Accelerated Primality Testing Using GPUs

Duncan McBain

August 23, 2013

Contents

Acknowledgements

I am very grateful for the help and support provided to me by Iain Bethune of the EPCC. His knowledge was invaluable in every stage of writing this dissertation. I would like to thank Aileen McBain for the advice, proofreading and patient nudging she provided.

1 Introduction

"Mathematicians have tried in vain to this day to discover some order in the sequence of prime numbers, and we have reason to believe that it is a mystery into which the human mind will never penetrate." - Leonhard Euler [1]

Mathematicians have attempted to understand the manner in which the primes are distributed throughout the integers, but have, as yet, been unable to do so. It has been known since Euclid that the primes are infinite in number but the exact sequence of primes is not predictable. The advent of computers, however, brought with it a way to investigate primes vastly larger than previous generations were able to calculate, resulting in a search for ever-larger primes, so that we might gain some insight into these strange mathematical beasts. This project aims to accelerate primality testing in a code known as LLRP using GPUs.

1.1 Motivation

Prime numbers underpin the security of the Internet. When a shopper buys an item on Amazon, their credit card details are protected by prime numbers. When someone logs in to digital banking, thier details are kept secret because of the sheer difficulty of factoring large integers quickly. However, it is not just for security purposes that volunteers across the world search for prime numbers. It is also for the sake of the knowledge itself, for the chance to participate in a global search that pushes the boundaries of our computing technology and in some cases advances our mathematical understanding. In January 2013, the largest prime yet discovered was found as a part of the Great Internet Mersenne Prime Search[3]. It is truly huge by everyday standards - it has 17,425,170 digits! Where previously research departments in Universities performed the majority of the computationally expensive primality tests, nowadays the software is mostly run by volunteers as part of distributed computing projects. Additionally the hardware which can run such tests has diversified to include Graphics Processing Units as well as Central Processing Units. In the last few years GPUs have increasingly been used to accelerate numerical computation in games, consumer software and even the largest supercomputers in the world [2]. It is through Application Programming Interfaces (APIs) like (CUDA and OpenCL) developed in the last few years that the power offered by these cards has been made available for non-graphical applications. Both leading GPU manufacturers, NVIDIA and AMD, provide high-performance drivers for their cards as well as specialised HPC editions of their consumer-level cards. They also provide software libraries for accelerating common mathematical codes, including linear algebra and fast Fourier transforms (FFTs). The floating-point capability of GPU hardware can vastly exceed that of CPUs, provided that the software is able to fit the massively-threaded pattern required by the architecture. While GPUs are not as versatile as CPUs, they offer a huge amount of computation if the algorithm can be structured to take advantage of it.

This project's aim is to port the numerical routines of LLRP (a freely available program for primality testing) to the Compute Unified Device Architecture (CUDA[4]), a platform for programming GPUs owned and developed by NVIDIA. The main computational task in the LLRP software is the execution of repeated forward and backward FFTs. Therefore it makes sense to attempt to accelerate certain numerical routines in LLRP by porting them to this platform in order to speed up their execution. It would be possible to attempt to write a new program from scratch, but that is likely infeasible. LLRP has been developed over many years and to throw that accumulated knowledge and effort away would be wasteful. Porting brings the advantages of a known-good code and the speed of a graphics accelerator without requiring a total rewrite and the effort that implies.

The decision to use CUDA, and particularly to use the cuFFT library, ties this project to running on NVIDIA hardware, and so is contentious given that AMD also has its own FFT library available[5]. An alternative platform called OpenCL (Open Computing Language [6]) has been standardised by the Khronos Group, a consortium of companies with a common interest in promulgating open standards. AMD is a part of this consortium and support the OpenCL standard on their GPUs. OpenCL allows programmatic access to GPU hardware but is not locked to one particular vendor, unlike NVIDIA's CUDA. One reason that CUDA was chosen is that CUDA has been in production use for longer than OpenCL, which is relatively new. Both NVIDIA and AMD have written mathematical libraries optimised for their hardware so this project would have had to target one library to the exclusion of the other. It would have been possible to target OpenCL and write an FFT implementation for this project specifically, to reduce fragmentation by introducing a single version of the code, but since one aim is to achieve the highest performance, using vendor-tuned libraries seems the most sensible option. For these reasons, a CUDA port with cuFFT providing the fast Fourier transforms was chosen.

Another project aim is to enable LLRP to make better use of the diverse hardware available in modern desktop computers. This software is run largely by volunteers interested in helping in the search for ever-larger primes, on many different hardware configurations. Making use of all the hardware that these volunteers have is desirable, particularly since using the GPU frees up CPU resources that might otherwise interrupt the volunteers' use of their computing systems. These volunteers run the software as part of large distributed computing projects. There is an infinite number of primes and for that reason projects seek to have highly efficient codes running on single cores, rather than some difficult parallelisation of Fourier transforms across multiple cores.

1.2 Project Proposal

Prime numbers are of central importance to encryption systems worldwide, yet the numbers themselves are interesting for a great many more reasons. Determining the factors of a large integer is known to be computationally very difficult, however there are methods which can check the primality of large integers of certain forms efficiently. There are a few such classes of number - Generalized Fermat Primes ($b^{2^n} + 1$), Proth and Riesel numbers ($k \times 2^n \pm 1$) and Mersenne Primes ($2^n - 1$). Just this year the 48th Mersenne prime was found (taking 39 days on a CPU), with a verification run being performed in only 4 days using a GPU. It has over 17 million digits and is currently the largest known prime. However, it is not known whether there are finitely many Mersenne primes or not.

Codes exist to test all these types of number and the tests are suited to optimisation by parallelisation on a GPU. Highly-optimised CPU code exists but is sometimes not portable, which is key as these projects are used in distributed computing projects like GIMPS and PrimeGrid. Therefore, efficient use of desktop machines is desirable.

The aim of this project is to port the plain-C LLRP code to the CUDA platform, with the aim of maximising the amount of work performed on the GPU and minimising the CPU usage. This will not only increase the efficiency of the code by minimising time spent transferring data between the CPU and GPU - often the slowest step - but will also allow the code to be run on desktop machines that have capable GPUs with minimal impact on the CPU. Performance comparisons between LLR, the new code and the existing LLRCUDA port will be made.

1.3 Background

Software that can test the primality of numbers is used in a number of online distributed computing projects, including the Great Internet Mersenne Prime Search[7] (GIMPS) and PrimeGrid[8]. These projects attempt to discover prime numbers not only for the sake of curiosity but also to satisfy certain hypotheses that, while lacking a mathematical proof, make some claim that can be searched exhaustively. The "Seventeen or bust" project[9] is an example of a group attempting to prove a hypothesis by exhaustive search. This project is attempting to solve the Sierpinski problem - to find the smallest Sierpinski number. A number k is a Sierpinski number if $k2^N + 1$ is not prime for all N; in 1967 Sierpiński and Selfridge conjectured that 78,557 was the smallest such number. Seventeen or Bust is searching for primes of the form $k2^N + 1$; there are very few ks left that are smaller than 78,557.

The software LLR was developed to test the primality of large integers. It utilises various algorithms to determine the primality of many different types of number. Numbers of the form $N = k \times 2^n - 1$ can be tested by the Lucas-Lehmer-Riesel algorithm[10] which is the code path this project aims to accelerate.

