha AA^T + \beta C$ which correspond to the SGEMM and SSYRK routines from level 3 of the BLAS. For simplicity their implementation only works for matrices that are a multiple of the block size used. Each matrix is stored in global GPU memory in column major layout and 64 threads per block is used as this was found to give the best performance through benchmarks. B is read into shared memory in blocks of 16×16 and transposed into row major layout with padding as this gives better data locality and reduces address arithmetic in the inner loop of the algorithm. This results in twice the performance than when storing B in column major layout in shared memory. A 64×16 block of C is kept in registers until all updates are complete. The choice of block size for C along with the number of threads used means that A can be read into registers in blocks of 64×1 as needed from global memory. It is possible to limit register usage using the CUDA compiler in order to fit more thread blocks on each multiprocessor however it is not needed in this case. Two dimensional thread blocks are used with a sophisticated indexing calculation used to map threads to non-zero elements of the matrix for the SSYRK routine. This is almost 70% faster than the alternative of scheduling nearly twice the number of threads needed and killing half of them as soon as they start. Double precision versions of the algorithms were created by replacing all single-precision variables with double-precision ones and padding the shared memory allocations to 32-bit boundaries to ensure efficient access when storing 64-bit variables. The authors remark that their resulting algorithm closely resembles an earlier algorithm for matrix multiplication on a Cray vector machine.

By counting the number of load and arithmetic instructions in the inner loop of the algorithm and multiplying by the number of cycles taken to execute each the performance of their algorithm can be expected to achieve 58% of the theoretical peak throughput. In reality their single precision matrix multiply achieves 60% of the peak throughput in comparison to the SGEMM routine distributed with CUBLAS 1.1 which achieves between 37% and 44%. Comparing their SSYRK and DSYRK routines to those distributed with CUBLAS 2.0 60% throughput was obtained compared with 36-44% for CUBLAS 2.0 SSYRK and 90% throughput versus 35% for CUBLAS 2.0 DSYRK. The higher proportion of peak performance obtained for double precision is due to less of it being available. If it were possible to store blocks of B in registers and reduce the cycle count for fetching B from shared memory, 90% of peak throughput would be also expected for their SGEMM routine. This is similar to that obtained by the SGEMM implementation in the Intel MKL when run on a Core 2 Duo.

Further analysing the CUBLAS 1.1 SGEMM kernel reveals that it uses a block size of 32×32 resulting in a bandwidth reduction of $32\times$ compared to $25.6\times$ of Volkov *et al.* [124]. It stores blocks of both A and B in shared memory simultaneously therefore requiring more shared memory but less than half the registers than in [124]. It also has less than half the inner loop arithmetic instructions than in [124] attaining only 36-44% of the peak theoretical throughput. The CUDA guidelines emphasise the importance of optimising for GPU occupancy to keep as much of the GPU busy as possible. Volkov *et al.*'s SGEMM implementation has less than half the occupancy of CUBLAS 1.1, however SGEMM has twice the performance and similar work on SGEMM optimising for occupancy attains only around 25% of the theoretical peak instruction throughput.

To demonstrate the utility of their fast SGEMM and SSYRK routines, Volkov *et al.* [124] use them to implement LU, QR and Cholesky factorisations using variants of the algorithms that expose maximum parallelism to the SGEMM routine. Again, their routines are limited to matrices that are multiples of the block size. A hybrid algorithm is used to perform part of the factorisation on the CPU, overlapping CPU and GPU computation using a lookahead technique. For the LU decomposition the matrices are stored on the GPU in row-major layout so as not to suffer poor performance when pivoting. The STRSM which has poor performance is replaced with an inverse operation on the CPU allowing a much faster SGEMM to be used on the GPU. For the hybrid QR factorisation step performed on the CPU the upper triangle of the result is not needed so is not transferred back onto the GPU. The block size for the QR factorisation is also tuned at each step. In contrast to similar work on a hybrid Cholesky factorisation by Baboulin *et al.* [18] the STRSM is performed on the GPU leaving the CPU only concerned with the block

diagonal factorisation. Two levels of recursive blocking are used in all algorithms in the same way as Barrachina *et al.*, however to increase parallelism coarse blocking is favoured more than fine blocking. A multi GPU version of the LU factorisation is also created that splits the columns of the matrix across two GPUs.

Benchmarks were run on a range of computer systems with different processors, graphics cards and software versions. A computer with a 2.67GHz Intel Core 2 Duo E6700 was used to run the hybrid factorisations. It was fitted with either a nVidia GeForce 8800GTX or GTX 280 graphics card. When using the 8800GTX 32-bit Windows XP was used with CUDA 1.1. 64-bit XP and CUDA 2.0 were used with the GTX 280. To contrast their algorithms with CPU-only versions another computer with an Intel Core 2 Quad CPU running at 3GHz was used. Intel MKL version 10.0 was used as the CPU BLAS/LAPACK library throughout although it was found to run slower in 32-bit than in 64-bit. Single precision floating point was also used across all machines and benchmarks along with pinned memory for the matrices in system memory which were also padded to an odd multiple of 64 elements. The input matrices were initialised with random uniform variates distributed over $[-1, 1]$ while a positive definite matrix was created for the Cholesky factorisation by multiplying the uniform matrix by its transpose and adding a scaled identity matrix. Accuracy was measured by taking the normalised maximum difference and was found to be comparable to the MKL when the matrix dimension is 8192.

The hybrid LU and QR factorisations run faster than a CPU only algorithm when the matrix dimension is greater than 1000. The Cholesky decomposition runs faster when the matrix size is greater than 600. The increases in speed are similar to that obtained with the standalone SGEMM implementation which shows the strong dependence of these algorithms on a fast matrix multiply routine. The overall performance is somewhat less than the theoretical maximum of a hybrid algorithm. The maximum performance obtained was with the hybrid LU factorisation running on two GTX 280 GPUs at 538 GFLOPs/s although even a single GTX 280 runs faster than two 8800 GTXs. Profiling the LU decomposition reveals that 10% of the computational time is spent on the CPU and 90% on the GPU.

A little over a year later, Volkov *et al.*'s work on a fast SGEMM algorithm for GT200-based GPUs [124] was improved upon by Lung-Sheng Chien [30]. Chien ran benchmarks to find the throughput of the two variants of single-precision floating-point multiply-add (`FMAD`) instruction of the form $d = a \times b + c$ used by CUDA. The variant used by Volkov *et al.* reads the argument b from shared memory and has a lower throughput than the variant which reads b from a register. This point is remarked upon in Volkov *et al.*'s paper when they state their

performance goal is 66% of the theoretical peak throughput due to using the `FMAD` instruction in shared memory. In order to use the higher throughput instruction Lung-Sheng Chien's SGEMM routine loads the current row of b from shared memory into registers before multiplying in the inner loop. This replaces the single `FMAD d, a, smem_b, c` instruction with a sequence of two instructions: `MOV b, smem_b; FMAD d, a, b, c`. This has lower throughput by itself as the latency of the two `MOV` and `FMAD` instructions is greater than the single `FMAD` instruction. However by allocating more registers the `MOV` instructions can be overlapped by the `FMAD` instructions improving overall throughput. This optimisation was hard to accomplish with the version of `nvcc` used in his research as the optimiser removes unnecessary register allocations and automatically replaces the fast register `FMAD` instruction with the slower shared-memory equivalent. To achieve this optimisation, Lung-Sheng Chien used an intermediate `MUL` instruction to multiply b from shared memory, storing the result in a register which is not optimised away by `nvcc`. A third-party CUDA binary assembler was then used to replace the `MUL` instructions with `MOV` instructions in the resulting binary code.

As well as improving the instruction throughput of Volkov *et al.*'s SGEMM algorithm, Lung-Sheng Chien also generalised it to handle cases where A, B and C are not multiples of the block size used without sacrificing performance. This improvement also requires extra registers to use as loop counters so thread block sizes were reduced to improve GPU occupancy by running less threads per block allowing more blocks to simultaneously fit on each GPU multiprocessor. Lung-Sheng Chien's optimisations result in an average 10% improvement in performance for all problem sizes on all the GT200-class GPUs used in his study.

When the new Fermi architecture of GPUs was developed by nVidia, Volkov *et al.*'s SGEMM and DGEMM implementations required further work to fully realise the performance improvements available from new architecture. Work had already been done to parameterise Volkov *et al.*'s SGEMM and automatically find the best values for the different layouts of global memory and number of multiprocessors available on different models of GPU of the same architecture. While their implementation will run faster on Fermi than on previous architectures it is unaware of any new architectural features introduced.

On the GPU architecture targeted by Volkov *et al.* shared memory access is almost as fast as register access but on the newer Fermi architecture registers are much faster. Nath *et al.* rewrote the tuning algorithms to take account of the additional level of memory hierarchy and this also required that the GEMM algorithms were rewritten to provide an extra level of blocking. This improved the performance of Volkov *et al.*'s SGEMM routine from 40% of the theoretical throughput on Fermi to 58%.

Nath *et al.* measured the performance of their new Fermi DGEMM routine when incorporated into the MAGMA library. The performance of the LU, QR and Cholesky decompositions was measured against CULA, the MKL and the reference LAPACK implementation. The new MAGMA implementation was found to be 63% faster than CULA, its nearest rival, reaching 240 GFLOPs/s on a C2050 GPU for a 10th of the cost of a 48-core CPU-based system also benchmarked.

Ltaief *et al.* [76] have published an analysis of a Cholesky factorisation implementation using the MAGMA library with the PLASMA scheduler. The tiled algorithm used is the same as the blocked algorithm used in the reference LAPACK implementation, relying on iterative applications of the level 3 BLAS routines `SYRK`, `GEMM` and `TRSM` with a blocked factorisation `POTRF` performed on diagonal tiles. This is known as the left-looking algorithm and their study focusses on the application of this algorithm to lower triangular matrices.

The PLASMA scheduler [13] is used to split the workload of the tiled algorithms into individual tasks representing the dependences between them as a graph. PLASMA computes the order in which to execute the tasks subject to the dependencies and schedules the tasks for execution using static pipeline scheduling which is simple and provides good data locality for dense linear algebra operations. Currently PLASMA keeps track of execution progress using a global progress table which could limit scalability.

The MAGMA library executes the tasks as hybrid linear algebra operations. It is concerned with selecting the best size and shape of the hybrid split and matching the tiles from the algorithm to the most effective processor. Small non-parallel tasks are overlapped with larger more parallel ones with non-parallel tasks that are on the critical path of the algorithm given higher priority.

The PLASMA scheduler was extended to distribute tiled hybrid tasks to execution units comprising of a CPU paired with a GPU. A data persistence strategy was implemented to minimise data transfer between GPUs and system memory. Overall, four optimisation passes were performed, the first keeping a tile in GPU memory until completely updated. The second pass minimised data transfers, the third replaced the factorisation of the diagonal tile with a hybrid algorithm while the fourth optimisation pass operated on the `SYRK` and `TRSM` operations. The `SYRK` operation was optimised by reordering the indices in the thread blocks to perform redundant computation and memory fetches to reduce branching overhead while the `TRSM` operation was optimised by replacing it with a kernel that computes the inverse of 32×32 matrix blocks then performs a simpler matrix-matrix multiplication.

On a GPU cluster comprising a nVidia Tesla S1070 containing four C1070 GPUs, con-

nected to a computer with two dual-core AMD Opterons running at 1.8 GHz Ltaief *et al.*'s Cholesky factorisation achieves 1.16 TFlops/s in single precision and 275 GFlops/s in double precision which is 73% and 84% of the theoretical peak throughput for single and double precision respectively.

2.4 Summary

In this chapter we have reviewed the variety of approaches previously considered for parallelising computer codes on CPU and GPU architectures. The use of GPUs for computational processing in parallel with a multicore CPU is increasing as hybrid architectures offer higher performance per watt and are cheaper to set up and run. Existing code needs to be ported to these architectures in order to exploit the efficiencies available and the use of standard libraries for numerical processing, in particular the BLAS and LAPACK specifications, helps in this regard. In addition, having an open source implementation of these specifications allows users to modify them to their specific needs and contribute their code back to the scientific community, speeding development. Under these criteria the MAGMA library can be viewed as the current state of the art for parallel BLAS and LAPACK on hybrid CPU and GPU architectures. MAGMA uses the reference algorithms for the LAPACK routines and implements them using the optimised PLASMA scheduler adapted from clusters to distribute workloads to CPU and GPU pairs. We believe that by considering the additional levels of parallelism available on hybrid architectures that further performance gains are achievable. This is mainly because the standard LAPACK algorithms used by MAGMA assume a homogeneous shared memory parallel architecture while hybrid multicore CPU and GPU systems are heterogeneous distributed memory parallel architectures.

We looked at Markov chain Monte Carlo as a motivating example of a widely employed algorithm that is not easily parallelisable due to its inherent sequential structure, and concluded that further parallelisation requires redesign of the underlying linear algebra routines that it makes frequent and heavy use of. In Markov chain Monte Carlo simulations that require it the Cholesky decomposition consumes most of the computational time. It is for this reason that we choose to focus our research on the gains in computational efficiency that are available when implementing the Cholesky decomposition on hybrid multicore CPU and GPU architectures while keeping our optimisations as general as possible so they may be applied to other blocked linear algebra routines on similar hybrid parallel architectures.

In the next chapter we shall introduce the technical details required to understand our novel algorithms for blocked linear algebra routines designed in particular for hybrid architectures

mixing CPU and GPU computing units.

Chapter 3

General Methodology

In this chapter we give a brief overview of the programming techniques and algorithmic background required to understand the contributions to blocked linear algebra routines presented in the later chapters. From this point onwards we assume our target environment is a Linux workstation with a multicore CPU and multiple GPUs. Our primary test system contains an Intel Core i7-965 Extreme Edition quad core CPU running at 3.2 GHz and two nVidia GTX 285 graphics cards. Specifications of the nVidia GTX 285 CPU are displayed in Table 3.1. Hyperthreading on the CPU presents 8 logical cores to the operating system, and the nVidia GTX 285 GPU has compute capability 1.3, which supports double precision and overlapping memory transfers with GPU computation. It does not support multiple GPU kernels running simultaneously, unlike graphics cards built around newer GPUs of Compute Capability 2.0 and higher. The features specified by Compute Capability 1.3 are shown in Table 3.2. The operating system installed is Gentoo Linux with kernel version 3.8.6 compiled with GCC version 4.7.2.

Our algorithms are implemented in C using the ISO C99 standard. This standard is well supported in most mainstream C compilers unlike the newer ISO C11 standard and provides many useful features for numeric programming such as built in support for complex types and arithmetic and type-generic math functions. In addition memory pointers can be marked with the `restrict` keyword to aid compiler optimisation and fixed-width integer types are provided to aid portability. C++-style comments, variable declarations and loop structures are also included over the previous ANSI/ICO C90 standard. Both GCC and ICC support the majority of features specified by C99 and can automatically vectorise code using SSE instructions. Both also support multithreading code using OpenMP compiler directives.

Our GPU code is compiled using version 5.0.35 of the nVidia CUDA toolkit. We create fat binary objects with PTX assembly and cubin binary code for multiple classes of compute capability 1.x GPU and embedded them in the resulting executable using the `bin2c` utility from the nVidia CUDA Toolkit. This is the same mechanism used by the nVidia CUBLAS

Processor	
Compute Capability	1.3
Multiprocessors	30
Clock Rate	1476 MHz
Memory	
Size	1 GB
Clock Rate	1242 MHz
Type	GDDR3
Interface	512 bit

Table 3.1: Specifications of the nVidia GeForce GTX 285 GPU used in this study

Threads per warp	32
Maximum threads per block	512
Maximum thread block dimensions	$512 \times 512 \times 64$
Maximum thread blocks per SM	8
Maximum threads per SM	1024
Shared memory per SM	16kB
Shared memory banks per SM	16
Number of 32 bit registers per SM	16384

Table 3.2: Summary of features of CUDA Compute Capability 1.3 GPUs

library. We concentrate on targeting nVidia GPUs of compute capability less than 2.0 although our code will run equally well on these (and later) devices.

3.1 Representing Matrices and Vectors in memory

Matrices and vectors are stored in memory as one-dimensional C arrays allocated on the heap. Vectors have an associated number of elements and a stride parameter which indicates the spacing between consecutive vector elements stored in the array. This allows a vector to be used as an alias for a row or column of a matrix. Matrices have an associated number of rows and columns and a leading dimension parameter indicating the number of elements between the start of one column and the next. The columns in a matrix are stored contiguously in memory. This is known as column-major layout and is used for compatibility with numerical libraries written in Fortran. The leading dimension of a matrix A, which we denote by lda, is used to calculate the linear array index from a two-dimensional i, j index as in Equation 3.3 and can also be used to enforce memory alignment requirements across columns. This is shown in Equations 3.1 and 3.2 where the 3×3 matrix A in Equation 3.1 is represented by the linear array in Equation 3.2 with zero-padding to ensure that each new column starts on a 4-element boundary. Two dimensional C arrays can also be used to store matrices but they use row-major layout.

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \tag{3.1}$$

$$A = [\,1,\, 2,\, 3,\, 0,\, 4,\, 5,\, 6,\, 0,\, 7,\, 8,\, 9,\, 0\,] \tag{3.2}$$

$$A_{(i,j)} = A[j * lda + i] \tag{3.3}$$

3.1.1 Host Memory

Loading data from memory addresses that are multiples of 16 bytes improves the performance of SSE instructions on Intel CPUs [58]. When allocating heap memory for use as a vector or matrix the address returned by `malloc` or `calloc` must be rounded up to the nearest multiple of 16 in order to get the best performance from functions implemented using SSE. The GNU C Library already aligns addresses returned by the `malloc` and `calloc` library calls to multiples of 16 bytes on 64-bit systems [11]. There are several alternative methods to align memory when a different standard C library is used. The Intel Math Kernel Library [4] provides an aligned memory allocation function as do C libraries conforming to the POSIX standard. The C11 standard also specifies the `aligned_alloc` aligned memory allocation function. If none of these options is available then the pointer can be manually aligned using pointer arithmetic to

round the address returned from `malloc` up to the next multiple of 16. This requires storing both the misaligned pointer to pass to `free` and the manually aligned pointer to pass to linear algebra library functions. The alternate methods can also be used to align memory pointers to arbitrary amounts.

Each column in a matrix also needs to be aligned on a 16-byte boundary. This is achieved by setting the leading dimension greater than or equal to the number of elements in each column, m. Since the leading dimension is expressed as a number of elements and not bytes the memory alignment is calculated in C as $16/\textbf{sizeof}(\mathbf{T})$ where **T** is the type of element being stored in the matrix. The alignment will always be a power of two and therefore the leading dimension, which we denote by ld, can be calculated with the bitwise operation $(m + align)\& \sim align$, where $align$ is the alignment expressed as a number of elements and $\sim$ in this context represents bitwise negation. The total memory consumed by a correctly aligned $m \times n$ matrix used to store elements of type **T** is $ld \times n \times \textbf{sizeof}(\mathbf{T})$.

Submatrices are created from a matrix stored as a linear C array by taking the address of the top left element of the submatrix using the address-of operator $\&$. This is known as pointer aliasing and allows elements in the matrix to be updated by writing to elements in the submatrix, provided that the matrix is not declared `const`. The leading dimension of the submatrix is the same as the larger matrix. Similarly subvectors are created by taking the address of the first element of the subvector and keeping the same stride. The subvector or submatrix will not necessarily be aligned correctly unless the offsets are multiples of the alignment expressed as a number of elements. In Equation 3.4 B is created as a submatrix of A from Equation 3.2 starting at $A_{(i,j)}$.

$$B = \&A[j * lda + i] \tag{3.4}$$

3.1.2 GPU Memory

Vectors and matrices are represented in GPU memory in the same way as in host memory using a stride for vectors and a leading dimension for matrices. Column-major matrix storage is used as the CUBLAS library expects matrices to be stored this way. The CUDA library provides memory allocation functions that automatically align memory addresses appropriately to store any type of variable [92]. For matrices a 2D memory allocation function is provided that computes the correct leading dimension to ensure each column is also aligned correctly. The leading dimension, or "pitch" in CUDA terminology, is returned from `cuMemAllocPitch` as a number of bytes between the start of one column and the next so must be divided by the element size to give a number of elements. The pitch will always divide evenly by the element size.

Subvectors and submatrices are constructed on the GPU in the same manner as on the host. From the host perspective, however, memory pointers on the GPU are simply integers so any offset must be explicitly multiplied by the element size. This is performed implicitly by the compiler when using a host memory pointer.

3.1.3 Copying Matrices and Vectors

The CUDA library provides functions to transfer areas of memory from the host to the device and back. The copy is done by the GPU using a DMA transfer over the PCI-Express bus and may be synchronous or asynchronous with respect to the CPU. The transfer may also be overlapped with GPU computation on GPUs with compute capability 1.1 and above. Additionally, the CUDA library provides functions to allocate or mark sections of host memory as page-locked or pinned which prevents the operating system from transferring it to disk when physical memory is exhausted. This increases transfer speed but reduces the amount of memory available to other applications.

In addition to providing a two-dimensional pitched memory allocation function the CUDA library includes two-dimensional copy functions. These can be used to transfer vectors with non-unit stride or matrices where the leading dimension is larger than the size of the columns. The GPU is able to iterate through individual linear copies asynchronously freeing the CPU to do other computation.

Each memory transfer incurs a fixed overhead in order to communicate the source and destination addresses to the GPU in addition to the time taken to transfer the data over the PCI Express bus. It will therefore be faster to transfer a vector with unit stride or a matrix that requires no padding as these can be copied in a single linear transfer.

The theoretical global memory bandwidth of a GPU is calculated by Equation 3.5 where DDR is set to 2 on GPUs with double data rate memory and 1 on those without.

$$\text{memory clock rate} \times \text{DDR} \times \text{memory interface width} \tag{3.5}$$

The theoretical internal bandwidth of the nVidia GeForce GTX 285 as calculated using Equation 3.5 and the specifications in Table 3.1 is 148.058 GB/s. The theoretical bandwidth of the PCI Express 2.0 x16 bus to which the GPU is connected is 8 GB/s.

In order to measure the actual attainable bandwidth when transferring blocks of memory across the PCI Express bus using CUDA, a benchmark was run similar to that outlined in Volkov *et al.* [124]. This benchmark times how long it takes to transfer a contiguous block of memory from the host onto the device and back. Each transfer is repeated a large number of times to obtain a mean bandwidth. This is then repeated for blocks ranging from 1MB in size up to

GPU with display attached	Bandwidth	Overhead
Host to Device	5594.92 MB/s	0.110ms
Device to Host	5441.69 MB/s	0.203ms
GPU without display attached	**Bandwidth**	**Overhead**
Host to Device	5722.98 MB/s	0.014ms
Device to Host	5465.04 MB/s	0.005ms

Table 3.3: Results from the PCI Express benchmark showing the attainable bandwidth and overhead in setting up a copy.

128MB in increments of 1MB. A linear least squares regression is performed to calculate the mean bandwidth and overhead for each transfer. Results from a sample run of the benchmark on the two GPUs in our test system are given in Table 3.3. There is a larger overhead when a GPU is being used to run a display as the graphics driver has to wait until the GPU has finished drawing the screen before a GPGPU operation can be performed.

This allows the time taken to upload and download vectors and matrices from the GPU to be calculated using Equations 3.6 and 3.7 respectively. Copying a vector with unit stride will be faster than copying a vector with non-unit stride and likewise copying a matrix with extra padding to enforce column alignment will be slower than copying one without.

$$t(n, stride) = \begin{cases} \frac{n \times \textbf{sizeof(T)}}{bandwidth} + overhead & \text{if } stride = 1 \\ n \times (\frac{\textbf{sizeof(T)}}{bandwidth} + overhead) & \text{if } stride \neq 1 \end{cases} \tag{3.6}$$

$$t(m, n, ld) = \begin{cases} \frac{m \times n \times \textbf{sizeof(T)}}{bandwidth} + overhead & \text{if } m = ld \\ n \times (\frac{m \times \textbf{sizeof(T)}}{bandwidth} + overhead) & \text{if } m \leq ld \end{cases} \tag{3.7}$$

3.2 Theoretical Instruction Throughput

The number of SM cycles needed to execute each instruction in the CUDA instruction set is detailed in section 5.4.1 of the CUDA Programming Guide [94] and summarised for floating point instructions on compute capability 1.x GPUs in Table 3.4.

The maximum theoretical instruction throughput of a GPU is calculated by Equation 3.8. It is possible for the 2 single precision special function units on a compute capability 1.x GPU to execute floating point multiplication in parallel with other operations on the 8 single precision arithmetic units. This is known as "dual-issue". When combining multiply-add operations with multiply operations this gives a theoretical throughput that is $1.5\times$ higher than executing

Instruction	Throughput
Single precision add, multiply and multiply-add.	8 instructions per clock cycle per multiprocessor
Double precision add, multiply and multiply-add.	1 instruction per clock cycle per multiprocessor
Single precision reciprocal, reciprocal square root, base-2 logarithm, base-2 exponential, sine and cosine.	2 instructions per clock cycle per multiprocessor

Table 3.4: Throughput for floating point instructions on CUDA Compute Capability 1.x GPUs

Processor	
Core count	4
Clock rate	3.2 GHz
SIMD width	128 bits

Table 3.5: Specifications of the Intel Core i7-965 Extreme Edition CPU used in this study

multiply-add operations alone.

$$\text{clock rate} \times \text{multiprocessor count} \times \text{cycles per instruction} \times \text{FLOPs per instruction} \quad (3.8)$$

A similar calculation can be used to find out the theoretical instruction throughput of a CPU. This is shown in Equation 3.9. The specifications of the CPU used in this study are given in Table 3.5. These give a theoretical instruction throughput of 102.4 GFLOPs for single-precision multiply-add operations and 51.2 GFLOPs in double precision which matches the processor documentation [10].

$$\text{clock rate} \times \text{core count} \times \text{SIMD lane width} \times \text{FLOPs per instruction} \quad (3.9)$$

The throughput benchmark from Volkov *et al.* [124] was run to measure the actual throughput attainable. This benchmark contains 4096 dependent single precision floating point multiply add operations in an unrolled loop executed 16 times. A single word is read from global memory at the start of the kernel and written back to global memory at the end to stop the compiler optimising away instructions. In total the kernel contains 131072 single precision FLOPs. In addition to this two additional benchmarks were run. One is a direct copy of the original kernel

	Theoretical	Actual	%
Single precision multiply-add	708.48 GFLOPs/s	700.37 GFLOPs/s	98.85
Single precision "dual-issue"	1062.72 GFLOPs/s	1053.47 GFLOPs/s	99.13
Double precision multiply-add	88.56 GFLOPs/s	86.15 GFLOPs/s	97.28

Table 3.6: Instruction throughput of the nVidia GeForce GTX 285 GPU

using double precision operations. The other interleaves independent single precision floating point multiplication instructions at a ratio of 1:1, or 2:1 FLOPs, in order to measure performance when using dual-issue. The results are shown in Table 3.6.

3.3 Design of Linear Algebra functions

Our linear algebra functions follow the BLAS and LAPACK interface for function names. The side, uplo, transpose and diag flags explained in Section 2.3.1 are implemented as C enumerations to enforce type safety and reduce the possibility of coding errors. The actual value of each flag is derived from the first character of the FORTRAN string representation unlike the CBLAS specification which defines the values of the enumerations as integer values starting at 100. Using character values allows the enumerations to be used directly when calling optimised Fortran BLAS and LAPACK libraries from C and aids debugging. Arguments for functions in CPU code follow the BLAS and LAPACK interface. Arguments for GPU functions are arranged in decreasing order of size. As GPU function arguments are stored in shared memory which has strict alignment requirements this minimises the amount of wasted memory used as padding. OpenMP is used to implement multithreaded parallelism across any independent loops in the CPU BLAS implementation. Additional GPU BLAS operations are derived from the SGEMM implementation using a similar technique to that described in Kagstrom *et al.* [66]. This is also used to implement BLAS operations across multiple GPUs by rewriting each operation as a sequence of independent matrix multiplications carried out on both GPUs simultaneously while the CPU executes the remaining operations with a call to the BLAS operation required.

3.3.1 Automatic Vectorisation of C code for the CPU

Our C code is compiled using both GCC and ICC. Both compilers are able to automatically recognise loops that are candidates for vectorisation and will output optimised assembly code using SSE instructions where applicable. Due to the differences in the way automatic vectorisation has been implemented in GCC and ICC both require some loops to be structured differently before they will vectorise them. Automatic vectorisation is only applicable to innermost loops

so the compilers will only consider these loops as candidates for potential vectorisation.

Listing 3.1 shows an implementation of a dot product operation that can be vectorised. ICC will vectorise this loop automatically while GCC requires an additional compiler flag to enable unsafe math optimisations which ICC will apply by default. Compiler optimisations must not alter the correctness of a program and floating point operations in limited precision are not associative. This means that changing the order of operations in a reduction will possibly change the result leading GCC to not apply them by default. The reduction operation may be any of the elementary binary mathematical operations add, subtract, multiply or divide and may also contain another operation to combine elements before reducing, as in the dot product.

Listing 3.1: GCC requires an extra unsafe math optimisation flag before it will vectorise reductions, unlike ICC which vectorises them by default

```
for (size_t i = 0; i < n; i++)
  res += x[i] * y[i];
```

Loading elements from main memory on a CPU using SSE instructions is fastest when loading consecutive elements from contiguous areas of memory. GCC will refuse to recognise loops where consecutive elements are non-contiguous as candidates for vectorisation. Scatter and gather instructions were added to SSE 4.1 and ICC uses these to vectorise loops that have non-contiguous vector elements across iterations where it seems profitable as these instructions are slower to execute. It is possible that current versions of GCC have not yet been updated to use these new instructions. A workaround for this is to produce two loops as shown in Listing 3.2. Here one loop handles the case with contiguous elements that both GCC and ICC will vectorise and the other handles non-contiguous elements that GCC will not vectorise but ICC will albeit with slower gather and scatter instructions.

