Project Number: BS2 MQP ABT2

An Arithmetic Library for Fully Homomorphic Encryption

A Major Qualifying Project Report

submitted to the faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

Robert Chase

Keith Colbert

Dominic Fusco

Austin Labastie

Date:

Approved:

Professor Berk Sunar, Major Advisor

Professor William J. Martin, Major Advisor

Abstract

As critical government, industry, and consumer applications move online, techniques to maintain data security and privacy must be updated. Traditional encryption methods leave data vulnerable when it is searched or modified. Homomorphic encryption fills the gap, enabling such operations on encrypted data and eliminating the vulnerable decryption step. We worked with the CUDA-accelerated Fully Homomorphic Encryption (cuFHE) Library, the fastest of its kind in the public domain, to create efficient arithmetic functions. We built upon and modified the existing gate primitives, arranging them to create functions which are hardware and application agnostic. The result is a fast platform upon which homomorphic applications can be built: applications which protect user privacy and data integrity.

Contents

Chapter 1. Introduction

Homomorphic encryption is a novel paradigm which will enable greater data security and privacy for a variety of applications. Put simply, homomorphic encryption enables a program to operate on secure data without decrypting it; arithmetic operations, searches, and modifications can all be performed on encrypted data without compromising its integrity. As homomorphic encryption becomes increasingly viable, it will change the way data is secured across industry and government.

1.1 What is Homomorphic Encryption

A generic encryption scheme is sometimes compared to a safe, given that once data is converted into ciphertext it is safe from any attacks on it. This is similar to an object being locked in a safe. In most cases though, the contents within the safe will need to be accessed at some point due to their importance [Gre12]. Otherwise, there would be no need for the safe. This means at some point the safe will have to be opened to utilize what is inside it, leaving these valuables compromised. This is like most encryption schemes where the ciphertext is useless until it is decrypted. These encryption schemes force the user to convert the data back into plaintext before doing any computations on it, then convert it back into ciphertext. This is the same as opening and closing the safe to touch its contents in anyway. This decryption needed to perform computations leaves the data susceptible to tampering. Homomorphic encryption is the first encryption paradigm being developed to conquer this issue.

Homomorphic encryption is a method that allows computations to be performed on encrypted data without first needing to decrypt it. These computations occur without any knowledge as to what the inputs or values being computed are. In [Gre14], the example of a homomorphically encrypted search engine is given, in which an encrypted search term is taken in and then compared to an encrypted index of the web. Since both encrypted terms can be compared to each other, everything on the outside is blind to what is being searched. The search engine is left ignorant not only of the keyword it used in its search, but also of the list of URLs it recommends to the user. This became a possibility when IBM researcher Craig Gentry developed the first homomorphic encryption scheme in 2009. Gentry gave a concrete analogy of homomorphic encryption by describing a jewelry store owner who constructs glove boxes, in which only the owner has the key to access the raw materials inside, but employees can manipulate the materials by using the gloves [Gen09]. Manipulations can be done to the raw materials using the gloves, similar to computations done on ciphertext. Then, when the time comes to access the raw materials the owner can unlock the box, similar to decrypting the data.

Since Gentry developed this scheme, a rich variety of alternative approaches have been proposed, but researchers have yet to bring homomorphic encryption to the point where it is efficient enough for practical uses. Although the optimization of homomorphic encryption has improved over time, it is still a long way away from real time processing.

1.2 Applications of Homomorphic Encryption

Some of the possible applied areas where homomorphic encryption could be implemented into are national security, health care, and voting systems. Homomorphic encryption could have a distinct, but significant impact on each of these areas, as described in the following sections.

1.2.1 National Security Applications

One future use for homomorphic encryption could be to monitor grid networks around the country. Take a smart grid network, for instance, where each node produces data which must be monitored by a larger smart grid. Data from a node must be sent to the smart grid to be continually monitored. As of now, computations and analysis of this data are done within the specific group who is monitoring the grid network. At a Crypto Standardization Workshop hosted by Microsoft Research, the possibility was discussed that this data can be outsourced to be monitored and computed on a public cloud with the use of homomorphic encryption [Arc14]. Homomorphic encryption would allow the government, or whatever body is overseeing the grid network, to securely outsource the data taken from it for analysis.

This outsourced data sent from a grid network might be analyzed to determine the grid's necessary distribution of power. Computations are performed on this data before it is analyzed, making it susceptible to exposure to others wishing to gain knowledge of it. Homomorphically encrypting the data will protect it from possible attacks aimed toward tampering with or shutting down a grid network. Homomorphic encryption was also proposed as the answer to this because of the reduction in hardware needed to implement it, as opposed to other encryption schemes. Other encryption solutions would require multiple secure servers in order to be implemented, while homomorphic encryption would only need a single server in the cloud.

1.2.2 Healthcare Applications

Healthcare systems handle data specific to individual patients that must be kept confidential to ensure each patient's personal privacy. Although such information on specific patients must be kept protected, it must still be available to professionals for the day-to-day operations to occur. Personal Health Information (PHI) must be protected in three phases: acquisition, storage and computation. Being able to perform analytics on PHI is necessary to ensure productive long-term health care. While current encryption schemes can handle the secure acquisition and storage of PHI, they do not allow for computation on it. This is valuable for the protection of the patient, but in order for significant progress to occur, computations need to be regularly performed on encrypted data. Many times, doctors from one hospital will have to send patient records to a specialist for review. Although these records are normally encrypted in a secure location, an unencrypted form of the records must be sent. Whenever unencrypted data changes location there is always the possibility for attacks. The only way to securely have the data change locations would be to keep it encrypted.

Homomorphic encryption is given as a viable alternative to the encryption schemes and data protection methods currently used. This would allow computations to be performed on encrypted patient data [Koc14]. Since PHI is the most common target for potential hackers, patient security will improve along with the healthcare benefits. Homomorphic encryption would allow analytics to be run on PHI through long term patient monitoring on a cloud-based system. Additionally, patient trust in healthcare organizations is vital for the best services to be offered. Securely storing PHI with homomorphic encryption will improve patient experience by knowing their data is secure. If effective, this could greatly increase and enhance the services offered through health care providers to their patients, while ensuring no personal information is at risk.

1.2.3 Voting Systems

Around the world, almost every country has implemented some type of voting system to allow citizens to have a say in who will lead them in the future. Most countries have now even expanded this to allow their citizens to vote on specific laws or bills brought up by the government to be put into place. In the United States, adults are able to vote for the next occupants of the Presidency, House of Representatives, and Senate in federal elections, but also have the right to vote in the state and local elections for the state in which they reside. Since each state runs these elections, the voting procedures vary from state to state when it comes these smaller elections [ACE16]. To the community, voting allows people to influence the decisions made by elected government officials. However, the integrity of this process is in danger when accusations of voter fraud are brought up. To ensure fair and honest elections, a voter's identity must be kept anonymous on their individual ballot and each vote must be counted equally when they are compiled.