The algorithm is an extension of the Lucas-Lehmer test used for Mersenne numbers and as such has similar structure. In short, given some starting value u_0 (which is determined by the constants that comprise N), the sequence $\{u_i\}$ is defined as $u_i = u_{i-1}^2 - 2$. If N divides u_{n-2}, then N is a prime number; otherwise, it is not. It is worth noting that in the case that N is a composite number none of its prime factors are revealed. Factorisation is more computationally demanding and as such these efficient

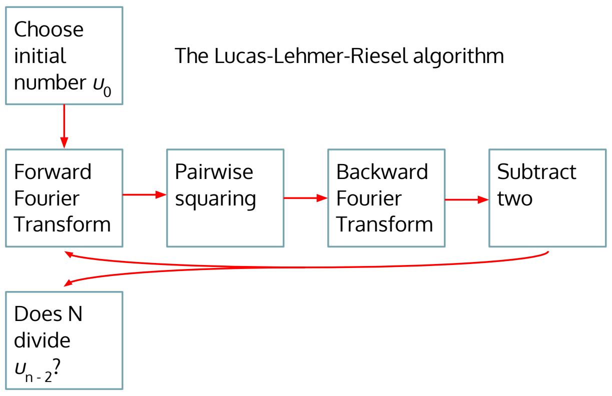
Choose initial number u_0
The Lucas-Lehmer-Riesel algorithm
Forward Fourier Transform
Pairwise squaring
Backward Fourier Transform
Subtract two
Does N divide u_{n-2}?

Figure 1: A flowchart showing the core of the Lucas-Lehmer-Riesel algorithm. The process loops $n-2$ times, at which point the program checks if $u_{n-2}/N = 0$; if it does, the number is prime.

prime tests do not provide a 'short cut' to obtain prime factors more quickly than integer factorisation is able to provide.

It can therefore be seen that the core of the algorithm is calculation of the sequence $\{u_i\}$. Normally integer multiplication is extremely fast on modern CPUs and is not a performance issue; however, the numbers being tested by this algorithm go far beyond the limits of even 64-bit floats. Naïve, or 'long' multiplication, has a time complexity of $O(n^2)$ (for multiplying two numbers of length n) which would take far too long to execute. Therefore, a faster multiplication method is required.

The method implemented in this code is due to Schonhage and Strassen[11] (see also [12]). The numbers are represented by floating-point arrays where each array element represents one 'digit' of the number (not necessarily in base ten, but this is an implementation detail and does not affect the description of the ideas of the algorithm). To obtain the product of the two numbers, they are first Fourier transformed, then each matching 'digit' is multiplied together. When the reverse transform is performed, the result will be the product of the two numbers. An additional normalisation step must be performed in order to ensure that each 'digit' is less than the base (analogous to the carry generated when multiplying in base ten, using long multiplication).

The complexity of the fast Fourier transform is $O(n\log(n))$ for a transform of length n; the piecewise multiplication has only n complex multiplications and so is essentially negligible. for sufficiently large n. The size of transforms performed in LLR is typically in the tens of thousands but can be much larger. It can be seen that this is an algorithmic speedup over naïve multiplication and is well worth the effort of implementing. This means that the most expensive operations in the main loop are now the forward and reverse transforms.

1.4 Related Software

The distributed computing projects, the GIMPS and PrimeGrid, make use of many different numerical codes to test their primes, depending on the class of number to be tested. These include Prime95[13] and Genefer [14] as two of the more common codes; however the ones of particular interest to this project are LLR, an implementation of the Lucas- Lehmer-Riesel algorithm using the gwnum library; LLRP, a plain-C version of the previous code and llrCUDA, a CUDA port of LLRP written by Shoichito Yamada.

1.5 Hardware

This project's test machine consists of an Intel Sandy Bridge 2500k with an NVIDIA GTX 570 running Arch Linux. It was developed targeting CUDA version 5. Since the majority of the computers running this software have consumer hardware rather than the High Performance Computing editions of this card, developing it on consumer hardware more accurately reflects the settings in which the software will be used.

2 Investigation of LLRP

2.1 LLRP Overview

LLR is a code developed by Jean Penné, utilising the gwnum library written by George Woltman, that implements the Lucas-Lehmer-Riesel algorithm. It is capable of testing the primality of many more types of number than Riesel primes using a variety of techniques, though these are of secondary importance to this project. The original gwnum library uses hand-coded assembly to perform its FFTs allowing it to do so very quickly - for example, the most recent versions are capable of taking advantage of AVX [15] instructions present in modern processors from AMD and Intel. However, when no assembly has been written for an architecture, the code simply will not run.

Penné has also written a version of the gwnum library which does not use the assembler present in the original but is instead designed to be portable to any system with a C compiler. This rewrite is called gwpnum and is embedded in the LLRP application. While it is slower, is a suitable starting point for a port to CUDA for two main reasons. Firstly, its portability; secondly, the interfaces to the cuFFT library are very similar to the API of the library known as "The Fastest Fourier Transform in the West" (FFTW [16]) which is used for the transforms in LLRP. It will run virtually anywhere but is considerably slower for this specific program.

The main interface to the program is through the command line, where users are able to describe a number to the program using mathematical operators like *, ^ and so on. An example invokation of the program might look like `./llrp -d -q"11*2^74726-1"`. This method of specifying the task allows it to be a part of distributed online computing projects like PrimeGrid such that the controlling software can distribute tasks to a number of volunteer machines, run these tasks and collect the output without user intervention. The program's output is in a standard format allowing it to be interpreted by automated scripts.

The program is split into several different object code files. The functions that gwnum provides are concatenated into a statically-linked archive allowing the main executable access to all the functionality of gwnum. This design is followed in LLRP since Penne re-uses the LLR-specific code and links gwpnum instead of Woltman's gwnum code. Therefore, the only difference between LLR and LLRP is in the numerical library, renamed gwpnum in LLRP. This modularity ensures that the two versions of the library are completely interchangeable removing the need to maintain two separate implementations of the Lucas-Lehmer-Riesel algorithm, and so on. One object contains the implementation of the Lucas-Lehmer-Riesel algorithm and organises the code to execute based on the number that it is to test (for example, the Lucas-Lehmer test for Mersenne primes, the Proth test and so on). The other objects contain the main startup code as well as utility functions for file handling, memory management and other important but, for this project, inconsequential functionality.

2.2 Data Flow

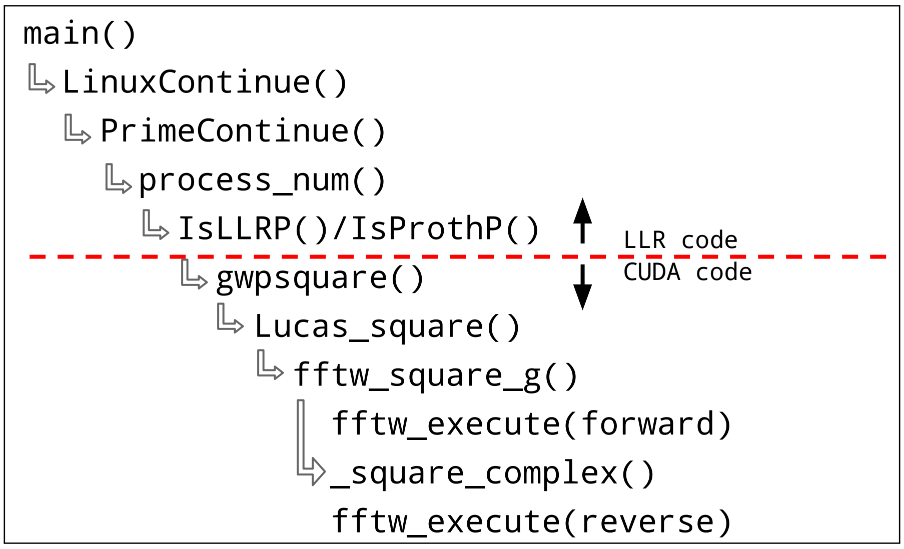

Figure 2: A graphical representation of the important functions called when running the Lucas-Lehmer-Riesel algorithm in LLRP