The loop dependency checking in both GCC and ICC will refuse to vectorise loops that update an element in a vector with elements from elsewhere in the vector, as shown in Listing 3.3. Both compilers incorrectly report a dependency between vector elements across iterations that would change the semantics of the loop if it were to be vectorised. A simple workaround for this is to explicitly alias the vector before entering the loop as shown in Listing 3.4. GCC and ICC will vectorise this loop.

These findings guide our design of the inner loops within our linear algebra functions in order to get the most efficient binary code output by both compilers.

Listing 3.2: ICC will vectorise the non-contiguous second loop as well as the contiguous first loop, whereas GCC will only vectorise the first.

```
if (incx == 1 && incy == 1) {
  for (size_t i = 0; i < n; i++)
    x[i] += alpha * y[i];
}
else {
  for (size_t i = 0; i < n; i++)
    x[i * incx] += alpha * y[i * incy];
}
```

Listing 3.3: GCC and ICC will incorrectly detect a data dependency across iterations of the inner loop and refuse to vectorise it.

```
for (size_t i = 0; i < n; i++) {
  for (size_t j = 0; j < i; j++)
    x[i] += y[j] * x[j];
}
```

3.3.2 Use of C++ templates for GPU kernels

Loop unrolling is an optimisation technique that removes the overhead of managing loop counters when the amount of instructions inside the loop is not large enough to hide the overhead itself. One method to unroll loops in C code is to use a preprocessor `#pragma unroll` directive. The amount of unrolling can be controlled by supplying an integer argument however the compiler is free to ignore this directive altogether. `nvcc` will automatically unroll small loops where the iteration count is known at compile time as explained in Appendix B.2.0 of the CUDA Toolkit Reference Manual [92]. This also has the effect of freeing registers that would be used for loop counters allowing them to be used for data instead. CUDA-C implements some features from C++ including template functions which provides an alternate means of loop unrolling. This is demonstrated in the reduction examples included in the nVidia SDK [53]. One difference using this method is that the amount of unrolling has to be left unspecified in any `#pragma unroll` directive as the C preprocessor is called earlier in the compilation sequence than the C++ template preprocessor when the value of the template argument is not yet known. This has the advantage that several instantiations of the same kernel with different block sizes and unrolling can be compiled from a common function template.

In addition to using templates to specify unrolling template arguments are also used to

Listing 3.4: GCC and ICC will correctly vectorise the inner loop after one array has been aliased to circumvent the dependency checker.

```
for (size_t i = 0; i < n; i++) {
  float * z = x;
  for (size_t j = 0; j < i; j++)
    x[i] += y[j] * z[j];
}
```

specify any parameter flags that are constant throughout the execution of a function, such as the `CBlasTranspose`, `CBlasSide`, `CBlasUplo` and `CBlasDiag` arguments. This allows them to be evaluated at compile time, cutting down on execution time and register usage, and produces several instantiations of the kernel function that can be chosen at runtime on the host by evaluating the BLAS flags there.

3.3.3 Generating Extra Precisions

While C++ templates can also be used to automatically generate the same function for several types, the CUDA architecture for compute capability 1 devices has enough differences when handling variables of different sizes to make this not worthwhile. Shared memory in CUDA is implemented in hardware as an array of 16 interleaved banks. For GPUs of compute capability less than 2.0 consecutive 32-bit words are stored in consecutive banks. A shared memory bank can only handle one request at a time so this allows maximum bandwidth when threads in a warp are accessing consecutive 32-bit elements. Single-precision `float` types can therefore be stored in shared memory in the same manner as they are accessed in global memory or registers. Larger accesses generated by accessing larger variables such as `double` or complex types generate bank conflicts where the hardware serialises warp requests. The nVidia Programming Guide suggests that it may be faster to generate multiple 32-bit requests in software than to rely on hardware serialisation so provides functions that can split 64-bit `doubles` into high and low bit patterns that may then be stored in shared memory as 32-bit integers [92]. A similar effect can be achieved for complex types by storing them in shared memory as separate real and imaginary parts. For complex double precision both techniques must be combined to produce 32-bit conflict-free accesses.

For every GPU kernel that is implemented in single precision we generate two kernels in `double` precision. The first is a direct translation of the single precision kernel where a search and replace is performed replacing each occurrence of `float` with the corresponding type. This kernel will contain bank conflicts when using shared memory and will cause the compiler

to serialise accesses. The other kernel is similar but modified to access shared memory as 32-bit words to avoid bank conflicts. `nvcc` defines a `__CUDA_ARCH__` preprocessor macro which contains the compute capability of the target GPU the code is being compiled for. This is used to select which set of kernels to compile and may also be disabled to test whether the software splitting is faster than the hardware warp serialisation.

3.3.4 Exploiting the differences between SIMT and SIMD

Several earlier papers on GPGPU work mistakenly described CUDA as a SIMD architecture [124, 112] when it is actually a SIMT architecture. While this still allowed the authors to write GPGPU code that executes extremely fast on GPUs it does not exploit the additional flexibility that SIMT has to offer. Each SM in CUDA multitasks several warps at once, each of which executes in a SIMD fashion and this is what nVidia refers to as SIMT. Earlier versions of the nVidia CUDA Programming Guide describe CUDA as a SIMT architecture and recommend that threads do not branch for best performance. Newer versions recommend that threads within a warp do not follow different code paths as each SM then has to execute each branch for each group of threads that follows that branch [94]. This suggests that code can branch on a warp, block or grid index without penalty unlike traditional SIMD architectures. Indeed this is the method used by the GPU to continue running a display while executing GPGPU kernels [75]. The GPU schedules thread blocks for execution on the SMs using static round robin scheduling. By having multiple thread blocks branch on the thread index each block can follow a different execution path. This provides a means of executing multiple kernels simultaneously on older GPUs that do not have this capability built in. The amount of shared memory and registers needed is the maximum of any execution path taken as `nvcc` is able to optimise resources away from thread blocks that do not use them. The number of threads per block must remain constant across blocks and must be set to the maximum required by any individual kernel.

This technique is used in our hybrid linear algebra functions to perform a block diagonal update simultaneously with an update of the trailing submatrix on the GPU when the block size is too small to offset the cost of copying the block diagonal matrix into CPU memory and back. One extra thread block is scheduled on the GPU and is added to a queue of blocks being multitasked on a particular SM in a similar manner to when all blocks are executing the same instruction path. The extra block branches away from the others and performs the diagonal update while the others execute a blocked matrix-multiply kernel. This is illustrated in Figure 3.1. The total time taken to execute the combined kernel is bound by maximum time taken for either kernel, instead of being the sum of both when executed sequentially.

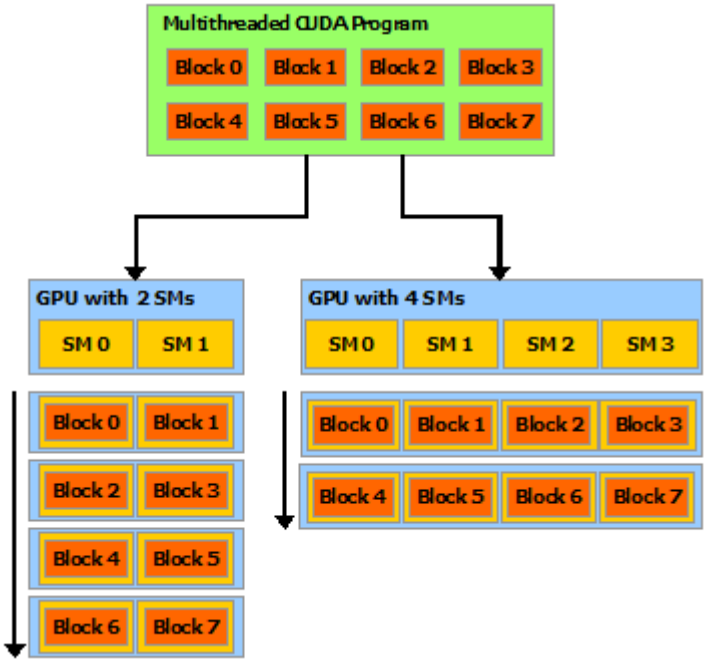


Figure 3.1: Exploiting the SIMT architecture to execute multiple kernels simultaneously. Here Block 7 is free to branch away from the others and follow a different execution path yet still is scheduled in the same manner by the CUDA hardware.

3.4 Using multiple GPUs

In order to use multiple GPUs within the same computer program a CUDA context must be created for each GPU device. GPU memory allocations, streams, modules and functions can only be used within the context they were created. GPU functions operate using the context current to the calling thread and each CPU thread may have only one GPU context current at any one time.

A multiGPU context structure was created in order to investigate which approach is best. A task structure was also created containing a pointer to the function to be executed, arguments for the function and a variable to hold the value returned from the function.

In the single-threaded case the multiGPU structure contains an array of CUDA contexts. The contexts are created with the `CU_CTX_SCHED_BLOCKING_SYNC` flag instructing them to block when synchronising with the GPU as this gives better performance in single-threaded programs. When a task is executed on a GPU the relevant context is made current to the calling thread using the `cuCtxPushCurrent` Driver API call. This function manages a stack of contexts for each thread with the current context at the top of the stack. The task is executed on the calling thread after which any previous context is restored using `cuCtxPopCurrent`. Asynchronous execution on the GPU continues after the context is no longer current to the

calling thread.

In the multi-threaded case the multiGPU structure contains an array of thread structures. Each thread structure contains a POSIX thread object and an unbounded array-backed queue of task structures protected by a mutex variable. When the thread starts it creates a CUDA context on a GPU and that context remains current throughout the life of the thread. Before terminating each thread destroys the context it created. Each thread monitors its own queue of tasks and sleeps until any appear. When executing a task the main thread places the task at the back of the queue for a particular thread and then wakes the thread prompting it to check the queue. The thread will remove the task from the head of the queue and begin executing it. Arguments for the task function must be copied onto the heap to ensure they can be accessed from other threads. In this case the task structure also contains a mutex variable to protect the result of the function. When destroying a task object the calling thread will block until the task is completed and the result is available.

3.5 Benchmarks and Error Analysis

3.5.1 GPU Occupancy

GPU occupancy is defined as the ratio of the number of warps that may be resident on a multiprocessor to the maximum number of resident warps supported by the GPU [94]. The number of warps that may be run concurrently on the same multiprocessor is defined by the amount of registers and shared memory resources required by each thread block compared to the resources available on each multiprocessor. The number of warps per thread block is chosen by the programmer as the thread block size. The CUDA Occupancy Calculator is a spreadsheet supplied with the CUDA Toolkit that allows programmers to enter the resource usage of a kernel and find out the GPU occupancy for different thread block sizes. The resource usage per thread for a kernel is output by `nvcc` when compiling to GPU binary code by using the `-Xptxas=-v` option. It used to be believed that higher GPU occupancy always means better performance as more of the processing hardware is able to be used, however this has been shown to be a false assumption for certain types of algorithms [123].

3.5.2 Timing Methods

The GNU implementation of the standard C library provides functions to time program execution on the CPU. The `clock_gettime` function is part of the POSIX standard and returns the number of seconds and nanoseconds since the Unix Epoch. The `clock` function returns the number of CPU clock ticks spent executing the program that called the function. By placing calls to these functions around a section of code the time taken to run the code can be

calculated. In the case of the `clock` method, the number of ticks must be divided by the `CLOCKS_PER_SEC` constant to convert clock ticks into seconds. The advantage of the latter method over the former is that it is immune to the system load while the program is being executed as it only counts clock ticks the CPU has spent processing the particular program. The `clock` method offers microsecond timing resolution while the `clock_gettime` method offers nanosecond resolution.

The CUDA tool kit provides functions to time GPU kernel execution and memory transfer to $0.5\mu s$ resolution. It presents timing through an event paradigm. Events are created on the device and recorded asynchronously with respect to the CPU. The CPU and GPU can synchronise on events. There is no documentation detailing which method the GPU uses to record events. A test program was written that records a start event on the GPU, waits for a second on the CPU then records a stop event on the GPU. If the elapsed time between the two events is less than a second then the GPU records the number of clock ticks, otherwise it records two time stamps. On the GPUs tested CUDA was found to use the time stamp method to record events.

For GPU programs the distinction between the two methods of timing does not matter as the GPUs in our test system cannot multi-task. However for hybrid functions there is a large difference in the amount of time measured using the two methods as the clock ticks method used by the CPU does not include GPU execution time. Our timings for hybrid and multi-GPU functions therefore use the POSIX `clock_gettime` method on a system with as few background processes running as possible.

3.5.3 Tuning the Block Size

When choosing the block size for a LAPACK implementation on a traditional CPU architecture the CPU cache size is the only factor that has to be considered. If the block size is too large the submatrices will not fit in the cache causing costly system RAM access. If the block size is too small then the cache will not be utilised to its full potential for data reuse.

When designing hybrid algorithms there are different factors to take into account such as the bandwidth and latency of the interconnect between the CPU and GPU. For heterogeneous hybrid architectures the difference in processing power of different processors must also be recognised. If the block size is too large, the transfers and CPU compute time will bound the performance of the algorithm and the GPU will be left idle waiting for the CPU. If the block size is too small, the GPU will not be operating to its full capacity and the CPU will be left idle waiting for the GPU. Therefore the optimum block size will be one that minimises the time any part of the system is waiting for the rest to catch up with processing.

For algorithms that perform a different amount of work on each loop iteration the optimum

block size for hybrid architectures can vary as the algorithm progresses. As a result we change the block size at each iteration of our linear algebra functions to balance the load and ensure maximum performance from all parts of the system.

3.5.4 Floating Point Error Analysis

The set of real numbers is continuous and infinite while computer memory is finite. A real number is therefore represented in computer memory by a discrete approximation with a certain amount of error. The IEEE has created a standard for representing real numbers on computers [7] which defines several floating-point formats which store real numbers as a fixed amount of digits with a known error [48]. Real numbers are stored in floating point format as a signed integer significand and exponent in the form $\pm d.ddd \times \beta^e$. The number of digits in the significand, p, is known as the precision and this, along with the exponent, e, and base, β characterise each floating point format. The IEEE standard defines three binary floating point formats (with $\beta = 2$) that use 32, 64 and 128 bits of memory divided among the significand and exponent. Each has a fixed value for p and a fixed range for e. These correspond to the 32 bit single-precision `float`, 64 bit double-precision `double` and 128 bit extended precision `long double` types in C. Any floating point implementation only has to implement one of these types to conform to the IEEE specification.

If the significand of a floating point number starts with $1.xxx$ then it is said to be normalised. The smallest difference between one normalised floating point number and the next is known as the Unit in Last Place or ULP. This refers to the value of the least significant bit of the significand when the exponent is zero and is used to measure the absolute error obtained when converting a real number in infinite precision into a floating point format with fixed precision. The maximum error when converting a real number to any IEEE floating point representation is 0.5 ULPs. An alternative measurement is the maximum relative error which is equal to $\frac{r-f}{r}$ where r is the real number and f is the number in floating point format. The upper bound of this formula is known as the machine epsilon, ε, and is available in C using the `FLT_EPSILON`, `DBL_EPSILON` and `LDBL_EPSILON` macros.

Floating point numbers must be denormalised when performing arithmetic in order to have their exponents match. The IEEE specification also defines five basic operations on floating point numbers ($+$, $-$, $\times$, $\div$ and $\sqrt{x}$) that must be implemented to less than 0.5ε precision and must use a guard digit when denormalising arguments. Other operations such as $\sin$, $\cos$, $\tan$, $\log$, $\exp$ and x^y may have greater error but most mathematical libraries compute these to within between 0.5ε and 1ε of the real answer [62, 11].

When sequences of elementary operations are applied successively the error incurred is ad-

ditive. This has lead to the design of special algorithms for summing that attempt to contain the error introduced [55]. When designing linear algebra operations on vectors and matrices there are additional issues such as optimising the memory access pattern to improve performance. This can lead to floating point operations being reordered and different implementations producing different results. In order to judge the correctness of our implementations we compare the output to that of an algorithm coded using the minimum number of floating point operations, and therefore the minimum amount of additive error. We also take into account the number of operations needed to compute the result.

3.6 Summary

We have introduced the necessary general methodology and technical information for developing novel blocked linear algebra functions, including discussions of the theoretical maximum bandwidths available on different GPU architectures. We proceed in the next chapter with first of our contributions by presenting novel approaches for accelerating Cholesky decompositions on hybrid architectures. After carefully detailing the underlying algorithms, we present simulation comparisons with the state of the art numerical linear algebra library, MAGMA.

Chapter 4

Hybrid Cholesky Decomposition

4.1 Introduction

Classical matrix decompositions from the linear algebra literature, such as the LU, QR and Cholesky decompositions [49], may all be used to solve systems of linear equations of the form $Ax = b$. All of these methods require A to be square but in the case where A is also symmetric and positive definite, then the Cholesky decomposition requires only around half the number of operations to compute. The Cholesky decomposition splits a matrix into the product of an upper (or lower) triangular matrix and its transpose, such that $A = U^T U$ or $A = LL^T$, and it can only be applied to symmetric, square, positive-definite matrices. Such matrices occur frequently in a wide variety of scenarios, for example in statistics where it is used to generate multivariate Gaussian random vectors with a known covariance [23]. Many machine learning algorithms also rely on the Cholesky decomposition, for example Gaussian processes [129, 103, 77], which are used for linear regression and prediction [130]. In computational statistics, the use of the Cholesky is ubiquitous in MCMC, for example Adaptive MCMC algorithms tune the proposal distribution of the sampler by adapting the covariance matrix at each iteration [107], which requires the Cholesky decomposition to be regularly reevaluated to propose a new sample. The Cholesky decomposition is therefore used heavily in statistical simulations and, particularly for high-dimensional problems, it is often the main bottleneck of the entire algorithm [59].

Each element of the Cholesky decomposition of a matrix A is defined recursively, as shown in Equation 4.2 for a lower triangular decomposition $A = LL^T$ and Equation 4.1 for an upper triangular decomposition $A = U^T U$. The subtraction of elements from $A_{i,j}$ is common to both cases i==j and i!=j in both upper and lower algorithms. In the case of the lower triangular algorithm, the calculation of each element $L_{i,j}$ depends on the prior calculation of all elements $L_{i,0\to j}$, $L_{j,0\to j}$ and $L_{j,j}$. Similarly in the upper triangular algorithm, the calculation of each element $U_{i,j}$ depends on the prior calculation of all elements $U_{0\to i,j}$, $U_{0\to i,i}$ and $U_{i,i}$. This

limits the parallelism inherent in the basic algorithm.

$$L_{i,j} = \begin{cases} \sqrt{A_{i,i} - \sum_{k=1}^{i-1} L_{i,k}^2} & \text{if } i = j \\ \frac{1}{L_{i,i}} \left(A_{i,j} - \sum_{k=1}^{i-1} L_{i,k} L_{j,k} \right) & \text{if } i < j \end{cases} \tag{4.1}$$

$$L_{i,j} = \begin{cases} \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2} & \text{if } i = j \\ \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right) & \text{if } i > j \end{cases} \tag{4.2}$$

The Cholesky decomposition is part of the LAPACK library specification [15] as the `SPOTRF` and `DPOTRF` subroutines in single and double precisions. Also part of the specification are the "unblocked" Cholesky decomposition routines `SPOTF2` and `DPOTF2`. The algorithms have $O(n^3)$ scaling and take approximately $\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$ FLOPs to calculate the Cholesky decomposition of an $n \times n$ matrix [22].

4.1.1 LAPACK Unblocked Algorithm

LAPACK relies on an optimised BLAS implementation being available on a target platform for high performance, as previously described in Section 2.3.1. This is because each LAPACK routine is implemented as a sequence of BLAS subroutine calls. The unblocked `SPOTF2` and `DPOTF2` routines are implemented in LAPACK as a sequence of BLAS 1 and BLAS 2 subroutine calls. This increases the parallelism available in the algorithm as the calculation of each element in the matrix-vector multiplication is independent and may be carried out simultaneously by the BLAS library. The calculation of the diagonal element involving the dot product and square root may also be carried out independently of the matrix vector multiplication. The unblocked algorithms from LAPACK are shown in Listings 4.1 and 4.2 for the unblocked Cholesky decomposition in the upper and lower triangles, respectively, using single precision BLAS. The BLAS routines used are `SDOT` to calculate the dot product of two vectors, `SSCAL` to multiply a vector by a scalar and `SGEMV` to perform matrix-vector multiplication. `sqrt` and `isnan` are C math library functions that calculate the square root and check whether a is the Not-A-Number value returned from an operation performed on invalid arguments, such as the square root of a negative number. The unblocked Cholesky decomposition routines return a non-zero integer in the `info` parameter if the matrix is not positive definite. This integer is the one-based index of the first diagonal element that is not positive definite.

Listing 4.1: Unblocked Cholesky Decomposition of an Upper Triangular Matrix

```
for (int i = 0; i < n; i++) {
  float aii = A[i * lda + i] - sdot(i, &A[i * lda], 1, &A[i * lda], 1);
  if (aii <= 0.0f || isnan(aii)) {
```

```
    A[i * lda + i] = aii;
    *info = i + 1;
    return;
  }
  A[i * lda + i] = aii = sqrtf(aii);
  if (i + 1 < n) {
    sgemv(CBlasTrans, i, n - i - 1, -1.0f, &A[(i + 1) * lda], lda, &A[i *
        lda], 1, 1.0f, &A[(i + 1) * lda + i], lda);
    sscal(n - i - 1, 1.0f / aii, &A[(i + 1) * lda + i], lda);
  }
}
```

Listing 4.2: Unblocked Cholesky Decomposition of a Lower Triangular Matrix

```
for (int j = 0; j < n; j++) {
  float ajj = A[j * lda + j] - sdot(j, &A[j * lda], lda, &A[j * lda], lda);
  if (ajj <= 0.0f || isnan(ajj)) {
    A[j * lda + j] = ajj;
    *info = j + 1;
    return;
  }
  A[j * lda + j] = ajj = sqrtf(ajj);
  if (j + 1 < n) {
    sgemv(CBlasNoTrans, n - j - 1, j, -1.0f, &A[j + 1], lda, &A[j], lda, 1.0
        f, &A[j * lda + j + 1], 1);
    sscal(n - j - 1, 1.0f / ajj, &A[j * lda + j + 1], 1);
  }
}
```

4.1.2 LAPACK Blocked Algorithm

Of the three variants of the blocked Cholesky decomposition in Table 2.2, we chose to implement variant 3 as it has the most scope for parallelism, both within the BLAS 3 operations it relies upon and between calls to the unblocked algorithm and matrix-multiply subroutines. The blocked Cholesky decomposition involves splitting the matrix into blocks as illustrated in Figures 4.1 and 4.2. The blocks are then updated according to Algorithms 1 and 2. The sequence of BLAS 3 subroutines may be reordered to perform the matrix multiply before or after the unblocked Cholesky decomposition as they operate independently on different parts of the matrix.

The Cholesky decomposition is unique among matrix decompositions in that it relies entirely upon level 3 BLAS operations. There is ongoing research into block algorithms for the

LU and QR decompositions to define them entirely using BLAS 3 subroutines. A fast matrix multiply implementation is key to the performance of the blocked Cholesky decomposition as it is the rate-determining step of the algorithm where most of the computation occurs. The call to the unblocked Cholesky decomposition lies on the critical path of the algorithm as it cannot overlap the symmetric rank-k update nor the triangular solve, both of which we describe in greater detail later in this chapter.

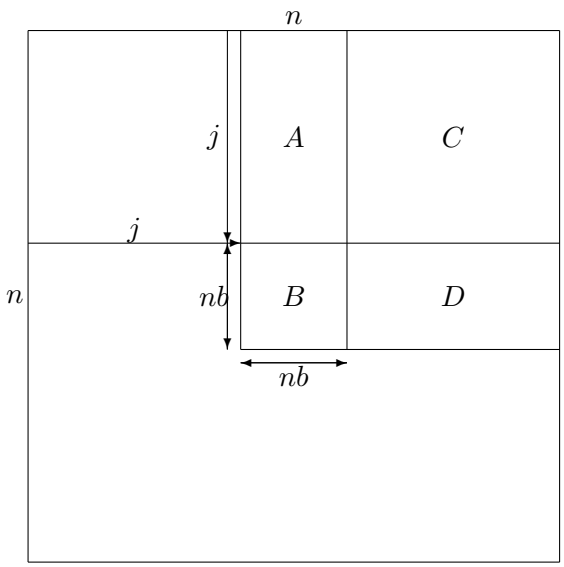


Figure 4.1: When performing the blocked Cholesky decomposition algorithm for upper triangular matrices the matrix is divided into the submatrices shown labeled as A, B, C and D. The block size nb is chosen by the programmer and the index j is the current iteration of the algorithm. By dividing the matrix into submatrices the size of the problem is reduced to a Cholesky decomposition of matrix B. The rest of the matrix can then be updated using highly parallel BLAS 3 subroutines.

4.1.3 Hybrid Blocked Algorithm

The choice to implement variant 3 of the blocked Cholesky decomposition algorithms is more easily motivated when considering a hybrid implementation. As the matrix multiplication and unblocked Cholesky decomposition steps of the algorithm are independent of one another they may be carried out simultaneously by different compute devices. Since GPUs are suited to parallel operations that consist of many independent calculations, we may therefore carry out the matrix multiplication step on the GPU. CPUs are faster at transcendental mathematical operations and branching, and so the the unblocked Cholesky step can be carried out more efficiently by the CPU. Modern GPUs can overlap memory transfers with computation, which allows the diagonal block transfers to and from host memory to be performed in parallel with

Algorithm 1 The upper triangular blockwise Cholesky decomposition algorithm expressed as a sequence of linear algebra operations on the submatrices defined in Figure 4.1. The call to `SPOTF2` performs the Cholesky decomposition of a smaller matrix B while the rest of the operations are performed using level 3 BLAS.

for j = 0,nb,...,n **do**

$B = B - A^T \times A$

$SPOTF2(\text{“}Upper\text{”}, B)$

$D = D - A^T \times C$

$D = B^{T-1} \times D$

end for

Algorithm 2 The lower triangular blockwise Cholesky decomposition algorithm expressed as a sequence of linear algebra operations on the submatrices defined in Figure 4.2. The call to `SPOTF2` performs the Cholesky decomposition of a smaller matrix B while the rest of the operations are performed using level 3 BLAS.

for j = 0,nb,...,n **do**

$B = B - A \times A^T$

$SPOTF2(\text{“}Lower\text{”}, B)$

$D = D - C \times A^T$

$D = D \times B^{T-1}$

end for

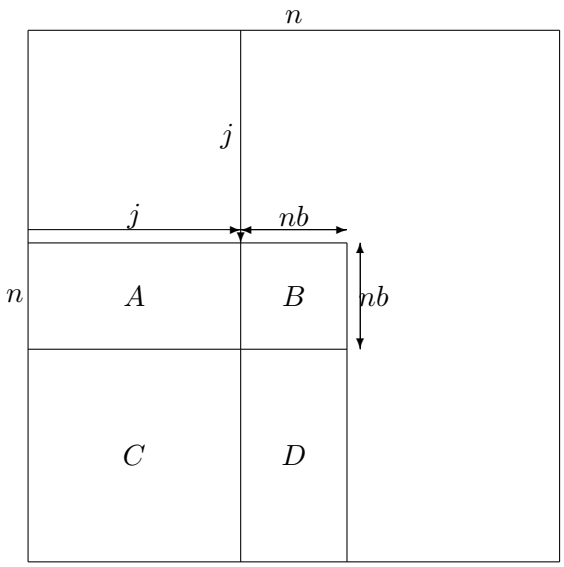


Figure 4.2: When performing the blocked Cholesky decomposition algorithm for lower triangular matrices the matrix is divided into the submatrices shown labelled as A, B, C and D. The block size nb is chosen by the programmer and the index j is the current iteration of the algorithm. By dividing the matrix into submatrices the size of the problem is reduced to a Cholesky decomposition of matrix B. The rest of the matrix can then be updated using highly parallel BLAS 3 subroutines.

the matrix multiply. If the block size is chosen correctly, the time taken to transfer the matrix block from GPU memory into host memory, perform the unblocked decomposition using the CPU and transfer the block back should be overlapped completely by the GPU matrix multiply, essentially performing the less efficient unblocked decomposition for no extra runtime cost.

4.2 Current State of the Art Methods

We now summarise the algorithmic approaches employed for routines in the linear algebra library MAGMA, which represents the current state of the art for hybrid Cholesky decomposition. The hybrid Cholesky decomposition requires three level 3 BLAS operations to be implemented for the GPU. In single precision these are the `SSYRK` operation which performs the symmetric rank-K update, `SGEMM` which performs matrix multiplication and `STRSM` which solves a triangular system of equations with multiple right hand sides. Volkov *et al.* [124] present an algorithm for single precision matrix multiplication on nVidia GPUs with compute capability 1.x, which includes the GPUs used in this study. As shown by Kågström *et al.* [66] the rest of the BLAS operations can be derived from an optimised matrix-multiply implementation, and so this is the approach taken here.

4.2.1 GPU Matrix Multiply

The `SGEMM` routine in the BLAS library performs single precision floating point matrix multiplications of the form $C = \alpha op(A)op(B) + \beta C$, where $op(A)$ is A or A^T and $op(B)$ is B or B^T. C is an $m \times n$ matrix which is updated in place, $op(A)$ is an $m \times k$ matrix and $op(B)$ is a $k \times n$ matrix, both of which are read-only.

