Implementing generic multiple precision arithmetic on GPUs

Alexandre Carlessi   Sahand Kashani-Akhavan

June 15, 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 in 2001, GPUs added programmable shading to their capabilities, which allowed 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 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 is shown on Figure 1.1. This unified design made GPUs architecturally closer to CPUs.

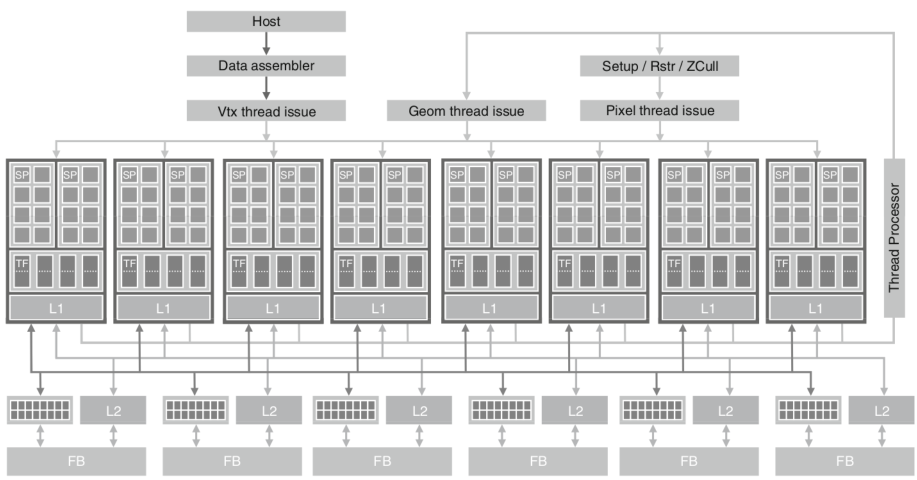

Figure 1.1: Unified programmable processor array of the GeForce 8800 GTX

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 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 or DirectX. 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, 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:

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

Listing 1.1: Vector definitions

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

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

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, 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

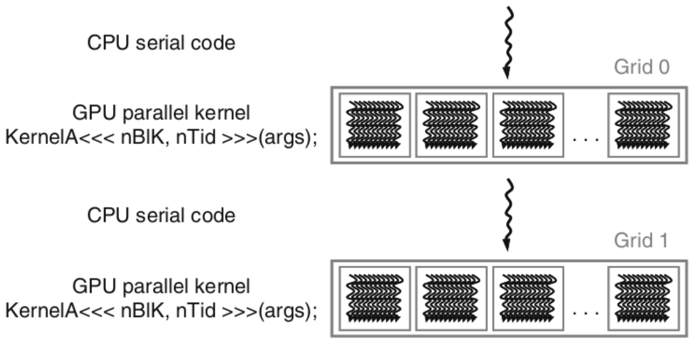

Figure 2.1: Cuda program execution phases

program execution flow is shown in Figure 2.1.

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.