Tracking data through the Lucas-Lehmer-Riesel execution path provided a great deal of insight into the operation of the program. Figure 2 shows the branch of the call tree which leads to the actual squaring of the large potential prime. There are other functions executed along other branches of the program but they are less relevant to this analysis of the data flow, as they will not be affected by the CUDA porting efforts. The main entry point to the program processes any command-line arguments which might have been passed to the program and then continues execution based on those arguments. The program ensures that no other copy is running (`LinuxContinue()`) then classifies the number according to the form it takes (say, $k \times b^n \pm c$) in `PrimeContinue()`. This chains into `process_num()` which will then call the appropriate `is*()` function, e.g. `isLLRP()`, depending on which algorithm is to be executed as decided by the form of the number. For this project in particular, the function of interest is `isLLRP()`, which implements the Lucas-Lehmer-Riesel algorithm.

Any code deeper in the tree than this is now executed in the gwpnum library. `isLLRP()`, after initialising several key variables in the gwpnum library, initiates the main loop of the program. It calls the function `gwpsquare()` whose main function is to call the lower-level `lucas_square`. The purpose of `lucas_square` is to perform many of the lower-level operations necessary before the actual Fourier transforms can be executed and then to normalise the output received from `fftwsquare_g()`. Optionally, some error checking is performed.

In `fftwsquare_g()`x, the main computation occurs. As can be seen in figure 2, the three most important functions are `fftw_execute` (called twice) and `_square_complex()`. This is where the actual squaring of the number happens, bounded by the forward and backward Fourier transforms.

Analysis of this program revealed that for the Lucas-Lehmer-Riesel algorithm, none of the boolean values `zp`, `generic` or `compl` are true. Therefore the sections of code executed when those variables are true have been excluded from this analysis. The lower-level functions of gwpnum are more generic than required for just the LLR algorithm and as such have alternative functionality controlled by these flags; for example, `zp` controls whether the FFTs are zero-padded, `generic` is used when the expression has an added component that is not ± 1 and `compl` controls transforming a real FFT of length N into a complext FFT of length $N/2$.

2.3 Results of Profiling

It is possible to compile programs on GNU/Linux with extra instrumentation code (using the compile-time flag -pg) that will profile the program at runtime, allowing the programmer to obtain useful information about how the code performs. The gprof tool can create a flat profile showing how often functions were called and for how long they were running, a useful tool to see where the code spends the most time. This can help the programmer's optimisation efforts ensuring that they do not waste time optimising slow code that is only executed a handful of times, for example. Additionally the instrumentation is capable of generating a call graph similar to the hand-produced graph in figure 2, albeit one that is much more complex and comprehensive. Significant portions of a flat profile for LLRP, testing the primality of the prime number $11 \times 2^{74726} - 1$, are reproduced in figure 2.3. However, it is worth noting that this profiler output is not exactly accurate for this code because it has not instrumented large parts of the FFTW code which are compiled in a separate listing. This is most apparent in that the running time is quoted as a little over five and a half seconds in the profiler but the actual time elapsed was a little over eleven seconds. The missing time is the time spent in the FFTW library.

It can be seen that the functions `inormalize`, `lucas_square` and `_square_complex` take the most time to execute (not including the missing Fourier transform time). This is a consequence of the core of the algorithm: the squaring happens $n - 2$ times; in this case, 74724 times. Some other constants must be determined before the algorithm can be executed and these are determined using a similar process to the algorithm's core. This explains the slightly higher execution count for the three functions.

The call tree, together with the profiler output, demonstrates what the key numerical routines at the heart of the Lucas-Lehmer-Riesel test section of the LLRP code are and therefore what the focus of this project should be.

2.4 Deciding Project Scope

The call tree shows a strong divide between the high-level algorithmic code and the lower-level mathematical code in LLRP. Similarly, most of the time spent in execution is in these lower-level mathematical functions. This indicates that the main efforts