Calculating one element of C requires reading k elements from A and k elements from B to calculate $C_{i,j} = \alpha \sum_{l=0}^{k} A_{i,l}B_{l,j} + \beta C_{i,j}$. A simple matrix multiply kernel for GPUs would schedule $m \times n$ threads and assign each one an element in C to calculate, reading elements in A and B from global memory as needed. This requires $2mnk$ words of bandwidth to calculate the entire matrix. Matrix multiplication kernels written for early GPUs without caches used this method and their throughput was bound by the speed of the memory interface. By dividing C into blocks of $mb \times nb$ the amount of bandwidth required can be reduced making the kernel compute bound. Additionally, by storing blocks of A and B in the cache memory they can be shared among elements of C without having to be re-fetched from global memory. This is illustrated in Figure 4.3. There are $\frac{m}{mb} \times \frac{n}{nb}$ blocks in C, $\frac{m}{mb} \times \frac{k}{kb}$ blocks in A and $\frac{k}{kb} \times \frac{n}{nb}$ blocks in B. Storing an $mb \times nb$ block of C in cache until all updates from A and B are accumulated requires reading $\frac{m}{mb}\frac{n}{nb}\frac{k}{kb}mbkb$ words of A and $\frac{m}{mb}\frac{n}{nb}\frac{k}{kb}kbnb$ words of B from global memory or

$$\begin{aligned} &\frac{m}{mb}\frac{n}{nb}\frac{k}{kb}mbkb + \frac{m}{mb}\frac{n}{nb}\frac{k}{kb}kbnb \\ =&\frac{mnk}{nb} + \frac{mnk}{mb} \\ =&mnk(\frac{1}{mb} + \frac{1}{nb}) \end{aligned} \tag{4.3}$$

words in total. This is independent of the block size kb which can be used to control the amount of cache memory needed to store A and B. The bandwidth reduction acquired by splitting a matrix C into blocks of $mb \times nb$ is therefore

$$\begin{aligned} &\frac{2mnk}{mnk(\frac{1}{mb} + \frac{1}{nb})} \\ =&\frac{2}{\frac{1}{mb} + \frac{1}{nb}} \end{aligned} \tag{4.4}$$

For a blocked matrix multiply kernel to be compute bound the amount of bandwidth reduction must be greater than the FLOP:word ratio of the compute device. FLOP:word ratios are shown in table 4.1.

Volkov *et al.*'s GPU matrix multiply kernel implements the $C = \alpha AB + \beta C$ and $C = \alpha AB^T + \beta C$ cases [124]. It holds an $mb \times nb$ block of C in registers on each GPU

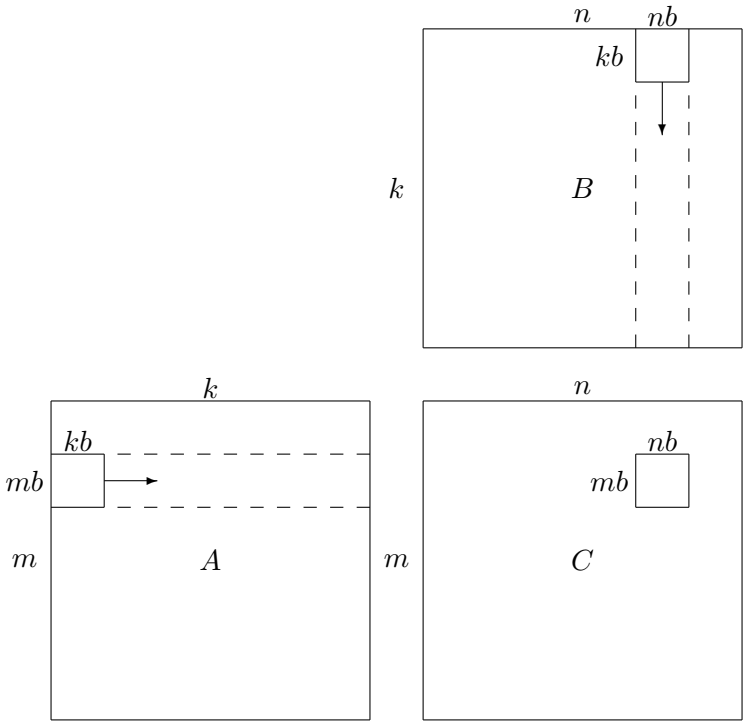


Figure 4.3: The blocked matrix multiply algorithm defines an $mb \times nb$ block of C which is held in registers. $mb \times kb$ blocks of A and $kb \times nb$ blocks of B are fetched into cache memory as needed and used to update the block of C. Blocking the matrix multiply algorithm in this way reduces the number of times A and B need to be read from main memory.

Single precision	multiply-add throughput	708.48 GFLOPs/s
	word bandwidth	39.744 Gwords/s
	FLOP:word ratio	17.826
Double precision	multiply-add throughput	88.56 GFLOPs/s
	word bandwidth	19.872 Gwords/s
	FLOP:word ratio	4.457

Table 4.1: This table shows the theoretical FLOP:word ratios of the nVidia GeForce GTX 285 in single and double precision. The theoretical throughput and bandwidth can be calculated from the GPU specifications in Table 3.1. To calculate the word bandwidth the theoretical bandwidth in bytes is divided by the number of bytes in a `float` or `double`.

multiprocessor and accumulates updates from $kb \times nb$ blocks of B held in shared memory which are broadcast to all threads in the thread block. As the block sizes are known at compile time shared memory is able to be allocated statically. It may also be allocated dynamically but this requires extra calls to the runtime library to define the amount of shared memory to allocate per block. Shared memory is allocated as a 2D array and padded to remove bank conflicts as recommended in the nVidia CUDA Programming Guide [94]. When reading B from shared memory the allocation must be padded to remove bank conflicts otherwise multiple threads within a warp will access the same element in shared memory reducing the bandwidth available by a factor of 16. When reading B^T from shared memory threads within a warp are already accessing elements from consecutive shared memory banks and padding is not necessary to achieve full bandwidth. A is read as needed from global memory. Thread blocks of 16×4 are used to read a 16×16 block of B into shared memory and transpose as required. The threads are then "unwrapped" to form a 64×1 block while reading A from global memory. This is used to update a 64×16 block of C held in registers giving a bandwidth reduction of 25.6 while using 1kb of shared memory. This is the $i, j/k, j$ form of the blocked algorithm with the first level of blocking handled by the CUDA hardware by splitting the workload into independent thread blocks. The inner j loop is unrolled manually 16 times, while the k loop is unrolled kb times automatically by the compiler using a precompiler `pragma unroll` directive. This results in a total of 256 floating point-multiply-add (`fmaf`) instructions in the unrolled inner loops executed by each thread, grouped as batches of 16 independent `fmaf` instructions. The innermost loop that iterates over the values of C must be unrolled completely in order to keep the array in registers which cost no extra clock cycles to access [93]. If the address of any element is taken, as happens implicitly with rolled loops, then the compiler will place the array in global memory which is costs an additional $470 - 720$ clock cycles to access [124]. As presented the kernel can only operate on matrices that are multiples of the block sizes chosen. This was generalised to any matrix size by Chien [30], who also introduces instruction optimisations using a third-party CUDA disassembler called decuda [122]. Chien found that of the two floating point multiply add instructions implemented by the GPU hardware, Volkov *et al.*'s kernel uses the slower one that operates with one argument in shared memory. By disassembling the GPU binary Chien was able to replace the slower instruction with a faster equivalent operating on arguments held entirely in registers.

Taking advantage of nvcc's support of C++ template functions, we implemented Chien's version of Volkov *et al.*'s SGEMM kernel in CUDA-C using template parameters for the transpose arguments and block sizes. These are evaluated at compile time using four template spe-

cialisations to produce all the functions needed for each SGEMM case from a single function template. In addition to the two transpose arguments for A and B, there are five template parameters that specify the block sizes. mb and nb are the number of rows and columns in the block of C held in registers. kb is the inner block size used to read blocks of A and B and this also controls the amount of unrolling applied to the inner k and j loops. bx and by are the x and y dimensions of the 2D block of threads used to update the block. These are equivalent to the CUDA built in variables $blockDim.x$ and $blockDim.y$, but having them as template variables allows the compiler to optimise away some calculations. When A is not to be transposed, only B is stored in shared memory, and therefore nb and kb dictate the amount of shared memory used. nb also controls register usage along with mb, as an $mb \times nb$ block of C is stored by each thread block. As the thread block is unwrapped to a linear array the size of the thread block $bx \times by$ must equal mb. For efficiency when B is not to be transposed bx must equal kb and when B is to be transposed bx must equal nb as this allows a single level of looping with no idle threads when fetching B into shared memory.

The register allocator in nvcc has improved since Volkov *et al.*'s SGEMM was written. Increased register reuse by the compiler allows the same code to use fewer registers, allowing more blocks to reside on one multiprocessor simultaneously and improving the performance of algorithms split into a large number of blocks. Despite this, the `-maxrregcount` compiler option is still needed to force the compiler to restrict the number of registers used to 32 per thread for the SGEMM kernels.

We were unable to implement the instructional optimisations introduced by Chien as the CUDA binary format has changed since and work on decuda has stopped. Current versions of the CUDA Toolkit include a disassembler for the new binary format. By comparing the output from disassembling our SGEMM kernel compiled with the current version of nvcc with that of Volkov's kernel compiled with the older version of nvcc, it would appear that the current version of nvcc still uses the slower of the two floating point multiply add instructions discovered by Chien. The higher performance of our SGEMM implementation when compared to Volkov *et al.*'s using the same GPU and compiler version must therefore be due to the newer compiler recognising more opportunities for optimisation in our code.

The lower triangular Cholesky decomposition implemented in MAGMA requires a matrix multiplication operation of the form $C = \alpha AB^T + \beta C$, which is already handled by Volkov *et al.*'s and Chien's kernels. The upper triangular Cholesky decomposition requires a matrix multiplication of the form $C = \alpha A^T B + \beta C$. This was implemented in their work by additionally caching an $mb \times kb$ block of A in shared memory and transposing it there. This means that

the block size mb now has an effect on the amount of shared memory used. The block sizes were adjusted to reduce the amount of shared memory needed so that a large number of thread blocks can still be accommodated on each GPU multiprocessor. During execution the thread block is unwrapped to 64×1 and rewrapped to 32×2 to update a 32×32 block of C held in registers. The wrapping of thread indices uses the modulo operator which is known to have low throughput on a GPU, however it is only used once per invocation and if the block size mb is a power of two the compiler will replace it with a faster sequence of bitwise operations. To limit the amount of shared memory needed kb is halved. This reduces the amount of instructions in the unrolled inner loops which lowers performance. The block sizes chosen give a bandwidth reduction of 32, which is higher than for the cases where A is not transposed, however due to requiring more shared memory overall the performance is lower when A is transposed. Block sizes for the four SGEMM functions are listed in Table 4.2 along with resource usage. The maximum amount of thread blocks that fit concurrently on a multiprocessor is limited by registers when A is not transposed and limited by shared memory when A is transposed.

Our GPU SGEMM implementation for double precision follows in a similar manner to the single precision version. This is appropriate for GPUs of compute capability 2.0 and above where shared memory is arranged as an array of memory banks with a 64-bit stride. On GPUs of compute capability less than 2.0 memory operations on 64-bit variables in shared memory need to be split into two 32-bit accesses to avoid bank conflicts. This can be performed in double precision using the `__double2loint`, `__double2hiint` and `__hiloint2double` functions built in to CUDA. The CUDA programming guide recommends to try implementations with and without bank conflicts as the hardware may be able to serialise requests with bank conflicts faster than executing two requests without bank conflicts. This is controlled in our code with a preprocessor macro that also checks the target compute capability requested via the `__CUDA_ARCH__` macro. It was found that both versions of the code execute at the same speed therefore the version without bank conflicts was chosen.

Another consequence of translating our GPU SGEMM to double precision is that register usage and shared memory requirements increase as each double variable is twice the size of a single precision floating point variable. As each 64 bit variable now occupies two 32 bit registers the block size nb needs to be halved to use the same number of registers to store a block of C. This also has the effect of halving the amount of shared memory needed for blocks of B. When A is transposed kb also needs to be reduced to reduce the amount of shared memory used to store blocks of A. mb is kept the same as it is linked to the number of threads per block which needs to be kept at 64 for maximum performance. bx and by are altered ensure no threads are

$op(A)$	$op(B)$	mb	nb	kb	bx	by	Threads	Registers	Shared memory	Blocks per SM
A	B	64	16	16	16	4	64	32	1172B	8
A	B^T	64	16	16	16	4	64	31	1108B	8
A^T	B	32	32	8	8	8	64	32	2292B	6
A^T	B^T	32	32	8	8	8	64	31	2260B	6

Table 4.2: These are the block sizes used for the single precision GPU SGEMM kernels. The number of blocks per multiprocessor depends on the number of threads, shared memory and register requirements defined by the block sizes. The block sizes are chosen to minimise resource usage and maximise performance. These kernels have 50% occupancy and a bandwidth reduction of $25.6\times$ when $op(A) = A$ and 37.5% occupancy with a bandwidth reduction of $32\times$ when $op(A) = A^T$.

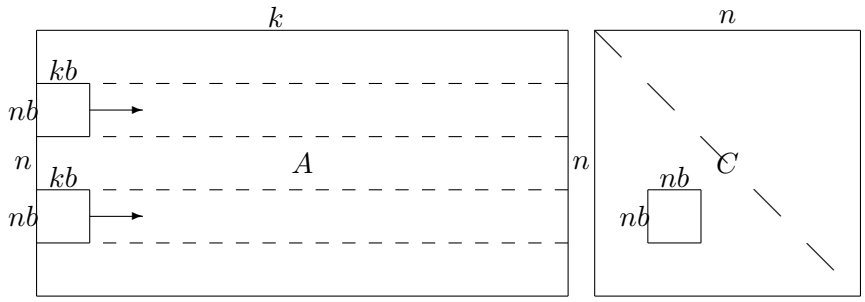
idle with the new block sizes when fetching A and B into shared memory. Block sizes for our GPU DGEMM are displayed in Table 4.3.

4.2.2 GPU Symmetric Rank-K Update

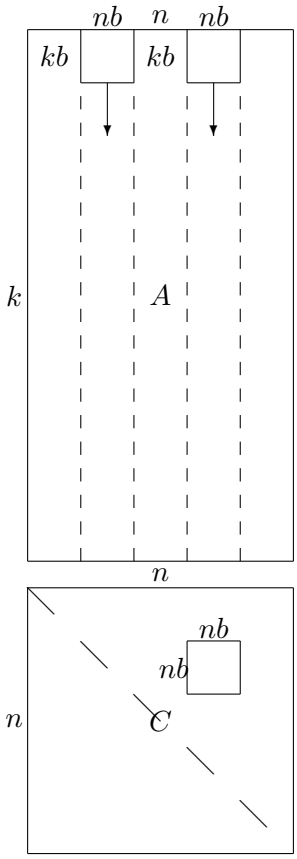
The symmetric rank-K update is an essential operation used within the overall Cholesky decomposition routine, which we will now describe. This operation is of the form $C = \alpha AA^T + \beta C$ or $C = \alpha A^T A + \beta C$ where only the lower or upper triangle of C, including the diagonal, is updated. C is an $n \times n$ square matrix and A is an $n \times k$ matrix in the first case and $k \times n$ in the second. It is implemented as the `SSYRK` subroutine in the BLAS library for single precision. There are four cases to handle depending on whether the upper or lower triangle of C is to be updated and whether the matrix multiply is AA^T or $A^T A$. The lower triangular Cholesky decomposition uses the $C = \alpha AA^T + \beta C$ form updating the lower triangle of C while the upper triangular Cholesky decomposition uses the $C = \alpha A^T A + \beta C$ form updating the upper triangle of C. These two cases are illustrated in Figure 4.4.

An optimised symmetric rank-K update kernel can be derived easily from an optimised matrix multiply kernel by taking the $C = \alpha AB^T + \beta C$ and $C = \alpha A^T B + \beta C$ cases and writing results to the upper or lower triangle of C only. This is illustrated in Figure 4.4 and requires two extra index variables to be allocated to map the two dimensional thread indices to global positions in C. These are not needed until writing the final block of C and can be placed in registers that are no longer required.

As the symmetric rank-K update is very similar to matrix multiplication the implementa-



(a) Blocked symmetric rank-K update for the lower triangular $C = \alpha AA^T + \beta C$ case. An $nb \times nb$ block of C is held in registers and updated by reading $nb \times kb$ blocks from rows of A in a similar manner to blocked matrix multiplication. This form of the symmetric rank-K update is used in the lower triangular Cholesky decomposition where blocks from the upper row of A are transposed.



(b) Blocked symmetric rank-K update for the upper triangular $C = \alpha A^T A + \beta C$ case. An $nb \times nb$ block of C is held in registers and updated by reading $kb \times nb$ blocks from columns of A in a similar manner to blocked matrix multiplication. This form of the symmetric rank-K update is used in the upper triangular Cholesky decomposition where blocks from the left row of A are transposed.

Figure 4.4: Blocked symmetric rank-K update. For diagonal blocks in both cases the rows of A overlap.

$op(A)$	$op(B)$	mb	nb	kb	bx	by	Threads	Registers	Shared memory	Blocks per SM
A	B	64	8	16	16	4	64	31	1234B	8
A	B^T	64	8	16	8	8	64	31	1116B	8
A^T	B	32	16	8	8	8	64	32	3484B	4
A^T	B^T	32	16	8	8	8	64	32	3420B	4

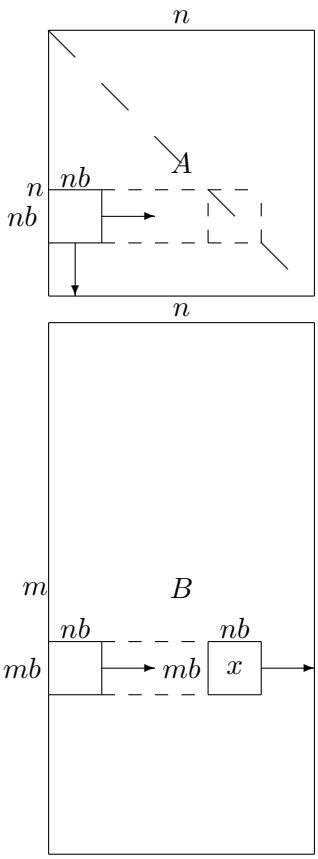
Table 4.3: These are the block sizes used for the double precision GPU DGEMM kernels. The number of blocks per multiprocessor depends on the number of threads, shared memory and register requirements defined by the block sizes. The block sizes are chosen to minimise resource usage and maximise performance. These kernels have 50% occupancy and a bandwidth reduction of $14.2\times$ when $op(A) = A$ and 25% occupancy with a bandwidth reduction of $21.33\times$ when $op(A) = A^T$. The shape of the thread block for the $C = \alpha AB^T + \beta C$ case is different than for single precision to ensure that $bx = nb$ when fetching B^T into shared memory in blocks of $nb \times kb$

tion for the GPU is also similar. As a result the block sizes, number of threads, resource usage and bandwidth reduction calculations from the matrix multiplication kernel in Section 4.2.1 also apply.

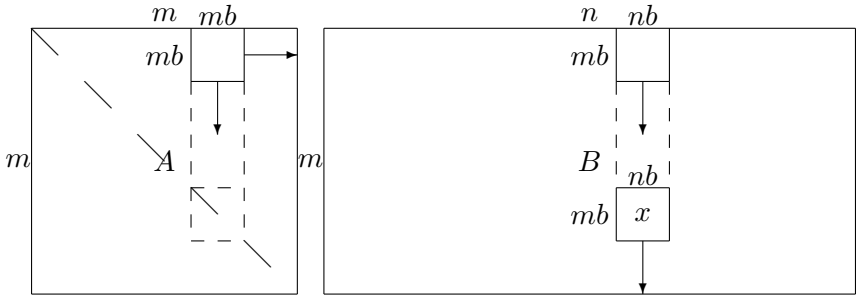
Volkov *et al.* [124] derive a `SSYRK` implementation from their optimised GPU `SGEMM` kernel in the same manner however they also use an additional technique to avoid scheduling extra thread blocks that will not compute any results in C. This involves allocating thread blocks only for blocks of C that contain at least one element. Using this technique we allocated a one dimensional grid of thread blocks and had them recalculate their positions in C but this was found to be slower than allocating a two dimensional grid of thread blocks for all of C and having those strictly above or below the diagonal exit as needed.

4.2.3 GPU Triangular Solve

The final operation required in the standard blocked Cholesky decomposition is a triangular solve. We present here an extension of the work in [124] rewriting the algorithm for use on the GPU itself, whereas the original was simply computed on the CPU. The triangular solve employs matrix equations of the form $op(A)X = \alpha B$ or $Xop(A) = \alpha B$. There are 16 cases in total depending on whether A multiplies X from the left or the right, is upper or lower triangular, is to be transposed or not and has a unit or non-unit diagonal. When A multiplies X from the left A is an $m \times m$ matrix and the system is solved by forming $X = \alpha op(A)^{-1}B$.



(a) Blocked triangular matrix solve for the lower triangular $XA^T = \alpha B$ case. This form of the triangular matrix solve is used in the lower triangular Cholesky decomposition. The block marked x starts on the left of B and is held in registers. It is updated by reading blocks from the current row of B and matching row in A. Blocks of A are transposed in shared memory. Only blocks in B to the left of x that have already been calculated are used to update the current x and as a result only the lower triangle of A is read. After x has been calculated it is written back to B and a new x is defined to the right of the old x. Each row of B is calculated by one processor.



(b) Blocked triangular matrix solve for the upper triangular $AX^T = alphaB$ case. This form of the triangular matrix solve is used in the upper triangular Cholesky decomposition. The block marked x starts at the top of B and is held in registers. it is updated by reading blocks from the current column of B and matching column in A. Blocks of A are transposed in shared memory. Only blocks in B above x that have already been calculated are used to update the current x and as a result only the upper triangle of A is read. After x has been calculated it is written back to B and a new x is defined below the old x. Each column of B is calculated by one processor.

Figure 4.5: Blocked triangular matrix solve

When A multiplies X from the right A is an $n \times n$ matrix and the system is solved by forming $X = \alpha B op(A)^{-1}$. In both cases X and B are $m \times n$ matrices. The in-place implementation in the BLAS specification overwrites B with X. The upper triangular Cholesky decomposition uses the $A^T X = \alpha B$ case where A is the $nb \times nb$ upper triangular submatrix on the diagonal and B is the matrix to the right of the diagonal block with $m <= n$. The lower triangular Cholesky decomposition uses the $XA^T = \alpha B$ case where A is the $nb \times nb$ lower triangular submatrix on the diagonal and B is the matrix below the diagonal block with $m >= n$. Both assume non-unit diagonal elements in A. These cases are illustrated in Figures 4.5a and 4.5b.

$$X_{i,j} = \begin{cases} \alpha B_{i,j} - \sum_{k=i+1}^{m} A_{i,k} X_{k,j} & \text{if } A \text{ is upper triangular and not transposed.} \\ \alpha B_{i,j} - \sum_{k=0}^{i} A_{i,k} X_{k,j} & \text{if } A \text{ is lower triangular and not transposed.} \\ \alpha B_{i,j} - \sum_{k=i+1}^{m} A_{k,i} X_{k,j} & \text{if } A \text{ is upper triangular and transposed.} \\ \alpha B_{i,j} - \sum_{k=0}^{i} A_{k,i} X_{k,j} & \text{if } A \text{ is lower triangular and transposed.} \end{cases} \tag{4.5}$$

$$X_{i,j} = \begin{cases} \alpha B_{i,j} - \sum_{k=0}^{j} A_{k,j} X_{i,k} & \text{if } A \text{ is upper triangular and not transposed.} \\ \alpha B_{i,j} - \sum_{k=j+1}^{n} A_{k,j} X_{i,k} & \text{if } A \text{ is lower triangular and not transposed.} \\ \alpha B_{i,j} - \sum_{k=j+1}^{n} A_{j,k} X_{i,k} & \text{if } A \text{ is upper triangular and transposed.} \\ \alpha B_{i,j} - \sum_{k=0}^{j} A_{j,k} X_{i,k} & \text{if } A \text{ is lower triangular and transposed.} \end{cases} \tag{4.6}$$

Each element of X is calculated using Equations 4.5 and 4.6 for $op(A)X = \alpha B$ and $Xop(A) = \alpha B$ respectively. If A has a non-unit diagonal then each $X_{i,j}$ is also divided by the corresponding diagonal element of A.

The equations show dependencies between elements of X that do not allow for an efficient GPU implementation. Elements of X cannot be calculated independently of one another. In the reference BLAS implementation the loops over i and j are reversed where needed to satisfy these dependencies. When using a GPU a high degree of synchronisation between GPU threads is needed to implement a correct solution. In addition when matrices are stored in column major layout the $op(A)X = \alpha B$ cases require sums down matrix columns which are implemented via reduction. This requires even more synchronisation and some GPU threads being left idle in order to fetch data from global memory at maximum bandwidth. The right cases require independent sums across matrix rows which can be carried out simultaneously by multiple threads fetching coalesced data from global memory.

The design of the GPU triangular solve algorithm follows that of the `SGEMM` implementation by Volkov *et. al* [124]. An $mb \times nb$ block of X is stored in registers by each GPU

multiprocessor and held there until all updates have been accumulated from blocks of A and B. X is initialised with values from B as in the reference BLAS implementation. When A multiplies X from the left reading and writing X from registers is done after transposing via shared memory. This forms $X^T = \alpha B^T (op(A)^{-1})^T$ for the left cases allowing sums to be accumulated independently by multiple threads as when A multiplies X from the right. B is fetched into shared memory in blocks of $kb \times nb$ for the cases where A multiplies X from the left and is fetched directly from global memory otherwise. For the cases where A multiplies X from the left and is not transposed A is fetched directly from global memory. For all other cases A is fetched into shared memory in blocks of $mb \times kb$ when not transposed and $kb \times mb$ when transposed. Up until the point where B is being read from the same block as X will be written to the operation performed is matrix multiplication. The `SAXPY` updates in the unrolled inner loop are modified to use subtraction rather than addition as in the reference BLAS implementation. When the block of B is in the same position as X each thread updates the elements in a column of X. This allows the dependencies between elements within a column to be satisfied while allowing each column to be processed independently. The length of the final inner loop executed by each thread is determined by another loop resulting in a triangular loop structure that `nvcc` is unable to automatically unroll. This results in `nvcc` storing the block of X in global memory rather than registers so that array offsets can be calculated. To enable X to be stored in registers the entire triangular loop needs to be manually unrolled.

When A multiplies X from the left a one-dimensional row of thread blocks is scheduled and when A multiplies X from the right a one-dimensional column of thread blocks is scheduled. The entire kernel is wrapped in a for loop to enforce data dependencies rather than relying on separate kernel launches to force synchronisation between thread blocks. The amount of work performed by the loop changes on every iteration. The block sizes used for each case in single and double precision are listed in Tables 4.4 and 4.5. Due to the increased register usage of the extra outer and unrolled inner loops the block size needs to be halved compared to the matrix multiplication kernels to avoid registers spilling into global memory.

There are a couple of alternative approaches that may be taken to implement a triangular solve kernel for GPUs, one of which is given in [40]. It involves forming A^{-1} on the GPU before using triangular matrix multiplication to form $B = \alpha A^{-1} B$ or $B = \alpha B A^{-1}$, and is implemented in the MAGMA library. The CUBLAS implementation of the triangular matrix solve for all precisions involves multiple alternating kernel launches of a matrix multiply kernel optimised for small matrices followed by a smaller triangular solve kernel. This removes the need for an outer for loop to force an ordering of updates and frees registers to allow more

	mb	nb	bx	by	Threads	Registers	Shared memory	Blocks per SM
$upper(A)X = \alpha B$	8	64	8	8	64	32	2396	6
$lower(A)X = \alpha B$	8	64	8	8	64	32	2396	6
$upper(A^T)X = \alpha B$	8	64	8	8	64	32	2428	6
$lower(A^T)X = \alpha B$	8	64	8	8	64	32	2428	6
$Xupper(A) = \alpha B$	64	8	8	8	64	31	348	8
$Xlower(A) = \alpha B$	64	8	8	8	64	32	348	8
$Xupper(A^T) = \alpha B$	64	8	8	8	64	32	316	8
$Xlower(A^T) = \alpha B$	64	8	8	8	64	32	316	8

Table 4.4: This table lists the block sizes for the single precision triangular solve kernels. In each case an $mb \times nb$ block of B is stored in registers and updated by a $bx \times by$ block of threads. Register usage is higher than for the corresponding matrix multiply kernel due to the extra loop required to enforce ordering of the updates to blocks of B. As a consequence nb is lower to fit the same number of blocks on each multiprocessor. This in turn reduces the amount of bandwidth reduction to $14.22\times$ which is lower than the GPU FLOP:word ratio of 17.826 making the kernels bandwidth bound.

thread blocks to fit concurrently on each GPU multiprocessor.

4.3 Improvements on the State of the Art

In this section we present the contributions that introduce novel approaches to performing unblocked Cholesky decompositions, firstly focussing on their implementation on CPUs, and subsequently on GPUs. We note that state of the art open source numerical linear algebra libraries do not currently make use of these approaches, and hence they may be used to further increase the performance of these routines.

