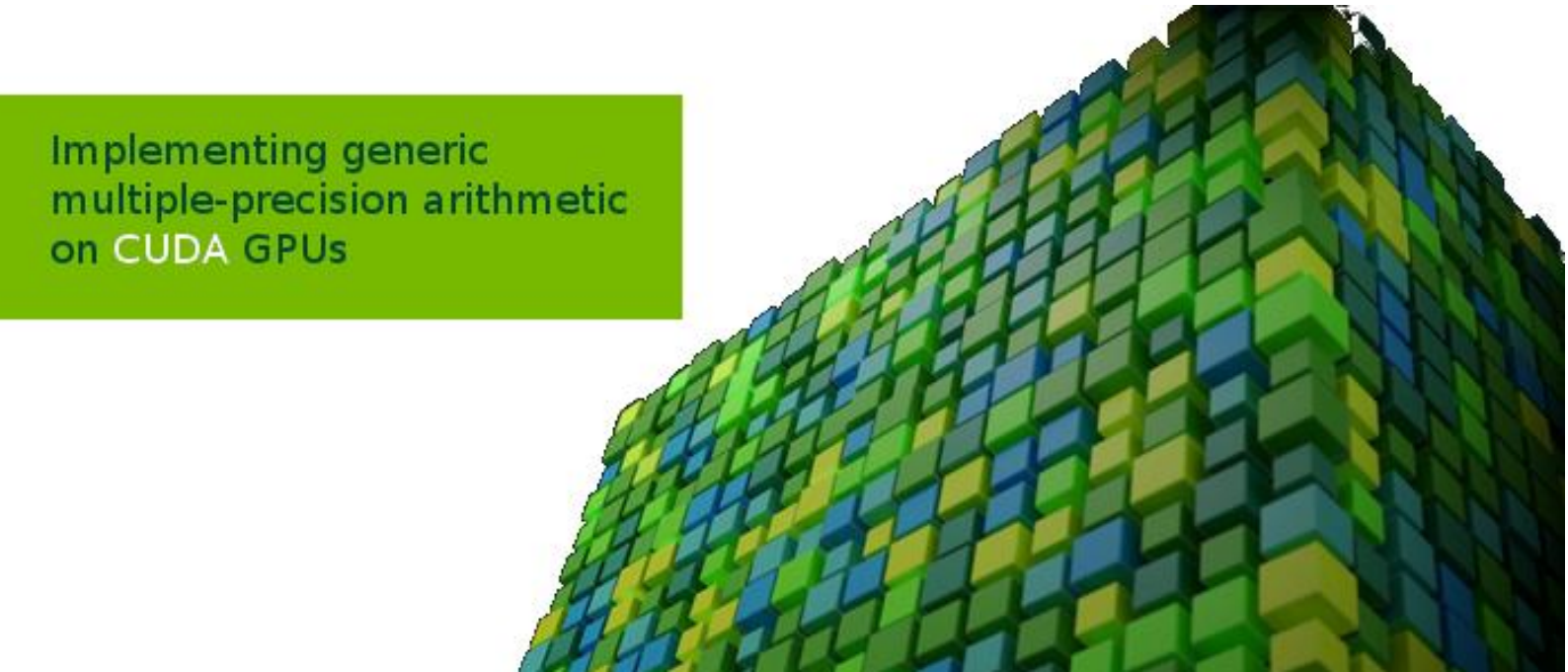
Semester Project Report

Implementing generic multiple-precision arithmetic on CUDA GPUs

Alexandre Carlessi **Sahand Kashani**

June 16, 2013



Contents

Chapter 1

Introduction

1.1 Background

For 30 years, one of the primary ways of speeding up electronic devices has been to increase CPU clock speeds. From around speeds of 1 MHz in the 1980s, clock speeds have risen to more than 4 GHz in 2013. Although increasing CPU clock speeds is by far not the *only* way to increase performance, it has been a reliable source for improvement. However, fundamental limits in the fabrication of integrated circuits makes it no longer feasible to just increase clock speeds of existing architectures as a way to gain more performance. For years, supercomputers have used another way to increase performance, which consists of performing more *parallel* work by increasing the *number* of processors used in machines.

To apply this idea to personal computers, the industry has steadily been shifting towards multi-core CPUs. This trend can easily be seen since the introduction of the first dual-core consumer CPUs in 2005, up to the current high-end 16-core workstation CPUs. As such, parallel computing is no longer a *niche* that only exotic supercomputers once claimed to perform. Indeed, more and more electronic devices have started to incorporate parallel computing capabilities as an effort to provide functionality well beyond those of their predecessors.

However, CPUs are not the first devices with parallel computing in mind, as GPUs have applied this idea earlier. A graphics processing unit, also known as a GPU, is a specialized electronic circuit initially designed for fast memory manipulations needed to accelerate the creation of images in a frame buffer, which are then outputted to a display for viewing. Nowadays, GPUs are present in almost all electronics, including, but not limited to embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs have highly parallel structures, making them much more effective than general-purpose CPUs for algorithms which process large blocks of data in parallel. Thus, GPUs have become very efficient at manipulating imagery, which consists of applying an algorithm parallely to all output pixels, which explains their abundant use in computer graphics.

In this report, we explore the implementation of a form of multiple-precision arithmetic on GPUs in order to leverage their high bandwidth capabilities.

1.2 GPU programmability evolution

GPUs were initially designed to accelerate texture mapping, polygon rendering, and geometry. The first GPUs had a fixed-function rendering pipeline, and could not be used for anything other than common geometry transformations and pixel-shading functions that were pre-defined in hardware.

With the introduction of the NVIDA GeForce 3[1] in 2001, GPUs added programmable shading to their capabilities. A programmable shader is one of a GPU's processing elements which programmers can access and for which they can write custom code. This allows developers to define their own straight-line shading programs to perform custom effects on the GPU. Bypassing the fixed-function rendering pipeline opened the door towards many future graphics novelties, such as cel shading, mip mapping, shadow volumes, oversampling, interpolation, bump mapping, and many others.

The 2 main shaders were the fragment shader (also known under the name of pixel shader), and the vertex shader (also known under then name of geometry shader). The vertex shader processed each geometric vertex of a scene and could manipulate their position and texture coordinates, but could not create any new vertices. The vertex shader's output was sent to the fragment shader for further processing. The fragment shader processed each pixel and computed its final color, as well as other per-pixel attributes by using supplied textures as inputs. Soon, shaders could execute code loops and lengthy floating point math instead of straight-line code, which pushed them to quickly become as flexible as CPUs, while being orders of magnitude faster for image processing operations. The shaders were written to apply transformations to a large set of elements parallely, such as to each pixel of the screen, or to every vertex of a 3D geometric model.

GPUs had different processing units for each type of shader, but with its introduction in 2006, the NVIDIA GeForce 8800 GTX[2] merged the separate programmable graphics stages to an array of unified processors, which allowed dynamic partitioning of the computing elements to the different shaders, thus attaining better load balancing.

The unified processor array of the GeForce 8800 GTX[3] is shown on Figure 1.1. This unified design made GPUs architecturally closer to CPUs.

1.3 Early GPGPU

1.3.1 Introduction

GPU hardware designs evolved towards more unified processors, and were getting more similar to high-performance parallel computers. Computations performed by programmable shaders mostly involved matrix and vector operations, for which GPUs were very well suited (processing blocks of data in parallel). The availability of high speed linear algebraic operations, as well as the unified procesors pushed

[1] `http://www.nvidia.com/page/geforce3.html`

[2] `http://www.nvidia.fr/page/geforce_8800.html`

[3] `http://www.clubic.com/article-61974-8-nvidia-g80-geforce-8800-gts-gtx.html`

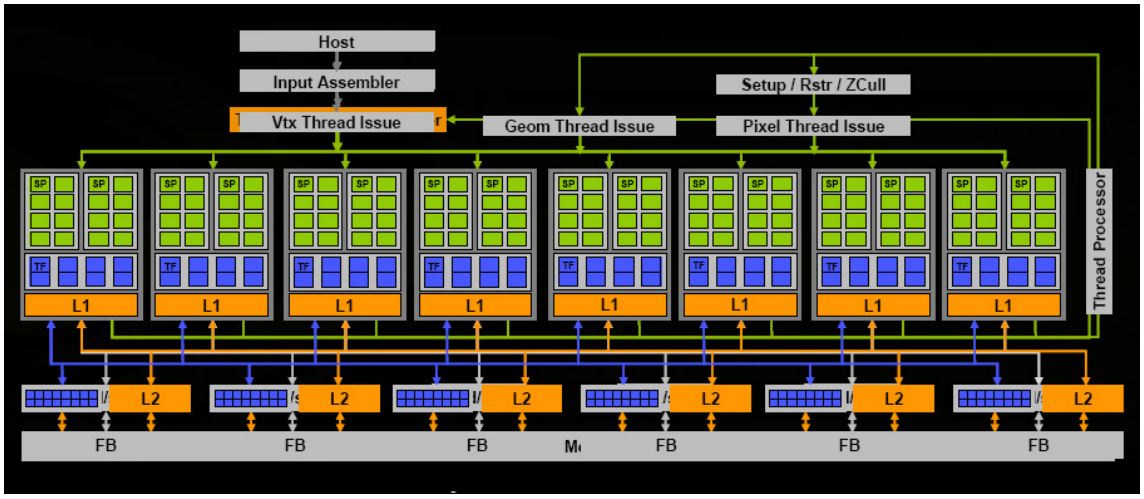


Figure 1.1: Unified programmable processor array of the GeForce 8800 GTX

scientists to start studying the use of GPUs for non-graphical calculations. This was achieved through the use of GPGPU techniques.

GPGPU, a shorthand for General Purpose computing on Graphics Processing Units, consists of using a GPU, which typically only handles computations related to computer graphics, to perform computations for applications which are traditionally handled by a CPU. Such a consideration is possible since GPUs support a functionally complete set of operations on arbitrary bits, and can thus compute any value.

At the time, programmers could only interface with GPUs through graphics APIs such as OpenGL[4] or DirectX[5]. However, APIs had been designed to only support features required in graphics. To access the GPU's computational resources, programmers had to map their problem into graphics operations so the computations could be launched through OpenGL or DirectX API calls. With this consideration in mind, programmers had to arrange their data in such a way to "trick" the GPU in performing the calculations defined in the programmers' shaders as if they were graphics calculations, whereas in reality, they were scientific computations.

1.3.2 GPGPU concepts

There are 4 main GPGPU concepts.

1. Data arrays are equivalent to GPU textures. The native data layout for CPUs is the one-dimensional array. Higher dimensional arrays are available for programmer convenience, but are actually implemented as one-dimensional arrays, and compilers use linear transformations to adapt the indices accordingly.

   On the other hand, GPUs use two-dimensional arrays as their native data layout and are, in fact, textures. To make a data array available to the GPU,

[4] `http://www.opengl.org/`
[5] `http://en.wikipedia.org/wiki/DirectX`

the CPU would need to create the data, then map it to a GPU texture which a shader could later read and process. In order to correctly use the memory available to a GPU, one needs to find a mapping between the CPU array indices, and the GPU texture coordinates. Once the mapping is done, the CPU would then transfer the data towards the GPU texture.

2. Computation code, also called a *kernel*, is equivalent to a shader. There is a fundamental difference between the computing model of CPUs and GPUs, and this impacts the way one needs to think algorithmically. GPUs follow the data-parallel programming paradigm, whereas CPUs follow the sequential programming paradigm.

   As such, CPU code is usually implemented as a loop-oriented program, since it has to iterate over all elements of a data structure, and apply a function to each one. In contrast, GPUs have highly parallel structures which can apply the same code to large blocks of data parallely, assuming that there is no dependency among the operations.

   To show the contrast in the programming model, let's compare how one would compute the addition of 2 N-element vectors and store the result in the first vector.

   Assume we have the following 2 vectors already pre-filled with their respective data:

```
1  int a[N];
2  int b[N];
```

Listing 1.1: Vector definitions

The CPU code to perform this vector addition would look like this:

```
1  for (int i = 0; i < N; i += 1)
2  {
3      a[i] = a[i] + b[i];
4  }
```

Listing 1.2: Vector addition

Note that the CPU will have to loop over all indices of the 2 arrays, and add each element one by one in a sequential way. It is important to note that each of the N computations are completely independent, as there are no data dependencies between elements in the result vector. For example, once we have computed `a[0] = a[0] + b[0]`, its result will be of no help when it comes to computing `a[1] = a[1] + b[1]`.

As such, assuming we have a computation unit with N parallel structures, we could be able to compute the vector addition without the need of any loop by assigning one vector element addition to each computation unit. This is easily done by adapting the index of the vector elements that are provided to each computation unit.

This is the core idea behind GPGPU computing: separating the identical, but independent calculations from one another, and assigning them to different

execution units which can then execute them at the same time. Algorithms are extracted into computational kernels which are no longer vector expressions, but scalar templates of the algorithm that form a single output value from a set of input values. These algorithms are implemented in shaders which will then calculate the independent computations parallely.

For the vector addition used above, the 2 vectors will have to be written into textures by the CPU, then the shader will read the appropriate elements from the texture to perform its independent computation.

3. Computations are equivalent to "drawing". Indeed, the final output of a GPU is an "image", therefore all computations have to, in some way or another, write their "results" to the frame buffer for it to be available to the programmer.