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
43.02      2.31     2.31    74730     0.00     0.00  inormalize
30.54      3.95     1.64    74727     0.00     0.00  lucas_square
22.91      5.18     1.23    74727     0.00     0.00  _square_complex
 0.56      5.21     0.03       35     0.00     0.00  fftinv_hermitian_to_real
 0.37      5.23     0.02  1123531     0.00     0.00  dd_real::dd_real
 0.37      5.25     0.02   157743     0.00     0.00  operator/
 0.37      5.27     0.02      689     0.00     0.00  check_balanced
 0.19      5.28     0.01  1620394     0.00     0.00  quick_two_sum
 0.19      5.30     0.01   358493     0.00     0.00  two_diff
 0.19      5.31     0.01   329821     0.00     0.00  operator*
 0.19      5.32     0.01    14337     0.00     0.00  exp
 0.19      5.33     0.01    14335     0.00     0.00  nint
 0.19      5.34     0.01       53     0.00     0.00  fft_real_to_hermitian
 0.19      5.35     0.01       35     0.00     0.00  addsignal
 0.19      5.36     0.01        3     0.00     0.00  lucas_mul
 0.19      5.37     0.01        1     0.01     5.37  isLLRP
```

Figure 3: Truncated output of the gprof profiling tool after instrumenting the program LLRP.

should be in the gwpnum library, accelerating the transforms and associated code, but maintaining encapsulation of the CUDA implementation below the point that the code in the algorithmic section might see.

It would be possible to alter more of the lower-level code such that other code paths (like, for example, the Proth primality test) might be accelerated too; however, it is felt that accelerating the code in the Lucas-Lehmer-Riesel test is sufficient for a project of this length.

3 Porting to CUDA

3.1 GPU Architecture and Theory

Throughout this discussion, “host” will refer to the CPU and main system memory and “device” will refer to the GPU, its memory and associated hardware.

GPU development has largely been driven by the need for stronger hardware to drive performance in games, with the architecture reflecting the strenuous requirements of generating three dimensional scenes at ever-higher resolutions, with higher triangle counts and larger textures, while still maintaining a sufficient number of frames per second. In contrast to CPUs, which traditionally only have a few hardware threads (though recently CPU design is heading for multi-core chips) GPUs are massively parallel, with core counts running easily into the hundreds. Games frequently require operations to be performed for every pixel on the screen, for every triangle and similar operations, and the “multiple threads working on multiple data” design reflects this. Originally there were two main APIs which exposed this functionality (abstracting the hardware, so that one piece of software could run on multiple devices): Direct3D and OpenGL. Although powerful, these APIs were designed to assist with two and three dimensional operations, and their use beyond this area was limited. However, recent years have seen the rise of GPGPU - General Purpose computing on GPUs. Through APIs like OpenCL and CUDA, programmers are now able to issue commands to the hundereds of cores in GPUs and have them accelerate processes, potentially leading to huge execution speedups. This project implemented a code in CUDA, so this discussion of GPU architectures will focus on NVIDIA products. The concept of multiple cores operating on multiple data streams is broadly applicable, however there are some differences between the two vendors’ hardware.

Figure 4 shows the structure of a Stream Multiprocessor in an NVIDIA GF100 chip. The GF100 is not the SM used in the GTX 570, but its design is useful for describing the operation of SMs in general. Each SM consists of thirty-two CUDA cores, however, it can be seen that there are far fewer scheduling units than cores. This design means that every core within an SM must execute the same sequence of commands since there are not enough schedulers to have multiple instruction streams. This is the main way in which GPUs parallelise in order to speed up execution - by applying the same operations to different data, which does put some limits on the algorithms that can be ported to GPU platforms. Figure 5 shows the composition of each core; each is much more simple than modern CPU cores. GPU cores typically have a much lower clock rate than CPUs as well, but is through the number of cores available that GPUs achieve their performance.

NVIDIA describes GPU computing as a “grid of thread blocks”, where blocks map to streaming multiprocessors and threads map to cores. A group of thirty-two threads is called a “warp”. Due to the low number of scheduling units, each thread in a warp must execute the same instruction, which places a heavy penalty on branching code. The

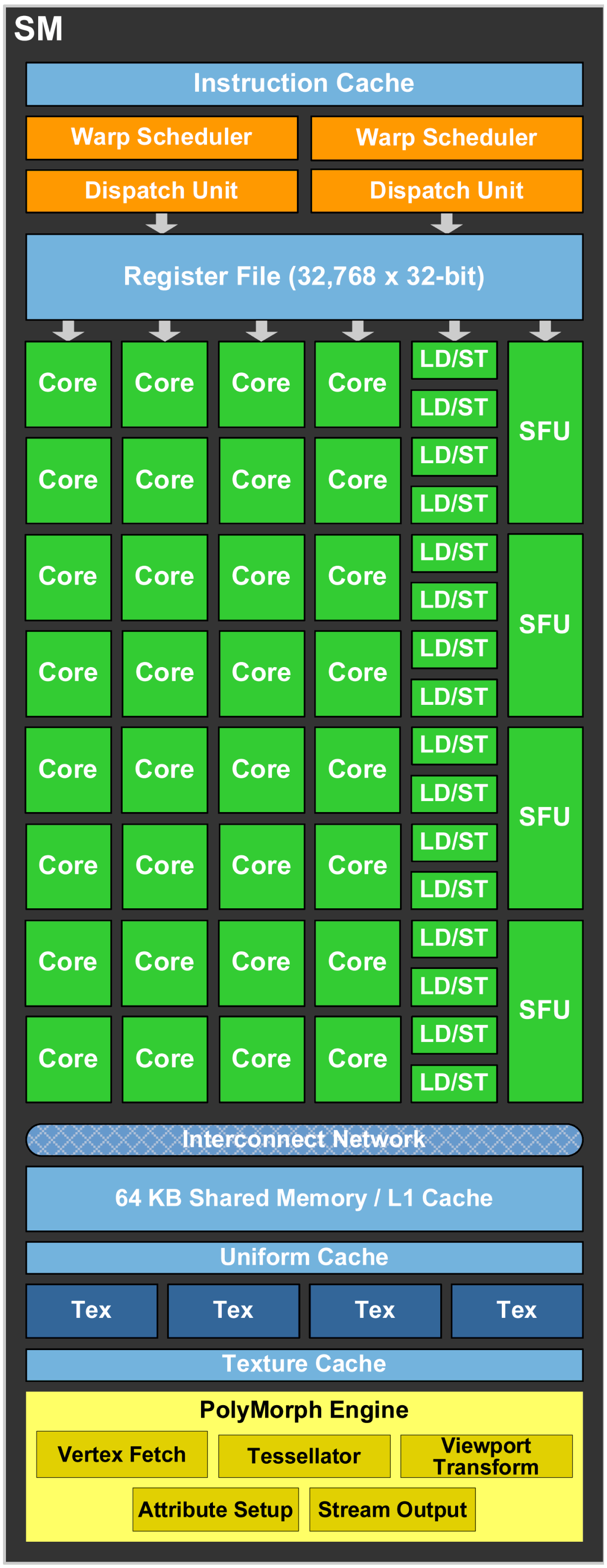

Figure 4: A Streaming Multiprocessor found in the GF100 chip made by NVIDIA, adapted from[17]

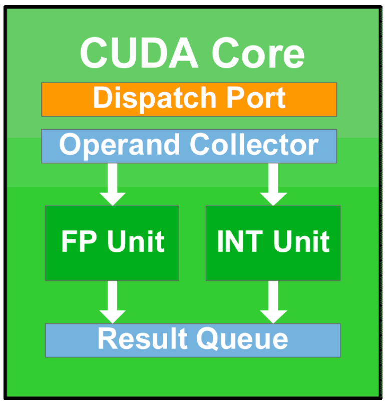

Figure 5: Close-up diagram of a CUDA core found in the GF100 SM, adapted from[17]

grid of thread blocks abstraction is useful as it allows the programmer to write code in a device-independent manner while maintaining high performance. When executing code on the GPU, the programmer indicates the number of threads to run in total by specifying the number of blocks per grid (corresponding to the number of SMs) and the number of threads per block (corresponding to the number of cores in an SM). While it would be possible to hard-code these numbers for different models, this would be inefficient and would require constant maintenance as new hardware is released. Instead, NVIDIA recommend oversubscribing both the blocks per grid and the threads per block to allow the thread scheduling engine to distribute the work across the available hardware. This allows the programmer to avoid situations that might arise from changes in more recent hardware, like an increase in core count in SMs. For example, each SM in the GF100 chip has thirty-two cores, but previous SMs had only eight. It took those units four times as many cycles to complete a warp since they only had one quarter of the number of cores. If programmers had hardcoded only eight threads per block, one quarter of the cores in GF100 chips would be unused, a clear inefficiency.

Oversubscribing the work to be performed per block and per core has other advantages, largely to do with memory access. GPU memory has been optimised for bandwidth (for example, to move large textures in games) but suffers from high latency. If there are sufficiently large number of threads, the scheduler can ‘hide’ the latency by swapping threads in and out of blocks when they make a request for global memory accesses. It is also advantageous for the programmer to ensure that the memory locations they access are adjacent, as the device can then coalesce these memory accesses for a peformance increase.

As well as global memory, there are other smaller sections of faster memory available. The programmer might choose to put constant but often-used data in the ‘constant’ memory section, or if there is mutable data to be shared among cores in an SM, it might be stored in the ‘shared’ address space.

Code declared as being a ‘kernel’ is executed on the GPU. Each thread will execute a copy of the function. It is common to have some large array and a task that must be performed for each element in the array. When writing the kernel, the programmer can ensure that each thread performs the correct unit of work by, for example, calculating the thread ID then operating on the array element indexed by that ID (this is the method primarily used in this project's CUDA code). Code that has multiple branching statements is likely to perform poorly on the GPU unless all cores within a warp take the same path of execution; if they do not, every core will still perform the same actions but will discard the data at the end, rather than saving the results. If all cores within a warp branch in exactly the same way this is not a problem.

Kernel launches are asynchronous (and are executed synchronously on the GPU unless the programmer specifies), allowing the programmer to perform other work while the GPU executes the kernels. This is similar to the overlapping communications and computation design pattern from, for example, MPI (Message Passing Interface[18]). The programmer is able to launch a kernel and simultaneously begin a memory transfer to or from the device, provided that the working area of the kernel and the transfers do not overlap. This is hugely important, because memory transfers between host and device are incredibly slow in either direction. Although a large amount of bandwidth is available (less than memory bandwidth, however), the latency is quite high, and memory transfers can easily ruin the performance of a CUDA code. It is best to avoid transferring data if at all possible; performing additional computations on the host and device while transferring is more efficient too.

In the last decade, there has been a large change in the design trends of CPUs. Previously, each new CPU generation brought a speed increase largely through an increase in clock rate (as well as a host of other important changes which are tangential to this discussion). The key difference between GPUs and CPUs was the massively parallel approach of GPUs, with a focus on simultaneous computation compared the CPU's sequential operation. However, the last decade has seen a halt in the increase of CPU clock speeds and a trend towards multiple cores in CPUs as well. While there are still many fewer cores in a CPU than a GPU, to obtain the best performance code must be prepared to take advantage of multiple cores. CPUs are also capable of other simultaneous operations like performing integer and floating point maths simultaneously as well as using Single Instruction, Multiple Data (SIMD) extentions like SSE and AVX to operate on multiple data elements at once.

In the future, designs might move towards increasing similarity between GPU and CPU devices. Intel's Larrabee project aimed to create a sort of middle ground between CPUs and GPUs. It was to support the x86 instruction sets like most CPUs but was to have simpler cores (e.g. no support for out-of-order execution) like GPU cores. In the end, the project was cancelled, but using it as a research project Intel recently released the Xeon Phi. Each card has over sixty cores; many more than any CPU but still far fewer than GPUs. It is capable of hosting a simple operating system based on Linux. The Chinese supercomputer Tianhe-2 has many of these devices installed as co-processors to its CPUs and as of June 2013 is the fastest supercomputer in the world.

3.2 The cuFFT Library

To encourage programmers to attempt to harness the power of GPU computing both hardware vendors provide highly-optimised mathematical libraries. Part of NVIDIA's CUDA platform is the cuFFT library, a code designed to perform fast Fourier transforms in both single and double precision in up to three dimensions[19]. The interface was designed to be very similar to FFTW's interface which is helpful in this project since LLRP uses FFTW for its transforms, but is also useful for other developers, helping reduce the amount that they have to learn when moving to a new platform.

cuFFT implements both the Cooley-Tukey and Bluestein algorithms for FFTs, falling back to the latter when the FFT size is not decomposable into sufficiently small prime factors. For this project, the FFT lengths are set such that they have small prime factors. The code to set this length is due to Shoichiro Yamada and is present in his code llrCUDA [20]. This means that cuFFT is always able to use the more accurate Cooley-Tukey algorithm, which has many optimised and hand-coded kernels in cuFFT.

3.3 Porting in Stages

It was decided that the porting attempt should consist of several stages: a naïve straight replacement attempt with no concern for memory transfers, potential bottlenecks and so on; a version which attempted to keep the data on the device between the two transforms, but copied it on and off either side of those; and finally a version which only transferred data to the device at the beginning of execution and transferred it off when execution was finished.This was decided because each version provided a clear path to the next iteration of the code, while still being relatively easy to implement. It also provided a chance to catch regressions as they happened without having to roll back every version of the code.

Immediately it was seen that it would be extrememly easy to implement a "drop-in" replacement, given the similarity of the APIs of cuFFT and FFTW. Therefore, code was written to create plans to execute the same transforms as the FFTW version. However, the data required was not on the device, so a synchronous data transfer was inserted before and after every transform. This comes with a severe performance penalty: execution of the code could not continue while the data was transferred, and such transfers are very slow. They are often a bottleneck in CUDA codes. Four such transfers were needed per iteration of the code; since the size of the data to be transferred as well as the number of times the loop executes increases with the size of the prime, the performance penalty was severe for larger primes. Figure 3.3 shows the pattern of transfers and transforms.