Homomorphic encryption could be the solution to the issues discussed regarding voting procedures. Proofs of concept have been created to utilize homomorphic encryption to give voters the security they deserve. Homomorphic encryption can be used in schemes to allow voters to securely vote at a given station, then check later that their vote was accounted for correctly [Hen16]. Although it does not seem feasible at the moment, through future optimization of homomorphic encryption and development of technology, a scheme like this could eventually be implemented into voting systems. The benefits of this implementation, as stated before, would be the security for voters to know there is no coercion or voter fraud occurring in any election having a direct impact on them. Note that this example indicated how the nature of the application dictates what sorts of computations on encrypted data are called for; applications involving only a limited instruction set are more easily achievable than others.

Chapter 2. Background

The Central Processing Unit (CPU) and Graphics Processing Unit (GPU) are two of the most important pieces of hardware for computing. Every year, big name manufacturers like Intel and NVIDIA work hard to make improvements to their products. Since both technologies are highly developed, slight boosts to the capabilities of these components make a big difference when it comes to computing power and, thus, to market share. Though the CPU is still "king" when it comes to its performance on highly serial compute tasks, the GPU has recently received more attention as programs take advantage of data parallelism inherent in applications such as graphics and gaming.

2.1 Central Processing Unit (CPU)

A central processing unit, or CPU, is the core of a computer: the CPU is able to handle tasks while allocating more complicated ones to specific chips. A CPU is built by placing thousands of transistors onto a single computer chip, allowing it to make the necessary calculations to run programs stored on the system's memory [Mar18]. The principle process of the CPU involves taking an instruction from the system's random-access memory (RAM), decoding the instruction, then executing the instruction or arithmetic. Originally, CPUs were built with only a single processing core. Modern CPUs now have multiple cores built into them, some with up to eight, twelve, or even more cores. The clock speed (rate at which the CPU can execute instructions) of modern CPUs has also drastically increased over the past decade. For example, the 4th Generation Intel Core i7-4900MQ processor, released in 2013, has 4 cores and a processing base frequency of 2.8 GHz. Compare this to the current Intel Core i7 iterations, the 9th generation Intel i7-9700k, which has 8 cores and a base processing frequency of 3.6 GHz, with the ability to increase to as much as 4.9 GHz [Int1].

Advanced CPUs can now implement multi-threading, which essentially creates virtual cores. This allows the processor to execute multiple instructions simultaneously. It is able to do this through the use of simultaneous multithreading (SMT). Simultaneous multithreading allows processors to work on more than one thread at a time by having the front end of the processor alternate to different threads through time sharing. Once the individual cores execute the instructions, the instruction will be matched to the correct stream later on. Although effective for certain tasks, there are ways to utilize more cores and threads for parallelization with the use of a graphics processing unit.

2.2 Graphics Processing Unit (GPU)

Graphics processing units (GPUs) take computer processing a step further than a CPU. Originally designed to handle high-speed 3D game rendering, the capabilities of GPUs are now being harnessed to increase general-purpose computing power [Kre09]. The main difference between this and the previously mentioned CPU is the number of cores. A basic GPU is designed with hundreds of cores that are able to run thousands of threads concurrently. In essence, a GPU is hundreds of CPUs all on a single chip. You can think of the CPU like the brain of the computer. It gives instructions to the body, jumping back and forth between different tasks. The GPU, on the other hand, is like the body; it takes instructions from the brain and executes them.

While GPUs are very powerful and can compute thousands of instructions at a time, they do have a few limitations. For starters, the processing cores that a GPU has can only do very basic calculations, while cores in the CPU can do more complex instruction sets such as manage virtual memory. Also, the CPU has a faster clock rate than the GPU, which is part of the reason why it can interact with all of the different input and outputs of the computer's components. A final constraint is the limited flexibility of Single Instruction Multiple Data (SIMD) in a GPU. In essence, SIMD is a processor's method for parallelizing work. In a CPU, it is able to pipeline multiple instructions to allow a program to run faster. However, it is difficult and time restrictive for a GPU to switch instructions, making it ideal for executing large volumes of data with the same instruction. The CPU is quick and nimble, able to switch between multiple processes and requests very fast, while the GPU is not agile but very fast and efficient at doing one set of instructions hundreds to thousands of times.

An example where the parallel capability of the GPU comes in handy is in graphics rendering. A modern GPU such as the NVIDIA GTX 1080 has 2560 shader cores. Thus, it can execute 2560 operations during one clock cycle. When you need to make a small change in the brightness of a display screen, for example, this is very helpful. By comparison, a four-core Intel i5 CPU can only execute four simultaneous instructions per clock cycle [Fox07].

The GPU we used for most of our testing was an Amazon Web Services (AWS) EC2 P3. The P3 instance uses an NVIDIA Tesla V100 GPU. The NVIDIA Tesla V100 is based on the Tesla GV100 architecture. The GV100 contains six GPU Processing Clusters (GPCs). Each GPC contains seven Texture Processing Clusters (TPCs), each with two Streaming Multiprocessors (SMs). This gives a total of 84 SMs. The Tesla V100, however, only uses 80 of the SMs available in the GV100, so 80 operating streams can be computed at a time. There are also eight 512-bit memory controllers, for a total of 4096 bits. Each SM contains 64 FP32 cores, 64 INT32 cores, and 32 FP64 cores. This gives totals of 5120, 5120, and 2560 of each type of core, respectively. The cores are named for the data types they operate upon: integer (INT) or floating point (FP) and 32 or 64 bits.