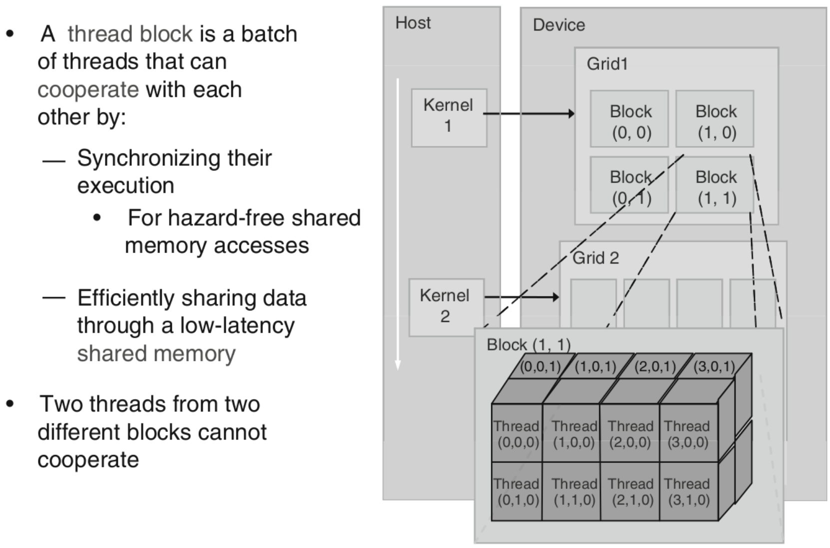

Figure 2.2: Two-level CUDA thread organization

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 (compute capability 2.0) and Kepler (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 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.

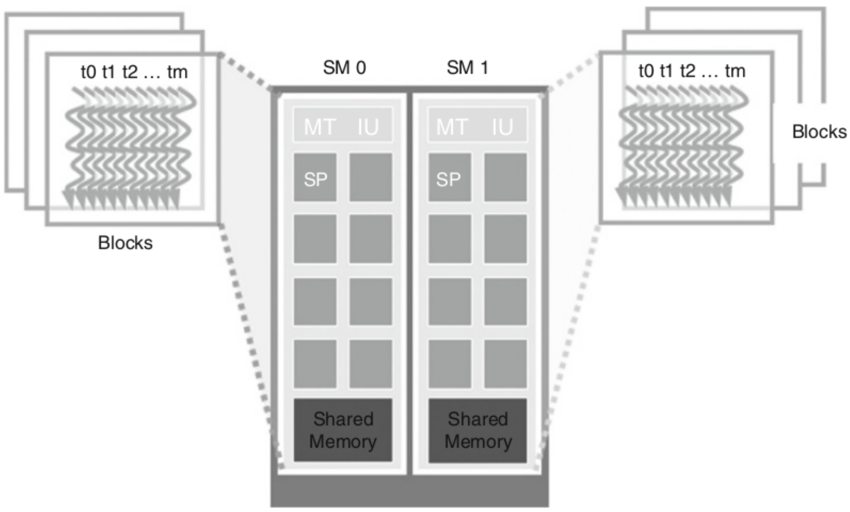

Figure 2.3: SM thread block assignment

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. Note that for simplicity, texture memory is not shown.

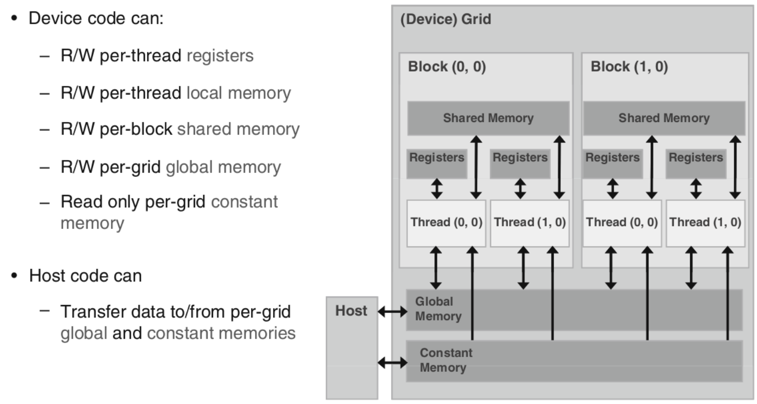

Figure 2.4: CUDA memory hierarchy

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 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:

```
for (int i = 0; i < 4; i++)
{
    for (int j = 0; j < 4; j++)
    {
        m[i][j] = ... ;
    }
}
```

Listing 2.1: Accessing matrix elements on a CPU

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

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

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):

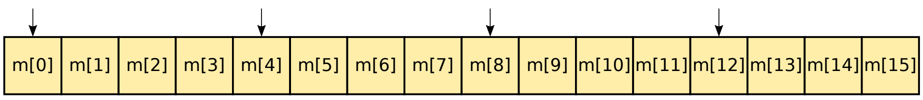

Figure 2.5: Non-coalesced memory access

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:

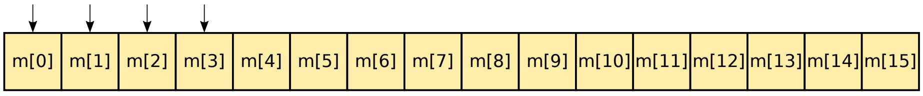

Figure 2.6: Coalesced memory access

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

Listing 2.3: Accessing matrix elements on a GPU (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

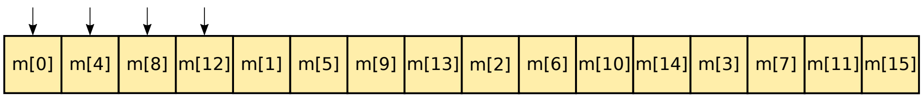

Figure 2.8: Correct coalesced memory access

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. 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 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 big number representation

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

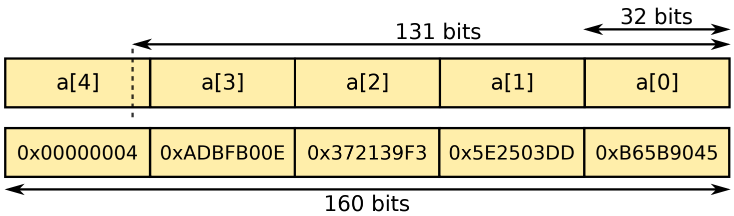

Figure 3.1: Most significant word first representation (MSW)

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.

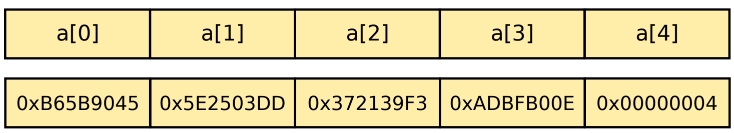

Figure 3.2: Least significant word first representation (LSW)

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 `1`s in its MSW.

a[0]	a[1]	a[2]	a[3]	a[4]
0xA93D318A	0x48EB68C1	0x7FB3AA74	0x9788816E	0xFFFFFFFE

Figure 3.3: Negative number representation

3.2 Operations

3.2.1 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`.

Let's consider the simple addition algorithm over two 63-bit numbers, `a` and `b`. What one would normally do is to start by computing `c[0] = a[0] + b[0]`, yielding a potential carry bit. Next, we compute `c[1] = a[1] + b[1] + carry`. Note that in this case, no carry bit can be created from the computation of `c[1]`, since `a` and `b` are 63-bit words, and we know that the result of an addition between a N-bit number, and a M number holds on at most $max(N, M) + 1$ bits, which is 64 bits here, and is thus fully representable on two 32-bit words.

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.

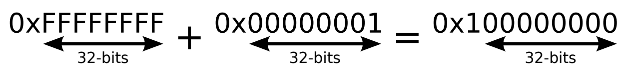

Figure 3.4: Overflow illustration on hexadecimal numbers

The trick to notice here is that if there were no overflow, then the lower 32-bits of the result would be *bigger* that the lower 32-bits of the 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, for our 63-bit addition example, in order to get carry flags in C for the computation of `c[1] = a[1] + b[1] + carry`, one would beforehand compute `carry = (c[0] < max(a[0], b[0]))`.

This is relatively simple for a basic addition algorithm, but starts getting complicated when trying to do more complex computations. Luckily, CUDA GPUs expose a low-level Parallel Thread Execution (PTX) 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`, `mul`, and `mad`, 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):

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

Listing 3.1: 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:

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

Listing 3.2: 5-word loop-oriented addition

The problem with this code is that its result is unpredicatable, since *carry flags are not presever across calls*, and the loop structure created in C around the assembly will *sometimes* make us lose the carry bit. We 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 bits.

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

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

Listing 3.3: Compiler unrolled 5-word loop-oriented addition

However, through a lot of tests, we found out 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 3.1.