```
// Copy onto the device, do the forward FFT, then copy off
cudaMemcpy(deviceArray, hostArray, size, cudaMemcpyHostToDevice);
cufftExecD2Z();
cudaMemcpy(hostArray, deviceArray, size, cudaMemcpyDeviceToHost);

// Executed on the host, not the device
_square_complex()

// Copy onto the device, do the backward FFT, then copy off
cudaMemcpy(deviceArray, hostArray, size, cudaMemcpyHostToDevice);
cufftExecZ2D();
cudaMemcpy(hostArray, deviceArray, size, cudaMemcpyDeviceToHost);
```

Figure 6: Pseudocode representing the order of memory transfers and Fourier transforms as in the first version of the code. Later versions removed the transfers either side of the squaring by implementing a CUDA kernel to perform the squaring.

3.3.1 Eliminating the interior memory transfers

To reduce the number of memory transfers, code to implement the element-by-element squaring present between the two Fourier transforms was chosen as the next target of the porting effort. This was the only operation present between the forward and backward transforms so it was quite wasteful to shuffle the data around so much for such a simple function. The code to square the numbers was straightforward, however squaring on the device instead of the host brought about a large improvement in performance simply by avoiding two time-consuming memory transfers per iteration.

Figures 7 and 8 show the similarity between the C code and the CUDA code for squaring large complex numbers. The numbers are represented as arrays of interleaved real and imaginary parts. The difference is that the CUDA code is designed to run on a single element only. The manner in which all elements of the array are transformed is to launch a kernel with as many threads as there are array elements (due to the division of the threads into blocks, the number of threads launched is actually equal to the smallest multiple of the number of threads per block greater than or equal to the number of array elements; this is why there is code to ensure that the thread index is not greater than the array length). This pattern of parallelising loops with independent iterations into CUDA kernels is a common idiom.

3.3.2 Porting the remaining functions

After the pairwise squaring was ported, only a small subset of the code remained on the host. Two such sections were simply a weighting of the array elements, necessary for the discrete weighted transform that this code implements.

```
void
_square_complex(
fftw_complex *b,
int n
)
{
    register int k;
    register double Reb;

    for (k=0; k<n; k++) {
        Reb = b[k][0]*b[k][0]-b[k][1]*b[k][1];
        b[k][1] = 2*b[k][1]*b[k][0];
        b[k][0] = Reb;
    }
}
```

Figure 7: Original C code to square an array of complex doubles element-by-element

```
__global__ void square_elem(cufftDoubleComplex *x, int length)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i >= length)
    return;
    double re = x[i].x * x[i].x - x[i].y * x[i].y;
    x[i].y = 2 * x[i].x * x[i].y;
    x[i].x = re;
}
```

Figure 8: CUDA code to square a single complex number out of an array; does nothing if its index is off the end of the array

The last remaining host function was inormalize. In the process of ‘long’ multiplication (the sort that is taught in schools) if a digit in a particular position should become too large then a ‘carry’ is created; the carry is equal to the overflow from the lower digit divided by ten. It is added on to the next digit in the sequence and the process repeats until every digit in the number is less than the base the number is represented in (most often ten). A similar operation must occur for the method of multiplication implemented in this code. While the multiplication happens in Fourier space, the normalisation, or carry-generating step is performed in real space, and is analogous to the same step found in long multiplication.

There are other operations which must happen in this normalisation code. The Fourier transforms are performed using double-precision numbers; however, the numbers that are being tested are obviously integers. There is some amount of rounding error associated with the transform in and out of Fourier space, and as such the digits must all be rounded to the nearest integer to ensure that the answer obtained is correct. In addition, LLRP adds the constant value (+2) to the number at this stage of the computation. This does not have to be done at this stage but it is convenient to do so. Figure 9 shows the first part of the normalisation procedure.

The digits are rounded by adding a sufficiently large value to the digit then taking it away again. In arbitrary-precision arithmetic, this is a null operation. However, floating-point mathematics is not associative, and when the large value is added to the digit some trailing bits of data are ‘lost’. With careful selection of the large value, the fractional part of the digit can be removed, leaving an integral value (all integral values below 2^{53} can be represented exactly in a double-precision float). This happens because the processor shifts the numbers to be operated on such that they have the same exponent. Values over a certain threshold only have enough precision to store units, not tenths or smaller, and so the fractional part is removed. The reason that the rounding is done this way is because it only takes two double- precision additions; using a cast operation can take upwards of eighty cycles in comparison! The technique of adding a large constant to the digits is reused to create the carry value, since a larger number will ‘wipe out’ even more of the lower bits of the digit. Again, a careful choice of number will ensure that the digit will remain less than the base the number is represented in.

One last pass must be made over the array in the case that the highest-position digit has a carry associated with it. The way it is dealt with is that the carry is rotated back to the lowest position of the array and, depending on the exact way in which the number is to be treated, is added back in the lower digits. One final normalisation pass is performed, since adding in any value might unnormalise a digit of the number.

This code works well on a sequential device like a CPU, since each carry value can potentially affect the digit above it. However, loops are most easily parallelised in the CUDA paradigm when their iterations are independent - this code is almost exactly the opposite. The only synchronisation available to CUDA threads is between threads in the same warp. Across the entire grid, there is none, save for atomic operations (which can be very slow if there is high contention). As a consequence the CUDA normalisation code became very complicated and difficult to implement. Several de-