   The programmer must tell the graphics API (either OpenGL or DirectX) to draw a rectangle over the whole screen, so that the fragment shader can apply its code to each pixel independently and output an answer. If the API were not instructed to draw something that fills the whole screen, then the fragment shader's code would not be applied to all the data in the textures we created, but only to a subset of it.

   By rendering the simple rectangle, we can be sure that the kernel is executed for each data item in the original texture.

4. On CPUs, data is read from memory locations, and results are written to memory locations. On GPUs, we just saw that the final output is written to the frame buffer.

   However, a huge number of algorithms are not straight-line code, and require the GPU's result to be used as input for another subsequent computation. To achieve this on a GPU, we need to execute another rendering pass. This is achieved by writing the current result to another texture, binding this texture as well as other input or output textures, and potentially also binding another shader for the algorithm to continue. This is known as the ping-pong technique since one has to keep juggling between textures until the algorithm is done and the result is outputted to the frame buffer.

To recap all that is needed for GPGPU on graphics APIs, one needs to create data on the CPU, map it to GPU textures, write shaders to perform computations based on the data in the textures, and finally write the result back to the frame buffer.

Early GPGPU programming can quickly become quite tedious, even for simple algorithms, as one must understand the complete graphics rendering pipeline in order to trick the GPU in thinking it's performing graphics calculations, whereas the programmers are actually manipulating their data on GPU textures, and writing their kernels in shaders.

Indeed, graphics APIs were not intended for scientific computations, and are thus not easily programmable. In order to fully benefit from the parallel processing power of GPUs without having to know anything about the graphics rendering pipeline, more computational oriented languages were created.

Chapter 2

NVIDIA CUDA

The Compute Unified Device Architecture, more commonly known under the name CUDA[1], is a parallel computing platform and programming model developed by NVIDIA in 2006, and implemented by the GPUs they produce.

CUDA was designed for GPGPU programming, as developers can compile C code for CUDA capable GPUs, thus avoiding the tedious work of mapping their algorithms to graphics concepts. Essentially, CUDA's main advantage is that developers have explicit access to the GPU's virtual instruction set, as well as its device memory. By using CUDA, developers can use GPUs in a similar way as CPUs, without having to know anything about the graphics rendering pipeline.

CUDA also exposes several GPU hardware features that are not accessible through graphics APIs, the most important of which is access to GPU shared memory, an area of on-chip GPU memory which can be accessed in parallel by several blocks of threads. CUDA also supports a thread synchronization primitive, allowing cooperative parallel processing of on-chip data, greatly reducing the high-latency off-chip bandwidth requirements of many parallel algorithms.

2.1 CUDA Program Structure

A CUDA program consists of multiple interleavings of CPU code segments, and GPU code segments. CPU code is called *host* code, whereas GPU code is called *device* code. The segments that exhibit little data parallelism are implemented as host code, whereas the data parallel segments are implemented as device code.

All the code is written in ANSI C extended with keywords for labeling data- parallel functions called *kernels*, and their associated data structures. The compilation process separates the host code from the device code, passing the host code to the host's standard C compiler, and the device code to the NVIDIA C compiler (nvcc).

In CUDA, computations are carried out by *threads*, a large number of which are generated by kernels to exploit data parallelism. Kernels specify the code to be executed by *all* threads during a parallel segment. Since all threads execute the same code, the CUDA programming model follows the *SIMT* (Single Instruction Multiple Thread) programming style. A representation of the CUDA program execution flow is shown in Figure 2.1.

[1] `http://www.nvidia.com/object/cuda_home_new.html`

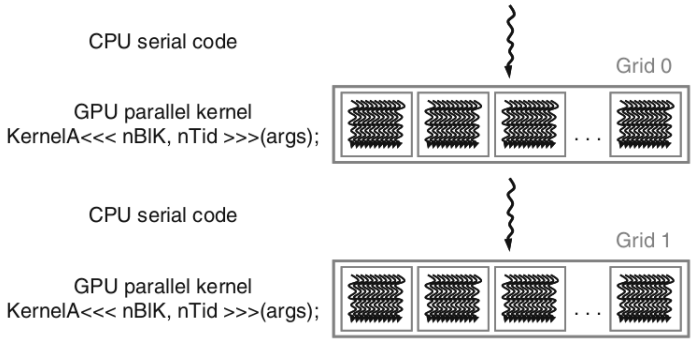


Figure 2.1: Cuda program execution phases

2.2 CUDA thread organization

When a kernel is launched, a *grid* of parallel threads are executed. Threads in a grid are organized into a two-level hierarchy, as shown in Figure 2.2. A grid consists of one or more thread blocks, each of which contain the same number of threads. Each thread block has a unique 1D, 2D, or 3D block identifier (note that for simplicity, a 2D identifier is drawn in Figure 2.2). Similarly, each thread within a block has a unique 1D, 2D, or 3D thread identifier.

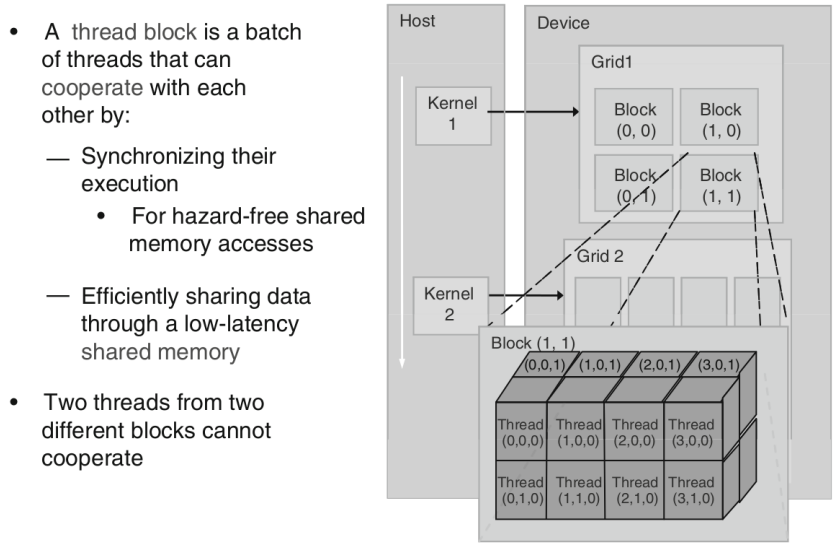


Figure 2.2: Two-level CUDA thread organization

Once a kernel is launched, the CUDA runtime generates the corresponding grid of threads, which are then assigned to execution resources on a block-by-block basis. To have some numbers, NVIDIA's Fermi[2] (compute capability 2.0) and Kepler[3] (compute capability 3.0) architectures can have a maximum of 1024 threads concurrently running in a block.

CUDA execution resources are organized into streaming multiprocessors (SMs), two of which are shown in Figure 2.3. A maximum number of blocks can be assigned

[2] `http://www.nvidia.com/object/fermi-architecture.html`
[3] `http://www.nvidia.com/object/nvidia-kepler.html`

to each SM (8 on Fermi GPUs, and 16 on Kepler GPUs) as long as there are enough resources to satisfy the needs of all the blocks. If any of the resources needed for the simultaneous execution of the blocks are unavailable, less blocks will be scheduled for execution on the SM. The remaining blocks will execute once the resources needed are available again.

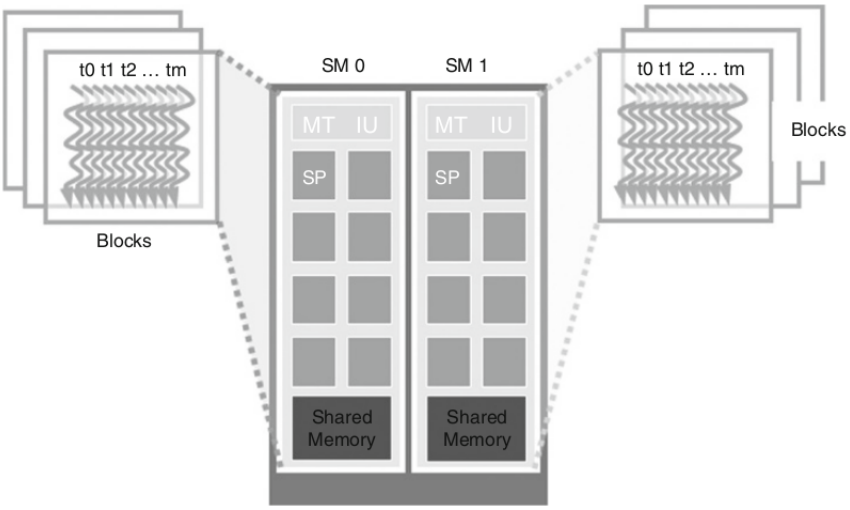


Figure 2.3: SM thread block assignment

Although blocks are scheduled to be run on an SM, it is the threads of that block that execute the computations. All threads of a block are scheduled for execution in structures called warps, each of which contain 32 continuous threads (identified by their $threadIdx$ values).

A crucial aspect about warps is that the hardware executes an instruction for all threads in the same warp *before* moving to the next instruction. This works well when all threads within a warp follow the same control flow path when working on their data. For example, *if-then-else* style branch statements work well when either all threads take the *then* statement, or all the threads take the *else* statement. If some threads execute the *then* part, and others execute the *else* part, the SIMT execution model no longer works and the warp will require multiple execution passes through the divergent paths, with one pass for each divergent path. These passes occur sequentially, thus increasing the execution time. The situation is even worse for *while* loops, since the each thread could potentially loop a different number of times compared to the others, therefore it is very important to try and keep thread divergence low.

2.3 CUDA memory structure

In CUDA, the host and devices have separate memory spaces, as GPUs are typically hardware cards that come with their own DRAM. In order to provide data to a kernel, memory needs to be allocated on the device, and data has to be transferred to the allocated memory. Similarly, after kernel completion, device results must be

transferred back from device memory to host memory. CUDA devices expose several different memories to developers, some of which are shown on Figure 2.4[4].

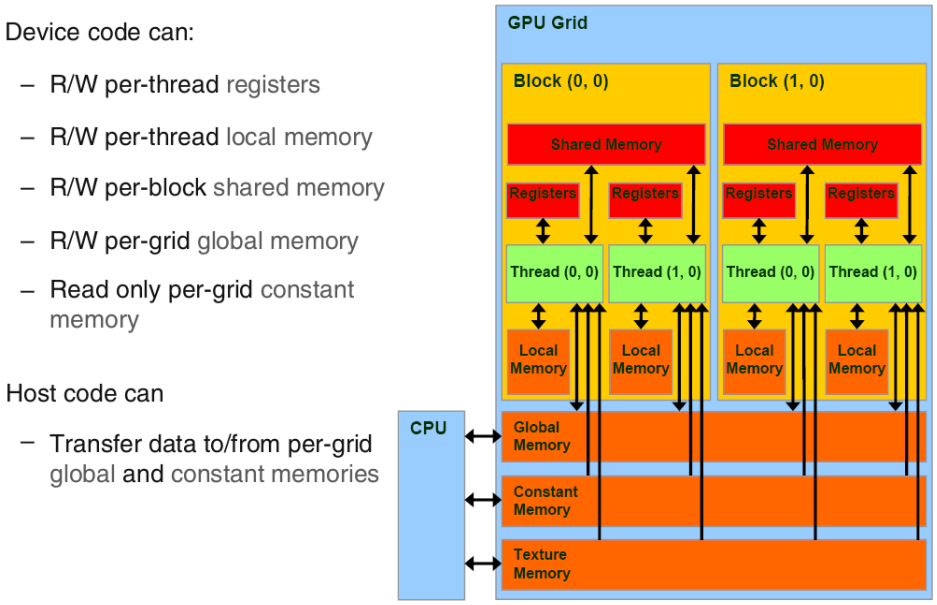


Figure 2.4: CUDA memory hierarchy

At the bottom of the figure, we see *global* memory, and *constant* memory, the 2 off-chip memories available on a CUDA GPU. Global memory, typically implemented as DRAM, can be written to, and read from the host. Because of its implementation technology, global memory suffers from long access latencies, and finite access bandwidth. In contrast, constant memory supports short-latency, high-bandwidth, read-only access by the device when all threads simultaneously access the same location. Registers and shared memory are on-chip memories, and are thus accessible at very high speeds, and in a parallel way.

2.4 Maximizing global memory bandwidth

Because of DRAM's high access latency, global memory is organized in such a way that when reading a certain location, several consecutive memory locations are returned. As such, if an application can make use of multiple consecutive global memory locations before moving to other locations, the DRAMs can supply the data at a much higher rate than if random locations are accessed.

When all threads in a warp execute a load instruction, the hardware detects if the threads are accessing consecutive global memory locations, and if it is the case, then it does not issue multiple separate load instructions, but will combine, or *coalesce* them into less load instructions. To achieve anywhere close to the peak advertised global memory bandwidth, it is important to take advantage of global memory coalescing, by organizing data in memory in such a way that each thread

[4] `http://lenam701.blogspot.ch/2012/02/nvidia-cuda-my-experience-with-high.html`

can read the data it needs at the same time as the other threads without requiring separate loads.