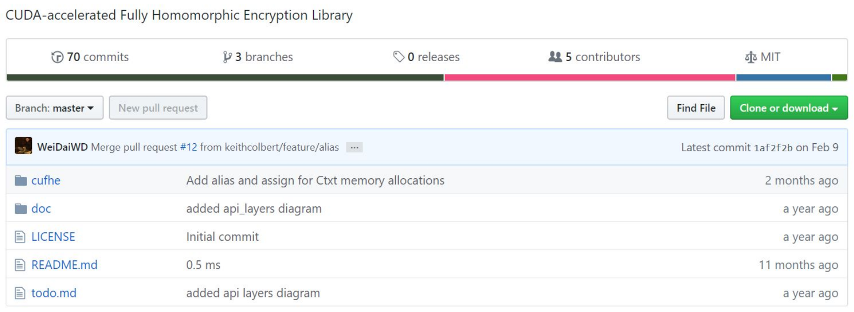
2.3 Benefits of using GPU over CPU for Homomorphic Functions

Homomorphic functions will inherently require high processor power. Unlike conventional functions that can be relatively small and work on plaintext, homomorphic functions work on ciphertexts which are orders of magnitude larger than plaintext. One plaintext bit equates to thousands of bits when encrypted. This means that for every bit a normal function would operate on, a homomorphic function must operate on thousands, causing the run times on homomorphic functions to be quite slow. Graphic processing units are used to speed up homomorphic function run times for two reasons: high core count, and parallel processing capability. The large number of cores is ideal when doing simple operations on large ciphertext values. A 32-bit CPU with only 4 cores would only be able to work on 128 bits at a time, while a GPU with thousands of cores can work on tens of thousands of bits. When doing the simple operations required for an ALU, it is clear these basic but plentiful cores are more efficient. In addition, many processes in homomorphic functions can be done in parallel, which means they can take full advantage of the parallel computing capabilities of the GPU. The GPU can run multiple different operations at a time on its multiple streaming multiprocessors, opening up the possibility for functions to speed up run time through parallel operations.

2.4 cuFHE

There are numerous implementations of homomorphic encryption available online. One such implementation is the `cuFHE` library, developed at WPI by Dai and Sunar. The `cuFHE` library is publicly available on Github and uses the CUDA[1] platform to accelerate the computation of homomorphic operations. It has been heavily optimized to achieve a high throughput of homomorphic operations. Using the NVIDIA Titan X GPU, the time per gate is 0.5 msec, making `cuFHE` the fastest available library for homomorphic encryption. This library provides homomorphic functions in two distinct languages: C++ and Python. We will primarily discuss the C++ implementation, but the Python implementation is very similar. In fact, the Python implementation provides wrappers to the C++ structures and functions. Additionally, the library provides both CPU only and GPU-accelerated versions of most functions. We will discuss the GPU-accelerated versions except when noted otherwise.

[1] Compute Unified Device Architecture



CUDA-accelerated Fully Homomorphic Encryption Library

70 commits | 3 branches | 0 releases | 5 contributors | MIT

Branch: master | New pull request | Find File | Clone or download

WeiDaiWD Merge pull request #12 from keithcolbert/feature/alias ... Latest commit 1af2f2b on Feb 9

cufhe	Add alias and assign for Ctxt memory allocations	2 months ago
doc	added api_layers diagram	a year ago
LICENSE	Initial commit	a year ago
README.md	0.5 ms	11 months ago
todo.md	added api layers diagram	a year ago

Figure 2.1. cuFHE Github.

The cuFHE library provides various structures to support the creation of and operation on plaintext and ciphertext data. The Ptxt and Ctxt structures represent single bits in unencrypted and encrypted form, respectively. The Ptxt structure contains an integer representation of the bit, either 1 or 0. The Ctxt structure contains pointers to the host and device memory which holds the encrypted bit. Due to the nature of the encryption scheme, the encrypted bit requires significantly more memory than the unencrypted one. The default in the cuFHE library is 2 kB. The host pointer points to memory on the CPU. This pointer will be passed to functions which operate on the CPU. The device pointer points to memory on the GPU, and this is passed to functions which operate on the GPU. Additionally, there are PriKey and PubKey structures which contain information about the keys. The PriKey contains the private key, used to encrypt and decrypt bits. The PubKey contains the public key, which is used by the homomorphic functions to operate on Ctxt structures.

The encrypt and decrypt functions occur in the CPU. The encrypt function takes the input Ptxt, the output Ctxt, and the PriKey which will be used for encryption. The decrypt function takes the input Ctxt, the output Ptxt, and the PriKey. The encryption and decryption operations are not as CPU intensive as the gate functions, so they do not require GPU acceleration. Additionally, this fits the client/server use cases of homomorphic encryption. The client only uses the encrypt and decrypt functions and sends encrypted bits to a server. The server performs operations on the bits and returns the encrypted results to the client. Finally, the client decrypts the results. In this model, the client does not need any special hardware beyond a basic CPU. The server is the only machine which requires GPUs.

The library provides various logic gate functions which operate on ciphertexts. These gates include Not, And, Or, Xor, Nand, Nor and Xnor. With the exception of Not, these functions follow a similar model. The part of the function which actually operates on the ciphertexts is called GatenameBootstrap, where "Gatename" is the name of the corresponding logic gate. These functions calculate the result and perform bootstrapping on the output. Since the host and device memory are separate, these functions also handle copying of the ciphertexts. First, the input ciphertexts must be copied from host memory to device memory. Then, the GatenameBootstrap operation is called. Finally, the resulting ciphertext is copied from device memory to host memory. The `Not` function is not as computationally expensive, so it is implemented entirely on the host side. This removes the need to copy memory to and from the device. Additionally, there is a `Copy` function, used to copy the value of one ciphertext to another, which does not require the device.

The GPU-enabled logic gate functions each take two input ciphertexts and one output ciphertext. The input ciphertexts are declared const, which acts as a promise that they will not be modified by the function. The output ciphertext is obviously modified, as it will hold the result of the operation. The last argument is a `Stream`. This is used to specify how the GPU will perform the operation. Generally, this will be a non-default stream. This causes the function to operate asynchronously. This means it can run simultaneously with other operations on the GPU. Additionally, the function will return immediately on the CPU rather than block until the operation is complete. If, instead, the default stream is passed, the operation will be synchronous with the host and within the GPU. In other words, only one operation will occur at a time in the GPU, and the CPU-side function will not return until computation on the device is complete. Additionally, the functions which copy memory will be synchronous to the host, so it is guaranteed that the ciphertext will be in host memory after the gate function returns.

2.5 Arithmetic Logic Unit (ALU)

The arithmetic logic unit (ALU) is a core component to much of modern computer hardware, including Central Processing Units (CPU) and Graphics Processing Units (GPU); a single CPU or GPU may contain multiple ALUs. The ALU performs bitwise functions (AND, OR, XOR, NOT), as well as mathematical functions on inputs stored in binary. Common ALU arithmetic functions are addition, subtraction, multiplication, division, and two's complement. ALUs also have status input and outputs for dealing with carry-ins, carry-outs, overflows, and more. These units can be split into integer and floating-point units, which work on integer or floating-point operands, respectively. In essence, an ALU is a binary calculator that forms one of the main building blocks of modern computing.