```
int j;
double * num;
double rounded, highBits, carry = 0.0;

for (j = 0; j < N; ++j)
{
    // Nearest-integer rounding
    rounded = (num[j] + bigVal) - bigVal;
    // Add the carry from previous iteration
    rounded += carry;
    // Removing the lower bits from the digit
    highBits = (rounded + limitbv[j]) - limitbv[j];
    // Compute the carry on next word
    carry = highBits * invlimit[j];
    // jth digit is now rounded and normalised
    num[j] = rounded - highBits;
}
```

Figure 9: Simplified code showing the sequence of operations in the inormalize function. It is not trivial to transform this loop into a CUDA kernel.

signs were attempted, which often contained subtle data hazards where potentially two threads would be attempting to access the same element at the same time, which CUDA provides no protection against. Naturally, these erroneous versions did not work.

The end design settled upon having multiple separate kernels, since it is guaranteed that all operations will have finished when the next kernel has been launched (by using a synchronisation function). This comes with an associated performance penalty as kernel launches are quite expensive. By using a global variable which is updated using an atomic operation, the device is able to indicate to the host how many carries remain, i.e. whether another pass needs to be performed on the array to ensure that all digits are less than the base of the number. There is a while loop which will, in order, add in the carries, set all of them save the highest position to zero, renormalise then check whether any carries remain. When no carries remain, the code leaves the loop and executes the 'propagate_carry' kernel, which is the code that deals with the high-position carry.

This code does not produce the right answer for primes of greater than a certain size which is disappointing, and the reason for its lack of correctness has not been discovered. It is felt that the difficulty of implementing a loop with such dependent iterations has led to a subtle error that prevents it from working as intended.

```
int j;
double rounded, high_bits, carry2, carry = carries[len];
if (carry)
{
    j = 0;
    if(wrapindex)
        carry2 = carry * wrapfactor;
    carry *= -1;
    while (carry || carry2)
    {
        if (wrapindex && !carry)
            j = wrapindex;
        rounded = *(x + j) + carry;
        if (wrapindex && j == wrapindex)
        {
            rounded += carry2;
            carry2 = 0.0;
        }

        high_bits = (rounded + limitbv[j]) - limitbv[j];
        carry = high_bits * invlimit[j];
        *(x + j) = rounded - high_bits;