4.3.1 Unblocked Cholesky on the CPU

The reference implementation of the unblocked Cholesky decomposition relies on subroutines from levels 1 and 2 of the BLAS as we saw in Section 4.1.1. This algorithm is shown in Listing 4.1 and is formed of a loop containing a dot product, square root, matrix-vector multiplication and vector scaling operation in that order.

The dependencies between the matrix elements in the Cholesky decomposition mean that only certain sequences of operations result in a correct implementation. The vector scaling

	mb	nb	bx	by	Threads	Registers	Shared memory	Blocks per SM
$upper(A)X = \alpha B$	4	16	4	4	16	32	732	8
$lower(A)X = \alpha B$	4	16	4	4	16	32	732	8
$upper(A^T)X = \alpha B$	4	16	4	4	16	32	764	8
$lower(A^T)X = \alpha B$	4	16	4	4	16	32	764	8
$Xupper(A) = \alpha B$	16	4	4	4	16	31	220	8
$Xlower(A) = \alpha B$	16	4	4	4	16	32	220	8
$Xupper(A^T) = \alpha B$	16	4	4	4	16	32	188	8
$Xlower(A^T) = \alpha B$	16	4	4	4	16	32	188	8

Table 4.5: This table lists the block sizes for the double precision triangular solve kernels. In each case an $mb \times nb$ block of B is stored in registers and updated by a $bx \times by$ block of threads. Register usage is higher than for the corresponding matrix multiply kernel due to the extra loop required to enforce ordering of the updates to blocks of B. As a consequence nb is lower to fit the same number of blocks on each multiprocessor. This in turn reduces the amount of bandwidth reduction to $6.4\times$, however, in contrast to the single precision case, this is still higher than the GPU FLOP:word ratio of 4.457 for double precision making these kernels compute bound.

operation depends on the result of the square root which itself depends on the outcome of the dot product so these operations must come in this order. The vector scaling operation also depends on the result of the matrix-vector multiplication but this can come before or after the dot product or square root. In the lower triangular case, it is advantageous to move the matrix-vector multiplication before the square root as this improves data locality for the vector scaling operation with the result of the square root already being held in a register.

In the first iteration of the loop, the dot product in both the upper and lower triangular cases is of a zero-length vector. In the last iteration, similarly the vector-matrix multiplication and vector scaling operations have no work to do. In the reference implementation an if statement checks for the latter case and avoids wasting instructions performing function calls that return immediately. By expanding all the BLAS calls inline all instructions used to perform function calls can be saved. The loops forming the dot product on the first iteration and the vector-matrix multiplication and vector scaling on the last iteration will be empty and skipped with the same cost of evaluating an if-statement. This also enables the compiler to peel the first and last iterations of the loop where the inner loops from the BLAS calls are empty.

In the lower triangular case the loop forming the dot product iterates over the same range as the inner loop forming the matrix-vector multiplication. In the upper triangular case the outer loop forming the matrix-vector multiplication iterates over the same range as the vector scaling operation. In both cases these loops can be fused together into a single loop. This gets rid of the overhead of initialising and managing loop counters for two loops and also gets rid of two inefficient BLAS 1 subroutine calls that use non-unit vector strides.

The final algorithm incorporating these optimisations is displayed in Listing 4.3. To summarise, by carefully considering the structure of this algorithm, and in particular the order of loops, we may eliminate strides across the memory and allow the compiler to insert vector operations that are particularly well suited to GPUs with wide vector units.

Listing 4.3: Optimised unblocked Cholesky decomposition algorithm.

```
if (uplo == CBlasUpper) {
  for (size_t i = 0; i < n; i++) {
    // Perform the dot product
    register float temp = zero;
    const float * restrict B = A;
    for (size_t k = 0; k < i; k++)
      temp += A[i * lda + k] * B[i * lda + k];

    // Calculate the diagonal element
```

```
      register float aii = A[i * lda + i] - temp;
      if (aii <= zero || isnan(aii)) {
        A[i * lda + i] = aii;
        *info = (long)i + 1;
        return;
      }
      aii = sqrtf(aii);
      A[i * lda + i] = aii;

      // Perform the combined matrix-vector multiplication/vector scaling
      for (size_t j = i + 1; j < n; j++) {
        temp = zero;
        for (size_t k = 0; k < i; k++)
          temp += A[j * lda + k] * A[i * lda + k];
        A[j * lda + i] = (A[j * lda + i] - temp) / aii;
      }
    }
  }
  else {
    for (size_t j = 0; j < n; j++) {
      // Perform the combined dot product/matrix-vector multiplication
      for (size_t k = 0; k < j; k++) {
        register float temp = A[k * lda + j];
        for (size_t i = j; i < n; i++)
          A[j * lda + i] -= temp * A[k * lda + i];
      }

      // Calculate the diagonal element
      register float ajj = A[j * lda + j];
      if (ajj <= zero || isnan(ajj)) {
        *info = (long)j + 1;
        return;
      }
      ajj = sqrtf(ajj);
      A[j * lda + j] = ajj;

      // Vector scale
      for (size_t i = j + 1; i < n; i++)
        A[j * lda + i] /= ajj;
    }
  }
}
```

4.3.2 Optimising Diagonal Block Transfer

We now consider a novel approach to speed up the transfer of matrices to and from GPU memory, which is generally a slow operation as the overhead in setting up a copy operation is repeated for each column. When the leading dimension of a matrix is equal to the number of rows, the matrix contains no alignment padding and is laid out contiguously in memory. This enables the matrix to be transferred in one linear copy operation with the overhead in setting up the copy incurred only once. The time taken to transfer a matrix with or without alignment padding can be calculated using Equation 3.7.

Submatrices with fewer rows than the original matrix will always contain padding around columns as they share the leading dimension with the original matrix. This is the case for the diagonal block transferred at each iteration of the hybrid Cholesky decomposition. By extending the size of the diagonal block so that it contains the same number of rows as the larger matrix a single linear copy can be used to copy the entire block column around the diagonal. Although this results in a larger amount of memory being copied it will be faster to copy a contiguous $n \times nb$ block column than an $nb \times nb$ submatrix if Equation 4.7 is satisfied.

$$\frac{n \times nb \times \textbf{sizeof}(\mathbf{T})}{bandwidth} + overhead < nb \times (\frac{nb \times \textbf{sizeof}(\mathbf{T})}{bandwidth} + overhead) \tag{4.7}$$

For the upper triangular hybrid Cholesky decomposition block column copy can be used to transfer the diagonal block from GPU memory and back as submatrix A in the column above the diagonal block B is constant in the current iteration and the lower triangle is always constant. For the lower triangular hybrid Cholesky decomposition block column copy can only be used to transfer the diagonal block from GPU memory as the submatrix D in the column below the diagonal block B is being updated by the GPU executing the matrix multiply in parallel. This is visualised in Figures 4.6 and 4.7.

4.3.3 Dynamic Block Sizing

We now propose the introduction of an additional level of blocking for heterogeneous computing environments consisting of multi-core CPUs with GPU accelerators working in parallel. These blocks are communicated between computing devices, which work on them using their own blocked or unblocked routines. The block size for the coarser level of blocking may then be chosen to balance the workload between the heterogeneous compute devices, helping to ensure that processors are not left idle when they could be doing useful computation.

The block size used for the blocked LAPACK routines for CPUs is chosen so that the working set for the call to the unblocked routine fits in the CPU cache. This is suitable for

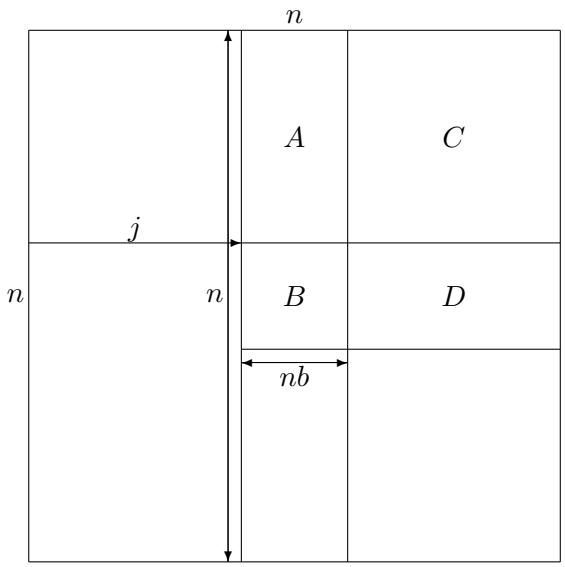


Figure 4.6: Defining a column around submatrix B that extends to the top and bottom of the matrix allows B to be copied to and from GPU memory in a single transfer if the matrix is not padded. This will be faster than copying each column of B separately if the overhead in setting up each copy is large in relation to the time taken to transfer the data. In the upper triangular Cholesky decomposition this optimisation can be used to transfer B into host memory and back into device memory as it overlaps the update of submatrix D to the right of B.

sequential algorithms executed on single core CPUs or parallel algorithms executed on homogeneous multi-core CPUs where each core has the same amount of cache.

In the hybrid Cholesky decomposition, the majority of the processing occurs in the matrix multiplication executed on the GPU. This is overlapped with a smaller Cholesky decomposition of the diagonal block executed by the CPU. Using a fixed block size the number of floating point operations consumed by the diagonal block Cholesky is constant across every iteration of the algorithm. The number of floating point operations taken by the matrix multiplication on the other hand changes on each iteration, increasing towards the midpoint of the algorithm and decreasing towards the end. This is shown in Figure 4.8 with the floating point operations consumed by the rank-K update and triangular solve routines removed for clarity. The area between the two curves for the matrix multiplication and Cholesky decomposition represent the difference in time taken to execute the two functions on a heterogeneous computing device. When the line from the matrix multiplication is lower the compute device executing it has to wait while the Cholesky decomposition of the diagonal block is completed. When the line is higher the device executing the Cholesky decomposition of the diagonal block finishes first and has to wait for the matrix multiplication.

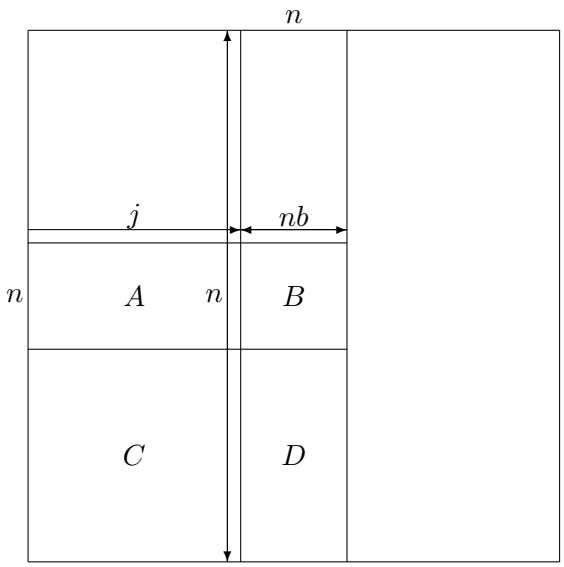


Figure 4.7: Defining a column around submatrix B that extends to the top and bottom of the matrix allows B to be copied to and from GPU memory in a single transfer if the matrix is not padded. This will be faster than copying each column of B separately if the overhead in setting up each copy is large in relation to the time taken to transfer the data. In the lower triangular Cholesky decomposition this optimisation can be used to transfer B into host memory only as the column overlaps submatrix D which is updated by the GPU in parallel.

Changing the block size on each iteration of the hybrid Cholesky decomposition would make better use of the available computing power from the CPU and GPU as neither would be waiting for the other to finish executing a function before proceeding. Ideally the block size would be changed on each iteration to minimise the area between the curves for matrix multiplication and Cholesky decomposition. The block size can be increased towards the midpoint then decreased towards the end which would cause the number of operations consumed by the Cholesky decomposition to increase then decrease in a similar manner to the matrix multiplication. Alternatively the block size can decrease towards the midpoint and increase towards the end to bring the curve from the matrix multiplication down towards the Cholesky decomposition. Both these approaches are shown in Figures 4.9 and 4.10 where the area between the curves is noticeably less than in Figure 4.8. The ideal block size for each iteration can be calculated analytically for homogeneous computing devices as the processing power for each core executing the different functions is the same. In a heterogeneous computing environment this cannot be done as the number of floating point operations consumed by each function needs to be normalised by the performance of the computing device executing it and there is a difference in theoretical and actual performance which also varies across architectures.

A tuning run was performed to measure the difference between execution times of the matrix multiplication by the GPU and the Cholesky decomposition on the CPU over a range of block sizes for a fixed matrix size. At each iteration the block size with the minimum time difference was selected. It was found that decreasing the block size then increasing it resulted in better performance however this tuning run would be costly to implement at runtime. We therefore choose a simpler scheme of starting the block size at $N/2$ and halving it at each iteration towards the centre, then doubling it until the end of the algorithm.

4.3.4 Unblocked Cholesky on the GPU

We now consider how the unblocked Cholesky can also be optimised on GPUs using vector optimisation. We note that we must now consider the problem of having less cache available to us than in the CPU version, which we can tackle by optimising the cache access pattern. When the block size is too small the time taken to transfer the diagonal block from GPU memory and back to perform the Cholesky decomposition using the CPU can far outweigh the time taken to perform the GPU matrix multiply which runs in parallel. The cost of copying the diagonal block can be avoided completely if the diagonal block decomposition can be performed by the GPU.

As the Cholesky decomposition involves a lot of data reuse it would be preferable to store the entire matrix in shared memory. As discovered by Volkov *et. al* [124] 64 threads per block is the minimum amount needed to get maximum performance. Using 64 threads would require storing a 64×64 matrix in shared memory which, in single precision, would use all available shared memory without leaving space for kernel parameters. To solve this problem triangular packed storage mode is used to store only the upper or lower triangle of the matrix including the diagonal. This requires storage space for $\frac{n(n+1)}{2}$ matrix elements for an $n \times n$ matrix. As well as making more efficient use of shared memory, using triangular packed storage mode does not require padding to reduce shared memory bank conflicts. Accessing data in consecutive rows is also as fast as accessing data in consecutive columns when using shared memory, unlike global memory.

The GPU kernel performing the unblocked Cholesky decomposition starts by reading the entire matrix into shared memory using triangular packed storage mode. Thread 0 reads the "info" error parameter from global memory, initialises it to zero and caches it in shared memory for faster access among threads in the block. A combined dot product and matrix vector multiplication is then performed similar to that introduced in Section 4.3.1 is then performed by all threads to update the current column or row in shared memory. At iteration j, thread j calculates the diagonal element using a square root and checks for positive definiteness updating the

info parameter in shared and global memory if necessary. A synchronisation barrier is inserted to ensure that updates to the info parameter and diagonal element are visible to all threads in the block. All threads check the value of the info parameter in shared memory and break from the loop if it is non-zero. The final operation is the vector scaling operation in which each thread scales one element of the vector using the diagonal element from shared memory. After the decomposition is completed in shared memory all threads unpack the matrix into global memory.

The unblocked Cholesky decomposition kernel requires two synchronisation points per iteration similar to the matrix multiplication kernel in Section 4.2.1. It uses a single thread block of 64×1 threads in single precision and each thread updates a row or column in shared memory. As the matrix is triangular this means that the number of threads actively executing is reduced by one at each iteration. Shared memory use is higher than with `SGEMM` but as this kernel is only executed by one thread block resource usage can be higher. For double, complex and double complex precisions the thread block size is 32×1. When the number of threads per block is less than or equal to the number of threads in a warp synchronisation barriers to ensure read-after-write dependencies in shared memory may be omitted and replaced with accesses through a pointer marked `volatile`[94].

The resource usage for this kernel in single and double precisions is shown in Tables 4.6a and 4.6b. For double precision nb, the maximum size of the matrix that will fit in shared memory when using triangular packed storage mode, is reduced by a factor of 2.

4.3.5 Combining Unblocked Cholesky and Inverse with Matrix Multiplication on the GPU

Let us now consider how these approaches might be overlapped on a GPU to further increase efficiency. Dynamic block sizing from Section 4.3.3 can make the block size too small for the GPU matrix multiplication to overlap the decomposition of the diagonal block using the CPU as using the CPU requires the block to be transferred into host memory and back. In this case the decomposition can be performed entirely using the GPU leaving the diagonal block in device memory. The GPU kernel introduced in Section 4.3.4 runs using one thread block on one GPU multiprocessor leaving the other multiprocessors idle. On newer GPUs that support concurrent kernels the kernel can be run on one stream in parallel with the matrix multiplication on another in order to use the rest of the GPU multiprocessors. On older GPUs without this capability a kernel must be written which combines the unblocked GPU Cholesky decomposition with the matrix multiplication as detailed in Section 3.3.4. Since the matrix multiplication kernel uses a two dimensional grid of thread blocks an extra row or column of blocks must be scheduled if the

	nb	Threads	Registers	Shared memory
Upper Triangular	64	64	9	8372B
Lower Triangular	64	64	10	8372B

(a) The unblocked Cholesky decomposition kernels for the GPU are executed using a single one dimensional thread block as the threads within the block all need access to the diagonal element stored in shared memory. As only one thread block is used it will be the only block residing on a GPU multiprocessor and can use more shared memory and registers than if it were having to share multiprocessor resources with other thread blocks.

	nb	Threads	Registers	Shared memory
Upper Triangular	32	32	22	4276B
Lower Triangular	32	32	22	4276B

(b) The unblocked Cholesky decomposition kernels for the GPU are executed using a single one dimensional thread block as the threads all need access to the diagonal element which is stored in shared memory. As the single thread block will

Table 4.6: The unblocked Cholesky decomposition kernels for the GPU are executed using a single one-dimensional thread block in order to communicate matrix elements between threads using shared memory. As a kernel using a single thread block will occupy an entire GPU multiprocessor to itself it can use more registers and shared memory than if it were sharing the multiprocessor with other thread blocks.

	Threads	Registers	Shared memory
Upper Triangular	64	32	4338B
Lower Triangular	64	31	1636B

(a) Resource usage for the combined Cholesky decomposition, inverse and matrix multiply GPU kernel in single precision.

	Threads	Registers	Shared memory
Upper Triangular	32	32	7692B
Lower Triangular	32	32	1338B

(b) Resource usage for the combined Cholesky decomposition, inverse and matrix multiply GPU kernel in double precision.

Table 4.7

matrix is lower or upper triangular, respectively, in order to minimise the number of extra blocks scheduled. In the upper triangular case the thread block with $blockIdx.x == gridDim.x - 1$ and $blockIdx.y == 0$ performs the Cholesky decomposition and inverse while the thread blocks with $blockIdx.x < gridDim.x - 1$ execute the matrix multiplication. Other thread blocks with $blockIdx.x == gridDim.x - 1$ and $blockIdx.y > 0$ exit immediately. Since the GPU Cholesky decomposition has a limit to the size of matrix that will fit in shared memory the combined kernel can only be used when the block size is small enough such that the Cholesky decomposition does not use more shared memory than the matrix multiplication.

4.3.6 Alternatives to GPU Triangular Solve

In this section we investigate how we might restructure the Cholesky decomposition such that our algorithm does not require a triangular solve, since this operation is slow on GPUs due to the dependencies between elements of the output matrix, as we saw in Section 4.2.3.

The triangular solve routine from the BLAS library solves matrix equations of the form $AX = \alpha B$ by forming $X = \alpha A^{-1}B$ or $XA = \alpha B$ by forming $X = \alpha BA^{-1}$. The same result can be achieved by forming A^{-1} separately then using triangular matrix multiplication to form $X = \alpha AB$ or $X = \alpha BA$. An in-place implementation overwrites B with X meaning there are similar data dependencies as when performing the triangular matrix solve. However, rather than each element of X depending on elements of X that have already been calculated, each element of X depends on elements of B that have not been calculated. This means that in an

out-of-place implementation each element of X is independent. As a result triangular matrix multiplication is much better suited to GPUs than triangular matrix solve.

As the name suggests triangular matrix multiplication is easily derived from regular matrix multiplication in a similar manner to a rank-K update. The key difference is that when forming a rank-K update of the form $C = \alpha AA^T + \beta C$ only the upper or lower triangle of C is updated while for a triangular matrix multiply of the form $X = \alpha AB$ only the upper or lower triangle of A is read.

This leaves the problem of calculating A^{-1}. The data dependencies involved in calculating a matrix inverse are the same as for the triangular solve routine which means that this is also not suited to GPUs. The inverse of a matrix can be formed via its Cholesky decomposition, where applicable, and this will be faster than calculating the inverse separately if the Cholesky decomposition has already been performed. As A in this case is the diagonal block from the hybrid Cholesky decomposition its decomposition has already been calculated by the CPU. This means that the CPU can additionally form the inverse of the diagonal block using its Cholesky decomposition and this will also be carried out in parallel with the GPU matrix multiplication.

Forming the inverse of a matrix from its Cholesky decomposition is done in two steps. The first calculates the inverse of the upper or lower triangle, in place, forming $A = A^{-1}$ where A is upper or lower triangular. The second step forms $A = AA^T$ or $A = A^TA$ to copy the inverse to the rest of the matrix. Since only the upper or lower triangle is read by the triangular matrix multiply only the first step needs to be implemented. This is performed out-of-place on the CPU so that it may overlap the upload of the Cholesky decomposition of the diagonal block onto the GPU. A temporary diagonal block is also allocated on the GPU to store the inverse.

As the GPU triangular matrix multiply is out-of-place, an additional temporary matrix is needed to store D. Rather than copy D directly it is populated with an out-of-place matrix multiply and the triangular matrix multiply copies the result back into the correct submatrix. This requires an extra $nb \times n$ or $n \times nb$ matrix to be allocated on the GPU.

4.4 Results

In order to test the overall performance of these suggested algorithmic changes involved in the Cholesky decomposition, we benchmarked our code by running it for values of N ranging from 64 to 4096 in steps of 64 for both single and double precision, and upper and lower triangular matrices. Each function benchmarked was timed using the appropriate method from Section 3.5.2 and an average was taken over 20 iterations, in order to remove any costs associated with loading the code onto the GPU when the function is first called.

Random symmetric, square, positive definite input matrices with condition number 2 were generated using Algorithm 3 on the CPU before being uploaded into GPU memory. A condition number of 1 gives the identity matrix, so 2 was chosen to give a random matrix that would pass the error analysis in Section 3.5.4. After performing the error analysis the input matrix was replaced with an identity matrix when benchmarking to avoid the algorithm exiting early due to the matrix becoming non-positive definite. Since our algorithm assumes the matrix is dense this does not effect the results.

Algorithm 3 Generating random positive definite matrices with desired condition number

Require: c the desired condition number

Ensure: A a matrix with condition number c

$\mathbf{A} \sim diag(U[1, c])$ with 1 and c at least once

$\mathbf{u} \sim U(0, 1)$

$\mathbf{v} = \mathbf{A}\mathbf{u}$

$t = \frac{2}{\mathbf{u}^T\mathbf{u}}$

$s = t^2 \frac{\mathbf{u}^T\mathbf{v}}{2}$

$\mathbf{w} = t\mathbf{v} - s\mathbf{u}$

$\mathbf{A} = \mathbf{A} - \mathbf{u}^T\mathbf{w} + \mathbf{u}\mathbf{w}^T$

Figures 4.11 and 4.12 show the performance of our hybrid algorithms in single precision. Maximum performance of our algorithms is 249 GFlops/s for the lower triangular algorithm, while performance for the upper triangular algorithm peaks at 201 GFlops/s. This is faster than the default implementation using the LAPACK algorithm which peaks at 225 and 166 GFlops/s respectively. Figures 4.13 and 4.14 show the performance of the same algorithms in double precision. Performance is significantly lower than single precision peaking at 64.3 GFlops/s for the lower triangular algorithm and 54.7 GFlops/s for the upper triangular algorithm, compared to 58.5 and 47.8 GFlops/s for the lower and upper triangular algorithms based on the default implementation using the LAPACK algorithm. Replacing the triangular solve operation with separate CPU inverse and GPU triangular matrix multiplication gives an increase in performance of all our hybrid Cholesky decomposition algorithms. This is shown in the figures as "Using STRMM" and "Using DTRMM" for single and double precision triangular matrix multiplication. Applying the additional optimisations of block column copy and dynamic blocking however appear to have a detrimental impact on the overall performance which we shall discuss at the end of the chapter.

A fixed block size of 256 is chosen for all implementations that do not use dynamic block-

ing. The spikes in Figure 4.11 are due to the GPU matrix multiply routine. In the upper triangular Cholesky decomposition the matrix multiply is of the form $C = \alpha A^T B + \beta C$. This requires more shared memory to store blocks of A and additional synchronisation when reading those blocks from shared memory. As a result, less thread blocks are able to occupy each GPU SM simultaneously and occupancy is reduced. This causes spikes in both Figures 4.11 and 4.12 of which the ones in Figure 4.11 for the upper triangular Cholesky decomposition are more pronounced.

Figures 4.15 and 4.16 compare the performance of our best hybrid algorithms in single precision with the same algorithms from the MAGMA library. For the lower triangular case our algorithms are approximately 15% faster on average and up to 50% faster for large n. For the upper triangular case our best algorithms approximately match the performance of the MAGMA library.

4.5 Using Multiple GPUs

Finally, we consider how we might make use of multiple GPUs connected to a single CPU. Being able to use multiple GPUs installed in a system has the obvious benefit of increased processing power possibly leading to faster computation. The overhead involved in transferring arguments and results to and from a single GPU is reduced when using multiple GPUs as the data is typically divided among GPUs and transfers can be done in parallel and asynchronously with respect to both the CPU and other GPU computation.

Only the matrix multiply kernel needs to be executed on the GPUs as all other BLAS operations can be derived from matrix multiply by combining it with the required BLAS operation executed on the CPU in the same way a blocked BLAS routine for CPUs or GPUs is designed.

As we noted in Section 3.4, execution of a kernel on multiple GPUs is asynchronous. When using the multiple GPU matrix multiply in the context of a Cholesky decomposition it is possible to overlap the execution of the matrix multiply on multiple GPUs with the factorisation of the diagonal block on the CPU as is done in the hybrid Cholesky decomposition executing on a single GPU.

The design of the matrix multiply algorithm for multiple GPUs follows the standard blocked matrix multiply design similar to single GPU matrix multiply. Each GPU uploads an $mb \times nb$ block of C into global memory and holds it there until all updates are completed. $C = \beta C$ is performed by the GPU while the first blocks of A and B are uploaded asynchronously into global memory. A is transferred in blocks of $mb \times kb$ while A^T is transferred in blocks of $kb \times mb$. Similarly B is transferred in blocks of $kb \times nb$ while B^T is transferred in

blocks of $nb \times kb$. $C = \alpha op(A)op(B) + C$ is then performed while the next blocks of A and B are uploaded. This is repeated until all updates from A and B are completed after which C is downloaded into the correct place in host memory.

mb and nb need to be chosen to be the minimum values that fully utilise the processing resources on the GPU in order to minimise the time taken to transfer C. For the single precision case when $op(A) = A$ each GPU multiprocessor computes a 64×16 block of C using 64 threads. Due to shared memory and register requirements a maximum of 8 thread blocks will fit concurrently on each multiprocessor. The nVidia GeForce GTX 280 has 30 multiprocessors so best performance should occur when a minimum of 240 thread blocks are scheduled on the GPU. To calculate the matrix dimensions that will provide 240 thread blocks all possible factors of 240 are taken and multiplied by the 64×16 block size used by the kernel. This is illustrated in Table 4.8. Given that each GPU multiprocessor is being given the maximum amount of work it is capable of each block size results in the same performance for a given value of k.

These calculations were repeated for the single precision case where $op(A) = A^T$ and each GPU multiprocessor computes 32×32 blocks of C using 64 threads. Due to increased shared memory usage only 6 thread blocks will run concurrently on each GPU multiprocessor. This means that a minimum of 180 blocks need to be scheduled on the GPU to get maximum performance. The results of these calculations are shown in Table 4.9.

Given mb and nb the bandwidth reduction provided by the block size can be calculated using Equation 4.4. As with other blocked matrix multiply routines the algorithm will be compute bound if the bandwidth reduction is greater than the FLOP:word ratio. When calculating the FLOP:word ratio the actual performance of the matrix multiply kernel is used along with the actual bandwidth of the PCI Express interface from Table 3.3. The bandwidth reduction is fixed for a particular block size while the word bandwidth is fixed for a particular precision. By rearranging the inequality in Equation 4.8 an upper bound on the performance of the GPU kernel can be calculated such that the algorithm remains compute bound. Since performance increases with k up to a point this gives a maximum value that can be used for kb for the multiple GPU algorithm. A single tuning run was performed using the block size that provides the maximum bandwidth reduction in order to find how performance scales with k. The maximum value of k that results in a compute bound algorithm is chosen for the value of kb. If the performance stops increasing the algorithm will never become bandwidth bound so the minimum value of k that gives the maximum performance is chosen for kb.

$$\text{bandwidth reduction} > \frac{\text{throughput}}{\text{word bandwidth}} \tag{4.8}$$

Factors of 240	Overall Block Size	Bandwidth Reduction	Maximum Performance	kb
1×240	64×3840	125.90	185.8	16
2×120	128×1920	240.00	354.3	112
3×80	192×1280	333.91	492.9	192*
4×60	256×960	404.21	596.7	192*
5×48	320×768	451.76	666.9	192*
6×40	384×640	480.00	708.6	192*
8×30	512×480	495.48	731.5	192*
10×24	640×384	480.00	708.6	192*
12×20	768×320	451.76	666.9	192*
15×16	960×256	404.21	596.7	192*
16×15	1024×240	388.86	574.1	192*
20×12	1280×192	333.91	492.9	192*
24×10	1536×160	289.81	427.8	192*
30×8	1920×128	240.00	354.3	112
40×6	2560×96	185.06	273.2	32
48×5	3072×80	155.94	230.2	16
60×4	3840×64	125.90	185.8	16
80×3	5120×48	95.11	140.4	16
120×2	7680×32	63.73	94.0	16
240×1	15360×16	31.97	47.1	16

Table 4.8: MultiGPU SGEMM block sizes for $op(A) = A$. Values for kb marked with an asterisk are minimum values that give maximum performance.