The "fetch–decode–execute cycle" is the basic operational process of a computer system. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction describes, and then carries out those actions. This cycle is repeated continuously by a computer's CPU, from boot-up until the computer has shut down. As can be seen in Figure 2.2, the ALU is a central component in this "fetch-decode-execute" style of operating. The ALU performs the "execute" part of the cycle, and thus makes up the last step in the cycle of operation.

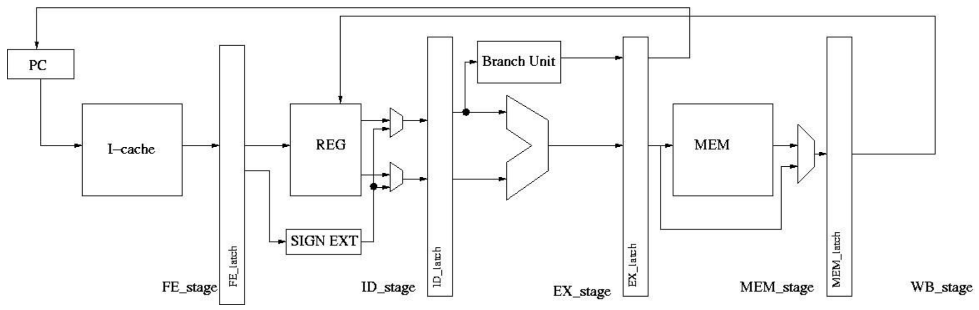


Figure 2.2: Fetch Decode Execute Cycle.

Traditional ALUs do not have a multiplexer as an operation; instead, the CPU uses a multiplexer (MUX) to evaluate conditional instructions such as branch and jump. We decided to include one in our set of homomorphic functions. Due to the added complexity from working on encrypted operations, the multiplexer proved a critical operation for almost all of the ALU functions. A multiplexer is a simple primitive which takes multiple inputs and selects one to connect to the output. The larger the number of inputs, the more selectors are needed for the output.

It is important to understand why creating an ALU for homomorphic encryption is difficult. Unlike normal ALUs, a homomorphically encrypted ALU must be able to operate on ciphertext instead of plaintext. This takes away the ability for the function to know what data it is operating on. Because of this we had to completely redesign the current arithmetic operations algorithms. In essence, homomorphic functions must act like circuits. For example, looking at the following conditional statement:

$$\begin{array}{l} \mathit{If}\ (X == 0) \\ \qquad Y = A; \\ \mathit{Else} \\ \qquad Y = B; \end{array}$$

The X value is a ciphertext, so there is no way of checking its value. To get around this, you must evaluate the results for both $X = 0$ and $X = 1$, and then use a multiplexer with X as the select bit to select the correct value. When trying to create more complex functions, this issue becomes much more difficult to deal with, causing a lot of inefficiency. However, there is room to lighten this inefficiency through smart manipulation of a function’s Boolean arithmetic equation. For example,

$$Y = A(1 - X) + BX = A + X(B - A)$$

Here, by manipulating the equation we can chose to do either one multiplication, shown in the first equation, or two multiplications, shown in the second equation. This can be used to help make functions more suited for parallelism, decreasing the overall run time of the equation.

Chapter 3. Integer Gate Functions

This section discusses the design and functionality of our integer gate functions within the `cuFHE` library. The final products were a multiplexer, an adder, a subtractor, a multiplier and a divider. We built multiple gate functions for some of the arithmetic operations in order to test different approaches and find the most efficient method in terms of runtime. Both design details and testing data are explained for each gate function in the following section.

3.1 Multiplexer 2-1

A 2-1 multiplexer (MUX) takes two *n*-bit inputs and one select bit and produces one *n*-bit output. This function requires *3n+1* gates to complete. In the construction of a basic MUX, the select bit is first inverted [Raj18]. Then, we use our `AND` function on the first and second inputs with the inverted select bit and the select bit, respectively. This has the effect of passing the selected bit forward while replacing the other input by a zero. Finally, we use our `OR` function on the results of the `AND` operations together to give the final result. See Figure 3.1 for a visual of these gate operations. Utilizing a multiplexer in our gate functions allows us to optimize these functions by decreasing the number of dependencies at the cost of discarding computational results computed in parallel. This specific multiplexer is used in some of our addition gate functions to choose between outputs. We can perform arithmetic operations by assuming all scenarios of the carry-in, then use a 2-1 multiplexer later to determine the correct output.

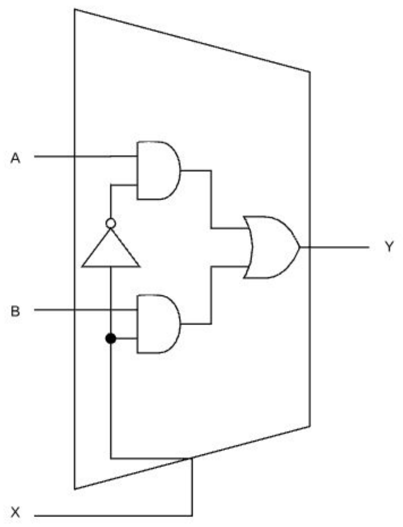


Figure 3.1. 1-bit 2-1 Multiplexer.

Below, Figure 3.2 shows our runtime data for our 2-1 MUX function. Our runtimes range from about 55 msec to 180 msec for 4-bit to 32-bit tests. Since the multiplexer is a fundamental building block and it is utilized by all of our gate functions, we optimized its design before moving on to more complex functions. It is important to note that there were two choices of design for this multiplexer. We will compare the two possible circuit designs later in section 3.2.4.

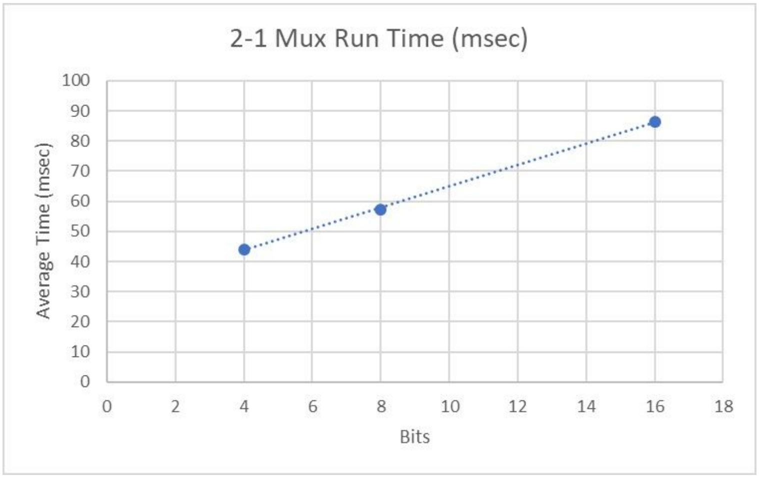


Figure 3.2. 2-1 MUX Runtime

3.2 Adders

In total, we constructed four different adder functions. These included a full adder, a carry propagate adder, a carry lookahead adder, and a carry select adder. This was done to see which could be optimized the most for ideal performance. Recall that these adders are working on encrypted data, so the gate operations that make up these functions run much more slowly than they would if working with plaintext data. Since each adder has a different design structure, various methods were explored to improve efficiency.