2.4.1 Example

Suppose 4 threads are trying to read a `4x4` matrix `m[4][4]`:

m[0]	m[1]	m[2]	m[3]
m[4]	m[5]	m[6]	m[7]
m[8]	m[9]	m[10]	m[11]
m[12]	m[13]	m[14]	m[15]

Normal CPU code for accessing such a matrix would ressemble the following:

```
1  for (int i = 0; i < 4; i++)
2  {
3      for (int j = 0; j < 4; j++)
4      {
5          m[i][j] = ... ;
6      }
7  }
```

Listing 2.1: Accessing matrix elements on a CPU

If we use 4 GPU threads, we get the following code:

```
1  for (int j = 0; j < 4; j++)
2  {
3      m[threadIdx.x][j] = ... ;
4  }
```

Listing 2.2: Accessing matrix elements on a GPU (non-coalesced version)

This matrix is stored in memory as a continuous 1D array of data, and accessing the data on the GPU with 4 threads parallely would look like this (first loop iteration):

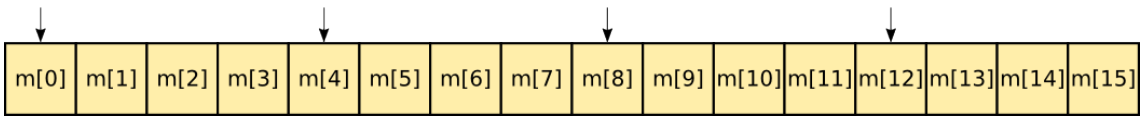


Figure 2.5: Non-coalesced memory access

Note that each thread tries to load its "line" at the same time as the others, resulting in 4 scattered global memory reads. What we would want to have, is reads of the following form:

The code corresponding to the memory access pattern in Figure 2.6 is:

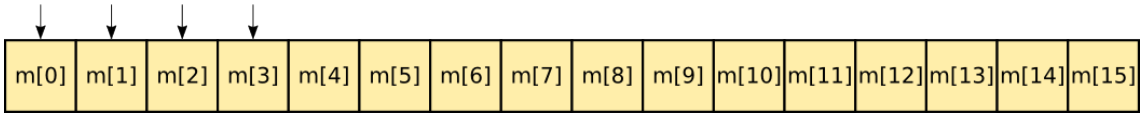


Figure 2.6: Coalesced memory access

```
1  for (int j = 0; j < 4; j++)
2  {
3      matrix[j][threadIdx.x] = ... ;
4  }
```

Listing 2.3: Accessing matrix elements on a GPU (non-coalesced version)

But then, each thread would no longer be accessing the element it initially wanted, as all threads in Figure 2.6 are accessing thread 1's "line". The solution to this problem is to transpose the initial matrix, thus yielding the correct memory access pattern, as well as the minimum number of global memory accesses:

m[0]	m[4]	m[8]	m[12]
m[1]	m[5]	m[9]	m[13]
m[2]	m[6]	m[10]	m[14]
m[3]	m[7]	m[11]	m[15]

Figure 2.7: Transposed matrix

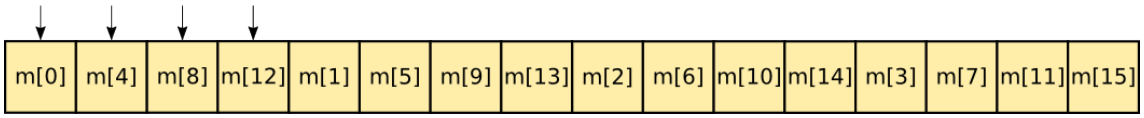


Figure 2.8: Correct coalesced memory access

Therefore, a "simple" port of CPU code is to transpose data arrays, since concurrent GPU threads can load columns more efficiently. One must strive for perfect per-warp memory coalescing by aligning starting addresses (may need padding for this), and accessing continuous memory regions, all in order to reduce global memory accesses, and to maximize bandwidth.

2.5 Performance summary

So, in order to make sure to take the most advantage of execution units, the following properties must hold:

1. Make sure *all* global memory reads and writes are coalesced whenever possible. If memory coalescing is not done, then all other so called "optimizations" would mostly be insignificant. In case global memory accesses are difficult to coalesce, it is better to try and use 1D texture lookups instead, as they are more suited for scattered access patters.

2. If any data is to be common between threads, do not use global memory with locks, but rather try to use shared memory as much as possible, since it is much faster.

3. Minimize the use of all divergent branches, or at least make them the shortest possible.

Chapter 3

Implementation

3.1 Data representation

3.1.1 CPU vs GPU representations

Nowadays, fields such as cryptography, require the ability to perform computations on very large integers. In C, multiple precision arithmetic is not built into the language, therefore one must use external libraries to have the feature. Until now, one of the reference libraries has been the GNU Multiple Precision Arithmetic Library, more commonly called GMP[1]. GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. GMP is a reference, because it is carefully designed to be as fast as possible, both for small operands, and for big operands by using fullwords as the basic arithmetic type, fast algorithms, and highly optimized assembly code.

GMP keeps integers in a structure called `mpz_t`. Each `mpz_t` structure stores numbers under sign and magnitude representation, which involves keeping the number's sign bit in a field, separate from its absolute value, which is contained in another field. The number's absolute value is represented as an array of unsigned integers. Each `mpz_t` may be composed of a different number of machine words, in order to maintain maximum memory efficiency.

It is important to note that while GMP uses *sign and magnitude* representation for its integers, native machine integers are stored in another form called *two's complement* representation. Two's complement notation has many advantages at the hardware level, but its main drawback is that it is bit-length dependent, and is thus impossible to use for multiple precision arithmetic, for which one does not know in advance what the precision of its operands are going to be.

By examining the source code of some operations over `mpz_t`, we can easily see that its efficiency comes from its ability to take care of all "corner cases" appropriately. This is done by using a huge number of branches in order to steer the execution towards the most efficient way to perform the computation. For example, let's briefly examine the GMP multiplication code. The GMP multiplication code

[1] `http://gmplib.org/`

performs the following:

1. Checks if one operand is bigger than the other, switching operand pointers and sizes if it is the case.
2. Checks if native long multiplication is supported by the processor.
3. Checks if the bigger operand is less than 2 machine words, in which case it decides to perform the "naive" schoolbook multiplication instead of asymptotically better algorithms, such as the Karatsuba-Offman algorithm, or the Toom-Cook algorithm, since their overhead makes them less efficient at small operand sizes. This operation is one machine instruction if the processor supports long multiplication natively, otherwise multiple instructions are used.
4. If the operands are not bounded by 2 machine words, then further tests are carried out so as to determine which multiplication algorithm should be used to compute the result more efficiently.
5. Once the computation is performed, the result is returned.

Of course, several memory management stages are interleaved within the computations to maintain space efficiency. Many more tests take place in the code, but the most important ones are listed above.

GMP has worked well until now, since it has been running on CPUs, which support very efficient sequential execution pipelines. However, if we were to use GMP directly on GPUs, program execution speed may be very disappointing. In section 2.2, we wrote about the SIMT execution model GPUs employ, and the importance for each thread to execute the same instruction. However, as seen previously, GMP code contains deepy nested branches, leading to high potential for divergent branches during execution. If one were to launch thousands of threads, each of which executes the GMP code for multiplication, in which each thread has a potential chance to follow a divergent branch, then performance would be disappointing.

In order to support efficient arithmetic on GPUs, we need to guarantee that each thread follows the most divergent-free path possible. Indeed, GPUs are *throughput*-oriented devices, whereas CPUs are *latency*-oriented devices. CPUs can have highly optimized code which can decide how to compute *each* single instance the most efficiently possible, whereas GPUs must provide more general algorithms which will compute *all* threads parallely, independently of the fact that each individual thread *could* have executed the computation in a more efficient way. It is a GPU's architecture which defines this behaviour. Therefore, the main idea to grasp here is that one cannot optimize operations on a thread-by-thread basis.

3.1.2 Our bignum representation

Our initial project idea was to work towards solving the 131-bit Certicom ECC challenge[2]. The challenge is to compute ECC private keys from a given list of

[2] `http://www.certicom.com/index.php/the-certicom-ecc-challenge`

ECC public keys and associated system parameters. This is the type of problem an adversary would face when attempting to defeat an elliptic curve cryptosystem.

This field uses *fixed*-precision modular arithmetic for computations, therefore it does not explicitly need dynamic runtime-level multiple precision arithmetic. This means that once a specific precision is chosen for the Certicom challenge, *all* operations will be bounded by a certain size constraint. For example, supposing we choose to solve the 131-bit Certicom challenge, we will know in advance that integers are going to be at most 131 bits in length. With this information, we can implement algorithms which operate on this exact precision the most efficiently possible. Even if we were to have to represent a 2-bit integer, we would use 131 bits of storage in order to make sure the representation is consistent between all threads. Note that we only need to be able to represent integers consistently, as ECC cryptography does not use any floating point values.

Unlike general multiple precision libraries like GMP, we decided not to use a sign and magnitude representation for our integers, since we have an upper bound on the precision of our operands. Thus, we decided to store our numbers in two's complement representation. The advantage of two's complement representation is that the fundamental arithmetic operations of addition, subtraction, and multiplication are identical to those for unsigned binary numbers (as long as the inputs are represented in the same number of bits).

In order to take the most advantage of a device, it is useful to represent data in its primary data type, which consists of 32-bit fullwords for CUDA GPUs. We will store our numbers as arrays of these fullwords, so as to have the most compact notation possible. In order to store X-bits using 32-bit words, one would need at least $\lceil \frac{X}{32} \rceil$ words. For 131-bit numbers, this means we need 5 words to store each number.

Figure 3.1 shows a possible number representation for the 131-bit number `0x00000004 ADBFB00E 372139F3 5E2503DD B65B9045`, assuming it is stored in an array named `a`.

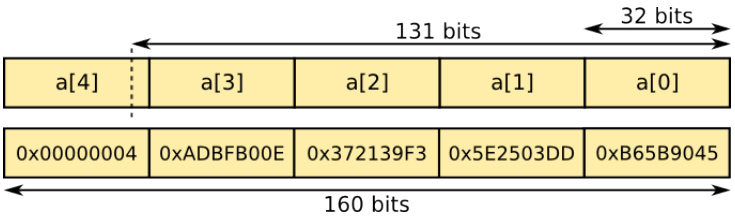


Figure 3.1: Most significant word first representation (MSW)

However, for convenience, we decided to store our numbers with their least significant word first, as all algorithms start from the least significant words and continue on from there. This would spare us the need to do constant index manipulations. Figure 3.2 shows this alternative representation.

Note that, since we are using two's complement notation, a negative number would be represented as shown on Figure 3.3, with several leading 1s in its MSW.

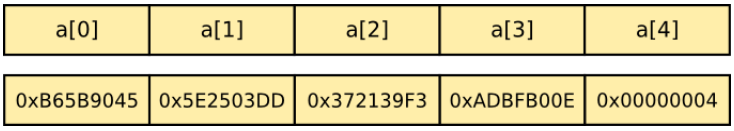


Figure 3.2: Least significant word first representation (LSW)

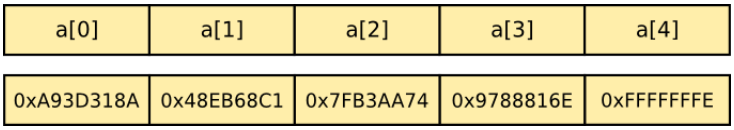


Figure 3.3: Negative number representation

3.2 A note about benchmarks

In order to benchmark our operations, we prepared several different operand arrays, each containing $1024 * 1024$ random bignums. Each operand file was generated for 5 different precisions:

- 109-bit
- 131-bit
- 163-bit
- 191-bit
- 239-bit

Throughout this document, all benchmarks consist of performing 10 consecutive executions of each operation. The amount of random numbers used in the benchmark depends on the total numbers of threads spawned by the kernel launch configurations, and is equal to $gridsize * blocksize$. For example, if we choose to launch a kernel with 1 block and 1 thread, then only 1 operand will be read from each file, but if we choose to launch a kernel with 16 blocks and 32 threads, then 512 operands will be read from each file. We only execute kernels with launch configurations consisting of power of 2 grid sizes and block sizes, so the available launch configurations are `{1, 2, 4, 16, 32, 64, 128, 256, 512, 1024}`.

We chose this way of benchmarking, because we can set the total amount of "work" the GPU must perform, and test what launch configurations work best for warp occupancy. For example, suppose we want to test what launch configurations lead to the best performance for a workload of 64 bignums. We can try to launch the kernel with the following launch configurations and see which one gives the best results:

- 1 block of 64 threads
- 2 blocks of 32 threads
- 4 blocks of 16 threads

- 8 blocks of 8 threads
- 16 blocks of 4 threads
- 32 blocks of 2 threads
- 64 blocks of 1 thread

3.3 Assembly vs. C

In order to implement arithmetic over multiple words, one needs be able to perform carry propagation. For example, in order to add two 5-word bignums `a` and `b`, each word in `a` must be added to the corresponding word in `b`, yielding a result, and a potential carry bit. In turn, this carry bit must be propagated to the next addition involving the next word of both `a`, and `b`.

We will now describe 2 different ways of accessing the carry bit, and explain which version we chose to work with, along with the reasons around the decision.

C

In C, we do not have access to processor carry flags. Nevertheless, it is possible to know if a carry was generated by a simple trick. When a carry flag is generated, it means that the result of an operation is too big to hold on 32 bits. This flag can only be generated, if, prior to the operation, there was a binary `1` in the leading bit of at least one of the 2 operands, such that the addition of this leading binary `1` with a potential other `1` in the computation causes the result to overflow. An illustration is provided in Figure 3.4.