Factors of 240	Overall Block Size	Bandwidth Reduction	Maximum Performance	kb
1×180	32×5760	63.65	93.9	8
2×90	64×2880	125.22	184.8	24
3×60	96×1920	182.86	269.9	80
4×45	128×1440	235.10	347.1	136$*$
5×36	160×1152	280.98	414.8	136$*$
6×30	192×960	320.00	472.4	136$*$
9×20	288×640	397.24	586.4	136$*$
10×18	320×576	411.43	607.4	136$*$
12×15	384×480	426.67	629.9	136$*$
15×12	480×384	426.67	629.9	136$*$
18×10	576×320	411.43	607.4	136$*$
20×9	640×288	397.24	586.4	136$*$
30×6	960×192	320.00	472.4	136$*$
36×5	1152×160	280.98	414.8	136$*$
45×4	1440×128	235.10	347.1	136$*$
60×3	1920×96	182.86	269.9	80
90×2	2880×64	125.22	184.8	24
180×1	5760×32	63.65	93.9	8

Table 4.9: MultiGPU SGEMM block sizes for $op(A) = A^T$. Values for kb marked with an asterisk are minimum values that give maximum performance.

4.6 Discussion

In this chapter we examined current state of the art approaches for performing a hybrid Cholesky decomposition. We proposed a number a novel algorithmic approaches in this context based on theoretical considerations, then carefully examined the resulting performance in practice.

In all of the Cholesky decomposition algorithms considered in this chapter the majority of the floating point operations are performed in the matrix multiply operation which is used to increase parallelism. On the GPU only B is stored in shared memory when performing $C = \alpha AB^T + \beta C$ while both A and B are stored in shared memory when performing $C = \alpha A^T B + \beta C$. This reduces the number of thread blocks that can fit simultaneously on each GPU multiprocessor and results in lower performance. As a result the lower triangular Cholesky decomposition algorithm, which uses the former case, has higher performance.

Replacing the slow triangular solve algorithm with the separate hybrid inverse and triangular matrix multiply operations provides the best performance increase for our algorithms, since they replace an algorithm that exhibits high data dependencies and limited parallelism with one that is highly parallel. The data dependencies are processed in the inverse, which is carried out by the CPU. This also provides another step of hybrid parallelism whereby the CPU performs more work concurrently with the GPU.

Our block column copy and dynamic blocking optimisations, on the other hand, do not improve the performance of our hybrid algorithms. The block column copy introduces extra calculations into the inner loop of the algorithm to calculate the theoretical time taken to transfer the matrix, and we discovered that in practice these additional calculations offset any performance gain the optimised copy might give. The block column copy should therefore only be applied when the overhead of setting up each copy is large, and performance will highly depend upon the driver versions used. Calculating the optimal block size at each iteration is not possible without applying a tuning run. Therefore a simpler scheme of halving and doubling the block size may be taken, which although not optimal prevents introducing too many extra calculations into the inner loop.

In conclusion, we recommend the use of the hybrid algorithm that uses out of place triangular matrix multiplication, as it has consistently better performance on the GPU than the in place triangular matrix solve variant. On older systems with high latency and lower bandwidth between the CPU and GPU, the column copy and dynamic blocking may improve the performance further and, additionally, if the GPU has considerably more power than the CPU then the concurrent kernel optimisation provides a means of performing the entire decomposition on a GPU.

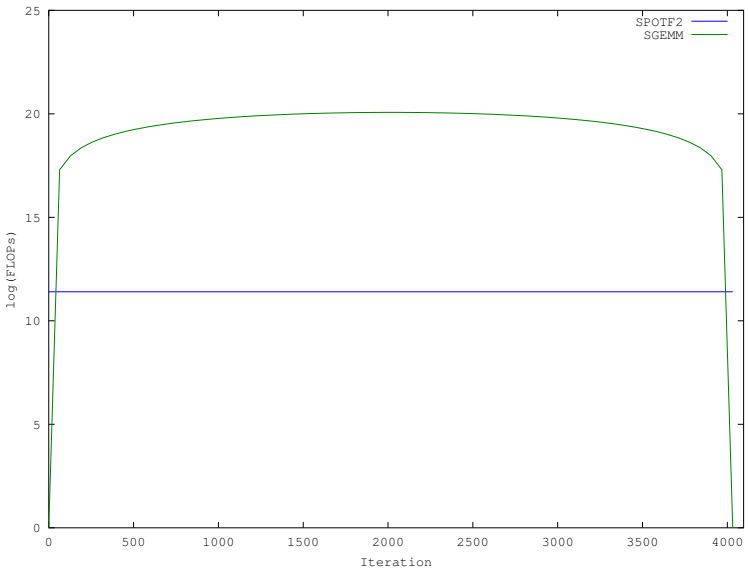


Figure 4.8: Using a static block size the number of FLOPs consumed by the SGEMM operation changes on each iteration. The number of FLOPs consumed by the SPOTF2 operation remains constant across iterations. The SGEMM is executed on the GPU in parallel with the SPOTF2 on the CPU and therefore the area between the two curves represents the time spent by the CPU waiting for the GPU to finish the SGEMM. The block size must be chosen to minimise this area in order to get maximum performance.

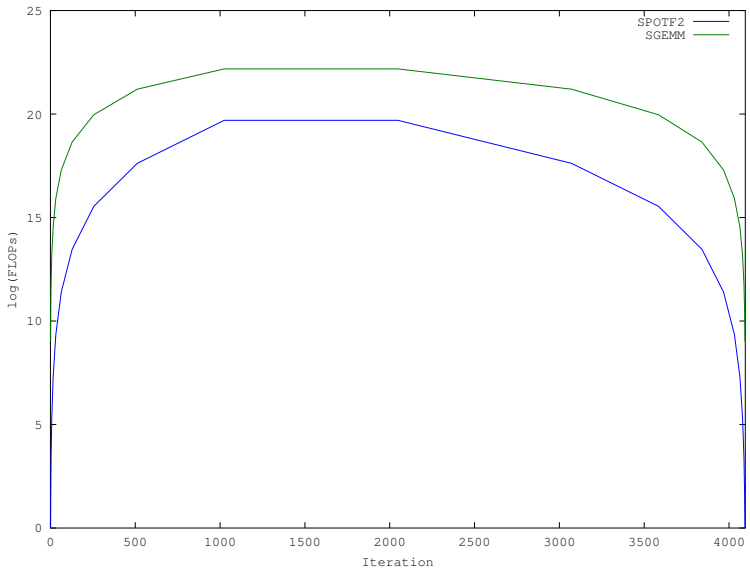


Figure 4.9: Starting with a small block size, increasing it towards the halfway point of the algorithm then decreasing it causes the number of FLOPs consumed by the SPOTF2 operation to curve upwards towards the SGEMM. This also has the effect of increasing the number of FLOPs consumed by the SGEMM pushing it further from the SPOTF2 curve.

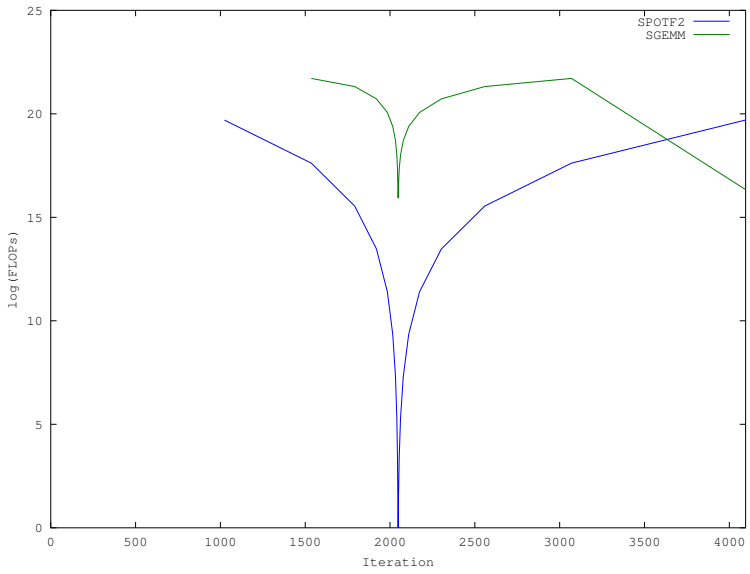


Figure 4.10: Starting with a large block size, decreasing it towards the halfway point of the algorithm then increasing it causes the number of FLOPs consumed by the SGEMM operation to curve downwards towards the SPOTF2. This brings the two curves closer together meaning that the CPU spends less time waiting for the GPU to finish. The two curves cross over towards the end of the algorithm when the CPU and GPU process the remaining elements using the current block size.

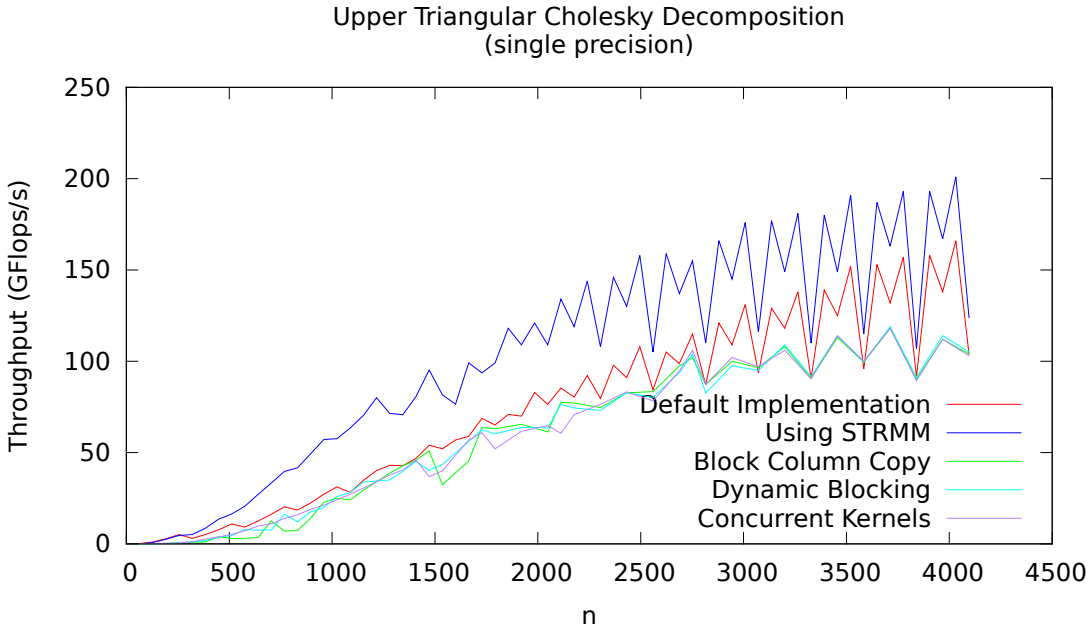


Figure 4.11: Performance of our upper triangular hybrid Cholesky decomposition in single precision. The optimisations are applied cumulatively from the default implementation, which follows the LAPACK algorithm with a hybrid step overlapping the CPU SPOTF2 with a GPU SGEMM.

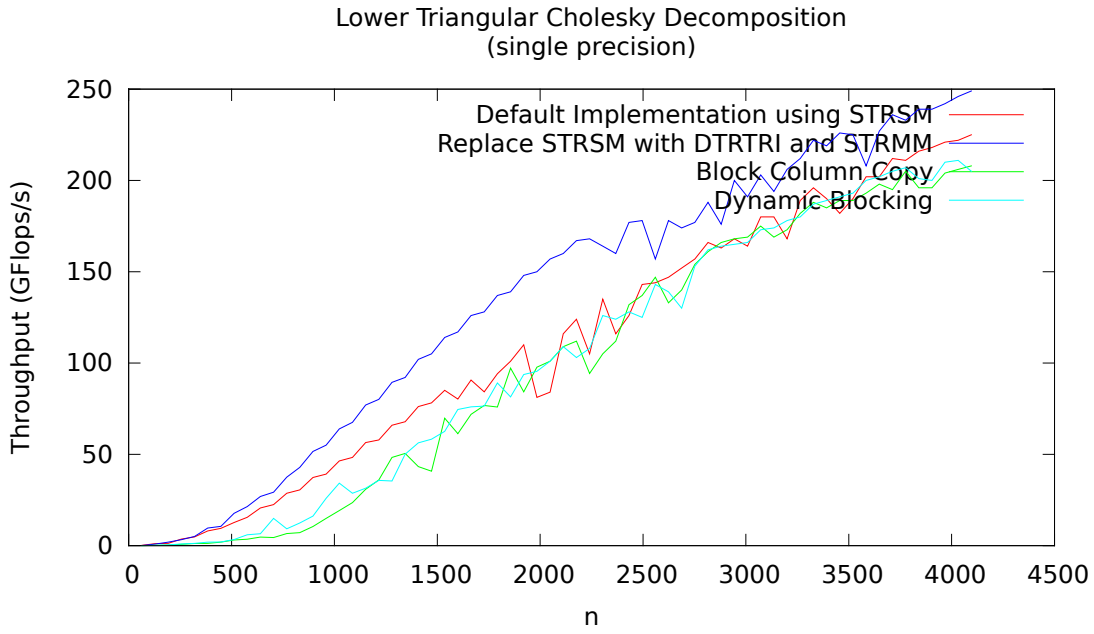


Figure 4.12: Performance of our lower triangular hybrid Cholesky decomposition in single precision. The optimisations are applied cumulatively from the default implementation which follows the LAPACK algorithm with a hybrid step overlapping the CPU SPOTF2 with a GPU SGEMM. The performance of the lower triangular Cholesky decomposition is higher than the upper triangular algorithm as the SGEMM operation performs AB^T in the lower triangular decomposition faster than $A^T B$ in the upper triangular decomposition.

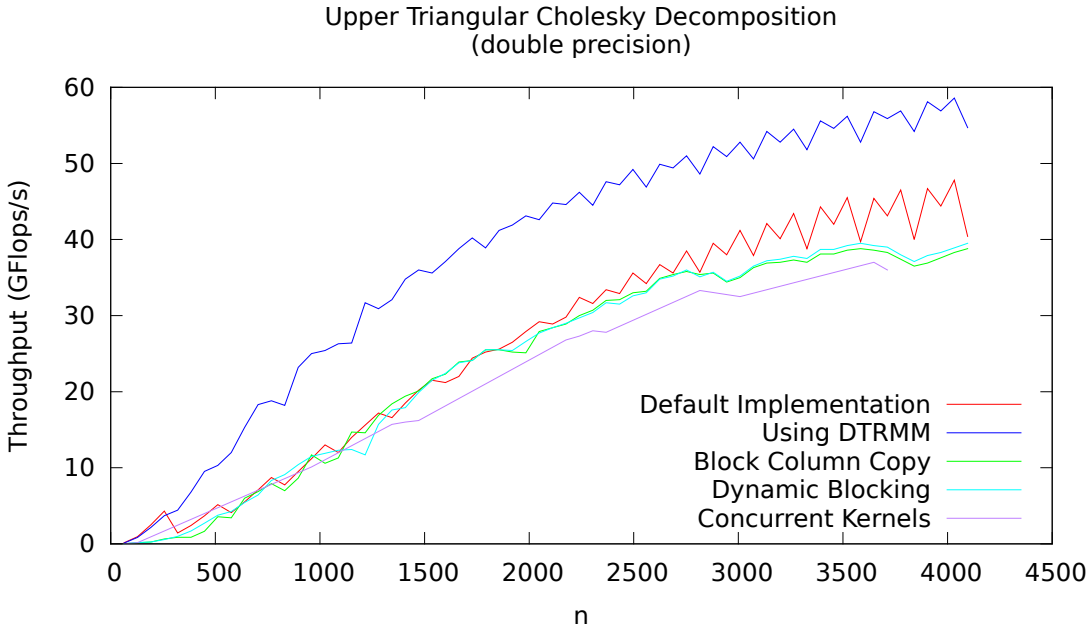


Figure 4.13: Performance of our upper triangular hybrid Cholesky decomposition in double precision. The optimisations are applied cumulatively from the default implementation as in the single precision case.

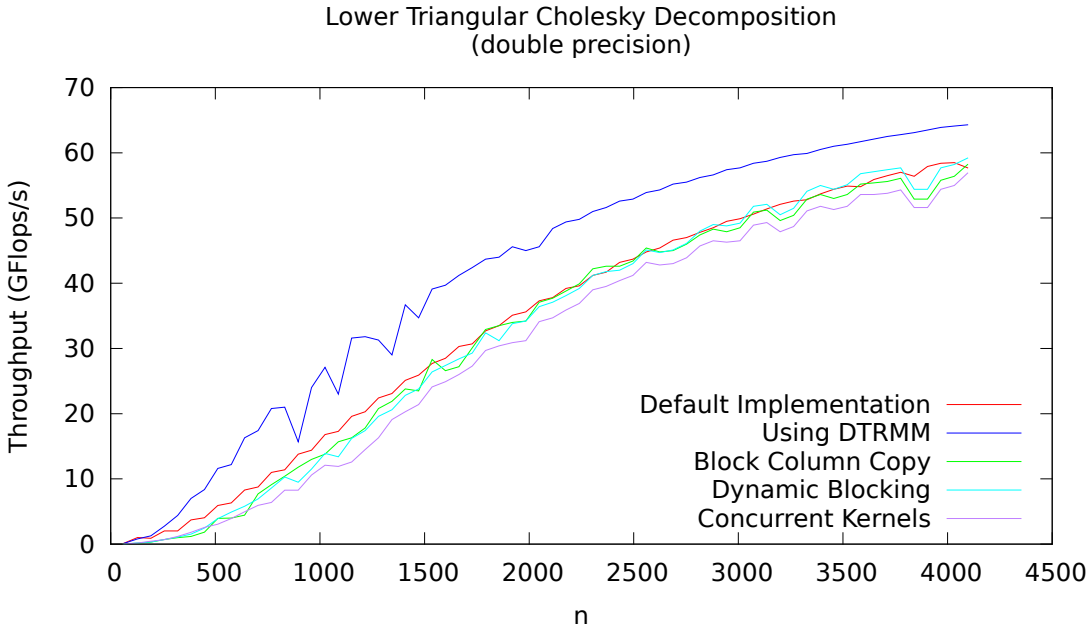


Figure 4.14: Performance of our lower triangular hybrid Cholesky decomposition in double precision. The optimisations are applied cumulatively from the default implementation as in the single precision case.

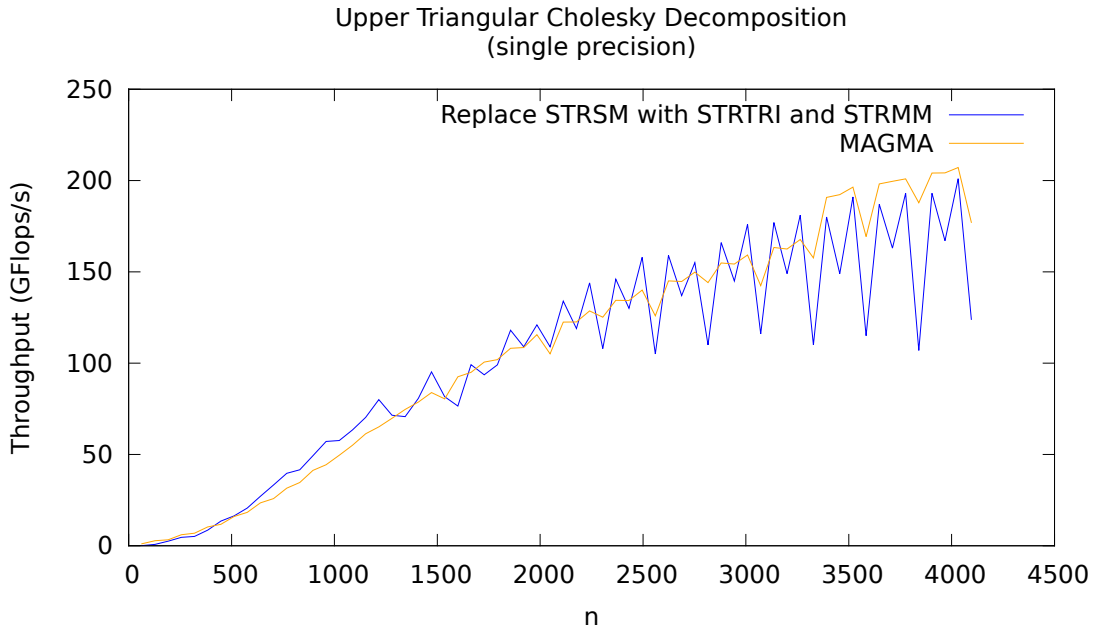


Figure 4.15: Performance of our upper triangular hybrid Cholesky decomposition compared to the MAGMA library in single precision. Our fastest implementation which replaces the triangular solve with a separate inverse and triangular multiply performs almost the same as the MAGMA library which does not include this optimisation. The MAGMA library implements the default algorithm using a highly optimised scheduler to get high performance.

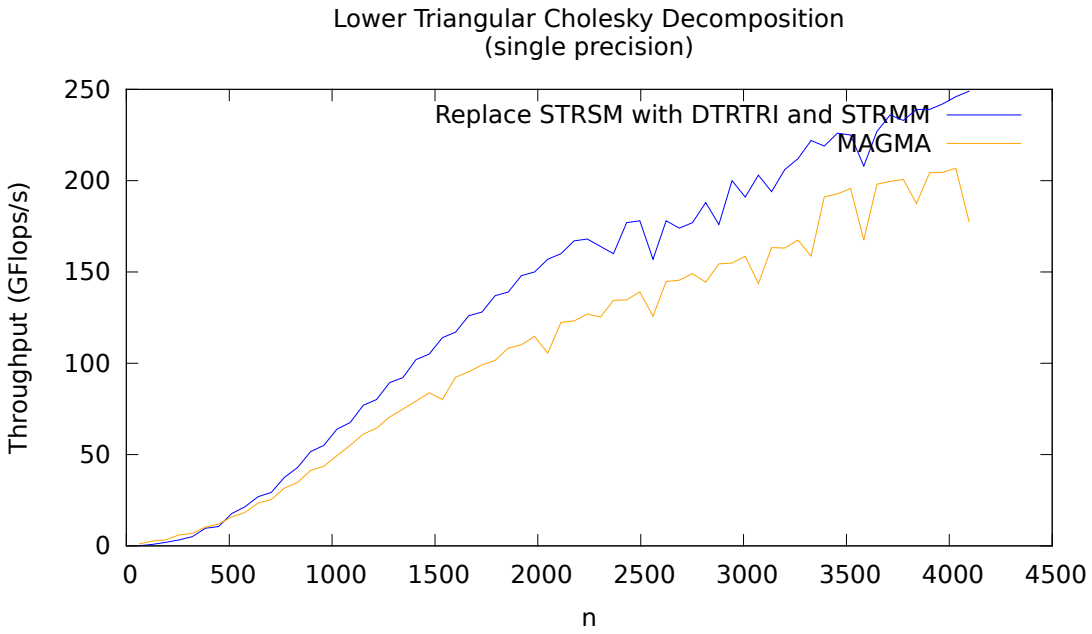


Figure 4.16: Performance of the lower triangular hybrid Cholesky decomposition compared to the MAGMA library in single precision. Our fastest lower triangular Cholesky decomposition in single precision provides an almost uniform increase in performance over the MAGMA implementation.

Chapter 5

Hybrid Cholesky Inverse

5.1 Introduction

We now demonstrate the general applicability of the novel approaches developed in the context of a hybrid Cholesky decomposition by investigating how they may be applied to the Cholesky inverse algorithm. The Cholesky inverse algorithm calculates the inverse of a matrix from the Cholesky decomposition and is also widely used in computational statistics. The inverse of a covariance matrix is needed for example when evaluating the probability density function of a multivariate Normal distribution. If there exists a matrix B such that $AB = I$ then B is said to be the inverse of A and is denoted A^{-1} [49]. The inverse can only be calculated for square matrices and may not exist, in which case the matrix A is said to be singular. The inverse of a covariance matrix is needed to evaluate the probability density function of the multivariate normal distribution and also for Gaussian process prediction [130]. The inverse of a matrix may be computed from its triangular decomposition faster than from its original form. Computing both the Cholesky decomposition and inverse requires more operations than other methods of calculating the inverse, so it is only advantageous if the decomposition is also required.

The LAPACK subroutines `SPOTRI` and `DPOTRI` calculate the inverse of a positive definite matrix from its triangular decomposition in single and double precisions, respectively [15]. These subroutines are composed of two further subroutines which calculate the inverse of the Cholesky decomposition and then compute the product to get the inverse of the original matrix. Both subroutines are available in blocked and unblocked forms.

5.1.1 LAPACK Unblocked Algorithm

As with the unblocked subroutines for the Cholesky decomposition the unblocked routines for the Cholesky inverse are implemented as a series of BLAS 1 and 2 subroutine calls. The unblocked inverse (`STRTI2`) subroutine which calculates the upper or lower triangular inverse in single precision is implemented using the vector scalar multiplication (`SSCAL`) and triangular

matrix vector multiplication (STRMV) subroutines from levels 1 and 2 of the BLAS specification. The STRTI2 subroutine also checks the matrix for singularity by ensuring the diagonal elements are greater than zero. The SLAUU2 subroutine calculates the matrix product $A = UU^T$ or $A = L^T L$ to give the inverse of the original matrix. It is implemented using the dot product (SDOT), vector scalar multiplication (SSCAL) and matrix vector multiplication (SGEMV) BLAS 1 and 2 subroutines. In the SLAUU2 subroutine the element on the diagonal may be calculated independently of the column above or below. The unblocked algorithms from LAPACK are shown in Listings 5.1, 5.2, 5.3 and 5.4 using single precision BLAS.

Listing 5.1: Unblocked Upper Triangular Inverse

```
for (size_t j = 0; j < n; j++) {
  float ajj;
  if (diag == CBlasNonUnit) {
    if (A[j * lda + j] == zero) {
      *info = (long)j + 1;
      return;
    }
    A[j * lda + j] = 1.0f / A[j * lda + j];
    ajj = -A[j * lda + j];
  }
  else
    ajj = -1.0f;

  strmv(CBlasUpper, CBlasNoTrans, diag, j - 1, A, lda, &A[j * lda], 1);
  sscal(j - 1, ajj, &A[j * lda], 1);
}
```

Listing 5.2: Unblocked Lower Triangular Inverse

```
for (int j = 0; j < n; j++) {
  float ajj;
  if (diag == CBlasNonUnit) {
    if (A[j * lda + j] == zero) {
      *info = (long)j + 1;
      return;
    }
    A[j * lda + j] = 1.0f / A[j * lda + j];
    ajj = -A[j * lda + j];
  }
  else
    ajj = -1.0f;
```

```
  strmv(CBlasLower, CBlasNoTrans, diag, n - j, &A[(j + 1) * lda + j + 1],
      lda, &A[j * lda + j + 1], 1);
  sscal(n - j, ajj, &A[j * lda + j + 1], 1);
}
```

Listing 5.3: Unblocked Upper Triangular Product

```
for (size_t j = 0; j < n; j++) {
  float aii = A[i * lda + i];
  if (i < n - 1) {
    A[i * lda + i] = sdot(n - i + 1, &A[(i * lda + i], lda, &A[i * lda + i],
         lda );
    sgemv(CBlasNoTrans, i - 1, n - i, 1.0f, &A[(i + 1) * lda], lda, &A[(i +
        1) * lda + i], lda, aii, &A[i * lda], 1);
  }
  else
    sscal(i, aii, &A[i * lda], 1);
}
```

Listing 5.4: Unblocked Lower Triangular Product

```
for (int j = 0; j < n; j++) {
  float aii = A[i * lda + i];
  if (i < n - 1) {
    A[i * lda + i] = sdot(n - i + 1, &A[i * lda + i], 1, &A[i * lda + i], 1)
        ;
    sgemv(CBlasTrans, n - i, i - 1, 1.0f, &A[i + 1], lda, &A[i * lda + i +
        1], 1, aii, &A[i], lda);
  }
  else
    sscal(i, aii, &A[i], lda);
}
```

5.1.2 LAPACK Blocked Algorithm

The blocked Cholesky inverse algorithm calls blocked versions of the triangular inverse and product subroutines. These involve splitting the matrix into blocks as illustrated in Figures 5.3, 5.4, 5.1 and 5.2. The blocks are then updated according to Algorithms 5, 4, 6 and 7, respectively. As with the blocked Cholesky decomposition all BLAS routines called as part of the blocked Cholesky inverse algorithms are from level 3 of the BLAS specification.