3.2.1 Full Adder (FA)

A full adder (FA) is a simple digital circuit that inputs two bits, as well as a carry-in bit, and outputs the sum of those numbers, as well as a carry-out bit. This is done with a total of five gate operations. Because the FA is a small, well known circuit, there were no serious complications caused by working with homomorphic bits. The FA is a building block for the more complicated functions in an ALU, therefore having a simple and reliable design is critical to the functionality of the ALU as a whole. See Figure 3.3 for details on the structure of a generic FA.

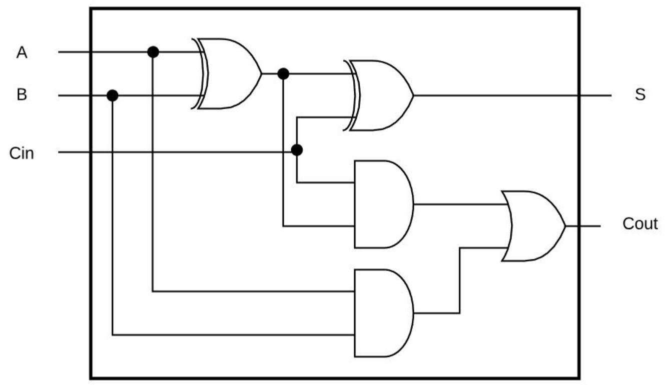


Figure 3.3. Full Adder Diagram

Because of its relative simplicity, our FA function had a fast runtime on the GPU of 0.17 msec. However, when trying to run a function that requires many calls to FA, such as a shift add multiplier (see Section 3.4.2), the FA function runtime accumulates to a significant overall operating time.

Intel i5 5200U CPU	Titan X GPU	Tesla v100
450 s	45 msec	0.17 msec

Table 3.1: Full Adder Runtime Hardware Comparison.

3.2.2 Carry Propagate Adder (CPA)

A carry propagate adder (CPA) is a digital circuit which produces the arithmetic sum of two binary numbers. The construction of a CPA consists of multiple full adders connected in a cascade, as shown in Figure 3.4, with the carry-out from each full adder connected to the carry-in of the next full adder in the chain [Che19].

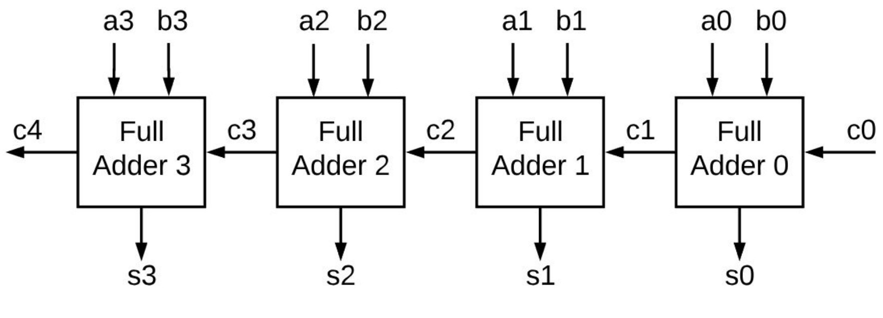


Figure 3.4: Carry Propagate Adder Diagram.

The benefits of the CPA approach are its simplicity and brief design time. This simplicity comes with a reduction in speed compared to more modern adder layouts. The slow speed is due to the delay caused by the propagation of the carry bit through the whole adder. Looking at Figure 3.4 above, Full Adder 1 cannot perform its addition until Full Adder 0 has completed its addition and sent the carry-out, *c1*, to Full Adder 1. Full Adder 2 cannot perform its task until Full Adder 1 has finished its addition and sent *c2*, and so on. In effect, no parallel work can be done, slowing down the total completion time of the CPA to the combined total runtime of all of the full adders that make up its design.

We implemented our `CPA` function by using a simple loop using our `FA` function, replacing the carry-out of the current addition with the carry-in of the next addition. This repeats until all of the bits have been added. Our implementation uses five gates per bit and has a depth of $5n$ for n bit inputs.

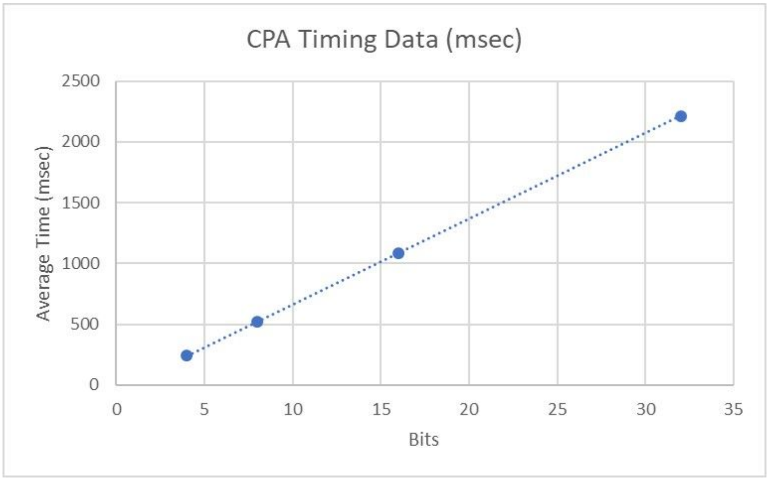


Figure 3.5. `CPA` Runtime.

The testing data for our `CPA` function shows us a linear relationship between depth and runtime. This is consistent with our equation for depth, which was $5n$, as stated above. Below in Figure 3.6, a timing diagram illustrates our `CPA` run with one, two and three streams. Stream 15 shows the `CPA` run with one stream. As you can see, it has a depth of 17 gate operations and an occupancy of 100%. As we add another stream, shown in Stream 16 and 17, the depth decreases to 9 gate operations, and the occupancy decreases to 94%. Finally, Stream 18, 19 and 20 shows three streams with a depth of 7 total gate operations and an occupancy of 81%. As we add streams, not all of the GPU is being utilized, but the overall latency improves. In scenarios when many `CPA` operations need to be done simultaneously, one stream would be the most efficient because it utilizes the GPU the most. In most cases, when latency is the most important factor, adding multiple streams will decrease the depth of the operation and improve the run time.