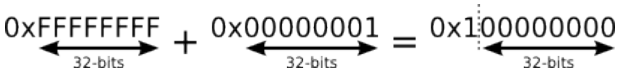


Figure 3.4: Overflow on hexadecimal numbers

The trick to notice here is that if there was no overflow, then the lower 32 bits of the result would be *bigger or equal* to the lower 32 bits of the biggest of the 2 operands. However, if an overflow occurred, then the lower 32 bits of the result is *smaller* than the lower 32 bits of the the biggest of the 2 operands. Therefore, in order to get the carry flag for the next computation in C, one would compute `carry = (c[i] < max(a[i], b[i]))`, then use it for the *next* addition, like `c[i+1] = a[i+1] + b[i+1] + carry`. Listing 3.1 shows the carry propagation technique for N-word additions.

```
1 c[0] = a[0] + b[0];
2 for (uint32_t i = 1; i < N; i++)
3 {
4     c[i] = a[i] + b[i] + (c[i-1] < max(a[i-1], b[i-1]));
5 }
```

Listing 3.1: N-word addition in C

Assembly

CUDA GPUs expose a low-level Parallel Thread Execution (PTX)[3] virtual machine and instruction set architecture which allows one to program in pseudo-assembly language. PTX assembly supports a large set of operations, such as `add`, `sub`, and `mul`, but in addition, at the time of this writing, it supports 6 extended-precision integer arithmetic instructions (organized into 3 categories), which are listed below.

1. `add.cc` (addition with carry-out), `addc` (addition with carry-in)
2. `sub.cc` (subtraction with borrow-out), `subc` (subtraction with borrow-in)
3. `mad.cc` (multiply-add with carry-out), `madc` (multiply-add with carry-in)

The special aspect of these instructions is that they reference an implicitly specified condition code register having a single carry flag bit, which can hold carry-in, carry-out, borrow-in, or borrow-out flags. Another important fact is that the condition code is *not* preserved across calls, and is only intended for use in straight-line code sequences for computing extended-precision integer addition, subtraction, and multiplication. PTX can be written in C programs as inline assembly, thus allowing one to write kernel sections involving normal arithmetic in C, and writing the extended-precision parts in PTX.

As an example, in order to calculate the extended-precision addition of two 131-bit numbers `a` and `b`, and store the result in `c`, one would use the following inline PTX code (remember that `a`, `b`, and `c` are 5-word arrays):

```
1 asm("add.cc.u32  %0, %1, %2;" : "=r"(c[0]) : "r"(a[0]), "r"(b[0]));
2 asm("addc.cc.u32 %0, %1, %2;" : "=r"(c[1]) : "r"(a[1]), "r"(b[1]));
3 asm("addc.cc.u32 %0, %1, %2;" : "=r"(c[2]) : "r"(a[2]), "r"(b[2]));
4 asm("addc.cc.u32 %0, %1, %2;" : "=r"(c[3]) : "r"(a[3]), "r"(b[3]));
5 asm("addc.u32    %0, %1, %2;" : "=r"(c[4]) : "r"(a[4]), "r"(b[4]));
```

Listing 3.2: 5-word hand unrolled addition in PTX

In the above addition code, notice that the assembly instructions are identical for the middle section, as the same operation is taking place. One would be tempted to write the code in a loop instead, such as the following:

```
1 asm("add.cc.u32  %0, %1, %2;" : "=r"(c[0]) : "r"(a[0]), "r"(b[0]));
2 for (uint32_t i = 1; i < 4; i++)
3 {
4     asm("addc.cc.u32 %0, %1, %2;" :
5         "=r"(c[i]) : "r"(a[i]), "r"(b[i]));
6 }
7 asm("addc.u32    %0, %1, %2;" : "=r"(c[4]) : "r"(a[4]), "r"(b[4]));
```

Listing 3.3: 5-word loop-oriented addition

The problem with this code is that its result is unpredicatable, since *carry flags are not preseved across calls*, and the loop structure created in C around the assembly will *sometimes* make us lose the carry bit. We initially thought a potential solution to this problem would be to make the compiler unroll the loop (since we know the number of loop iterations) with the `#pragma unroll` directive, as doing so would give us straight-line assembly code, therefore preseving the carry bit.

[3] `http://en.wikipedia.org/wiki/Parallel_Thread_Execution`

```
1 asm("add.cc.u32  %0, %1, %2;" : "=r"(c[0]) : "r"(a[0]), "r"(b[0]));
2 #pragma unroll
3 for (uint32_t i = 1; i < 4; i++)
4 {
5     asm("addc.cc.u32 %0, %1, %2;" :
6         "=r"(c[i]) : "r"(a[i]), "r"(b[i]));
7 }
8 asm("addc.u32    %0, %1, %2;" : "=r"(c[4]) : "r"(a[4]), "r"(b[4]));
```

Listing 3.4: Compiler unrolled 5-word loop-oriented addition

However, extensive testing showed that this approach does not guarantee that the carry is preserved either. The only solution which works well with inline assembly is that the code be fully hand-unrolled, such as in Listing 3.2.

Why we chose assembly over C

This project was initially about implementing multiple precision arithmetic in order to be able to implement Pollard's Rho algorithm, and to study the effect of warp occupation on performance. Since the arithmetic used in this algorithm consists of modular operations, we knew in advance that we only needed operations to support a certain precision. So, the problem is defined as trying to implement operations the most efficiently possible, by knowing the precision of the operands in advance. The basic principle we used for our implementation was to make the operations work for a specific precision, then try to generalize it from there, in order for it to work on different ones.

At first, we were trying to implement one pure C function per operation. These functions would take the precisions of their operands as inputs, then execute the operation to the best they could. This was easy to do for addition and subtraction, because they use very regular algorithms and could be implemented by a simple loop along with the carry propagation trick.

Problems started to arise once we took on the multiplication implementation. Indeed, for the multiplication to work efficiently, we sometimes needed to test a condition with respect to the operand precisions, in order to know if some element even needs to be evaluated. For example, we know that the multiplication of two N-bit numbers can yield a number of at most $2N$ bits. One can also generally say that the multiplication of two k-word numbers can yield a number of at most $2k$-words. However, there are some cases where optimizations could be performed. For example, 131-bit numbers are representable on 5 words, and their multiplication could yield numbers of at most $2 * 131 = 262$ bits in size. But, 262-bit numbers can be represented on $\lceil \frac{262}{32} \rceil = 9$ words instead of $5 * 2 = 10$ words. An optimization one could do here is to test if the size of the multiplication result does need 10 words, and to perform the computation only if necessary. Since all threads are performing computations on operands of the same precision, the test will not cause any thread divergence, and all threads would perform the "optimization". However, it is wasteful to perform a test if all threads are going to respond identically to it, as it becomes equivalent to pure overhead.

What would be best is if one could have precision-specific code at compile time, so that each thread would only be executing straight-line code, and would not have any branches to decide upon. It is possible to do this with the C preprocessor,

however there are some other things we would like to do that are not possible with the preprocessor, such as code loops ...

The solution we came up with was to use a form of metaprogramming by having pre-compilation scripts which would, in function of the precision provided, proceed to "create" the code for each operation. The scripts would apply all optimizations needed for the specific precision, and one would be left with functions which could perform the operations as straight-line code.

One important note about our scripts is that they do not work for *all* given precisions. What we wanted was for the scripts to create functions that will perform computations for the user as a black box which simply returns the exact answer. In C, we do not have the guarantee that the result of an addition is exact (it is possible an overflow will occur), however, our scripts choose to *not* handle precisions for which unexact answers would be possible by making sure that *addition* results would *not* need more storage than its operands. For example, our script will not work on 128-bit numbers, because the addition of two 128-bit numbers will potentially give a 129-bit number, which in turn would require 1 more word to be represented. Therefore, our scripts will not work for precisions which are multiples of 32. We took this decision to be sure users could use the same array size to hold addition or subtraction operands and results, as well as modular addition or subtraction operands and results. They would not need to keep track of separate arrays, each of different sizes, in order for each operation to have enough precision for its result.

Something that was very important for us was that the scripts be the most generic possible, so others could potentially re-use its output for future projects, since it is a bit wasteful to have to implement arithmetic again and again for each new precision one chooses. One could then focus on their application, instead of the "basic" arithmetic, which turned out to be not so basic in the end.

However, through experimentation, we have to say what C achieves to do better than our assembly statements.

- By choosing to use PTX assembly, we have limited ourselves to CUDA-capable GPUs only, instead of all the AMD GPUs also available.
- Compiler technology has advanced a lot and it is way ahead of us when it comes to optimizations, as it can more aggressively tune C code compared to our assembly, sometimes leading to less register usage. The performance difference between the C and assembly implementations are negligeable. We performed a simple benchmark to estimate the difference in execution speed, and the results for different grid and block sizes are shown in Figure 3.5.

  As one can see, the difference between the 2 versions is negligable. Figure 3.6 shows the difference in register usage between the 2 versions.
- C code would be more portable than our assembly statements. It would also be safer to use, and much easier to debug (this project made us learn this the hard way.)

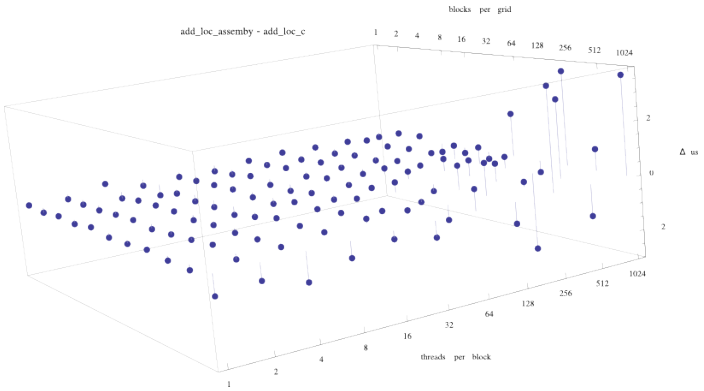


Figure 3.5: Assembly vs. C 131-bit addition execution time difference (local vectors)

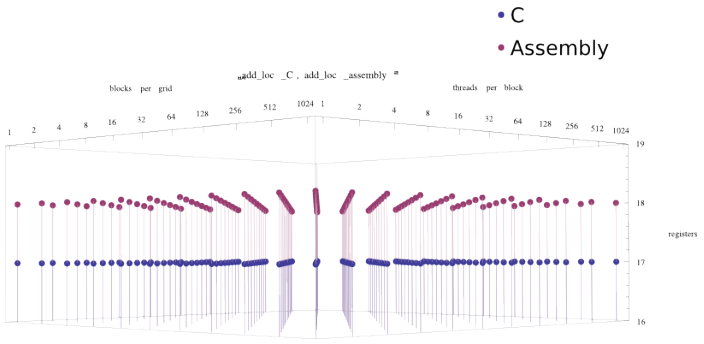


Figure 3.6: C vs. assembly 131-bit kernel register usage (local vectors)

3.4 Interleaved vs normal operand layout

Since we are working with binary operators, we tried to find out if some alternative memory layout would be beneficial to reduce global memory latency.

We tried 2 memory layouts, the first having operands interleaved, whereas the second had separate arrays for each operand.

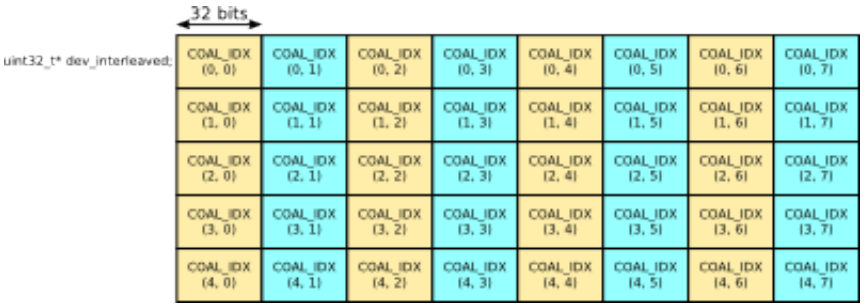


Figure 3.7: Interleaved bignum array layout

We initially though the interleaved technique was going to be better, as the compiler would know that the 2 operands are on the same memory line, and the GPU could read both operands with one global memory read. However, when compiling to PTX, we see that there are actually 2 separate load instructions performed one after the other, and no optimization was performed by the compiler. Benchmarking

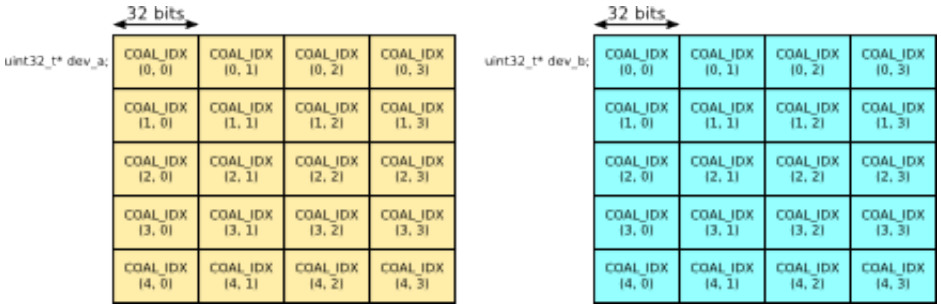


Figure 3.8: Normal bignum array layout

the 2 layouts showed that having the operands in 2 different memory locations was better.