        if (++j == len)
        {
            j = 0;
            if (wrapindex)
                carry2 = carry * wrapfactor;
            carry *= -1;
        }
    }
}
```

Figure 10: Code that reconciles the carry values from the end of the array by adding the carry back onto the lower digits of the number. The translation from C to CUDA left it virtually unchanged.

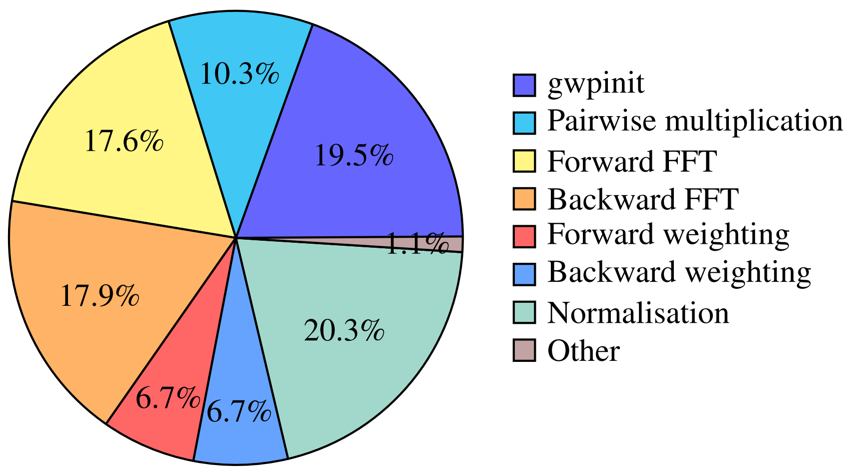

Figure 11: The percent time spent in each section of code in the original LLRP code

4 Performance Analysis

4.1 Comparison of Times for Various Primes

The code has many timing functions available for both debug and informational purposes. To have a more fine-grained profile of how long the program spent executing certain sections of the code, timing functions were inserted at key points in the program's squaring code. These were used to track how long the data transfers ran for, or how quickly the transforms were executed. These times are stored cumulatively for the entire running time of the program. Figures 11 to 14 shows the percentage time spent in key areas of the program for select primes. The timers cover the time spent executing the transforms, the time spent normalising, the time spent transferring data, the complex squaring, the weighting of the number before and after the transforms and the time taken to initialise the system. Any discrepancy between the total time recorded by these timers and a timer that covers the execution of the program is represented as 'other'.

The primes were chosen to cover a range of sizes while staying small enough to be tested in a matter of seconds to ensure rapid testing when developing the code. Finally a very large prime was chosen to investigate the behaviour of the program at large N. The trend is that proportionally less time is spent executing the FFTs compared to the normalisation, which still runs on the CPU, and time spent waiting for memory transfers to complete, which are necessary for the normalisation to run. For example, in figure 12 the FFTs take approximately 20% of the running time, dropping to a little over 10% of the running time in figure 14.

Figure 11 shows the time spent inside the plain LLRP code. The next three show the same code sections but in the final version of the llrgpu code, with FFTs and so on performed on the GPU. It can be seen that the FFTs take proportionally less time in the

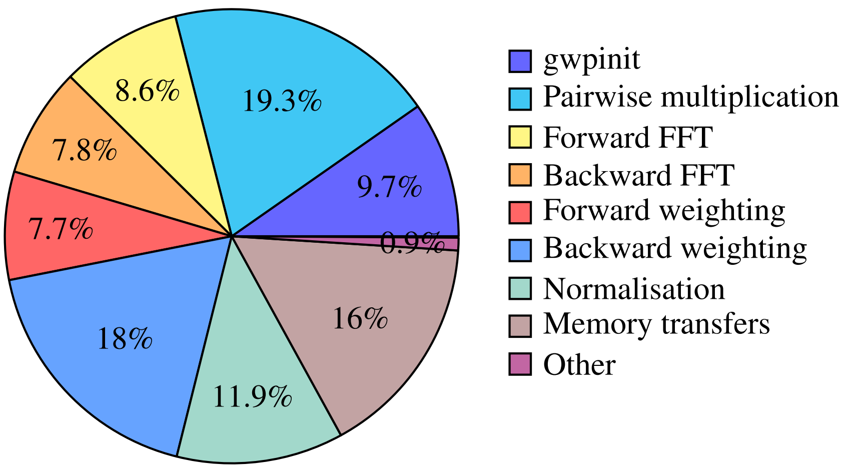

Figure 12: llrgpu timings for $11 * 2^{74726} - 1$

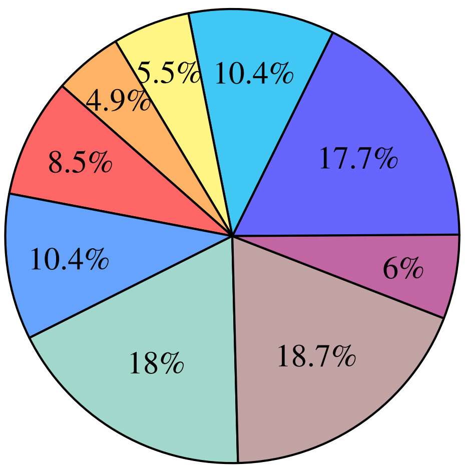

Figure 13: llrgpu timings for $13 * 2^{166303} - 1$

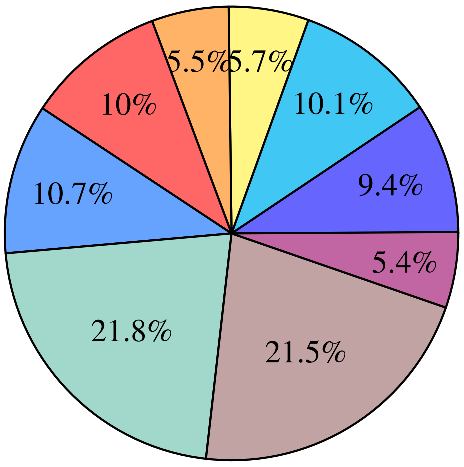

Figure 14: llrgpu timings for $13 * 2^{233207} - 1$

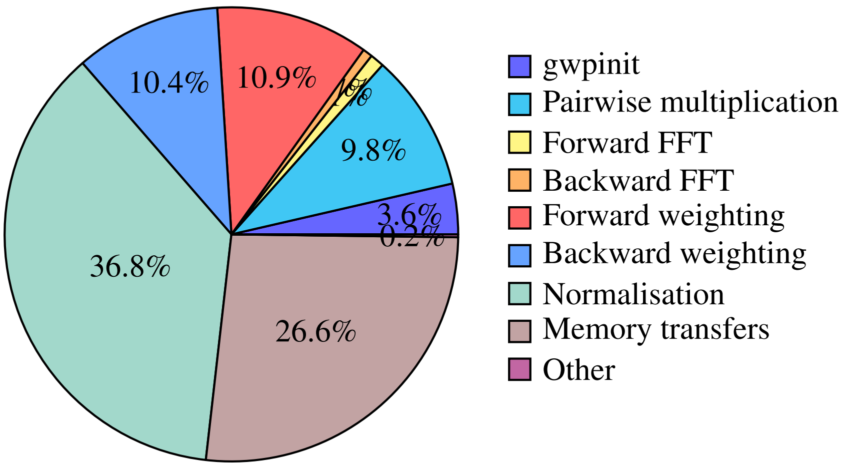

Figure 15: llrgpu timings for $3 * 2^{3136255} - 1$

GPU accelerated code, indicating that the GPU code is faster. This aspect of the code could be considered highly successful.

The size of the prime being tested increases across the three charts. The charts indicate that the proportion of time spent transferring data and normalising the number is now the bottleneck. If a suitably fast GPU normalisation routine were written, it would solve both problems at once; however, implementing such a function correctly and quickly remains very difficult.

To test how the code could deal with very large primes, the prime number $3 \times 2^{3136255} - 1$ was tested. The total running time was 6670 seconds, including time taken for the library to initialise; figure 15 shows the breakdown of times. Overwhelmingly the most time was taken in normalising the number. The transforms took less than 2% of the running time of the program. It is a testament to the power available in GPUs that

Primes	$11 \times 2^{74726} - 1$	$13 \times 2^{166303} - 1$	$13 \times 2^{233207} - 1$	$17 \times 2^{2946584} - 1$
llr	1.260 s	6.795 s	11.762 s	2514.362 s
llrCUDA	35.336 s	81.192 s	112.283 s	*
llrGPU	9.552 s	33.686 s	50.896 s	6857.843s

Table 1: A table of primality test timings. Each code was tested with several primes in order to compare their relative performance. llrCUDA crashed during the testing of the largest prime.

Primes	$11 \times 2^{74726} - 1$	$13 \times 2^{166303} - 1$	$13 \times 2^{233207} - 1$
llrgpu	13.948 s	44.388 s	74.007 s

Table 2: A table showing results for an in-development version of the code; the code is significantly slower due to the additional memory transfers that are required each iteration.

the worse-scaling section of the code should execute so much more quickly than the essentially linear normalisation. Of course, the memory transfers also take up a large amount of time and it would be a strong optimisation to be able to remove them from the code.

In addition to timing different sections of the same code, comparisons were made between the new GPU code and the original LLR code which is highly optimised to run on CPUs. It was also compared against the existing CUDA port known as llrCUDA; the results are in table 1.

The table indicates that LLR (using the hand-optimised functions of gwnum) is still the fastest implementation of the Lucas-Lehmer-Riesel algorithm. Nevertheless, an improvement over the current CUDA port has been obtained, which is very promising. If the GPU's power could be leveraged for the normalisation routines, not only would the memory transfers be eliminated, reducing the run time considerably, it might even prove faster than current CPU code, making it exceedingly competitive with LLR.

During development the code transitioned through many different versions, as features were developed and refined, but it is interesting to investigate the effects of memory transfers on performance. Table 4.1 shows how the first in-development version of the code performed. In the first, the data was only moved onto the device when a transform was required and was taken off the device immediately after. This resulted in a heavy load of four memory transfers per iteration and performance suffered as a result.

The version of the code featuring a CUDA normalisation function enters an infinite loop making it impossible to time.