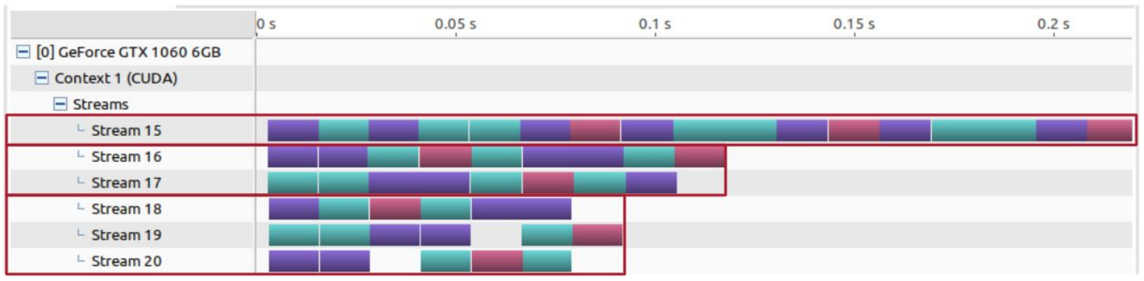


Figure 3.6: CPA Timing Diagram.

3.2.3 Carry Look Ahead Adder (CLA)

The limiting factor in the carry propagate adder model is the propagation delay caused by each adder block having to wait for the carry-in to arrive from the previous block. The total propagation time is equal to the propagation delay of an individual adder block multiplied by the number of adder blocks involved in the circuit. The carry lookahead adder (CLA) reduces this propagation delay by reducing the carry logic for each group of bits to a two-level logic [Wan13]. See Figure 3.7 below for a diagram showing how this is possible.

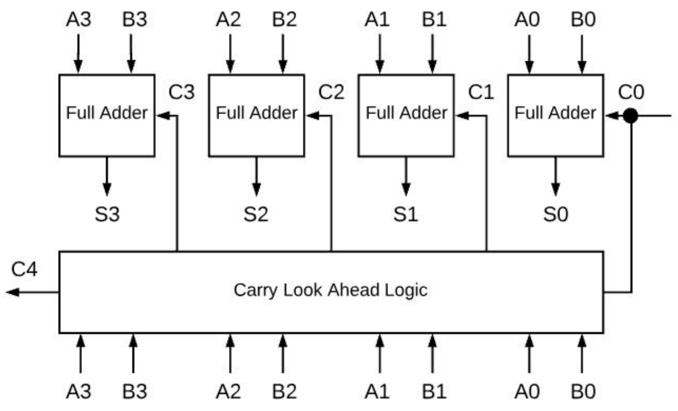


Figure 3.7: Carry Look Ahead Adder Diagram.

If you break down each full adder circuit into Boolean equations, you can see how the propagation delay is reduced. Start with the equation for the propagation bit (P_i) and the carry generated bit (G_i).

$$P_i = A \oplus B$$
$$G_i = AB$$

Use these equations to find the sum (S_i) and carry-out (C_{i+1}).

$$S_i = C_i \oplus P_i$$

$$C_{i+1} = G + P_i C_i$$

Using these equations to demonstrate a 2-bit carry look ahead adder, it can be shown that no output value is dependent upon a previous output.

$$C_1 = G_0 + P_0 C_{in}$$
$$C_2 = G_1 + P_1 C_1$$
$$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$$

Therefore, C_2 does not need to wait for C_1 to finish and can propagate at the same time as C_1. In other words, C_1 and C_2 are independent equations that can be processed on different devices. If this were a 4-bit add or larger this same pattern would continue.

3.2.4 Carry Select Adder

A carry select adder utilizes two adder functions and a multiplexer to calculate the sum of two *n*-bit numbers. Each bit is calculated twice using the adders, once assuming the carry-in is zero, and the second with the carry-in set as one [Par13]. The correct sum and carry-out is then selected using a multiplexer once the carry-in is known. The addition should ideally be broken into blocks, with a certain number of bits in each block, to properly optimize its performance. This can be designed with a recursive structure to have the multiplexer choose between two *n*/2-bit inputs. A block diagram of this structure is shown below in Figure 3.8 below.

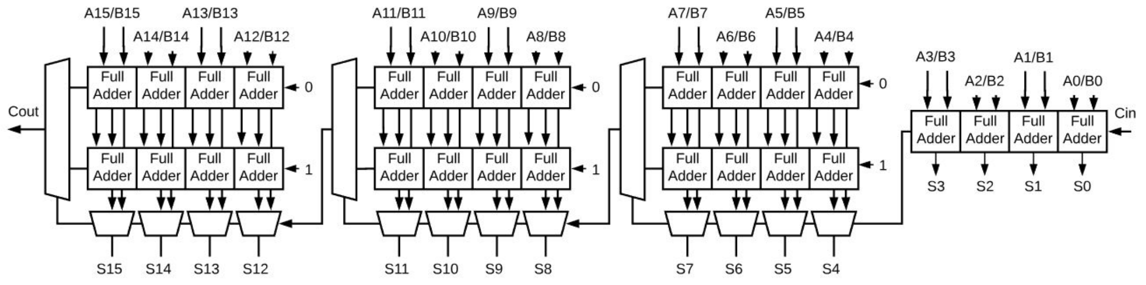


Figure 3.8: `Carry Select Adder` Diagram.

In our design, we used the `CPA` function we previously created and a modified 2-1 `MUX` function. Each bit of addition calls the `CPA` function twice, setting the carry-in to zero and one. The 2-1 `MUX` then uses a series of AND and OR gates to check the carry-out of the previous bit. The carry-out of the previous bit is used as the carry-in for the current bit being added. With the correct carry-in chosen, the 2-1 `MUX` function then finds the correct sum to use. For example, if the carry-out of the previous bit was 0, then the sum using a carry-in of zero is used. We utilized recursion to split the addition into blocks as previously mentioned. We did this by using an IF-ELSE statement. If the argument passed by our `Carry Select Adder` is two or more bits, then our `Carry Select Adder` will be called recursively with a length of *n/2* bits. The recursive operation is on *x+(n/2)*, where *x* is a pointer to one of the arguments passed into the function. In essence, this adds the first half with the second half of the argument. If $n < 2$, then our normal carry propagate adders are called (because no more splits can be done). Once this IF-ELSE statement is complete we use our original multiplexer to find the final output.