3.5 Project File Structure

There are 2 main folders in our project, the `scripts` folder, which contains all our python scripts, and the `src` folder, which contains all the C code for actual testing on the GPUs.

3.5.1 Scripts folder

The `script` folder contains the following files:

`constants.py` This is the central control point for the whole project, as it contains all the constants used in the project, and influences the code that is generated. An extract of the most important constants is given in Listing 3.5, along with a short explanation. Some of the constants defined in this file are needed by the C code, therefore constants.py modifies all C files such that they contain all the up-to-date values it contains.

```
1  import math
2
3  precision = 239
4  bits_per_word = 32
5  blocks_per_grid = 1
6  threads_per_block = 1
7  file_name_operations_h = r'../src/operations.h'
8  number_of_bignums = threads_per_block * blocks_per_grid
9  min_bignum_number_of_words = math.ceil(precision/bits_per_word)
10 max_bignum_number_of_words = math.ceil((2*precision)/bits_per_word)
11
12 assert min_bignum_number_of_words == math.ceil((precision+1)/bits_per_word)
```

Listing 3.5: Important constants in constants.py

`precision` Precision the generated operations should support. Any value is supported here, except all multiples of 32, for reasons specified in section 3.3. This condition is asserted at the end of the file.

`threads_per_block` Number of threads contained in a grid block. Supported values are anything from 1 up to 1024, as the Kepler and Fermi GPUs do not currently support any larger number.

`blocks_per_grid` Number of blocks contained in a grid. Supported values are anything from 1 up to 1024, as the Kepler and Fermi GPUs do not currently support any larger number.

`bits_per_word` Primary data type of the device. This field is useful if GPUs were to move from their current 32-bit primary data type to a higher precision one. It causes all the code generation algorithms to adapt accordingly.

`min_bignum_number_of_words` This field represents the minimum number of words that a bignum needs for its representation. No bignum will ever use less storage place than this value.

`max_bignum_number_of_words` This field represents the maximum number of words that a bignum needs for its representation. This value is used for storage requirements of the result a multiplication yields. No bignum will ever use more storage place than this value.

`number_of_bignums` This field represents the total number of threads that are being launched.

`file_name_operations_h` This field tells the script which file it should generate all the functions to.

`conversions.py` This file handles the conversions of numbers from Python's internal unlimited-precision integer arithmetic to our GPU's number representation, which is 2's complement notation. The functions in this file are mostly used to verify that the results of our GPU operations are correct. Integers are kept as Python's basic signed integers, however, all binary and hexadecimal values are kept as strings, as we have to transform them into two's complement form. Signed integers are converted to two's complement by getting their binary string in signed and magnitude form. If it is a positive number, then it is simply returned. Otherwise, we apply the following forumla to get the representation of the negative version of the number's absolute value (let's call it B): $-B = \bar{B} + 1$.

`input_output.py` This file defines functions for writing bignums to a file in coalesced form, as well as functions for reading coalesced bignums from a file, and reconstructing their integer value from the two's complement representation.

`operation_checker.py` This file verifies if the bignums the GPU computed are correct.

`operation_generator.py` This is the main file of the project which generates our functions. Note that the generated *macros* are put in the file defined by `file_name_operations_h` in `constants.py`.

`random_number_generator.py` This file generates random numbers of the precision specified by `precision` in `constants.py`.

3.5.2 Src folder

The `src` folder contains the following files:

`constants.h` This file is the central control point for the C part. It contains some of the fields that were defined by `constants.py`, which are needed in order to know where to write results to, how much memory needs to be allocated, ...

The important fields are listed below:

`THREADS_PER_BLOCK` Equivalent to `threads_per_block` in `constants.py`.

`BLOCKS_PER_GRID` Equivalent to `blocks_per_grid` in `constants.py`.

`NUMBER_OF_BIGNUMS` Equivalent to `number_of_bignums` in `constants.py`.

`bignum_types.h` This file contains the definitions `MIN_BIGNUM_NUMBER_OF_WORDS`, and `MAX_BIGNUM_NUMBER_OF_WORDS`, which are equivalent to their counterparts defined in `constants.py`. It also defines 2 important macros needed in order to know where a given bignum is located in memory. These macros are `IDX(i, j, is_long_number)` and `COAL_IDX(i, j)`, and are explained below:

```
1 #define IDX(i, j, is_long_number) (((i) * ((is_long_number) ? (
      MAX_BIGNUM_NUMBER_OF_WORDS) : (MIN_BIGNUM_NUMBER_OF_WORDS))) + (j))
2 #define COAL_IDX(i, j) (((i) * (NUMBER_OF_BIGNUMS)) + (j))
```

Listing 3.6: bignum_types.h

`IDX(i, j, is_long_number)` This macro allows one to select bignums from a *non*-coalesced array. Input `i` must satisfy `0 <= i < NUMBER_OF_BIGNUMS`, and chooses which bignum one wants to select. Input `is_long_number` specifies if the bignums represented in the array are bignums that are represented on `MIN_BIGNUM_NUMBER_OF_WORDS` words, or `MAX_BIGNUM_NUMBER_OF_WORDS` words. One needs to know if the data is a multiplication result, or just normal bignums. Input `j` must satisfy `0 <= j < MIN_BIGNUM_NUMBER_OF_WORDS` or `0 <= j < MAX_BIGNUM_NUMBER_OF_WORDS`, depending on the value of `is_long_number`. Input `j` allows one to select a particular word of a bignum.

`COAL_IDX(i, j)` This macro allows one to select bignums from a *coalesced* array. Input `i` must satisfy `0 <= i < MIN_BIGNUM_NUMBER_OF_WORDS`, or `0 <= i < MAX_BIGNUM_NUMBER_OF_WORDS`, depending if the array is containing multiplication results, or normal bignums. Input `j` must satisfy `0 <= j <= NUMBER_OF_BIGNUMS`. Input `i` selects which element of a bignum must be accessed, and input `j` selects which thread's bignum must be accessed. Normally, one would use this in kernels when accessing coalesced global memory, by looping over index `i`, and using `threadIdx.x` as the value in `j`.

`benchmarks.cu` This file contains all our kernels, with which we launch our operation benchmarks. A benchmark consists of computing 10 instances of each

operation one after the other. Every kernel reads its operands from the files `random_number_generator.py` created, performs the operation, then writes the results back to a file in coalesced form for verification. Each operation has 2 kernels, one that only uses coalesced global memory reads for the 10 instances, and another which reads the operands to some local memory before performing the operations. This will be used so we can compare the latency difference between the 2 approaches.

`input_output.cpp` Contains functions to read coalesced bignum arrays from files, and to write coalesced bignum arrays to files.

`main.cu` Selects which benchmarks have to be run.

`operations.h` Contains all the generated operations that have been created by `operation_generator.py`.

3.6 Operations

All the code fragments that are showed next are extracts from the `operation_generator.py` file.

3.6.1 Output precision

First of all, let's define some functions we will often be calling to find out more about precision and storage questions.

The number of words needed to store a number of `precision` bits is defined as:

```
1 def number_of_words_needed_for_precision(precision):
2     return math.ceil(precision / bits_per_word)
```

The precision of the result of an addition is always 1 bit bigger than the precision of the biggest operand.

```
1 def add_res_precision(op1_precision, op2_precision):
2     res_precision = max(op1_precision, op2_precision) + 1
3     return res_precision
```

The precision of the result of a multiplication is the sum of the precisions of its operands. However, this does *not* hold if one of the operands has a precision of 1 bit, as the maximum value that is representable on one bit is the number 1, and the multiplication of a number by 1 gives a result that holds on the original precision.

```
1  def mul_res_precision(op1_precision, op2_precision):
2      res_precision = op1_precision + op2_precision
3
4      # res_precision = op1_precision + op2_precision does not hold
5      # if one of the operands has precision 1. In that case, you
6      # need to reduce the precision of the result by 1 bit.
7      if (op1_precision == 1) or (op2_precision == 1):
8          res_precision -= 1
9
10     return res_precision
```

3.6.2 Generic function interfaces and outputs

The ultimate goal of the operator generation script is to create macros which the C programs can later use. These macros will be defined somewhat like the following:

```
#define add_loc(c_loc, a_loc, b_loc)\
{\
    // code of the operation referencing a_loc, b_loc, and c_loc
}
```

Listing 3.7: Heading of the local addition macro

There are 2 types of functions in `operator_generator.py`.

The first type sets up macro headings, one of which is shown in Listing 3.7. These functions are named after the macro they will export to the C code. For example, the function that exports the macro named `add_loc` will also be called `add_loc`. An important note is that these functions do *not* create the contents of the macro, but *only* the heading.

The second type of functions sets up the code of the macro, and always have the word `_generic_` in their name. Note that these functions do not know the name of their operands, so they must be provided to the function.

An example of the first type of function is the `add_loc()` function, shown in Listing 3.8, and an example of the second type of function is the `add_loc_exact_generic()` function (This can be found in our source code).

To define these macros, we use code like the following:

```
def add_loc():
    indent = 1

    asm = []
    asm.append('#define add_loc(c_loc, a_loc, b_loc)\\')
    asm.append('{\\')
    asm += add_loc_generic(precision, precision, 'a_loc', 'b_loc', 'c_loc', 0, 0,
    0, indent)
    asm.append('}' + '\n')
    return asm
```

Listing 3.8: Macro generator

The code shown in Listing 3.8 defines an empty list called `asm`, to which it will then append a string representing *each* line that will appear in the final macro. This list of lines is later printed to a file. This code sets up the macro's interface, along with the names of the operands and results that will be exposed to the C programs.

The `_generic_` functions have an interface that ressembles the following:

```
def add_loc_exact_generic(op1_precision, op2_precision, op1_name,
                          op2_name, res_name, op1_shift, op2_shift,
                          res_shift, indent = 0):
```

Let's explain what these fields mean:

`op1_precision, op2_precision` Precisions of the first and second operands. The precision of the result is normally calculated inside the function.

`op1_name, op2_name, res_name` Name of the first and second operands, as well as the name of the result. Remember that the `_generic_` functions do *not* know the names of their operands, therefore we need these fields to know how to

label the data that will be used in the generated code. We decided to do this so we could call the same `_generic_` function multiple times with different data.

`op1_shift, op2_shift, res_shift` Index in the array named by `op1_name`, `op2_name`, and `res_name` which the algorithm should consider as the least significant word. For example, if we have a bignum that holds on 5 words, but wish to add the last 3 of its words with another 3-word number, then we would set its `_shift` index to 2, and the 3-word bignum's `_shift` index to 0.

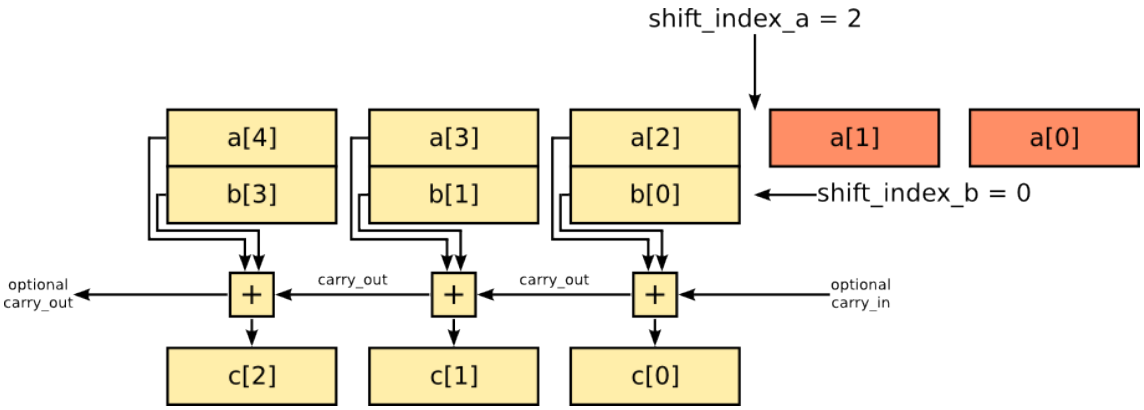


Figure 3.9: Addition with shift index

`indent` Used for recursive algorithms in order to indent them correctly in the printed source code. This field can be ignored.

Note: Each assembly instruction which is printed by the code generator is in a separate statement. This is due to the fact that assembly statements are limited to 30 registers, but since our scripts work for higher precisions, it is possible that some functions would require more than that limit. Placing each assembly intruction in a separate statement is a way to avoid this problem.