The triangular inverse (`STRTRI`) blocked algorithm requires the triangular matrix multi-

Algorithm 4 The upper triangular blockwise matrix inverse algorithm expressed as a sequence of linear algebra operations on the submatrices defined in Figure 5.3. The call to `STRTI2` performs the matrix inverse of a smaller matrix B while the rest of the operations are performed using level 3 BLAS.

for j = 0,nb,...,n **do**

$B = A \times B$

$B = -B \times C^{-1}$

$STRTI2(\text{“}Upper\text{”}, B)$

end for

Algorithm 5 The lower triangular blockwise matrix inverse algorithm expressed as a sequence of linear algebra operations on the submatrices defined in Figure 5.4. The call to `STRTI2` performs the matrix inverse of a smaller matrix B while the rest of the operations are performed using level 3 BLAS.

for j = 0,nb,...,n **do**

$B = A \times B$

$B = -B \times C^{-1}$

$STRTI2(\text{“}Lower\text{”}, B)$

end for

Algorithm 6 The upper triangular blockwise matrix product algorithm expressed as a sequence of linear algebra operations on the submatrices defined in Figure 5.1. The call to `SLAUU2` performs the product of a smaller matrix B while the rest of the operations are performed using level 3 BLAS.

for j = 0,nb,...,n **do**

$A = A \times B^T$

$SLAUU2(\text{“}Upper\text{”}, B)$

$A = A + C \times D^T$

$A = A + D \times D^T$

end for

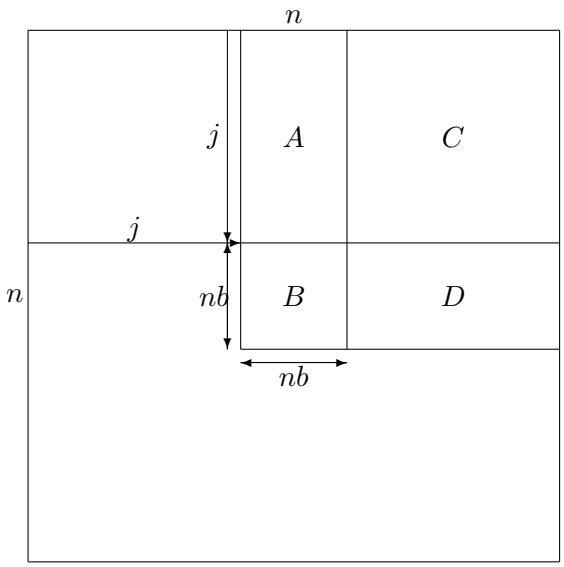


Figure 5.1: When performing the blocked matrix product algorithm for upper triangular matrices the matrix is divided into the submatrices shown. By dividing the matrix into submatrices the size of the problem is reduced to a product of matrix B. The rest of the matrix can then be updated using highly parallel BLAS 3 subroutines.

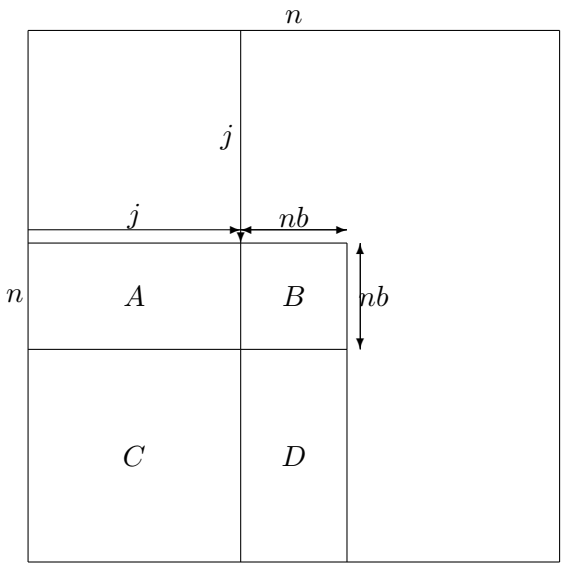


Figure 5.2: When performing the blocked matrix product algorithm for lower triangular matrices the matrix is divided into the submatrices shown. By dividing the matrix into submatrices the size of the problem is reduced to a product of matrix B. The rest of the matrix can then be updated using highly parallel BLAS 3 subroutines.

plication (`STRMM`) and triangular matrix solve (`STRSM`) subroutines from the BLAS as well as the unblocked triangular inverse subroutine (`STRTI2`). There are dependencies between each of the subroutines called in this algorithm so they must be executed in the specific order listed

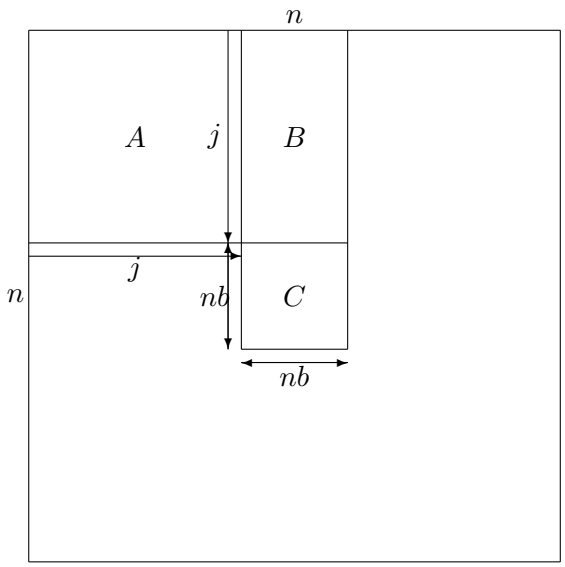


Figure 5.3: When performing the blocked matrix product algorithm for upper triangular matrices the matrix is divided into the submatrices shown. By dividing the matrix into submatrices the size of the problem is reduced to a product of matrix B. The rest of the matrix can then be updated using highly parallel BLAS 3 subroutines.

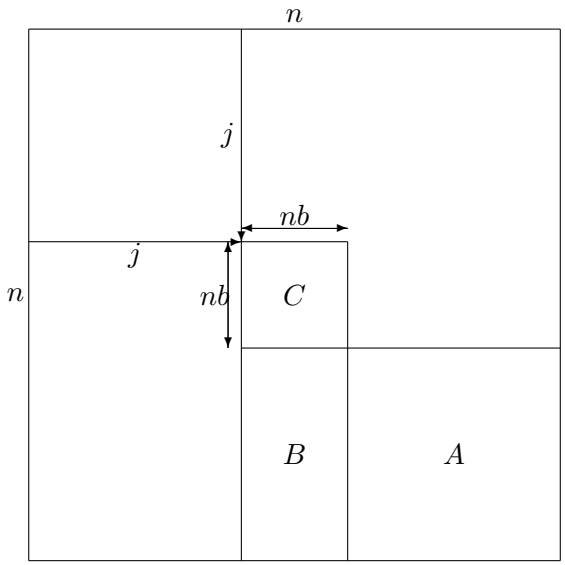


Figure 5.4: When performing the blocked matrix product algorithm for lower triangular matrices the matrix is divided into the submatrices shown. By dividing the matrix into submatrices the size of the problem is reduced to a product of matrix B. The rest of the matrix can then be updated using highly parallel BLAS 3 subroutines.

in Algorithms 5 and 4. Most of the computation is performed in the `STRMM` subroutine so its performance is key to the performance of the blocked algorithm.

The matrix product (`SLAUUM`) blocked algorithm requires the triangular matrix multipli-

Algorithm 7 The lower triangular blockwise Cholesky decomposition algorithm expressed as a sequence of linear algebra operations on the submatrices defined in Figure 5.2. The call to `SLAUU2` performs the product of a smaller matrix B while the rest of the operations are performed using level 3 BLAS.

for j = 0,nb,...,n **do**

$A = B^T \times A$

$SLAUU2(\text{“}Lower'', B)$

$A = A + D^T \times C$

$B = B + D^T \times D$

end for

cation (`STRMM`), matrix multiplication (`SGEMM`) and rank-K update (`SSYRK`) subroutines from the BLAS as well as the unblocked matrix product subroutine (`SLAUU2`). The calculation of the diagonal block is independent of the matrix multiplication and rank-K update subroutines and so may be overlapped with them. Most of the computation occurs within the `SGEMM` subroutine.

5.1.3 Hybrid Blocked Algorithm

It is possible to implement hybrid versions of the Cholesky inverse algorithms in the same manner as the Cholesky decomposition. For both the hybrid triangular inverse and triangular product algorithms the diagonal block is transferred into host memory to be processed by the CPU while the GPU overlaps this with other computation, however due to data dependencies it is not possible to combine the two operations into a single loop.

In the hybrid triangular inverse algorithm the diagonal block is transferred into host memory and inverted by the CPU while the GPU overlaps this with the triangular matrix multiply and triangular matrix solve routines to update the rest of the block column. The GPU must finish processing before the diagonal block is transferred back however as the original value is used in the triangular matrix solve. This limits the amount of processing that can be carried out in parallel across the CPU and GPU.

In the hybrid matrix product algorithm the diagonal block is transferred into host memory to be calculated by the CPU, while the GPU overlaps this with the triangular and regular matrix multiply operations. The rank-K update depends on the result of the diagonal block.

5.2 Improvements on the State of the Art

In this section we present novel approaches that may be employed to calculate inverse matrices by utilising the Cholesky decomposition routines developed in the last chapter for hybrid architectures.

The hybrid Cholesky inverse algorithm is implemented as two separate hybrid algorithms to calculate the inverse of a triangular matrix and to compute the product of this to get the inverse of the original matrix. This allows a different block size to be chosen for each algorithm. In addition to the BLAS functions required for the hybrid Cholesky decomposition both the hybrid triangular inverse and hybrid matrix product algorithms require a triangular matrix multiply kernel to be implemented for the GPU. For performance reasons this is implemented out of place, however these algorithms require an in place implementation, which is achieved by using an additional copy operation but this is slow. The triangular matrix multiply kernel can also be used to replace the slow triangular matrix solve kernel used in the triangular inverse algorithm in a similar manner to that described in Section 4.3.6. This allows the out of place implementation to be used when combined with an out of place matrix multiply in the matrix product algorithm.

5.2.1 GPU Triangular Matrix Multiply

The triangular matrix multiply operation performs $X = \alpha op(A)B$ or $X = \alpha Bop(A)$, where B and X are $m \times n$ matrices. In the first case A is an $m \times m$ matrix and in the second case A is $n \times n$. As with the triangular matrix solve there are 16 cases depending on whether A multiplies B from the left or right, $op(A) = A$ or $op(A) = A^T$, A is upper or lower triangular and whether A has unit or non-unit entries along the diagonal.

The triangular matrix multiply operation also has similar dependencies between elements as the triangular matrix solve operation. However each element of the output matrix X depends on elements of B rather than elements of X that have already been calculated. The in place BLAS reference implementation overwrites B with X and uses reverse loops in places where the triangular matrix solve would use forward loops and vice versa in order to satisfy the dependencies.

Our novel approach to this is to consider an out-of place implementation with X stored separately from B. As each element of X is independent of other elements of X an out of place implementation would remove the need to update elements of B in a particular order. This is better suited to a GPU implementation which benefits from many independent calculations being carried out by separate threads. The implementation of an out of place triangular matrix multiply closely follows the conversion of the regular matrix multiply to a rank-K update. A regular matrix multiply operation of the form $C = \alpha AB + \beta C$ is converted to a rank-K update by either setting B to A^T for $op(A) = A$, or A to A^T and B to A for $op(A) = A^T$, and only writing to the upper or lower triangle of C. When converting a regular matrix multiply to a triangular matrix multiply C becomes X. When A multiplies B from the right A and B are swapped. Only the upper or lower triangle of A is read in either case while all of C (or X)

	mb	nb	kb	bx	by	Threads	Registers	Shared memory	Blocks per SM
$upper(A)X = \alpha B$	64	16	16	16	4	64	31	1088	8
$lower(A)X = \alpha B$	64	16	16	16	4	64	31	1088	8
$upper(A^T)X = \alpha B$	32	32	8	8	8	64	32	2240	6
$lower(A^T)X = \alpha B$	32	32	8	8	8	64	32	2240	6
$Xupper(A) = \alpha B$	64	16	16	16	4	64	32	1088	8
$Xlower(A) = \alpha B$	64	16	16	16	4	64	32	1088	8
$Xupper(A^T) = \alpha B$	64	16	16	16	4	64	32	1088	8
$Xlower(A^T) = \alpha B$	64	16	16	16	4	64	32	1088	8

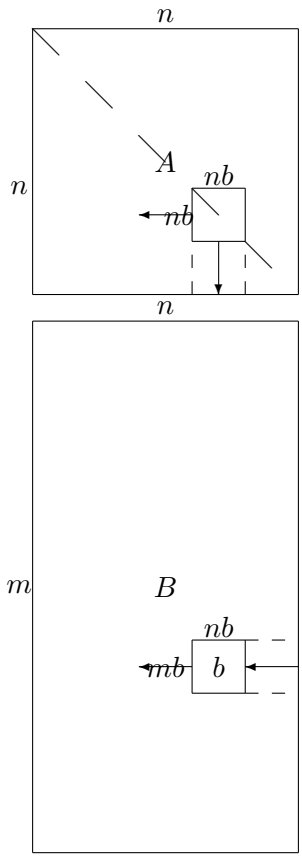
Table 5.1: The block sizes, shared memory and register usage for the GPU triangular solve algorithm in single precision. Although the block diagram for the two operations is similar the independence between blocks of the triangular matrix multiply means we can use the block sizes from the matrix multiply kernel in Table 4.2 instead of those for the triangular solve in Table 4.4. This gives the triangular matrix multiply kernels similar performance to the general matrix multiply kernels.

is written to. Figures 5.5a and 5.5b show how the matrices are split into blocks when using a matrix multiplication kernel to perform triangular matrix multiplication. Converting matrix multiply GPU kernels in this way gives triangular matrix multiplication performance similar to matrix multiply rather than triangular solve, even though the triangular operations are similar in notation. The thread block sizes and resource usage for the kernel is also similar to matrix multiplication and is shown for the single precision kernel in Table 5.1.

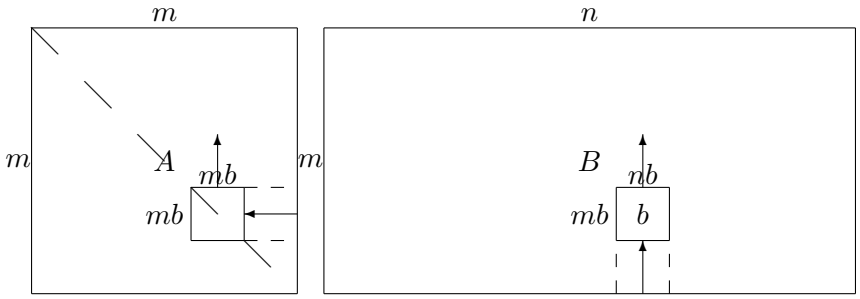
5.2.2 Unblocked Triangular Inverse on the CPU

Here we consider application of the loop reordering optimisations from Section 4.3.1, which enable the compiler to automatically vectorise our code, to the unblocked algorithm calculating the inverse from the Cholesky decomposition.

The unblocked triangular inverse function for the CPU calculates the inverse of the matrix column by column using subroutines from levels 1 and 2 of the BLAS specification. The diagonal element is inverted first in each column independently of the rest of the matrix. The rest of the column above or below the diagonal is then updated using triangular matrix vector multiplication with the submatrix that has already been inverted. This requires the loop for the lower triangular case to be reversed so that it is updated from the bottom right to the top left. The



(a) Blocked triangular matrix multiply for the right/lower triangular/transpose case. This form of the triangular matrix multiply is used in the lower triangular Cholesky decomposition. The block marked b starts on the right of B and is held in registers. It is updated by reading blocks from the current row of B and matching row in A. Only blocks in B to the right of b that have not been calculated yet are used to update the current b. This is the opposite case from the triangular matrix solve which allows the blocks to be processed independently and out-of-place.



(b) Blocked triangular matrix multiply for the left/upper triangular/transpose case. This form of the triangular matrix multiply is used in the upper triangular Cholesky decomposition. The block marked b starts at the bottom of B and is held in registers. It is updated by reading blocks from the current column of B and matching column in A. Only blocks in B that have not been calculated yet are used to update the current b. This is the opposite case from the triangular matrix solve which allows the blocks to be processed independently and out-of-place.

Figure 5.5: Blocked triangular matrix multiply

column vector is then scaled by the diagonal element. Since the calculation of the diagonal element is independent of the rest of the matrix it may come before or after the multiplication but there are no advantages nor disadvantages to reordering the operations. As with the unblocked Cholesky decomposition function for the CPU the calls to the BLAS are expanded inline. Unlike the unblocked Cholesky decomposition however, the loops within the functions are already arranged to get maximum performance from SIMD instructions and there are no opportunities to merge loops over identical ranges. The code for the unblocked triangular inverse on the CPU is shown in Listing 5.5.

Listing 5.5: Optimised unblocked triangular inverse algorithm.

```
if (uplo == CBlasUpper) {
  for (size_t j = 0; j < n; j++) {
    register float ajj;
    if (diag == CBlasNonUnit) {
      if (A[j * lda + j] == 0.0f) {
        *info = (long)j + 1;
        return;
      }
      A[j * lda + j] = 1.0f / A[j * lda + j];
      ajj = -A[j * lda + j];
    }
    else
      ajj = -1.0f;

    for (size_t k = 0; k < j; k++) {
      register float temp = A[j * lda + k];
      if (diag == CBlasNonUnit) A[j * lda + k] *= A[k * lda + k];
      for (size_t i = 0; i < k; i++)
        A[j * lda + i] += temp * A[k * lda + i];
    }
    for (size_t i = 0; i < j; i++)
      A[j * lda + i] *= ajj;
  }
}
else {
  size_t j = n - 1;
  do {
    register float ajj;
    if (diag == CBlasNonUnit) {
      if (A[j * lda + j] == 0.0f) {
```

```
        *info = (long)j + 1;
        return;
      }
      A[j * lda + j] = 1.0f / A[j * lda + j];
      ajj = -A[j * lda + j];
    }
    else
      ajj = -1.0f;

    for (size_t i = n - 1; i > j; i--) {
      register float temp = A[j * lda + i];
      if (diag == CBlasNonUnit) A[j * lda + i] *= A[i * lda + i];
      for (size_t k = i + 1; k < n; k++)
        A[j * lda + k] += temp * A[i * lda + k];
    }
    for (size_t i = j + 1; i < n; i++)
      A[j * lda + i] *= ajj;
  } while (j-- > 0);
}
```

5.2.3 Unblocked Triangular Inverse on the GPU

Having created an optimised unblocked inverse algorithm using vector instructions for the CPU, and expanding all the BLAS 1 and 2 calls inline, we can now extend it to use GPU vectorisation. The unblocked triangular inverse is calculated column by column from left to right. This is performed by a single one-dimensional thread block running on one GPU multiprocessor as for the unblocked Cholesky decomposition. The size of the thread block is the same as for the unblocked Cholesky decomposition kernel. As the matrix is triangular and being calculated column by column left to right the number of active threads in the block increases at each iteration in the upper triangular case and decreases at each iteration in the lower triangular case.

Elements of the matrix that have been calculated are shared among threads of the block to calculate subsequent elements so are stored in shared memory. Triangular packed storage mode is used to store the matrix in shared memory so that the number of threads can be kept as high as possible. The shared memory required is the same as for the unblocked Cholesky decomposition kernels. The kernel is composed of three subsequent outer loops which read the matrix into shared memory, compute the inverse of the upper or lower triangle and then write it out to global memory.

5.2.4 Unblocked Triangular Product on the CPU

Our loop reordering and vector optimisations from Section 4.3.1 are now applied to the triangular product operation for the CPU. The triangular product operation forms $A = UU^T$ or $A = L^T L$ where U is the upper triangle of A and L is the lower triangle of A. In the upper triangular case each element of $A_{i,j}$ with $i <= j$ is calculated as $A_{i,j}A_{j,j} + \sum_{k=j+1}^{n} A_{i,k}A_{j,k}$. In the lower triangular case each element of $A_{i,j}$ with $i >= j$ is calculated as $A_{i,j}A_{i,i} + \sum_{k=i+1}^{n} A_{k,i}A_{k,j}$. In both cases there are no dependencies between elements and the sums are computed as three nested loops similar to matrix multiplication. In the upper triangular case the loops are ordered j, k, i to perform multiple updates down each column using SSE. In the lower triangular case the loops are ordered j, i, k in order to perform reduction down each column using SSE. The code for the unblocked triangular product on the CPU is shown in Listing 5.6.

Listing 5.6: Optimised unblocked triangular product algorithm.

```
if (uplo == CBlasUpper) {
  for (size_t j = 0; j < n; j++) {
    register float ajj = A[j * lda + j];
    for (size_t i = 0; i <= j; i++)
      A[j * lda + i] *= ajj;

    for (size_t k = j + 1; k < n; k++) {
      register float temp = A[k * lda + j];
      for (size_t i = 0; i <= j; i++)
        A[j * lda + i] += temp * A[k * lda + i];
    }
  }
}
else {
  for (size_t j = 0; j < n; j++) {
    for (size_t i = j; i < n; i++) {
      A[j * lda + i] *= A[i * lda + i];

      for (size_t k = i + 1; k < n; k++)
        A[j * lda + i] += A[i * lda + k] * A[j * lda + k];
    }
  }
}
```

5.2.5 Unblocked Triangular Product on the GPU

Our vectorised CPU implementation of the unblocked triangular matrix product algorithm is now extended to an unblocked GPU kernel. Due to dependencies between elements of the output matrix the unblocked triangular product needs to be calculated column by column in the upper triangular case and row by row in the lower triangular case. This is achieved using a single one-dimensional thread block on the GPU as for the unblocked Cholesky decomposition kernel. The number of threads in the block is the same as for the unblocked Cholesky decomposition kernels. The number of active threads in the block increases in both cases as the kernel processes each column or row from the top left to the bottom right. The current column or row is stored in registers and updated by reading the rest of the matrix.

The entire matrix is stored in shared memory using triangular packed storage in order to have the highest number of threads per block without overloading shared memory. This is also the same as for the Cholesky decomposition so the shared memory usage is similar. The current row or column is updated by reading elements from the current row or column broadcast to all threads from shared memory and multiplying them by elements from the rest of the matrix. As the values are being calculated column by column in the upper triangular case and are not reused once calculated they may be written straight to global memory. In the lower triangular case the matrix is being calculated row by row so the values must be stored in shared memory to write them to global memory column by column later.

5.2.6 Alternatives to GPU Triangular Solve

As in the hybrid Cholesky decomposition the hybrid triangular inverse algorithm depends on the triangular solve operation being executed on the GPU. This is slow to run on a GPU so we now consider ways we may replace it with a faster triangular matrix multiplication.

The original algorithm to form the triangular inverse involves updating the current column B using the matrix A that has already been calculated to the upper left or lower right. This uses the triangular matrix multiplication operation which is implemented out-of-place on the GPU to reduce the dependencies between elements of the output matrix. In order to convert this to an in-place implementation an additional copy is needed which reduces performance.

The triangular solve operation is used to update B using C. The triangular solve forms the inverse of C temporarily on the GPU and uses it to update B. This is immediately followed by the unblocked inverse routine which forms the inverse of the C in-place. This means that the inverse of C is computed twice: first by the triangular solve on the GPU, then again by the unblocked inverse routine on the CPU. By moving the unblocked inverse routine before the triangular solve the latter can be replaced with a triangular multiplication operation. This

calculates the inverse of the diagonal block only once and, as both triangular multiplication operations are now out-of-place, this also removes the need for the additional copy operation to restore B to its correct position. The updated algorithm is shown in Listing 5.7 with an out-of-place triangular matrix multiplication routine called `strmm2`.

Listing 5.7: Triangular Inverse using matrix multiplication in place of matrix solve.

```
if (uplo == CBlasUpper) {
  for (size_t j = 0; j < n; j += nb) {
    const size_t jb = min(nb, n - j);
    // Triangular matrix multiplication storing the result in X
    strmm2(CBlasLeft, CBlasUpper, CBlasNoTrans, diag, j, jb, one, A, lda,
        &A[j * lda], lda, X, ldx);
    strti2(CBlasUpper, diag, jb, &A[j * lda + j], lda, info);
    if (*info != 0) {
      *info += (long)j;
      return;
    }
    // Triangular matrix multiplication restoring the result to A
    strmm2(CBlasRight, CBlasUpper, CBlasNoTrans, diag, j, jb, -one, &A[j *
         lda + j], lda, X, ldx, &A[j * lda], lda);
  }
}
else {
  size_t j = (n + nb - 1) & ~(nb - 1);
  do {
    j -= nb;
    const size_t jb = min(nb, n - j);
    strmm2(CBlasLeft, CBlasLower, CBlasNoTrans, diag, n - j - jb, jb, one,
         &A[(j + jb) * lda + j + jb], lda, &A[j * lda + j + jb], lda, X,
        ldx);
    strti2(CBlasLower, diag, jb, &A[j * lda + j], lda, info);
    if (*info != 0) {
      *info += (long)j;
      return;
    }
    strmm2(CBlasRight, CBlasLower, CBlasNoTrans, diag, n - j - jb, jb, -
        one, &A[j * lda + j], lda, X, ldx, &A[j * lda + j + jb], lda);
  } while (j > 0);
}
```

5.2.7 Improving Diagonal Block Transfer

Here we discuss the application of our optimisation from Section 4.3.2 to both the triangular inverse and matrix product algorithms that make up the Cholesky inverse operation. The original hybrid algorithm to compute the triangular inverse overlaps the update of the diagonal block, C, on the CPU with an in-place update of the rest of the column, B, using the GPU triangular matrix multiply. Applying the column copy optimisation from Section 4.3.2 directly would require the CPU to be one iteration ahead of the GPU to avoid overwriting the updates to B. After replacing the triangular matrix multiply with an out-of-place implementation the updates to the rest of the column are applied to a temporary matrix X, allowing the CPU to overwrite the rest of the column with its original contents. The columns around the diagonal blocks are therefore defined as in Figures 5.8 and 5.9.

In the upper triangular matrix product algorithm the upper part of the column is being updated by the GPU in parallel with the diagonal block by the CPU. For the same reasons as the lower triangular Cholesky decomposition this means that the entire column can only be copied into host memory as copying it back would overwrite updates made by the GPU. The lower triangular matrix product algorithm is updated row by row so the column can be copied from and to the device without overwriting GPU results.

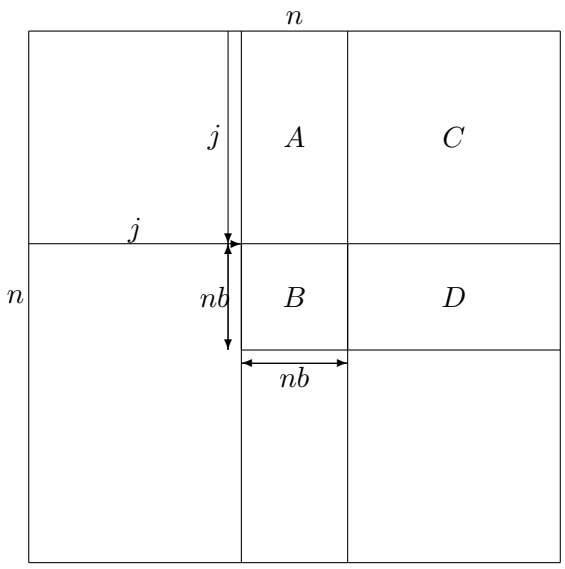


Figure 5.6: When performing the upper triangular matrix product algorithm the diagonal block B may be extended to a column in exactly the same way as for the Cholesky decomposition. This will provide a decrease in transfer time when the overhead of setting up each copy operation is large.

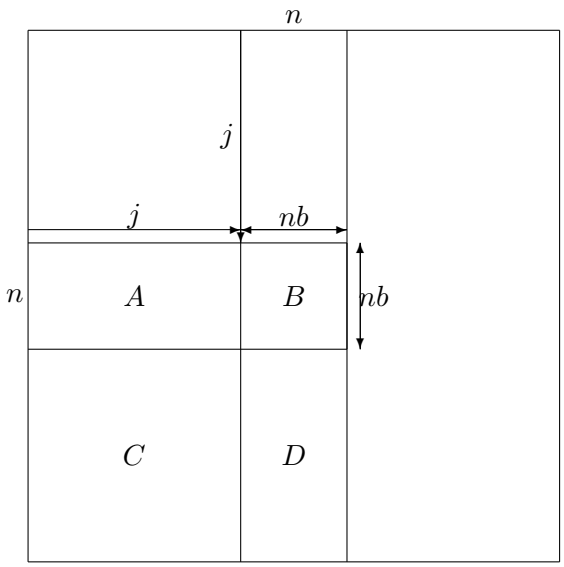


Figure 5.7: When performing the lower triangular matrix product algorithm the diagonal block B may be extended to a column in the same way as for the Cholesky decomposition. This will provide a decrease in transfer time when the overhead of setting up each copy operation is large.

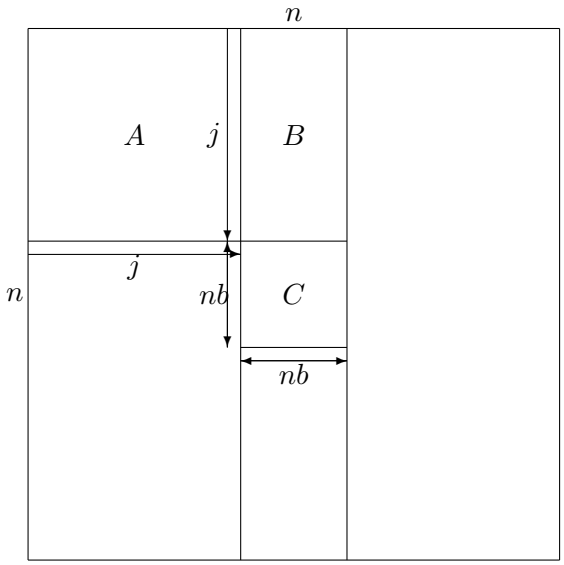


Figure 5.8: The upper triangular matrix inverse algorithm processes the matrix column by column. The diagonal block at the bottom of each column may be extended to the whole column as shown in the diagram.