We came to a design decision when deciding which 2-1 `MUX` we wanted to use. Our first 2-1 `MUX` worked by doing two AND gates followed by two more AND gates, followed by a final two OR gates, while the other did four AND gates followed by two OR gates. In effect, the first design took one extra gate delay longer than the second but used only two streams as opposed to the four streams used by the second `MUX`. We decided the faster `MUX` would be best for our design, as time efficiency was our goal and we wanted to have the `Carry Select Adder` be as parallel as possible. After this decision, we can compute the total number of parallel operations (and therefore streams needed) with the following equation:

$$\#\ of\ parallel\ operations\ = N * 1.5^{s}$$

In this equation, *N* is the number of bits and *s* is the number of splits in the `Carry Select Adder`. For example, an 8-bit number with 3 splits would require around 27 streams. This is less than the 30 SMPs in the Titan X GPU, which is why we decided to stick with three splits for our `Carry Select Adder`. With better hardware more splits could be made to speed up the operation, but for an 8-bit number three splits are the maximum that can be done anyways.

The final implementation of our `Carry Select Adder` function takes two parameters which control the depth of the recursion: the number of bits (*n*) and the number of available SMs. Our `Carry Select Adder` requires *n* SMs to efficiently multiplex the results in parallel. Our `CPA` only requires one stream. Since the function splits the input into three additions of *n*/2 bits, 3*n*/2 streams are required to perform these additions using the `Carry Select Adder`. If enough streams are available, the additions will be performed using recursive calls to the `Carry Select Adder`. Otherwise, they will be performed using our `CPA` function. This achieves the greatest possible parallelization with the available resources. It is important to recall that, for the functions we created, we treated processors as unlimited, with the primary goal being the fastest completion time possible for a given operation.

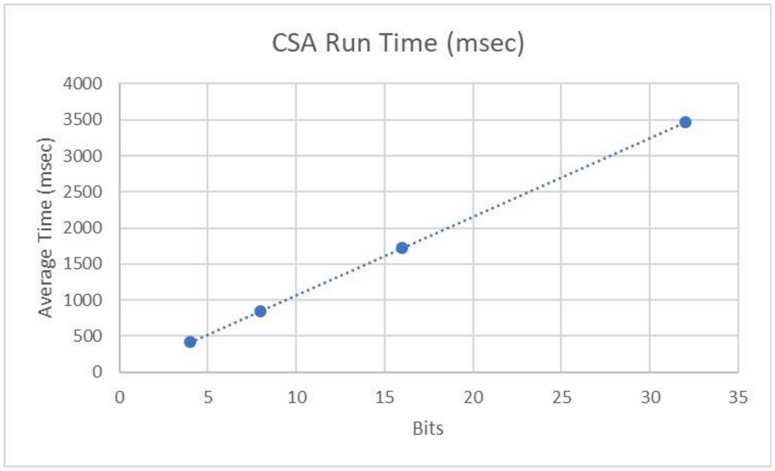


Figure 3.9. `Carry Select Adder` Runtime.

Although, in theory, our `Carry Select Adder` should have a faster runtime than our `CPA`, testing results showed it to be significantly slower. In fact, it ran at an average of two times slower than the `CPA`, as seen below in Figure 3.10.

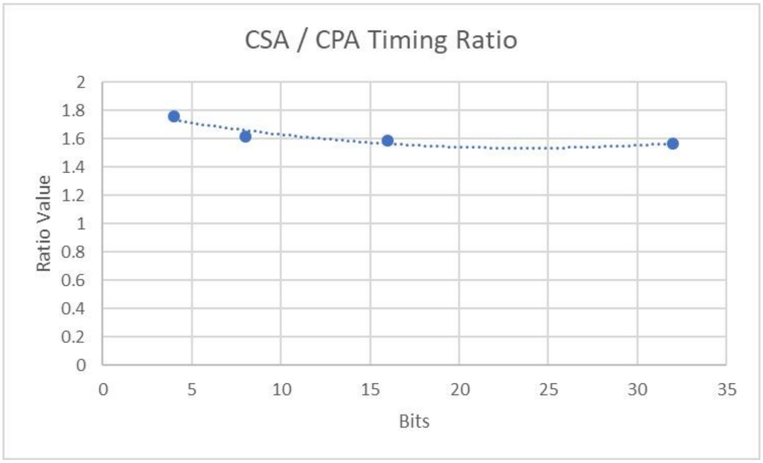


Figure 3.10. Ratio of `Carry Select Adder` over `CPA` Timing.

The slow speed of out `Carry Select Adder` versus the `CPA` is due to an issue with the GPU not running operations in parallel. One option was to run each `CPA` in parallel on different streams. Instead the GPU delays the starting of a new stream until the stream before it has completed its action. This takes away the main advantage of the `Carry Select Adder`: parallel processing. As our `Carry Select Adder` function is used in many other functions, solving this parallelization problem should significantly improve runtimes on future function.

To further explain our parallelization issue, we created timing diagrams for our `Carry Select Adder`. Below, Figure 3.11 shows a timing diagram for a single 8-bit addition using the non-recursive `Carry Select Adder`. The first three blocks of operations are 4-bit additions using the `CPA`. These should run concurrently, as they are performed in different streams and there is no dependency. After the additions, the final result is selected from the speculative results. This is done using a multiplexer, which does run in parallel. The total depth is 26 gate operations.

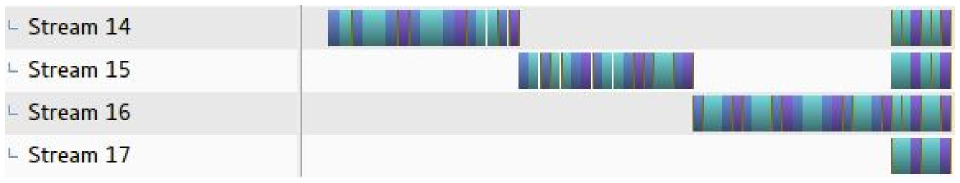


Figure 3.11: Timing diagram for non-recursive `Carry Select Adder`.

Below, Figure 3.12 shows an 8-bit addition using the fully recursive `Carry Select Adder`. Note that not all of the streams are shown. Again, there is an issue preventing the operations from running concurrently, but the stream utilization increases, and the total depth is less (~20 operations). This could be further improved by using *2n* streams for the multiplexer wherever they are available. For example, the block of operations in the upper right of Figure 3.11 could be performed using 16 streams with a depth of two gate operations. This would reduce the depth further, to approximately 10 gate operations for this example.