3.6.3 Local vs. Global macros

For each macro, we implemented 2 different versions which have identical code, but which differ in their memory access patterns. Each macro has a `_loc_` version, and a `_glo_` version. The `_loc_` version expects its operands to be local vectors, whereas the `_glo_` version expects its operands to be in global memory. Note that the `_glo_` version needs to have the thread id of the bignum it is to access.

```
1 #define add_glo(c_glo, a_glo, b_glo, tid)\
2 {\
3     asm("add.cc.u32  %0, %1, %2;" : "=r"(c_glo[COAL_IDX(0, tid)]) : "r"(a_glo[
      COAL_IDX(0, tid)]), "r"(b_glo[COAL_IDX(0, tid)]));\
4     asm("addc.cc.u32 %0, %1, %2;" : "=r"(c_glo[COAL_IDX(1, tid)]) : "r"(a_glo[
      COAL_IDX(1, tid)]), "r"(b_glo[COAL_IDX(1, tid)]));\
5     asm("addc.cc.u32 %0, %1, %2;" : "=r"(c_glo[COAL_IDX(2, tid)]) : "r"(a_glo[
      COAL_IDX(2, tid)]), "r"(b_glo[COAL_IDX(2, tid)]));\
6     asm("addc.cc.u32 %0, %1, %2;" : "=r"(c_glo[COAL_IDX(3, tid)]) : "r"(a_glo[
      COAL_IDX(3, tid)]), "r"(b_glo[COAL_IDX(3, tid)]));\
7     asm("addc.u32    %0, %1, %2;" : "=r"(c_glo[COAL_IDX(4, tid)]) : "r"(a_glo[
      COAL_IDX(4, tid)]), "r"(b_glo[COAL_IDX(4, tid)]));\
8 }
```

Listing 3.9: 5-word global memory addition

```
1 #define add_loc(c_loc, a_loc, b_loc)\
2 {\
3     asm("add.cc.u32  %0, %1, %2;" : "=r"(c_loc[0]) : "r"(a_loc[0]), "r"(b_loc[0]))
      ;\
4     asm("addc.cc.u32 %0, %1, %2;" : "=r"(c_loc[1]) : "r"(a_loc[1]), "r"(b_loc[1]))
      ;\
5     asm("addc.cc.u32 %0, %1, %2;" : "=r"(c_loc[2]) : "r"(a_loc[2]), "r"(b_loc[2]))
      ;\
6     asm("addc.cc.u32 %0, %1, %2;" : "=r"(c_loc[3]) : "r"(a_loc[3]), "r"(b_loc[3]))
      ;\
7     asm("addc.u32    %0, %1, %2;" : "=r"(c_loc[4]) : "r"(a_loc[4]), "r"(b_loc[4]))
      ;\
8 }
```

Listing 3.10: 5-word local memory addition

3.6.4 Addition

The general algorithm for performing addition can be seen in Figure 3.10.

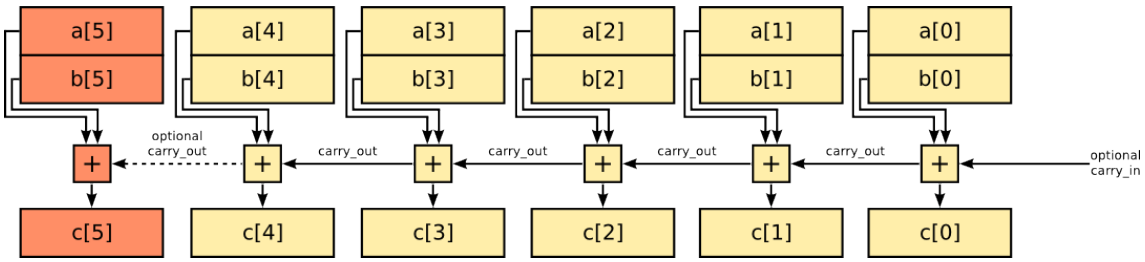


Figure 3.10: Addition algorithm

The following algorithm concerns the `add_loc_generic()` (in yellow on Figure 3.10, and `add_loc_exact_generic()` (yellow part of Figure 3.10, potentially with the additional red part) functions. `add_loc_generic()` returns a result that holds on the same precision as the *biggest* of its 2 operands, similar to how hardware adders work. In contrast, `add_loc_exact_generic()` returns the actual exact answer of the addition. The only difference between these 2 functions is the upper bound used in their internal loops. `add_loc_generic()` makes use of the number of words its biggest operand uses, whereas `add_loc_exact_generic()` uses the output of `add_res_precision()` to calculate the number of words it should loop until. Note that `add_loc_exact_generic()` is only used in the implementation of the Karatsuba algorithm.

The addition algorithm starts by finding out which of the 2 operands needs less words in its representation. If the operands use the same number of words for their representatoin, then the basic ripple carry addition is performed, and code such as the one in Listing 3.2 is outputted. However, if the operands do not have an identical word count, then the function performs the normal addition algorithm until the smaller bignum has no more words left, after which case it modifies the assembly instructions and replaces the smaller bignum's placeholder with `0`.

Listing 3.11 shows the code that would be generated for the addition between a 5-word and a 3-word bignum.

```
1 asm("add.cc.u32  %0, %1, %2;" : "=r"(c[0]) : "r"(a[0]), "r"(b[0]));
2 asm("addc.cc.u32 %0, %1, %2;" : "=r"(c[1]) : "r"(a[1]), "r"(b[1]));
3 asm("addc.cc.u32 %0, %1, %2;" : "=r"(c[2]) : "r"(a[2]), "r"(b[2]));
4 asm("addc.cc.u32 %0, %1,  0;" : "=r"(c[3]) : "r"(a[3]));
5 asm("addc.u32    %0, %1,  0;" : "=r"(c[4]) : "r"(a[4]));
```

Listing 3.11: Addition between 5-word and 3-word bignums

3.6.5 Subtraction

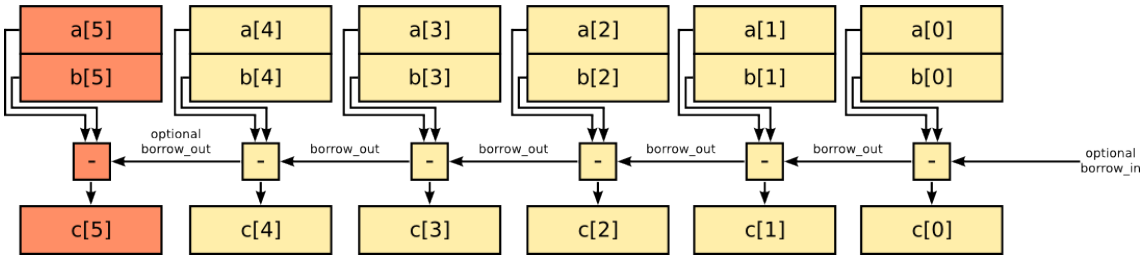


Figure 3.11: Subtraction algorithm

The subtraction functions, `sub_loc_generic()` and `sub_loc_exact_generic()`, work exactly as their addition counterparts work, but they use the `sub` instruction as opposed to the `add` instruction. Actually, these functions are implemented by taking the code that is generated by `add_loc_generic()` and `add_loc_exact_generic()`, and by replacing all instances of the `add` instruction by `sub`. An example is shown below:

```
1 asm("sub.cc.u32  %0, %1, %2;" : "=r"(c[0]) : "r"(a[0]), "r"(b[0]));
2 asm("subc.cc.u32 %0, %1, %2;" : "=r"(c[1]) : "r"(a[1]), "r"(b[1]));
3 asm("subc.cc.u32 %0, %1, %2;" : "=r"(c[2]) : "r"(a[2]), "r"(b[2]));
4 asm("subc.cc.u32 %0, %1, %2;" : "=r"(c[3]) : "r"(a[3]), "r"(b[3]));
5 asm("subc.u32    %0, %1, %2;" : "=r"(c[4]) : "r"(a[4]), "r"(b[4]));
```

Listing 3.12: 5-word subtraction

3.6.6 Modular Addition

The algorithm we used for modular addition supposes that the 2 operands are smaller than the modulus, and is not guaranteed to work if any of the operands do not satisfy this condition. If we have the following condition, `0 <= a, b < m`, where `a`, and `b` are operands, and `m` is the modulus, then we know that `a + b < 2*m`. Therefore, if

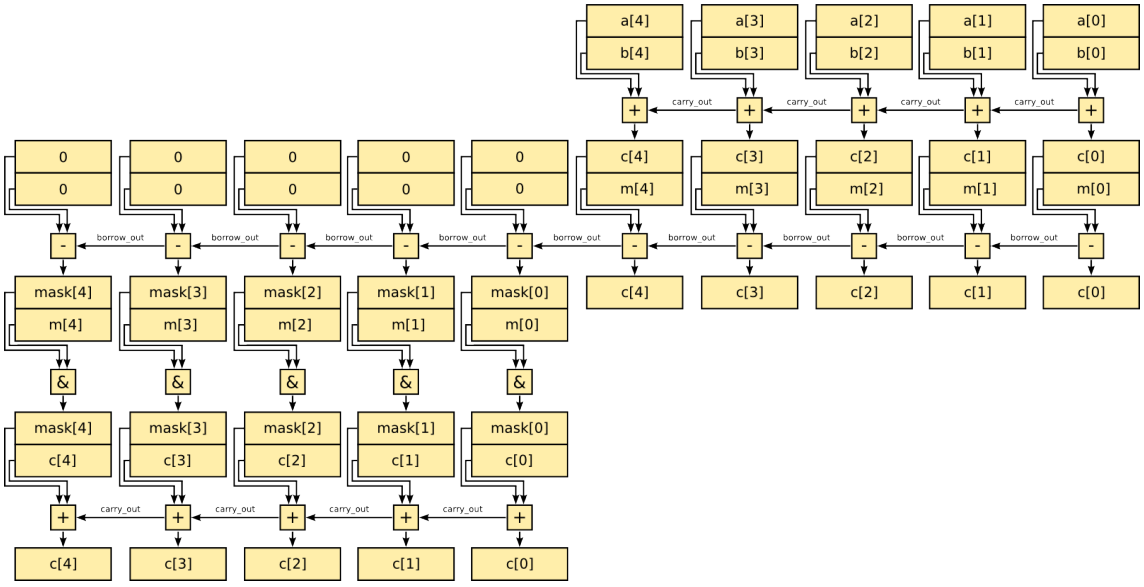


Figure 3.12: Modular addition algorithm

the addition yielded a number bigger than `m`, a single subtraction by `m` would give us back the modular addition's result.

Normally, a simple conditional subtraction would be used to perform the modular addition, as shown in Listing 3.13. Note that we use a simple 1-word bignum as an illustration.

```
1  uint32_t a = ...;
2  uint32_t b = ...;
3  uint64_t c = a + b;
4
5  if (c > m)
6  {
7      c -= m;
8  }
9
10 return (uint32_t) c;
```

Listing 3.13: Conditional subtraction in modular addition

However, this method introduces a conditional statement that may cause some thread divergence. We stated earlier why we wanted to avoid thread divergence at all costs, therefore we decided to implement the conditional subtraction differently.

The algorithm we used is the following:

```
1  c = a + b
2  c = c - m
3  mask = 0 - borrow
4  mask = mask & m
5  c = c + mask
```

Listing 3.14: Branch-less modular addition algorithm

The key part to notice is on lines 2-3. By subtracting `m` from `c`, if the result were to be negative, then a borrow bit would be emitted. Subtracting the borrow bit from `0` would yield a number that only contains binary `1`s in its representation. One can then use this number as a mask to know if the modulus should be added back to the result. Note that if no borrow bit was emitted, then the subtraction on

line 2 yielded a positive number, and the mask would be 0, thus no modulus would be added back to the result.

An illustration is shown in Figure 3.12.

The Python code to generate this macro is short, and is shown below. It uses all some of our `_generic_` functions.

```
1  def add_m_loc():
2      indent = 1
3
4      asm = []
5      asm.append('#define add_m_loc(c_loc, a_loc, b_loc, m_loc)\\')
6      asm.append('{\\')
7      asm.append(" " * 4 * indent + 'uint32_t mask[' + str(min_bignum_number_of_words
       ) + '] = ' + str([0] * min_bignum_number_of_words).replace('[', '{').replace(']
       ', '}') + ';\\')
8
9      asm += add_loc_generic(precision, precision, 'a_loc', 'b_loc', 'c_loc', 0, 0,
       0, indent)
10     asm += sub_cc_loc_generic(precision, precision, 'c_loc', 'm_loc', 'c_loc', 0,
       0, 0, indent)
11     asm += subc_loc_generic(precision, precision, 'mask', 'mask', 'mask', 0, 0, 0,
       indent)
12     asm += and_loc_generic(precision, precision, 'mask', 'm_loc', 'mask', 0, 0, 0,
       indent)
13     asm += add_loc_generic(precision, precision, 'c_loc', 'mask', 'c_loc', 0, 0, 0,
        indent)
14
15     asm.append('}' + '\n')
16     return asm
```

Listing 3.15: Modular addition macro generation function

3.6.7 Modular Subtraction

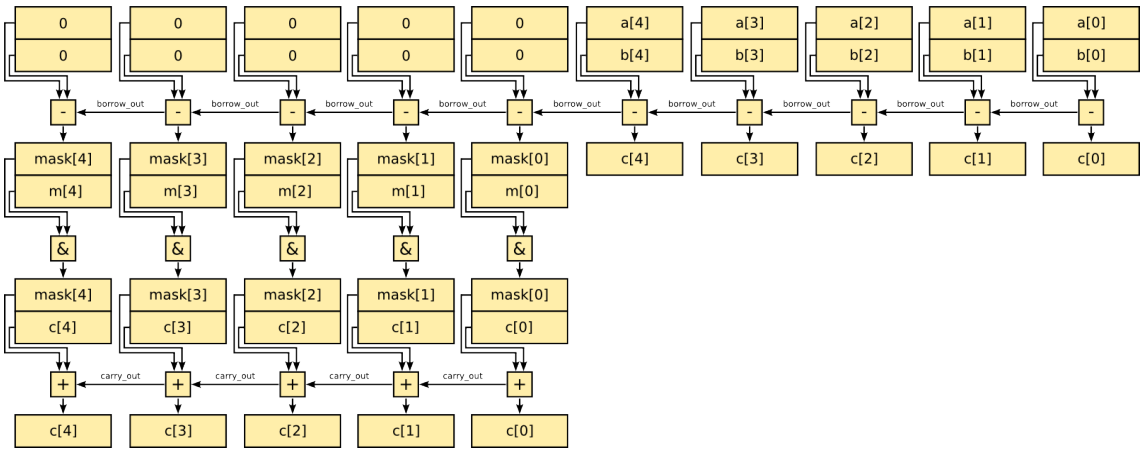


Figure 3.13: Modular subtraction algorithm

Modular subtraction is analogous to its addition counterpart. The only difference is that the first operation is a subtraction instead of an addition, and that one doesn't need to subtract the modulus from the result of the first operation.