5.2.8 Dynamic Block Sizing

Dynamically changing the block size used in both the hybrid triangular inverse and matrix product algorithms during execution has similar advantages to the hybrid Cholesky decomposition. In the hybrid triangular inverse algorithm the majority of the floating point operations are performed in the triangular matrix multiply executed by the GPU in parallel with the unblocked

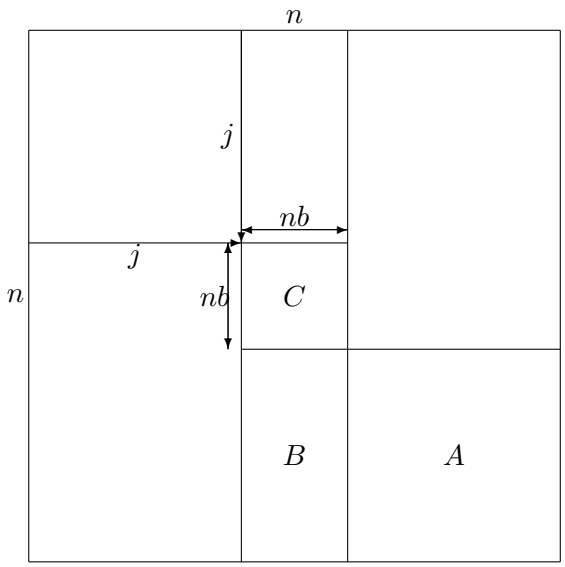


Figure 5.9: The lower triangular matrix inverse algorithm processes the matrix column by column. The diagonal block may be extended to a column as shown in the diagram to improve transfer time.

triangular inverse by the CPU. The size of submatrix used in the triangular matrix multiply increases with a fixed block size as the algorithm proceeds. By increasing the block size used at each iteration an increasing amount of work is carried out by the CPU to offset the increase in work carried out by the GPU.

In the hybrid triangular matrix product algorithm the majority of the floating point operations occur in the matrix multiply routine carried out by the GPU and overlapping the unblocked operation on the CPU. As with the Cholesky decomposition the size of the submatrices involved in the matrix multiply increase and then decrease as the algorithm proceeds. As with the Cholesky decomposition the block size is decreased and then increased to reduce the amount of work carried out by the matrix multiply in the middle of the algorithm to match that carried out by the unblocked triangular product.

5.2.9 Combining Unblocked kernels with Matrix Multiplication on the GPU

Using dynamic block sizes for the triangular inverse algorithm can leave the GPU waiting for data to be transferred back from the CPU before continuing. To work around this it is possible to execute the unblocked triangular inverse of the diagonal block on the GPU and have it execute in parallel with the triangular matrix multiplication kernel using the method explained in Section 3.3.4.

The kernel from Section 5.2.3 is combined with the out-of-place triangular matrix multiply kernel from Section 5.2.1 and used to overlap execution of the first triangular matrix multiply of

each iteration with the unblocked inversion. The first triangular matrix multiply is overlapped as the output from the inverse step is used in the second triangular multiply.

Similarly the unblocked matrix product kernel from Section 5.2.5 is combined with the matrix multiply kernel from Section 4.2.1 to overlap both operations on the GPU when performing the blocked hybrid matrix product.

5.3 Results

To analyse the effect our optimisations for the Cholesky decomposition have on the performance of the Cholesky inverse algorithm, we ran a similar benchmark on square matrices for values of N from 64 to 4096 in steps of 64 for both single and double precisions, and upper and lower triangular matrices. Again, each function was timed using the appropriate method from Section 3.5.2 and an average over 20 iterations was taken. Random input matrices with condition number 2 were generated with Algorithm 3 but did not have the Cholesky decomposition applied to them before computing the inverse. After performing error analysis to check that the result is correct the matrix was replaced with the identity matrix when benchmarking to avoid the algorithm exiting early due to the matrix becoming singular after repeated applications of the inverse algorithm.

Performance of our single precision hybrid Cholesky inverse implementations with the optimisations we have developed are shown in Figures 5.10 and 5.11. Performance reaches 205 GFlops/s for the lower triangular inverse and 275 GFlops/s for the upper triangular inverse. This is faster than the default hybrid implementation of the LAPACK algorithm which peaks at 166 GFlops/s for the lower triangular inverse and 219 GFlops/s for the upper triangular inverse. Double precision results for the same algoritms are shown in Figures 5.12 and 5.13. For double precision performance reaches 56.9 GFlops/s for the lower triangular inverse and 68.4 GFlops/s for the upper triangular inverse. As with the Cholesky decomposition the replacement of the triangular solve with the out of place triangular matrix multiply provides the largest improvement in performance and this is shown in the figures as “Use STRMM” and “Use DTRMM” for single and double precision. Similarly the diagonal block transfer, dynamic block size and combined kernel optimisations lower performance although for the Cholesky inverse routine they show an improvement over the default algorithm.

Figures 5.14 and 5.15 compare the performance of our best hybrid algorithms for the Cholesky inverse in single precision with the same algorithms from the MAGMA library. We choose to compare the performance of our algorithms with competing implementations from the MAGMA library as it contains many similarities to our work and is considered the state of

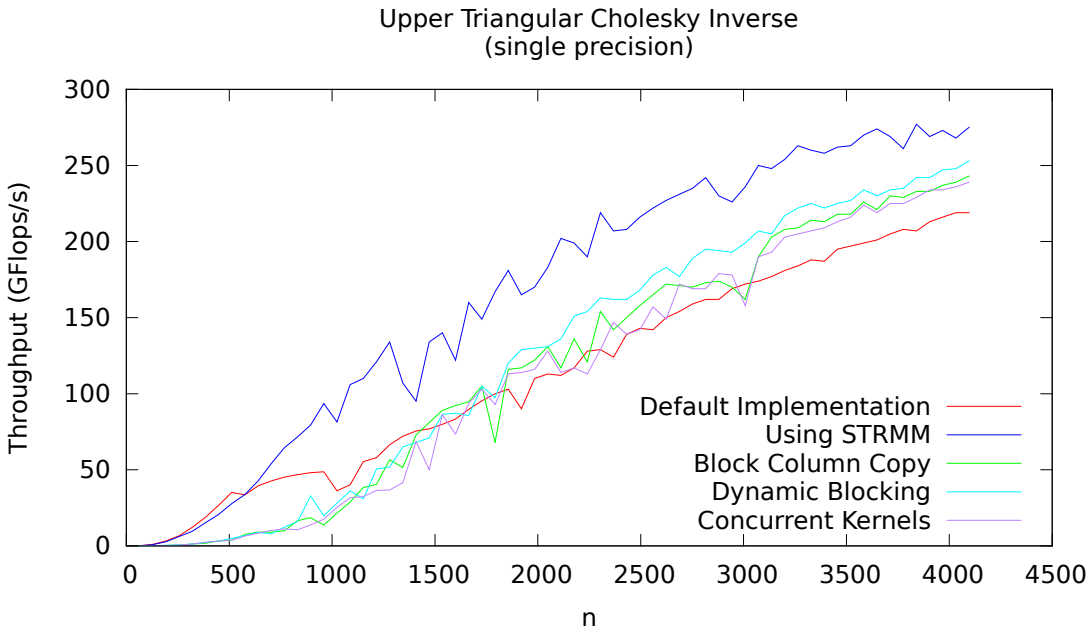


Figure 5.10: Performance of the upper triangular hybrid Cholesky inverse in single precision. Our optimisations are applied cumulatively from the default implementation of the hybrid LAPACK algorithm. A performance increase is provided by replacing the triangular solve with the out-of-place triangular matrix multiply while the block column copy and dynamic blocking decrease the performance due to the extra calculations introduced into the main loop.

the art in open source hybrid multicore CPU and GPU linear algebra libraries. Our proposed Cholesky inverse implementation in single precision which replaces the slow triangular solve STRSM with a faster hybrid CPU triangular inverse STRTRI and out of place GPU triangular matrix multiply STRMM outperforms the equivalent algorithms from the MAGMA library by an average of 150% for the lower triangular case and 275% for the upper triangular case. The maximum difference in performance is $2.7\times$ for the lower triangular inverse and $4.3\times$ for the upper triangular inverse. Our hybrid Cholesky inverse implementations significantly outperform those from the MAGMA library.

5.4 Discussion

Our contribution to the state of the art is a fast hybrid Cholesky inverse algorithm for a single GPU and multicore CPU that operates on a matrix in GPU memory and outperforms existing algorithms in single precision by up to $2.7\times$ to invert a lower triangular matrix and $4.3\times$ for an upper triangular matrix.

We reorder the operations in the triangular inverse routine to remove the slow triangular solve and in place triangular matrix multiply. These are replaced by two calls to a faster out

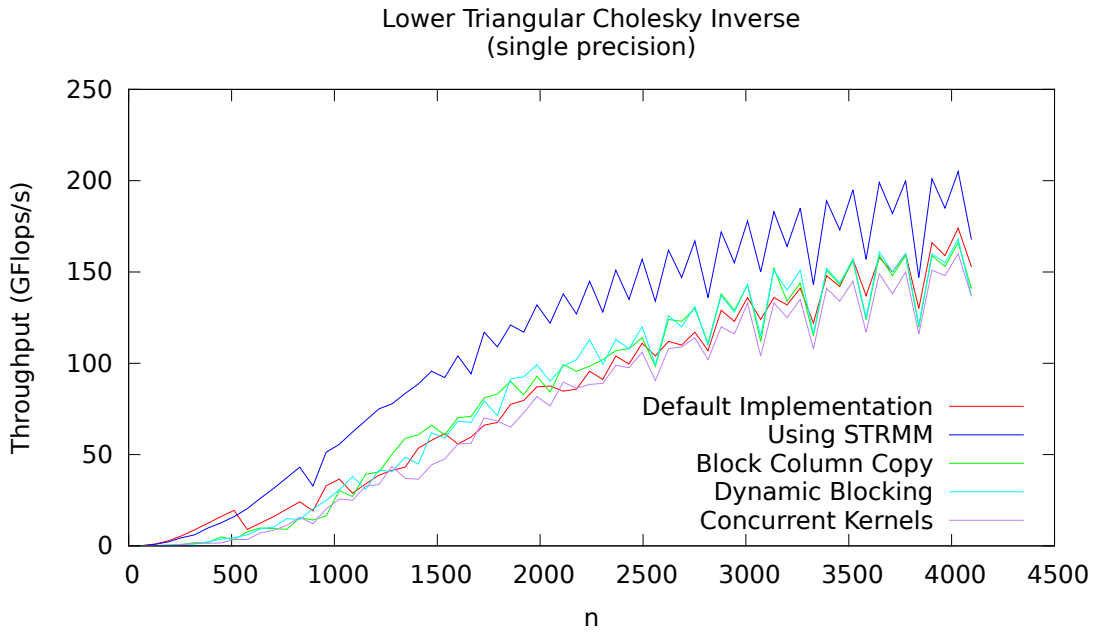


Figure 5.11: Performance of the lower triangular hybrid Cholesky inverse in single precision. Our optimisations are applied cumulatively from the default implementation of the hybrid LAPACK algorithm. A performance increase is provided by replacing the triangular solve with the out-of-place triangular matrix multiply while the block column copy and dynamic blocking decrease the performance due to the extra calculations introduced into the main loop.

of place triangular matrix multiply which also avoids calculating the inverse of the diagonal block twice. We also replace the in place triangular matrix multiply with our out of place implementation in the matrix product routine using an out of place general matrix multiply to copy the results back into the correct position in the matrix. This results in an increase in speed over the standard algorithms. The triangular solve operation has limited parallelism between columns in the left hand case used in both routines. The in-place triangular matrix multiply has similar data dependencies but an out of place implementation removes these allowing each element to be calculated independently resulting in large speedups on parallel architectures such as GPUs. This requires allocating a matrix to store the temporary out of place result. The cost of memory allocation depends on the version of the CUDA library used. In the current version, 5.0.35, it is fast enough such that allocating a temporary matrix on each invocation of the triangular matrix multiply does not degrade performance but in the future a persistent allocation mechanism should be implemented.

Our diagonal block column transfer optimisation reduced the performance of our algorithm in the both lower and upper triangular cases but still gives a performance increase over the

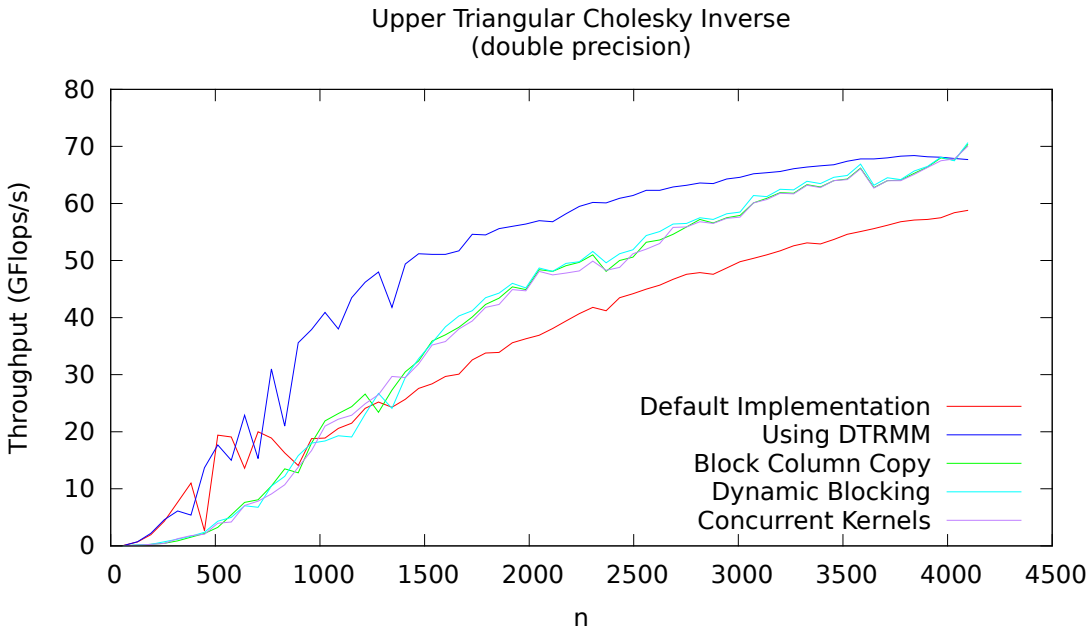


Figure 5.12: Performance of the upper triangular hybrid Cholesky inverse in double precision. Our optimisations are applied cumulatively with the same results as for single precision.

standard algorithm. The overhead in setting up memory transfers between host and device is highly dependent on the graphics driver version. As the block size changes on each iteration of the algorithm the time taken to transfer the diagonal block and block column needs to be recalculated on each iteration of the loop. This introduces extra instructions into the main loop of the algorithm for an optimisation that is not used on every iteration. Improvements in the CUDA hardware mean that the GPU can now perform 2D memory copies asynchronously with respect to CPU and GPU computation so this optimisation may only give a performance increase on older GPUs.

Dynamically changing the block size on every iteration of the algorithm reduces the performance of our inverse algorithm further. As well as introducing the need to reestimate the diagonal block and column transfer times on every iteration it also introduces extra calculations of its own to calculate the optimal block size. Calculating the optimal block size to effectively balance the workloads between the different architectures of the CPU and GPU is a hard problem and the solution needs to balance simplicity with effectiveness of load balancing. Currently the calculation is based on the difference in FLOP counts between the overlapping matrix multiply and unblocked Cholesky decomposition routines at each iteration. The GPU and CPU differ in their efficiency of executing these routines so using FLOP counts to measure differences in performance provides a poor estimate. A better solution would be to provide a tuning

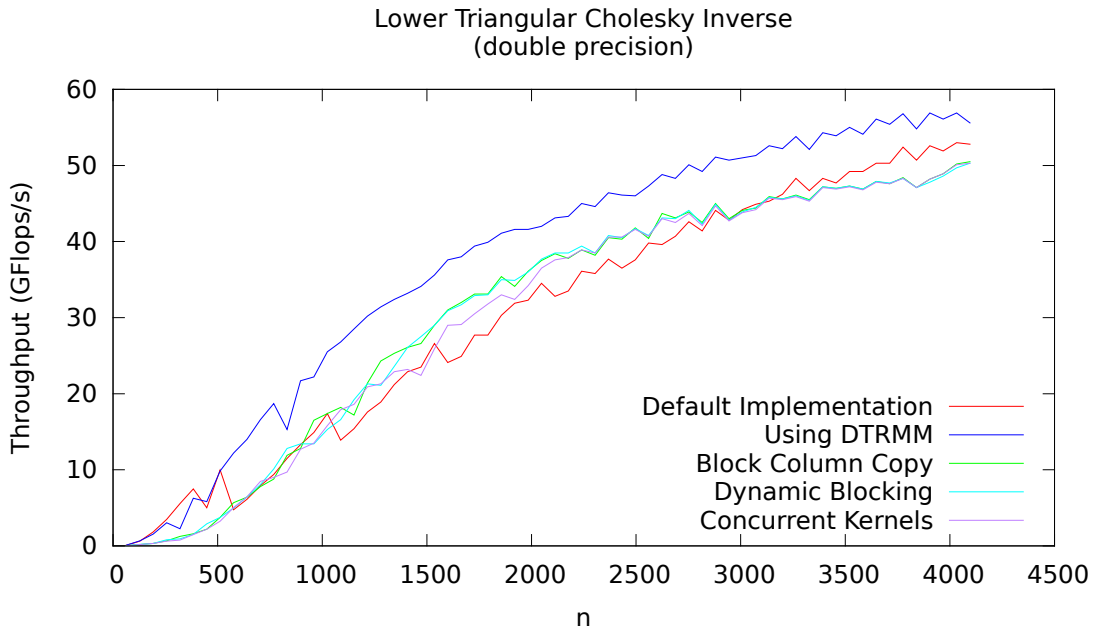


Figure 5.13: Performance of the lower triangular hybrid Cholesky inverse in double precision. Our optimisations are applied cumulatively with the same results as for single precision.

routine that benchmarks the overlapping operations for a range of block sizes at each iteration. This would result in the optimal execution configuration which could be reused for a particular matrix size.

The difference in performance between upper and lower triangular versions of the algorithm is due to the matrix multiply operation in the matrix product operation which forms part of the inverse. In the lower triangular inverse the matrix multiply is of the form $A^T B$ which is slower on GPUs than the AB^T form used in the upper triangular inverse. As with the Cholesky decomposition algorithms replacing the triangular solve with an out-of-place triangular matrix multiply provides the best performance increase as it replaces a slow algorithm with a lot of data dependencies with one that is highly parallel. The block column copy and dynamic blocking optimisations lower the performance of the inverse algorithms for the same reasons as the Cholesky decomposition. Also similar to the results for the hybrid Cholesky decomposition are the spikes in the graph due to the choice of block size.

Here we have shown the generality of our optimisations developed for the Cholesky decomposition by applying them to the inverse operation. Our optimisations give a larger improvement in performance when applied to the hybrid Cholesky inverse algorithm than when used in the hybrid Cholesky decomposition algorithm. The reasons for this are twofold. Firstly the Cholesky inverse involves two subroutines which can both have all our optimisations ap-

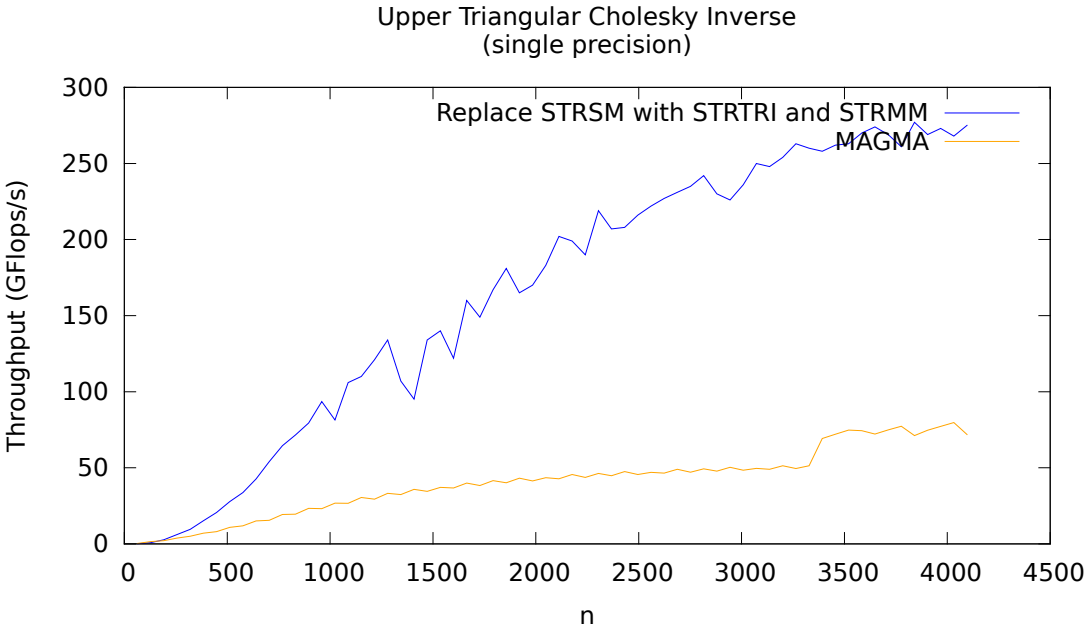


Figure 5.14: Performance of our upper triangular hybrid Cholesky inverse compared to the MAGMA library in single precision. Our fastest implementation which replaces the triangular solve with a triangular multiply outperforms the MAGMA library by an average of 275%.

plied to them. This presents a larger scope for increasing performance. Secondly, the default implementations of the triangular solve and in-place triangular matrix multiply are both used in the triangular inverse operation and can both be replaced by the faster out of place triangular matrix multiply. In the Cholesky decomposition algorithm only the triangular matrix solve is used.

We would recommend that our optimised hybrid Cholesky inverse algorithms should always be used as they are significantly faster than the current state of the art and require no extra effort to implement given the code we have written.

In the next chapter we consider a modification to our existing hybrid Cholesky decomposition algorithm that allows the determinant to be computed faster when the matrix is stored in GPU memory.

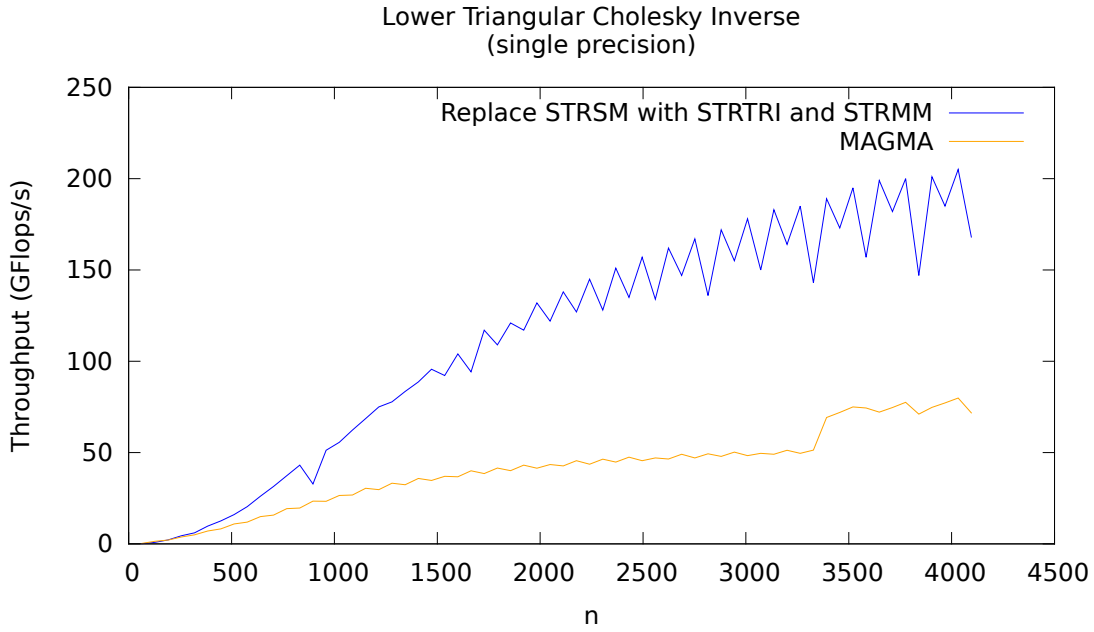


Figure 5.15: Performance of the lower triangular hybrid Cholesky inverse compared to the MAGMA library in single precision. Our fastest lower triangular Cholesky decomposition in single precision provides an average 150% increase in performance over the MAGMA implementation.

Chapter 6

Hybrid Cholesky Determinant

In this chapter we investigate how to efficiently obtain the log determinant of a matrix from its Cholesky decomposition. We show a simple modification to our hybrid Cholesky decomposition algorithm that improves the performance of the log determinant calculation on a GPU.

6.1 Introduction

The Cholesky decomposition also provides a fast means of calculating the determinant of a matrix. The determinant is used in the multivariate normal probability density function along with the inverse of the covariance matrix. The determinant of the covariance matrix may be calculated from its Cholesky decomposition using Equation 6.1.

$$det(A) = (\prod_{i=0}^{n} A_{i,i})^2 \tag{6.1}$$

Since the square of the product of the diagonal elements can overflow the numerical type used to calculate it, it is common to convert the product into a sum via logarithms and calculate the log of the determinant according to the formula shown in Equation 6.2.

$$log(det(A)) = 2 \times \sum_{i=0}^{n} log(A_{i,i}) \tag{6.2}$$

A product or sum over a vector that produces a scalar is known as a reduction operation. Reduction operations are inherently sequential as the accumulation of the elements usually relies on updating a running total. Several parallel reduction algorithms do exist however which compute several partial sums in parallel and then sequentially accumulate these into a final result. The number of operations required to reduce a vector is always less than the number of elements in the vector therefore the performance of all reduction algorithms is bound by memory bandwidth.

6.2 Methods

We now review existing parallel reduction methods on GPUs, then show how the Cholesky determinant method may be further improved by increasing the memory bandwidth.

Maximum bandwidth is obtained on a GPU when reading contiguous blocks of memory. Summing the diagonal elements of a matrix will result in poor performance as the elements are spaced far apart in memory. The diagonal of a matrix can be represented as a vector with stride one greater than the leading dimension of the matrix. As shown in the memory bandwidth benchmark in Section 3.1.3, the memory bandwidth decreases rapidly as the stride increases.

6.2.1 Parallel Reduction on the GPU

The CUDA SDK contains a sample implementation of parallel reduction using partial sums. It uses multiple threads to compute partial sums of elements in memory. Each thread block then accumulates the partial sums within the block using shared memory to create a total for the block. One thread from each block then writes this to a temporary vector in global memory. The last thread block to store its partial sum then computes the final sum from the partial sums stored in the temporary vector by repeating the first step. In total it takes $O(logN)$ operations to compute the sum of a vector of length N.

As there is no global synchronisation barrier implemented in CUDA a workaround is needed to ensure the final block does not start processing the temporary vector while the partial sums are still being calculated. As thread blocks are scheduled asynchronously, some may end up waiting on a synchronisation barrier while others are not running. If the number of blocks scheduled is greater than the number of multiprocessors available then this will cause a deadlock. It is possible to use multiple kernels with decreasing numbers of blocks to implement parallel reduction with each kernel launch acting as a global synchronisation point, however an alternative is to use CUDA's built in atomic operations on variables stored in global memory. These are available on nVidia GPUs with compute capability 1.1 and higher when using the CUDA Toolkit version 2.2 and later. A counter is stored in global memory visible to all blocks and initialised to zero. As each block finishes it atomically reads and increments the counter. The last block to update the counter will receive the size of the grid and will know that it is the last block to finish and can safely accumulate the partial sums from the temporary vector to produce the final result.

In order to implement the Cholesky log determinant algorithm, the reduction sample from the CUDA SDK version 5.0.35 was copied and modified to calculate the log of the elements as they are read from global memory. The partial sums are accumulated within the block as

normal, however the results from each block are doubled while being stored in the temporary vector. The multiplication is performed at this stage of the reduction as there may only be one block running, in which case the temporary vector will not be read again for the last block to apply the multiplication to the final result.

6.2.2 Improving Memory Bandwidth

In order to improve the memory bandwidth when calculating the determinant, the hybrid Cholesky decomposition is modified to store a copy of the diagonal in a contiguous vector. Only the unblocked routines on both the CPU and GPU calculate the diagonal elements, and therefore only these need to be modified. Since the same issues with reading non-contiguous elements occur on CPUs with SSE this also improves performance of the log determinant calculation using parallel reduction on the CPU. This paradigm is recognised as a candidate for automatic vectorisation by both GCC and ICC as explained in Section 2.1.4.

The diagonal vector is allocated in GPU memory and copied into host memory along with the diagonal block, which ensures that the diagonal vector is already in GPU memory when the reduction is started. We note that copying the diagonal vector from host memory into device memory to run a bandwidth bound algorithm would double the runtime unnecessarily, and it is for this reason that it would not make sense to implement a multi-GPU version of the log determinant reduction algorithm.

6.3 Results

Figures 6.1 and 6.2 show the performance of the Cholesky log determinant reduction algorithm in single and double precision when implemented on a GPU using contiguous and strided addition down the diagonal of the matrix. Since the algorithm is bandwidth rather than compute bound, we employ the bandwidth as a performance metric. The time benchmarks were run for N up to 15360 in single precision and 7680 in double precision in steps of 1024. These values were chosen as they are the maximum matrix dimensions that will fit in 1GB of graphics memory. Rather than compute the full Cholesky decomposition, only the diagonal was initialised using a uniform distribution on the interval $(0, 1)$, excluding zero and negative values to avoid errors with computing the logarithm of the elements. As with the other benchmarks a sum was also computed on the CPU using Kahan summation to improve the accuracy. The GPU and CPU results were compared as for the other benchmarks.