```
==4457== Profiling application: ./llrgpu -d -q11*2^74726-1
==4457== Profiling result:
Time(%)      Time  Calls       Avg       Min       Max  Name
 32.79%  1.50685s  74690  20.174us  19.780us  20.924us  void dpVector2048D::
                                                        kernelTex<fftDirection_t=-1>
 32.59%  1.49766s  74690  20.051us  19.787us  20.795us  void dpVector2048D::
                                                        kernelTex<fftDirection_t=1>
  9.67%  444.31ms  74725  5.9450us  5.8870us  17.151us  [CUDA memcpy DtoH]
  7.59%  348.65ms  74729  4.6650us  4.4150us  11.936us  [CUDA memcpy HtoD]
  6.35%  292.04ms  74724  3.9080us  3.5150us  4.8020us  void dpRealComplex::
                                                        preprocessC2C_kernelMem
  5.35%  245.67ms  74724  3.2870us  2.7340us  6.5590us  void dpRealComplex::
                                                        postprocessC2C_kernelMem
  3.80%  174.75ms 149448  1.1690us     893ns  2.0120us  convolution_elem
  1.87%  85.802ms  74724  1.1480us     925ns  1.6960us  square_elem
```

Figure 16: Sample output from the command-line profiler. It shows that on the device, the transforms take the most computation. The average, minimum and maximum times are useful for showing that sometimes memory transfers can take much longer than the average (in this example, 17 microseconds, over three times as long as the average.

4.2 NVIDIA Profiling Tools

The CUDA toolkit comes with many additional tools as well as libraries. One such tool is the NVIDIA profiling tool, a program that is the GPU equivalent of gprof. The profiler tool will measure how long the graphics card device spends executing each function and can even suggest areas for performance improvement, by indicating whether a kernel is slowed down by memory latency, memory bandwidth and so on. In this project it was useful in proving that one of the major sources of bad performance was the time spent in device driver code waiting for memory transfers to complete, even though they took less time to execute on the device. It also showed that for the very simplest of the kernels being executed by the device, the main barrier to better performance was memory latency, since often kernels only required a few bytes of data from main memory. As a result, kernels were often waiting on memory accesses to complete before continuing execution.

It is worth noting that the profiler tools do have some associated overhead, though the tools try to account for time spent acquiring profile data in the output. However, the slowdown can make running tests on large data sets infeasible as the program takes longer and longer to run with the profiling enabled.

Figure 4.2 shows the output of the profiler (function arguments have been removed for greater clarity). It is useful to see where the device spends its time as it is difficult for the programmer to interact directly with the GPU device.

There is a GUI version of the profiler called the NVIDIA visual profiler. It attempts to show similar information in a more intuitive way with bars of colour representing what computation is taking place at any given interval. The amount of data that can be collected by the tools is huge and can help the programmer completely understand the reasons behind the performance of his or her code.

5 Further Work

5.1 More CUDA Code

In the most recent version of this project's code only the path corresponding to Lucas-Lehmer-Riesel numbers has been ported to CUDA. It was felt that porting this subset of functionality was an appropriate amount of work for this project but by no means does it have to be the end point. The changes required to be able to test other types of number range from trivial, in the case of fftwmultiply_g, to considerably more complex; for example, functional and fast code for the normalisation procedure. Moving more of the LLRP code to the CUDA platform would be an interesting avenue of research and would allow for even more types of potential primes to be tested quickly.

5.2 AMD/OpenCL Port

The CUDA programming language is owned and developed by NVIDIA and will only run on their hardware. However, approximately 33%[21] of the consumer desktop market has AMD graphics hardware installed, whose cards have shown very strong performance in a variety of software. AMD actively support the OpenCL platform on their graphics devices, a standard which is also supported by NVIDIA. AMD also have developed an accelerated mathematics library which will perform fast Fourier transforms using the resources of the GPU. While this project focussed on NVIDIA and CUDA, a port which targeted OpenCL instead is equally feasible. The choice then would be either to support the AMD APPML (Accelerated Parallel Processing Maths Libraries) or to write an OpenCL-based FFT library which could then run on any hardware that supported OpenCL. This hand-written version might perform more slowly than the vendor-supplied code, but would have the advantage of reducing code fragmentation by introducing one single code that could run anywhere.

5.3 Other Hardware

AMD are introducing new technologies which might prove useful for codes such as this. One of these is known as AMD Accelerated Processing Unit (formerly AMD Fusion), a model which presents the programmer with a unified memory space between the GPU and the CPU. It has been confirmed that Sony's soon-to-be-released PlayStation 4 console will use this hardware, allowing completely shared access to the console's 8 GB GDDR 5 (Graphics Double Data Rate, version 5) memory. This hardware would eliminate many of the difficulties faced in this project - there would be no need for costly data transfers or inefficient GPU code as the programmer could simply choose to execute on whichever hardware was more suited to the task. However, although the exact specifications are not yet known, the PS4's CPU is likely to be considerably slower than many desktop processors, instead opting to provide computational power through multiple cores. Nevertheless the concept is interesting and could prove a powerful alternative to the current model of separate host and device memory spaces.

6 Evaluation of project

Overall the project was quite successful. The core aim was to port the FFTs to CUDA and this was completely achieved. The performance available from the GPU device is exceptionally large; however, some aspects were less successful. A stated aim was to minimise memory transfers (from CPU to GPU) as they often seriously impact the performance of GPU codes. A complicated numerical routine which was highly linear in nature proved very difficult to port in not just an efficient manner but even just a basic, working code. It might have been possible to predict that the normalisation routines were the most difficult parts of the project and allocate more time to them earlier, but due to the code's structure the normalisation was seen as the last step.

Risks were largely averted in this project. The HECToR GPU machine was briefly unavailable during the project but since all development work was done on a local machine this was of no impact. There was no data loss during the project either and multiple backup copies existed but were not required. However, as mentioned, the code was not fully ported to CUDA. There was a risk that the project was infeasible and this was not the case; however, a subset of the work allocated did prove too difficult to complete, and the result is a much less performant code.

7 Conclusions

This aim of this project was to port the FFT routines of LLRP to CUDA, which was done successfully. This success is quantified as the FFT portions of the program executing in a much shorter time than the FFTW transforms. The project shows that GPUs are certainly viable for use in numerical codes such as this which hopefully will encourage further research in this area. However, due to the difficulty of porting some of the code to CUDA, the performance was severely limited by memory transfers. If better normalisation code were to be written, even stronger performance gains could be realised.

References

[1] G. F. Simmons, *1992 Calculus Gems* McGraw-Hill

[2] June 1013 | TOP500 Supercomputer Sites, http://www.top500.org/lists/2013/06/

[3] 48th Known Mersenne Prime Discovered, http://www.mersenne.org/various/57885161.htm

[4] Parallel Programming and Computing Patform | CUDA | NVIDIA, http://www.nvidia.com/object/cuda_home_new.html

[5] Accelerated Parallel Processing Math Libraries (APPML) | AMD, http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-math-libraries/

[6] OpenCL - The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/

[7] GIMPS Home, http://www.mersenne.org/

[8] PrimeGrid, http://www.primegrid.com/

[9] Seventeen or Bust: Distributed Computing, http://www.seventeenorbust.com/

[10] Riesel, Hans (1969), "Lucasian Criteria for the Primality of $N = h \times 2^n - 1$", Mathematics of Computation (American Mathematical Society) **23** (108): 869-875

[11] Schönhage, A and Strassen, V (1971), "Schnelle Multiplikation großer Zahlen", Computing **7**, pp. 281-292

[12] Crandall, Richard and Fagin, Barry (1994), "Discrete weighted transforms and large-integer arithmetic", Math. Comp. **62** pp. 305-324

[13] Free Software - GIMPS, http://mersenne.org/freesoft/default.php

[14] Genefer - Family of GFN Testing Programs, http://www.underbakke.com/genefer/

[15] Intel Instruction Set Architecture Extensions | Intel Developer Zone, http://software.intel.com/en-us/intel-isa-extensions

[16] FFTW Home Page, http://www.fftw.org/

[17] NVIDIA GF100 Whitepaper, retrieved from www.nvidia.com/object/IO_86775.html

[18] Message Passing Interface (MPI) Forum Home Page, http://www.mpi-forum.org/

[19] CUFFT | NVIDIA Developer Zone, https://developer.nvidia.com/cufft

[20] llrCUDA - mersenneforum.org, http://www.mersenneforum.org/showthread.php?t=14608

[21] Steam Hardware & Software Survey, http://store.steampowered.com/hwsurvey/