The algorithm is shown in Listing 3.16, and an illustration is shown in Figure 3.13.

```
c = a - b
mask = 0 - borrow
mask = mask & m
c = c + mask
```

Listing 3.16: Branch-less modular subtraction algorithm

3.6.8 Multiplication

We will now study how to adapt the schoolbook multiplication algorithm for GPUs.

In primary school, we learned to perform long multiplications by using a an iterative algorithm, as shown in Figure 3.14a. The approach here is to accumulate the result of every sub-multiplication before passing on towards the next one. At first, we were implementing the schoolbook multiplication ordering mostly in C, with assembly for the parts that needed to access the carry bit. However, we kept losing carries with this ordering, and we decided to reorder the multiplications as shown on Figure 3.14b.

This new ordering allowed us to fully compute the final value of one of the result's indices, before passing on to the next one. There were 2 difficult parts here:

1. Generating the list of index tuples which are to be multiplied (visibile in Figure 3.14b). We know that a result column's index is equal to the addition of the operand indices, so we computed the tuples by using the following code:

```
# generate tuples of indexes in arrays A and B that are
# to be multiplied together. "i" represents "b_loc" and "j"
# represents "a_loc"
mul_index_tuples = []
# min shifting value is 0 and max is (res_number_of_words - 1)
for shift_index in range(res_number_of_words):
    shift_index_tuples = []
    for j in range(op1_number_of_words):
        for i in range(op2_number_of_words):
            if i + j == shift_index:
                shift_index_tuples.append((i, j))
    if shift_index_tuples != []:
        mul_index_tuples.append(shift_index_tuples)
```

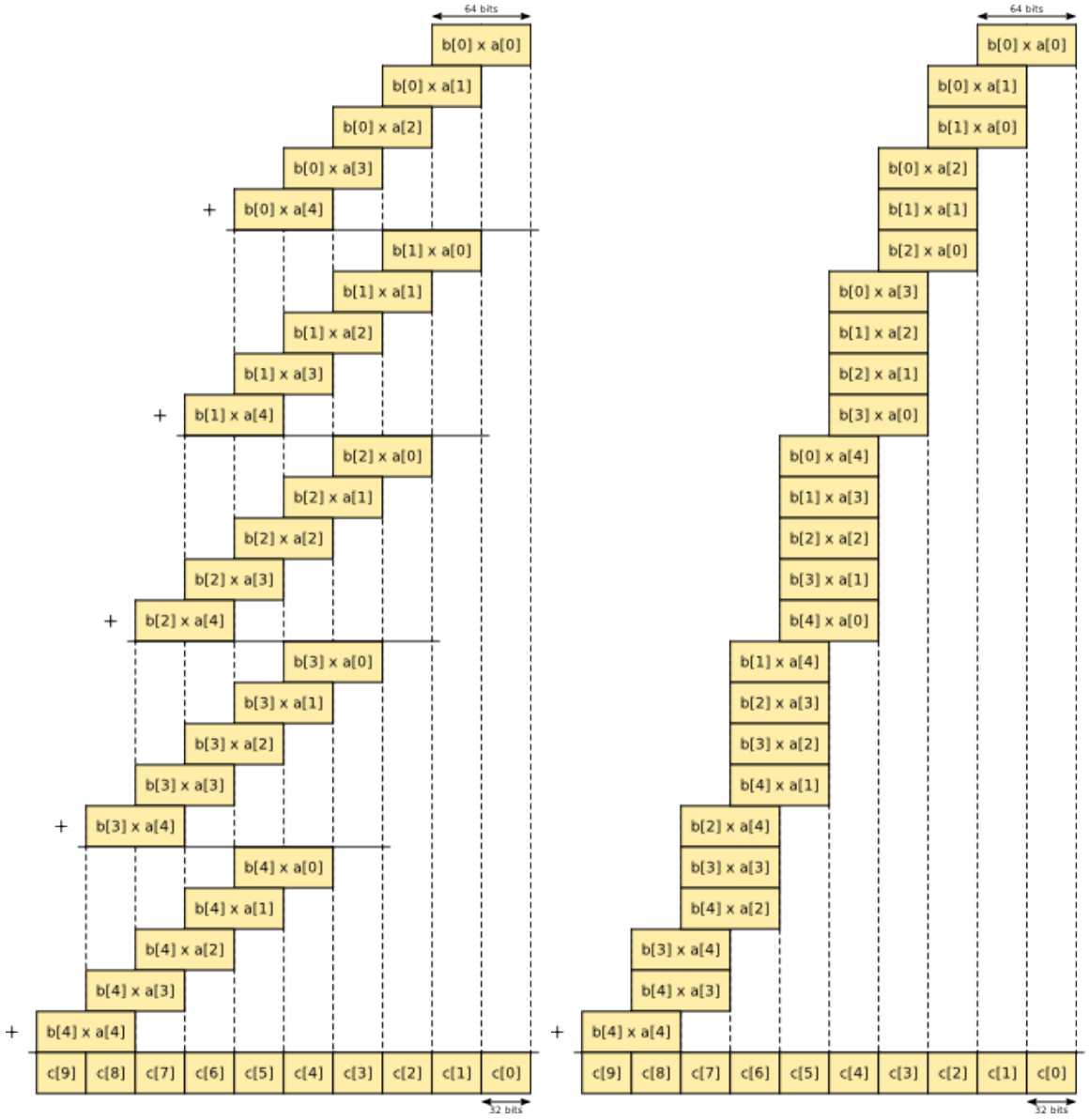
Listing 3.17: Index tuple generation for multiplication

2. How to accumulate multiple carries from one column to the next. To solve this, we used a single register as our carry holder, and accumulated the partial carries to it after each multiply-add instruction, as shown in the following code:

```
asm.append('asm("mad.hi.cc.u32 %0, %1, %2, %0;" : "+r"(c_loc[' + str(c_index
    + res_shift) + ']) : "r"(b_loc[' + str(b_index + op2_shift) + ']), "r"(
    a_loc[' + str(a_index + op1_shift) + ']));\\')
asm.append('asm("addc.u32      %0, %0,  0    ;" : "+r"(carry));\\')
```

Listing 3.18: Carry accumulation

Once the final value of the column had been fully calculated, the carry register contains multiple accumulated carries These accumulated carries are used to



(a) Schoolbook ordering (b) Ordering for carry propagation

Figure 3.14: Multiplication ordering

initialize the result of the next column, then the calculation of that column continued just as the previous one. We also reset the carry after it has been propagated to the next column.

```
1 asm.append('asm("add.u32       %0, %1,  0    ;" : "=r"(c_loc[' + str(c_index
      + res_shift) + ']) : "r"(carry));\\')
2 asm.append('asm("add.u32       %0,  0,  0    ;" : "=r"(carry));\\')
```

Listing 3.19: Carry propagation to next result column

Note, there is a little optimization that avoids the calculation of the final result index if it is not needed. For example, the multiplication of two 131-bit numbers yields, at most, a 262-bit integer. However, 262-bit integers are representable on 9 words, instead of $2 * 5 = 10$ words. Therefore, the final index of the multiplication does not need to be performed.

3.6.9 Karatsuba Multiplication

Unlike all the other macros we implemented, the Karatsuba mutliplication one only works on 2 operands of the *same* precision.

Schoolbook multiplication is asymptotically of n^2 complexity. More theoretically efficient algorithms based off the divide-and-conquer paradigm exist, such as the Karatsuba multiplication algorithm. The idea behind Karatsuba's algorithm is to compute the product of two large numbers using three multiplications of smaller numbers, each with about half as many digits as the initial operands, plus some additions and digit shifts [4]. Karatsuba's algorithm can be used recursively to attaign a complexity of $n^{1.585}$, however, because of difficulties performing so in assembly, we decided to only implement a 1-level recursive algorithm.

The algorithm is as follows:

1. Take 2 operands a and b represented as N-digit strings in some base B. For any positive integer $M < N$, one can represent a and b alternatively as follows:
   - $a = a_1 B^M + a_0$
   - $b = b_1 B^M + b_0$
2. The product ab can then be obtained by $c_2 B^{2M} + c_1 B^M + c_0$, where
   - $c_0 = a_0 b_0$
   - $c_2 = a_1 b_1$
   - $c_1 = (a_1 + a_0)(b_1 + b_0) - c_0 - c_2$

Theoretically, the algorithm is the most efficient when the operands are divided in 2 equal, or one-off parts, however, since our machine operates on full-words, we decided to split the operands at word boundaries. We split the operands by keeping the potentially smaller partition over the least significant words, as shown on Figure 3.15.

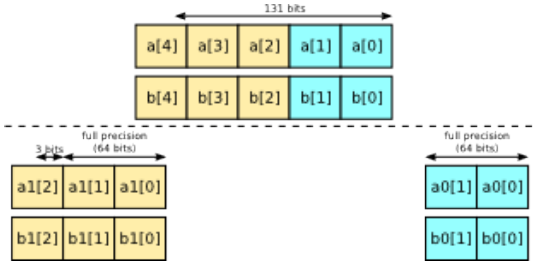


Figure 3.15: Karatsuba operand decomposition for 131-bit operands

Note: all red colorings in the future figures reference potential optimizations that can occur depending on the precision of the operands.

The reason we did this arose when we were initially trying to implement the recursive karatsuba algorithm. When we had chosen the most significant words to be the smaller part of the decomposition, we were entering an infinite loop in our operation generation script. The solution we found was to just choose the other part of the bignum as the smaller part.

The computation of c_0 and c_2 is straightforward, as we just have to use the `mul_loc_generic()` function we implemented previously. Note that c_0 is the result of the multiplication of 2 full-precision operands, and has to perform all multiplication steps, however, c_2's operands are not always full-precision numbers, and it is possible that their multiplication would not need to perform one step, as shown on Figure 3.16 for 131-bit numbers.

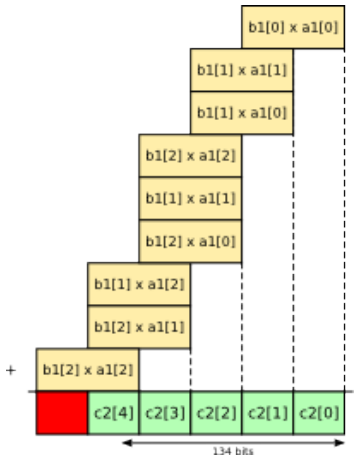


Figure 3.16: Karatsuba multiplication optimizations when computing a_1b_1 for 131-bit operands

The Karatsuba algorithm is the only place in the code where we use `add_loc_exact_generic()`, instead of `add_loc_generic()`. This is shown in Figure 3.17. Indeed, this generic karatsuba function works for any operand precision, and sometimes, it is possible that the result of the addition in $a_0 + a_1$ or $b_0 + b_1$ will not hold on the same precision as the operands. We then would need the exact value of this addition, which is what the `add_loc_exact_generic()` function provides us

[4] `http://en.wikipedia.org/wiki/Karatsuba_algorithm`

with.

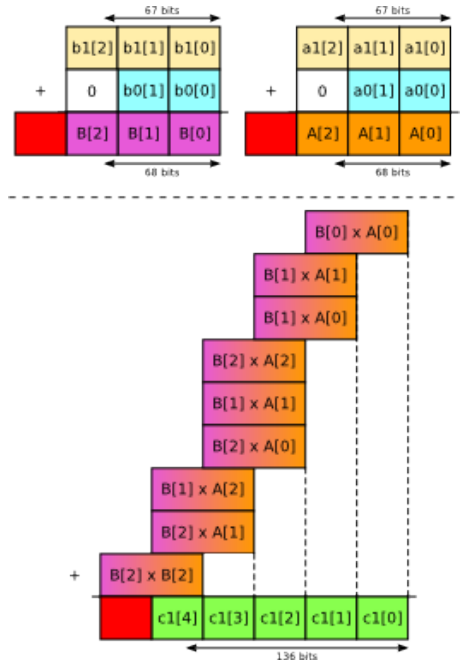


Figure 3.17: Potential use of add_loc_exact_generic in Karatsuba algorithm for 131-bit operands

The Karatsuba algorithm is always slower than the classical multiplication algorithm when $a_0 + a_1$ or $b_0 + b_1$ require an additional word. Since multiplication is an n^2 operation in terms of complexity, adding a single extra word will greatly increase the number of steps that will be needed, especially if the operand's precision is already large. For example, if we wanted to use the Karatsuba algorithm to multiply two 128-bit numbers, we will see that $x_0 + x_1$ will need an additional word for storage, therefore the multiplication $(x_0 + x_1)(y_0 + y_1)$ would need to perform a multiplication of two 3-word numbers, instead of 2-word numbers, and the number of multiplication instructions that would be executed as extra would make the algorithm slower.