The performance of the algorithm reaches 1.8 GB/s in both single and double precision when the diagonal of the matrix is stored separately as a contiguous vector. When the diagonal is non-contiguous the bandwidth drops to a maximum of 0.456 GB/s in single precision and 0.565

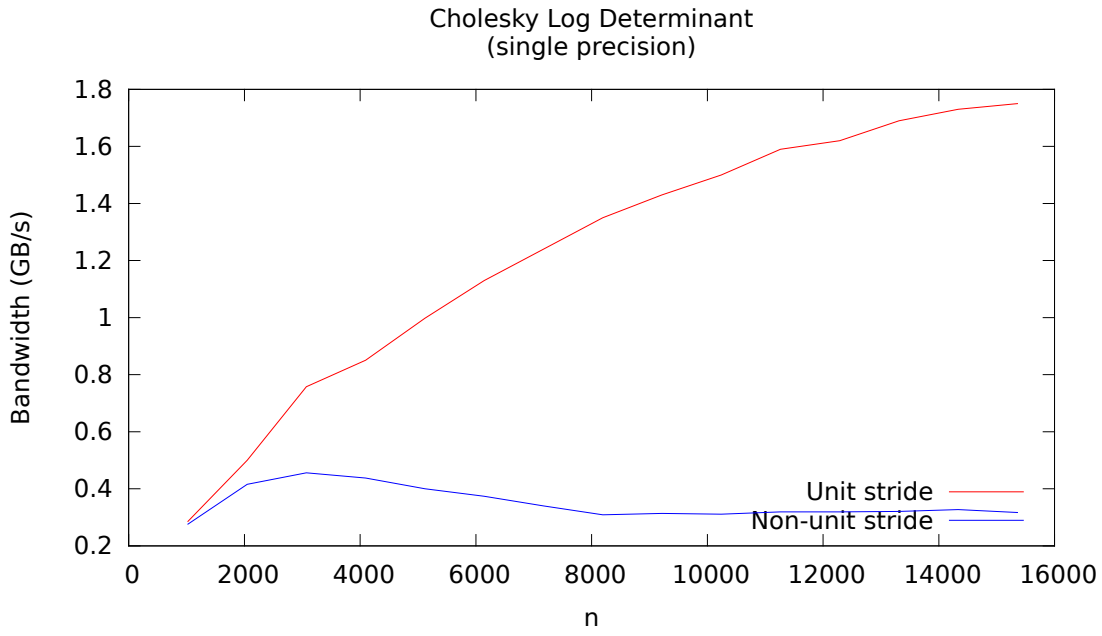


Figure 6.1: Performance of the GPU Cholesky log determinant algorithm in single precision. Our modifications to the hybrid Cholesky decomposition to store the diagonal separately as a contiguous vector result in a $4\times$ increase in bandwidth when calculating the determinant.

GB/s in double precision. Modifying the hybrid Cholesky decomposition to store the diagonal vector separately therefore gives around $3.5\times$ to $4\times$ increase in performance. Assuming a larger matrix would fit in GPU memory, our log determinant algorithm runs at around 30 GB/s for $N = 1048576$ in both single and double precision. The maximum bandwidth attained is therefore still far from the theoretical maximum bandwidth of 159 GB/s on the GPU being benchmarked. The modifications we have made to the reduction example from the CUDA SDK have reduced its performance to 50% due to the extra logarithm instructions applied when reading elements from global memory.

6.4 Discussion

Since reduction is a bandwidth bound algorithm it is not suited to GPUs, which have more processing power than bandwidth. It is also not suited to SIMD computation due to the dependencies between elements. Reading diagonal elements of a matrix with a large stride results in drastically reduced bandwidth, which limits the performance of this algorithm.

We have taken the reduction example from the nVidia CUDA SDK and modified it to read elements from the diagonal of a matrix and compute the log determinant. The reduction example shows how to compute the sum of a contiguous vector. Modifying this to compute the log determinant from a Cholesky decomposition entails calculating the log of the elements

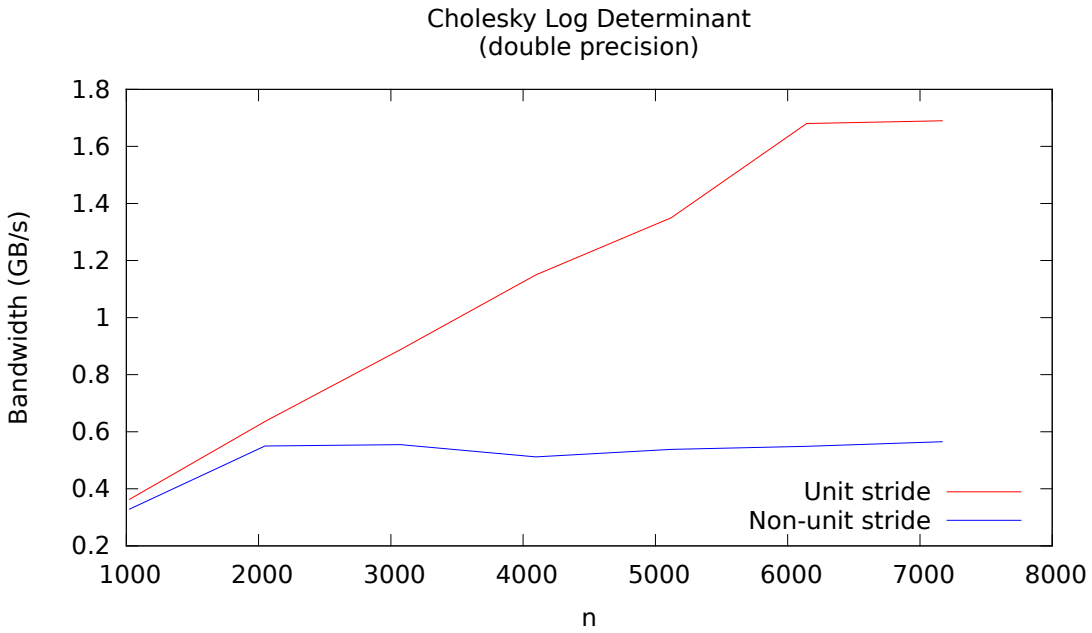


Figure 6.2: Performance of the GPU Cholesky log determinant algorithm in double precision. The performance for double precision is similar to that for single precision due to the algorithm being bandwidth bound.

when they are read from global memory, doubling each partial sum computed by the thread block and introducing a stride parameter to read non-contiguous vectors.

The faster implementation also involves adjusting the Cholesky decomposition to copy the diagonal elements into a contiguous vector. This requires extra bandwidth and memory for the Cholesky decomposition but it remains compute bound and provides a $3.5\times$ to $4\times$ increase in performance when calculating the determinant for matrix sizes that fit in GPU memory.

Chapter 7

Conclusions and Discussion

Linear algebra routines, such as the Cholesky decomposition and matrix inverse, are commonly the performance bottlenecks in many machine learning and computational statistics algorithms. For example, such routines are often very heavily used in Markov chain Monte Carlo, and other algorithms that make use of multivariate Gaussian distributions. MCMC in particular has an inherently sequential structure and so parallelisation of the underlying linear algebra routines offers an approach to further improve its performance. The dependencies between matrix elements in the Cholesky decomposition and inverse algorithms, however, make efficient parallel implementations non-trivial. Blocked matrix algorithms help overcome this limitation by dividing the matrix into sub-blocks and using a parallel BLAS library to achieve high performance.

There is a trend in modern day computing towards the use of mixed architectures, combining CPUs with GPU accelerators, which are well suited to algorithms with a large number of independent operations that can be executed with a high degree of parallelism. Such algorithms can be found in level 3 of the BLAS library and are vital for many algorithms found in the LAPACK library, such as the Cholesky decomposition. However, many numerical linear algebra libraries were originally designed for single core processors with memory hierarchies. There is therefore much value to be gained by examining these algorithms and investigating the possible improvements that might be made when computing in mixed architecture environments.

In this thesis we have developed novel low-level algorithms for Cholesky decomposition using hybrid and heterogenous architectures, such as systems with a multicore CPU and GPU accelerator. In addition, we have demonstrated their applicability to other blocked linear algebra algorithms by applying them to the Cholesky matrix inverse algorithm, as well as to a routine for efficiently calculating log matrix determinants.

Our optimistation which replaces the triangular solve step in the Cholesky decomposition with separate triangular inverse and triangular matrix multiply operations provides the biggest increase in performance. This is because it removes a step with limited parallelism which is

performed on a GPU with an additional hybrid step which may be performed in parallel by the CPU and GPU. The CPU is tasked with performing the triangular inverse which it is better suited to as it has a higher clock rate and lower number of cores than a GPU. The GPU performs an out of place triangular matrix multiply which has increased parallelism when compared to the triangular matrix solve. This shows that when considering parallel architectures it is more important to consider algorithms that have increased independence and if possible rewrite the algorithm to use more parallel operations. It also shows that on hybrid architectures it is important to fit the type of serial or parallel workload to the processor more able to execute it.

Traditionally blocked matrix algorithms have used a constant block size which is static throughout the algorithm and is related to the amount of data that can be stored in the processor cache. Static block sizes are suited to homogeneous multicore environments whereas a dynamic block size that changes to better balance the computational workload to the processor executing it is more suited to hybrid heterogeneous multicore and accelerator environments. The accelerator in this case need not be a GPU and this optimisation is equally applicable when using an FPGA or another computer. The state of the art MAGMA library uses a static block size.

By defining a column around the diagonal block the number of memory transactions needed to transfer the diagonal block to or from an accelerator can be reduced to a single transaction. This is particularly important for GPUs, where the cost of setting up each transaction is large when compared to the amount of data. This optimisation is generally applicable to hybrid blocked algorithms and we have demonstrated its utility by adapting it for the matrix inverse routines, which contain data dependencies between the blocks defined on a column. It can also be used in environments that store matrices in row major order by defining a row around the block. The MAGMA library uses an optimised scheduler to distribute matrix blocks to be processed on CPU and GPU pairs. It has been optimised to reduce the number of times each block needs to be transferred on and off each GPU but does not include our novel optimisation which reduces the time taken for each transfer.

We have also developed a general method of running multiple GPU kernels simultaneously on older GPUs that do not have this capability built in. This exploits the difference between traditional SIMD architectures, which GPUs are commonly thought to be restricted to, and the more modern SIMT architecture, which GPUs are actually capable of. As the MAGMA library is now concentrating on higher performance from later GPUs which include this capability in hardware it does not include this optimisation which enables similar performance from older GPUs.

Our hybrid Cholesky decomposition has slightly higher performance than the implemen-

tation from the MAGMA library, while our hybrid Cholesky inverse is up to $2.7\times$ faster for lower triangular matrices and $4.3\times$ faster for upper triangular matrices. The optimisations we have developed differ from those used in state of the art MAGMA library. In particular, the MAGMA library employs an optimised static scheduler [76] for blocked linear algebra algorithms and does not use any further optimisations of the type we have developed. This would suggest that incorporating an advanced scheduler algorithm into our code may improve the performance further.

The blocked Cholesky decomposition and inverse algorithms use delayed updates to offload parallelism to BLAS 3 subroutines. This increases parallelism but moves the dependencies into the unblocked routines and triangular matrix solve. nVidia provides a complete implementation of BLAS routines for its range of graphics cards and it is easy to implement a hybrid LAPACK library using a CPU implementation and calls to CUBLAS. Better performance is attainable, however, if we consider the algorithm as a whole and try to replace operations that are not suited to GPUs with equivalent ones that are. By considering the operations that the blocked algorithms use, we have been able to increased the parallelism available to the GPU and shift more of the dependencies into unblocked routines that may be carried out by the CPU.

Our hybrid GPU implementations may be used as drop in replacements for existing algorithms that require a Cholesky decomposition, inverse or determinant of a matrix stored in GPU memory. When the matrix is not stored in GPU memory the multiGPU versions of our algorithms should be preferred as these hide the cost of transferring the matrix into GPU memory. If the target system only has a single GPU then the multiGPU versions offer no benefit over an optimised CPU implementation unless the entire matrix will not fit in GPU memory.

We developed our optimisations in the context of a hybrid Cholesky decomposition and demonstrated their generality by applying them to the Cholesky inverse algorithm. In the future, we intend to research the performance improvements available by applying the optimisations we have developed to other blocked subroutines from the LAPACK library. Our optimisations involve analysing the algorithm as a whole and replacing less parallel operations with ones better suited to GPU computation. This contrasts with the approach taken by the MAGMA library, which implements the standard algorithms using an optimised scheduler to send matrix blocks to CPU and GPU pairs. The MAGMA approach has the advantage of being more generic and as a result may be applied to other classes of algorithms, including those that are not necessarily blocked linear algebra operations.

Currently our algorithms are implemented in single and double precision. The Cholesky decomposition and inverse can also be performed in complex and double complex precisions

and these are supported by CUDA-C. We intend to port our code to these extra precisions at some point in the future.

We note that the nVidia GeForce GTX 285 GPU used in our study, while state of the art at the start of this work, is now several years old and more recent GPUs include several architectural improvements [90]. An investigation of how our code performs on more recent hardware remains as future work. Newer classes of GPU implement hardware features that can be utilised by the algorithms presented in this thesis. GPUs can now schedule and run multiple kernels simultaneously in hardware rather than relying on our method to combine thread blocks from different kernels. This should improve performance since it allows the individual thread blocks executing different kernels to have different sizes and have different amounts of shared memory and registers allocated. While our method of running multiple kernels simultaneously on a single GPU would still work, it has been obsoleted by the scheduling hardware in newer GPUs which can perform the same task more efficiently. Replacing the triangular matrix solve operation with separate inverse and triangular matrix multiply operations can be applied equally as well to newer GPUs. One caveat is that older GPUs implement floating point division as two separate reciprocal and multiply operations. This allows the triangular solve to be replaced by the inverse and triangular matrix multiply with no additional loss of precision on older GPUs. On newer GPUs the effect on accuracy may be greater. New GPUs also have an additional level of memory hierarchy and larger amounts of memory at each level. Other architectural improvements include a wider SIMD width per multiprocessor, as well as more multiprocessors allowing increased levels of parallelism. Double precision performance has also been improved recently. Our other optimisations which optimise data transfer and use a dynamic block size are equally applicable to newer GPUs and indeed other distributed memory parallel architectures such as CPU clusters. Interestingly, the PCI Express bus has the capability of allowing individual devices to communicate with each other without intervention from the CPU. This allows GPUs to transfer data between one another independently and may be used to implement multi GPU algorithms that do not require the CPU to participate.

In the future, we will also analyse the performance of our implementations in the wider context of a complete MCMC simulation. For our hybrid GPU implementation to perform well, the entire simulation would have to be performed on the GPU and incorporate the GPU PRNG algorithms from Section 2.2.1. This would result in an entire MCMC simulation being carried out on the GPU. We are already taking steps towards this aim, to widen the reach of our algorithmic improvements and increase their accessibility. In particular, our implementations currently use CUDA to target nVidia graphics cards, yet our algorithms would run equally well

on other GPUs from rival vendors. Porting our codes to OpenCL, for example, would widen the use of our algorithms to other platforms. We also plan to contribute our algorithms to standard libraries for statistical simulation, such as the Shogun machine learning toolbox [114], to widen the reach of our optimisations and share our developments with end users. As mentioned earlier in this thesis there are already a number of libraries performing numberical linear algebra on GPUs, including CUBLAS from nVidia which implements the BLAS specification on nVidia GPUs. It is unclear whether nVidia would be interested in extending this work to implement the LAPACK specification. We have had no communication with nVidia about this or any other aspect of our research.

Bibliography

[1] AMD Accelerated Parallel Processing Math Libraries (APPML). http://developer.amd.com/tools/hc/appmathlibs/Pages/default.aspx.

[2] AMD Core Math Library. http://developer.amd.com/cpu/Libraries/acml/Pages/default.aspx.

[3] Amdahl's law. http://en.wikipedia.org/wiki/Amdahl%27s_law.

[4] Intel Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/.

[5] nVidia CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[6] AMD Close to Metal Technology Unleashes the Power of Stream Computing. 2006.

[7] IEEE Standard for Floating-Point Arithmetic. Technical report, August 2008.

[8] CULA Tools: GPU Accelerated Linear Algebra, 2010.

[9] *OpenCL 1.1 Specification*, September 2010.

[10] *Intel Core i7-900 Desktop Processor Extreme Edition Series*, June 2011.

[11] GNU C library reference manual. http://www.gnu.org/software/libc/manual/, 2012.

[12] M. Aaftab. OpenCL Specification Version 1.0. http://www.khronos.org/registry/cl/, December 2008.

[13] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1):012037+, August 2009.

[14] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):491–542, 1987.

[15] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, C. Bischof, D. Sorensen, and A10. LAPACK: A portable linear algebra library for high-performance computers. In *Supercomputing '90. Proceedings of*, pages 2–11, 1990.

[16] C. Andrieu, N. de Freitas, A. Doucet, and M. Jordan. An Introduction to MCMC for Machine Learning. *Machine Learning*, 50(1-2):5–43, January 2003.

[17] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, March 2009.

[18] M. Baboulin, J. Dongarra, and S. Tomov. Some Issues in Dense Linear Algebra for Multicore and Special Purpose Architectures. Technical report, University of Tennessee, Knoxville, 2008.

[19] S. Barrachina, M. Castillo, F. Igual, R. Mayo, and E. Quintana-Ortí. Solving Dense Linear Systems on Graphics Processors. In E. Luque, T. Margalef, and D. Benítez, editors, *Euro-Par 2008 Parallel Processing*, volume 5168 of *Lecture Notes in Computer Science*, chapter 79, pages 739–748. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[20] J. Besag, P. Green, D. Higdon, and K. Mengersen. Bayesian computation and stochastic systems. *Statistical Science*, pages 3–41, 1995.

[21] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic intra-register vectorization for the intel
textregistered architecture. *International Journal of Parallel Programming*, 30(2):65–98, 2002.

[22] S. Blackford and J. Dongarra. Installation Guide for LAPACK. Technical report, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, June 1999.

[23] G. E. P. Box and M. E. Muller. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, June 1958.

[24] A. E. Brockwell. Parallel Markov chain Monte Carlo Simulation by Pre-Fetching. *Journal of Computational and Graphical Statistics*, 15(1):246–261, March 2006.

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

[26] J. M. R. Byrd, S. A. Jarvis, and A. H. Bhalerao. Reducing the run-time of MCMC programs by multithreading on SMP architectures. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, April 2008.

[27] R. Calkin, R. Hempel, H. C. Hoppe, and P. Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, 1994.

[28] B. Chapman, G. Jost, and R. van van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, October 2007.

[29] B. M. Chapman and F. Massaioli. OpenMP. *Parallel Computing*, 31(1012):957–959, 2005. ¡ce:title¿OpenMP¡/ce:title¿.

[30] L. S. Chien. Hand Tuned SGEMM on GT200 GPU. Technical report, Department of Mathematics, Tsing Hua University, Taiwan, February 2010.

[31] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127. IEEE, October 1992.

[32] P. Coddington. Random Number Generators for Parallel Computers. In *The NHSE Review*, 1997.

[33] M. Cowles and B. P. Carlin. Markov Chain Monte Carlo Convergence Diagnostics: A Comparative Review. *Journal of the American Statistical Association*, 91(434):883–904, June 1996.

[34] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, January 1998.

[35] V. Demchik. Pseudo-random number generators for Monte Carlo simulations on Graphics Processing Units. March 2010.

[36] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. Technical report, CAPS Entreprise, 2007.

[37] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.

[38] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subroutines. *ACM Trans. Math. Soft*, 14(1):1–17, 1988.

[39] U. Drepper. What Every Programmer Should Know About Memory. 2007.

[40] P. Du, P. Luszczek, S. Tomov, and J. Dongarra. Mixed-Tool Performance Analysis on Hybrid Multicore Architectures.

[41] A. E. Eichenberger, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, and Others. Optimizing compiler for the cell processor. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 161–172. IEEE, 2005.

[42] M. P. I. Forum. *MPI: A Message-Passing Interface Standard*, September 2012.

[43] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In D. Kranzlmüller, P. Kacsuk, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, chapter 19, pages 97–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[44] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *Gnu Scientific Library: Reference Manual*. Network Theory Ltd., February 2003.

[45] D. Gamerman. *Markov chain Monte Carlo : stochastic simulation for Bayesian inference*. Chapman & Hall, 2 edition, May 1997.

[46] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[47] C. J. Geyer. Practical Markov Chain Monte Carlo. *Statistical Science*, 7(4):473–483, 1992.

[48] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[49] G. H. Golub and C. F. van Loan. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)(3rd Edition)*. The Johns Hopkins University Press, 3rd edition, October 1996.

[50] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[51] J. L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[52] J. M. Hammersley and D. C. Handscomb. *Monte Carlo methods*. Methuen; Wiley, 1964.

[53] M. Harris. Optimizing Parallel Reduction in CUDA. Technical report, nVidia, 2008.

[54] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, April 1970.

[55] N. J. Higham. The Accuracy of Floating Point Summation. *SIAM Journal of Scientific Computing*, 1993.

[56] L. Howes and D. Thomas. Efficient Random Number Generation and Application Using CUDA. In *GPU Gems*, chapter 37. 2009.

[57] D. Husmeier. Sensitivity and specificity of inferring genetic regulatory interactions from microarray experiments with dynamic Bayesian networks. *Bioinformatics*, 19(17):2271–2282, November 2003.

[58] Intel. *Intel C++ Compiler XE 12.1 User and Reference Guides*.

[59] P. Jacob, C. P. Robert, and M. H. Smith. Using parallel computation to improve Independent Metropolis--Hastings based estimation. October 2010.

[60] J. H. Jung and D. P. O'Leary. Cholesky Decomposition and Linear Programming on a GPU. Master's thesis, University of Maryland, 2006.

[61] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40+, January 1965.

[62] W. Kahan. A Logarithm Too Clever by Half. Technical report, University of California, Berkeley, August 2004.

[63] A. D. Kennedy. The Hybrid Monte Carlo algorithm on parallel computers. *Parallel Computing*, 25(10-11):1311–1339, September 1999.

[64] S. Kestur, J. D. Davis, and O. Williams. BLAS comparison on FPGA, CPU and GPU. In *Proceedings of the 2010 IEEE Annual Symposium on VLSI*, ISVLSI '10, pages 288–293, Washington, DC, USA, 2010. IEEE Computer Society.

[65] D. E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, third edition, November 1997.

[66] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. In *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE*, pages 268–302, 1998.

[67] P. J. Krause. Learning probabilistic networks. *The Knowledge Engineering Review*, 13(4):321–351, February 1999.

[68] W. B. Langdon. A fast high quality pseudo random number generator for nVidia CUDA. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2511–2514, New York, NY, USA, 2009. ACM.

[69] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[70] P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53(1):77–120, December 1994.

[71] P. L'Ecuyer. Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators. *Operations Research*, 47(1), 1999.

[72] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4), August 2007.

[73] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[74] E. Lindholm, M. J. Kligard, and H. Moreton. A user-programmable vertex engine. In *the 28th annual conference*, pages 149–158, New York, USA, 2001. ACM, ACM Press.

[75] E. Lindholm and S. Oberman. Nvidia geforce 8800 gpu. In *Hot Chips*, volume 19, 2007.

[76] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra. A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators. Technical report, University of Tennessee, Knoxville, 2010.

[77] D. J. C. Mackay. Introduction to Gaussian Processes. Technical report, Cambridge University, 1997.

[78] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, July 2003.

[79] G. Marsaglia. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. http://www.stat.fsu.edu/pub/diehard/, 1995.

[80] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: a scalable library for pseudo-random number generation. *ACM Trans. Math. Softw.*, 26(3):436–461, September 2000.

[81] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[82] M. Matsumoto and T. Nishimura. Dynamic Creation of Pseudorandom Number Generators. *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pages 56–69, 2000.

[83] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[84] J. Montrym and H. Moreton. The GeForce 6800. *IEEE Micro*, 25(2):41–51, March 2005.

[85] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[86] F. Mueller. Pthreads Library Interface, 1994.

[87] D. Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[88] N. Nandapalan, R. P. Brent, L. M. Murray, and A. Rendell. High-Performance Pseudo-Random Number Generation on Graphics Processing Units. *Lect. Notes Comput. Sc.*, 7203:609–618, 2012.

[89] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–14, New York, NY, USA, 2008. ACM.

[90] nVidia. Fermi Architecture White Paper. Technical report, nVidia, 2009.

[91] nVidia. *CUBLAS Library User Guide*. nVidia, v5.0 edition, October 2012.

[92] nVidia. *CUDA API Reference Manual*, 5.0 edition, October 2012.

[93] nVidia. *CUDA C Best Practices Guide*, October 2012.

[94] nVidia. *CUDA C Programming Guide*, pg-02829-001_v5.0 edition, October 2012.

[95] nVidia. *CUDA Toolkit 5.0 CURAND Guide*. nVidia, pg-05328-050_v02 edition, September 2012.

[96] A. O'Hagan, J. Forster, and M. G. Kendall. *Bayesian inference*. Arnold London, 2004.

[97] 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.

[98] M. Paprzycki and P. Stpiczynski. *A Brief Introduction to Parallel Computing*, volume 20052841, pages 3–42. Chapman and Hall/CRC, December 2005.

[99] C. Peeper and J. L. Mitchell. Introduction to the directx
textregistered 9 high level shading language. *ShaderX2: Introduction and Tutorials with DirectX*, 9, 2003.

[100] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *Micro, IEEE*, 16(4):42–50, August 1996.

[101] P. Pierce. The NX/2 operating system. In *Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues-Volume 1*, pages 384–390. ACM, 1988.

[102] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *Micro, IEEE*, 20(4):47–57, 2000.

[103] C. E. Rasmussen. Gaussian processes for machine learning. MIT Press, 2006.

[104] B. D. Ripley. *Stochastic simulation*, volume 316. Wiley, 1987.

[105] C. P. Robert. *The Bayesian Choice: From Decision-Theoretic Foundations to Computational Implementation*. Springer, 2nd edition, May 2007.

[106] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer-Verlag, 1 edition, August 1999.

[107] G. O. Roberts and J. S. Rosenthal. Examples of adaptive MCMC. *Journal of Computational and Graphical Statistics*, 18(2):349–367, 2009.

[108] J. S. Rosenthal. Parallel computing and Monte Carlo algorithms. In *Far East Journal of Theoretical Statistics*, pages 207–236, 1999.

[109] W. G. Rudd. X3H5 Parallel Extensions for Programming Language C. Technical report, Corvallis, OR, USA, 1993.

[110] J. Sacks, W. J. Welch, T. J. Mitchell, and H. P. Wynn. Design and Analysis of Computer Experiments. *Statistical Science*, 4(4):409–423, 1989.

[111] M. Saito and M. Matsumoto. SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In A. Keller, S. Heinrich, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, chapter 36, pages 607–622. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[112] M. Saito and M. Matsumoto. Variants of Mersenne Twister Suitable for Graphic Processors, March 2012.

[113] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[114] S. Sonnenburg, G. Rätsch, S. Henschel, C. Widmer, J. Behr, A. Zien, F. Bona, A. Binder, C. Gehl, and V. Franc. The SHOGUN Machine Learning Toolbox. *J. Mach. Learn. Res.*, 99:1799–1802, August 2010.

[115] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in science & engineering*, 12(3):66–72, May 2010.

[116] J. A. Stuart, P. Balaji, and J. D. Owens. Extending MPI to accelerators. In *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, ASBD '11, pages 19–23, New York, NY, USA, 2011. ACM.

[117] C. Tan. On Parallel Pseudo-Random Number Generation. In V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, and C. Tan, editors, *Computational Science ICCS 2001*, volume 2073 of *Lecture Notes in Computer Science*, chapter 68, pages 589–596. Springer Berlin / Heidelberg, Berlin, Heidelberg, July 2001.

[118] S. Thakkur and T. Huff. Internet Streaming SIMD Extensions. *Computer*, 32(12):26–34, December 1999.

[119] M. M. Tibbits, M. Haran, and J. C. Liechty. Parallel multivariate slice sampling. *Statistics and Computing*, 21(3):415–430, July 2011.

[120] L. Tierney. Markov chains for exploring posterior distributions. *the Annals of Statistics*, pages 1701–1728, 1994.

[121] P. Trancoso and P. Evripidou. Parallel Computer Architecture. In *Handbook of Parallel Computing and Statistics*, Statistics: A Series of Textbooks and Monographs, pages 43–73+. Chapman and Hall/CRC, December 2005.

[122] W. J. van der Laan. Decuda and cudasm, the CUDA binary utilities package. https://github.com/laanwj/decuda.

[123] V. Volkov. Better performance at lower occupancy. In *GPU Technology Conference*, 2010.

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

[125] D. W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, April 1994.

[126] J. Walker. HotBits: Genuine random numbers, generated by radioactive decay. *online at www. fourmilab. ch/hotbits*, 2001.

[127] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *CONFERENCE ON HIGH PERFORMANCE NETWORKING AND COMPUTING*, pages 1–27, 1998.

[128] D. Wilkinson. *Parallel Bayesian Computation*, volume 184, pages 477–508. Chapman and Hall/CRC, December 2006.

[129] C. K. I. Williams. Gaussian Processes, 2002.

[130] C. K. I. Williams. Prediction with Gaussian processes: From linear regression to linear prediction and beyond. In *Learning in graphical models*, pages 599–621. Springer, 1998.

[131] K. Williams and S. A. Williams. Implementation of an Efficient and Powerful Parallel Pseudo-random Number Generator. In *Proceedings of the Second European PVM Users' Group Meeting*, 1995.