Computing c_1 is easily done with `add_loc_generic()`, and the final shifted addition of $c_2B^{2M} + c_1B^M + c_0$ can be obtained through the use of the shift indices of the function. An illustration is shown on Figure 3.18, along with a potential optimization for 131-bit operands.

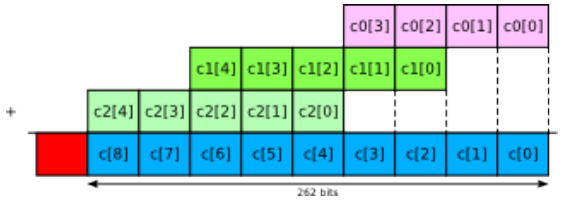


Figure 3.18: Shifted addition (final step of Karatsuba algorithm)

3.7 Montgomery Reduction

Algorithm Montgomery reduction

INPUT: integers $m = (m_{n-1} \cdots m_1 m_0)_b$ with $\gcd(m, b) = 1$, $R = b^n$, $m' = -m^{-1} \bmod b$, and $T = (t_{2n-1} \cdots t_1 t_0)_b < mR$.
OUTPUT: $TR^{-1} \bmod m$.

1. $A \leftarrow T$. (Notation: $A = (a_{2n-1} \cdots a_1 a_0)_b$.)
2. For i from 0 to $(n-1)$ do the following:
   2.1 $u_i \leftarrow a_i m' \bmod b$.
   2.2 $A \leftarrow A + u_i m b^i$.
3. $A \leftarrow A/b^n$.
4. If $A \geq m$ then $A \leftarrow A - m$.
5. Return(A).

Figure 3.19: Montgomery reduction algorithm

We spent some time trying to implement Montgomery reduction by following the algorithm defined in the Handbook of Applied Cryptography[5], and which is shown in Figure 3.19. In order to test if our implementation was correct, we generated the T operand that is needed in the algorithm, performed the computations on the GPU, and proceeded to verify the results. Our testing infrastructure kept warning us that the results were incorrect, so we went on to try and debug the implementation to see what was wrong.

On line 3 of the Montgomery reduction algorithm, it is stated that `A` must be right-shifted by a certain amount, which was by 131-bits for our implementation. For this right-shift to be valid, there must only be binary `0` bits on the least significant bits of `A`, otherwise we would be losing information through the operation. After thorough testing, we believe our implementation of steps 2, 2.1, and 2.2 is correct, as we found that the least significant bits of our `A` value were all `0`, just as they are supposed to be.

We then verified that the right shift operation was implemented correctly, and it was the case.

However, our operation tester kept telling us that the results were false.

From there, we had 2 interpretations:

- Either our implementation of the Montgomery reduction is wrong, and it is not computing what we expect it to compute, however, we have debugged this code line-by-line, and it seems it is executing exactly what we intended it to do.
- Or, we have misunderstood how Montgomery reductions actually work, and are probably not providing the correct operands to the algorithm.

We belive our problem lies with the operand generation.

[5] `http://cacr.uwaterloo.ca/hac/`

3.8 Weird Experiences

3.8.1 Erroneus compiler optimization

When we wanted to launch some benchmarks, we tried to launch an operation for a large number of iterations. We though basically adding a `for` loop around our macro, along with a big counter would suffice to launch the benchmarks for multiple iterations. This worked well for addition and subtraction, however, once we tried it for the multiplication macros, we started getting wrong results.

We spent quite some time trying to debug the multiplication code, and after a while compiled the code to PTX, and realized that some of our assembly instructions were *not* being put in the final PTX. The instructions in this case were the ones that reset the carry counter back to 0 once a column was finished being calculated.

We believe the compiler is trying to optimize out some instructions, but it mistreating our algorithm. To work around this problem, we had to resort to unrolling loops in C as well, which means that if you want to run 1000 iterations of the multiplication algorithm, you have to paste the invokation of the macro 1000 time. Note that this also greatly increased compile time, as macros are expanded in place, and the multiplication code being quite lengthy didn't help.

Chapter 4

Benchmarks

4.1 Local vs. Global

As we expected, using local vectors is much faster that performing coalesced global memory reads, especially if the data will be accessed multiple times. Figure 4.1 shows the performance difference between local and global modular addition for 239-bit arithmetic.

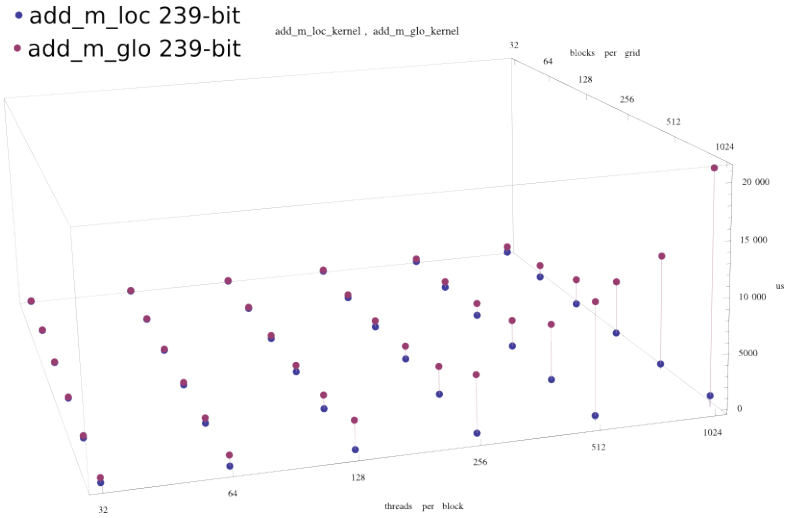


Figure 4.1: add_m_loc vs add_m_glo performance for 239-bit operands (Kepler)

4.2 Schoolbook multiplication vs. Karatsuba multiplication

4.2.1 Kepler

Figure 4.2 shows the performance difference between schoolbook multiplication, and Karatsuba multiplication on Kepler GPUs. For 131-bit arithmetic, we can see that Karatsuba multiplication is faster for any grid size larger than 512 blocks, and any block size larger than 256 threads. Note that Karatsuba multiplication uses more

registers than the schoolbook algorithm (34 vs. 25 registers), but still is slightly faster.

However, for 239-bit arithmetic, the situation is the opposite, and Karatsuba's algorithm is slower for all combinations of grid and block sizes. This can be explained by the very high register usage of 49 registers, compared to the 33 registers that the schoolbook multiplication uses. There is therefore some register spilling to global memory, which affects execution time.

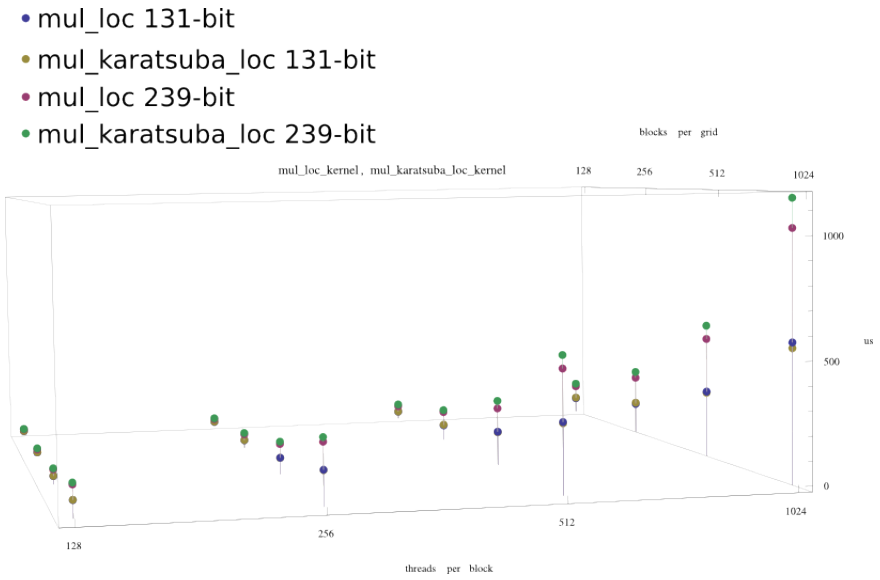


Figure 4.2: mul_loc vs mul_karatsuba_loc performance on Kepler GPUs for 131-bit and 239-bit operands

4.2.2 Fermi

Figure 4.3 performs the same performance comparisons between Karatsuba's algorithm, and the schoolbook algorithm, but on smaller grid and block sizes to see the effect of warp occupancy better. Every top-right directed diagonal represents an equivalent amount of data to be computed, but with different launch configurations each time. Observe how the execution time decreases as one uses less blocks containing many threads, as opposed to more blocks containing few threads. Indeed, CUDA GPUs use warps as their basic execution block. If a block is not fully populated with threads, then it will have to be executed over multiple passes. One pass would be used for the threads that are actually doing some calculations, and another one would be used for those that are not. Of course, this effect accumulates, and can give the performance differences we see in Figure 4.3.

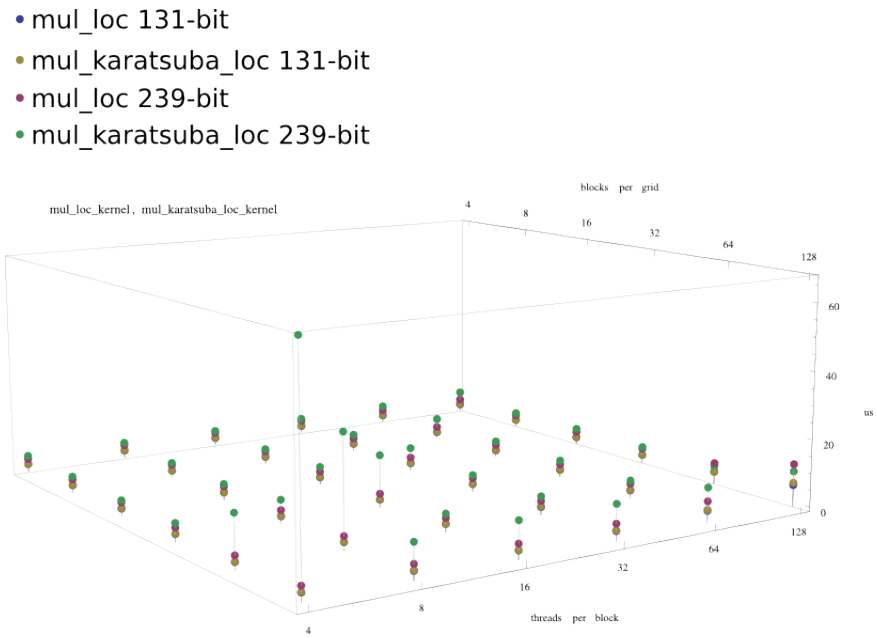


Figure 4.3: mul_loc vs mul_karatsuba_loc performance on Fermi GPUs for 131-bit and 239-bit operands

Chapter 5

Conclusion

This project was very challenging, as it was our first step in the domain of GPU programming. At first, we spent a lot of time trying to read about how GPUs were structured, and to grasp their programming model. We then took our first steps in GPU implementations with CUDA.

We learned how to do the basics of GPU programming. We now know how to transform a sequential algorithm to a parallel one. We also learned the importance of having good tools for work, as we lost a lot of time when trying to implement this ourselves.

We had a rough start, since we started to implement everything in C, but ran into problems. We then realized it was more convenient for us to use assembly for accessing carries. Since we now wanted to implement fully generic arithmetic, we went with the idea to write scripts which would generate generic assembly code for our instructions.

We have learned, perhaps in a hard way, that it may not be worth all the effort to produce all the code in assembly, as our benchmarks showed that C code had neglectable performance differences compared to assembly, in addition to being easier to write.

Chapter 6

Bibliography

6.1 Websites

- Cover image: http://pc-internet-zone.blogspot.ch/2011/12/nvidia-cuda-parallel-programming.html
- http://en.wikipedia.org/wiki/Graphics_processing_unit
- http://en.wikipedia.org/wiki/Programmable_shader
- http://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units
- http://en.wikipedia.org/wiki/Functional_completeness
- http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html
- http://en.wikipedia.org/wiki/CUDA
- https://developer.nvidia.com/cuda-faq
- https://developer.nvidia.com/get-started-cuda-cc
- http://gmplib.org/
- http://en.wikipedia.org/wiki/Two%27s_complement
- http://en.wikipedia.org/wiki/Parallel_Thread_Execution

6.2 Books

- CUDA By Example - An Introduction to General-Purpose GPU Programming (Jason Sanders, Edward Kandrot), ISBN 978-0-13-138768-3
- Programming Massively Parallel Processors - A Hands-on Approach (David B. Kirk and Wen-mei W. Hwu), ISBN: 978-0-12-381472-2

- Handbook of Applied Cryptography (A. Menezes, P. van Oorschot, and S. Vanstone), ISBN: 0-8493-8523-7

6.3 Articles

- A Review of Modular Multiplication Methods and Respective Hardware Implementations (Nadia Nedjah and Luiza de Macedo Mourelle)
- Comparison of Modular Arithmetic Algorithms on GPUs (Pascal Giorgi, Thomas Izard, and Arnaud Tisserand)
- Implementation of Multiple-precision Modular Multiplication on GPU (Kaiyong Zhao)
- Montgomery Multiplication (Duncan A. Buell)
- Supporting Extended Precision on Graphics Processors (Mian Lu, Bingsheng He, and Qiong Luo)