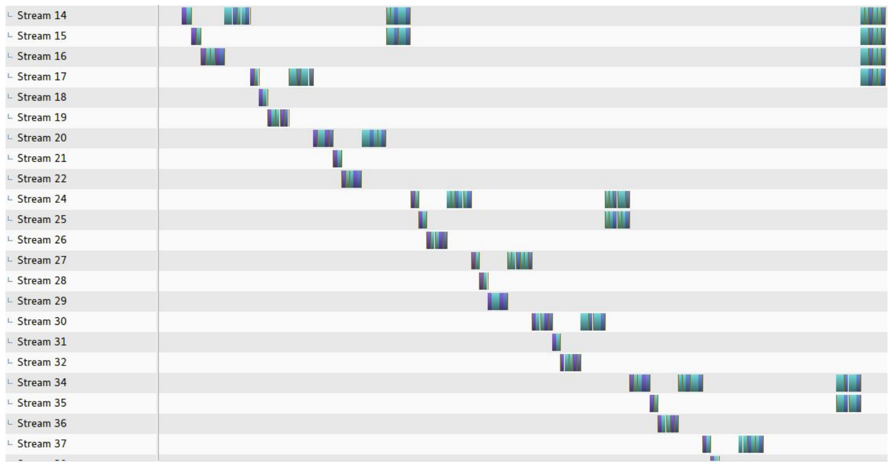


Figure 3.12: Timing diagram for recursive `Carry Select Adder`.

To resolve the problem of memory accesses causing synchronization with the host, we implemented a memory management unit for both the host memory and device memory. We pre-allocate enough blocks for the temporary ciphertexts used by functions such as `MUX` and `Carry Select Adder`. Pointers to this memory are provided during function execution, but this does not cause synchronization with the host.

Unfortunately, the available SMs are not being used to their full potential. For each gate function, the two input ciphertexts are copied to the device. In Figure 3.13, these memory operations are shown as very narrow lines. There appears to be a large delay between the first and second arguments being copied for some of the function calls. We suspect this could be an issue with priority. The desired order of the function execution differs from the order in which they are sent to the GPU. If the GPU prioritizes memory access in issue order, this could be the cause for the delays.

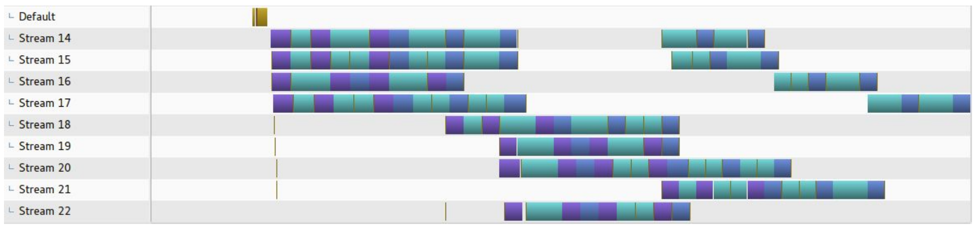


Figure 3.13. Timing Diagram for 8-bit `Carry Select Adder`.

As the reader can see in Figure 3.14 below, with these memory access issues finally resolved, the 8-bit `Carry Select Adder` runs much more efficiently. This now has a depth of only 13 gate operations, which is half of what the total depth was with the synchronization issues. One can also see the GPU is utilized much more than in the previous examples. This example uses 6 total streams and has an occupancy of 88%. If more streams were utilized for this example, the GPU would not be utilized to its fullest, but the latency could be improved even more. As previously mentioned, with so many cores available with the GPU, total usage is not as high a priority as latency in these examples.

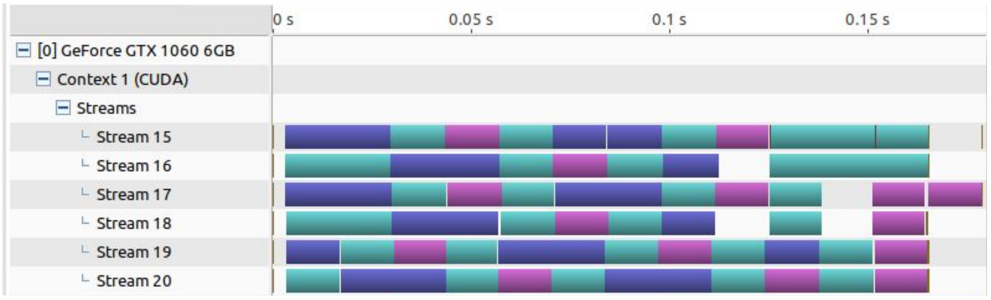


Figure 3.14. Timing Diagram for 8-bit `Carry Select Adder` with Parallelization Fix.

3.3 Subtractor

In order to create a subtraction function, our first step was to implement a two's complement element to change the notion of one of the inputs. The premise of two's complement is inverting all of the bits of a number and then adding one to it. This is a schematic way of changing the binary representation of a number to make it negative. We were able to accomplish this in our function by looping through the input being subtracted and applying a NOT gate to all of the bits. A `CtxtList` of the same length as this input is created and filled with zeros. By applying a NOT gate to the first bit of this `CtxtList`, we were able to give it a value of one. Using the carry propagate adder, we added our inverted input with the `CtxtList`. With this step, we compute the two's complement of the input.

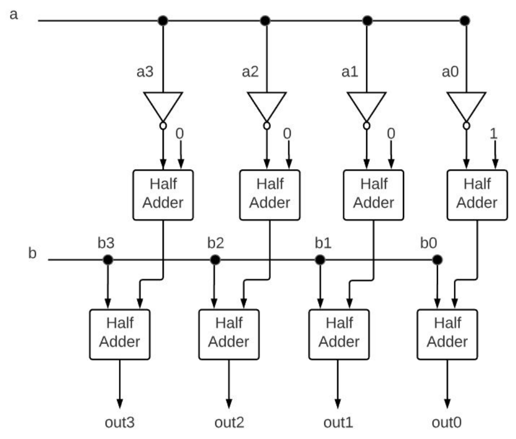


Figure 3.15: 4-bit subtractor diagram

After implementing two's complement, we now had one of our original inputs and a negative notion of the other input. We were able to add our two's complement result with the other input to find the difference of the two original inputs. In Figure 3.15 above, a diagram of this functionality is shown. The image shows a result if the user wanted to compute b - a.

When moving on to testing our subtractor, some issues arose. Specifically, we ran into a problem when it came to printing negative numbers. In order to prevent confusion in future testing, we added a clause to the `Decrypt` function that converts a negative number to a positive one before printing. For example, when we have a bit value of 1110, instead of printing "14" it will print "-2".

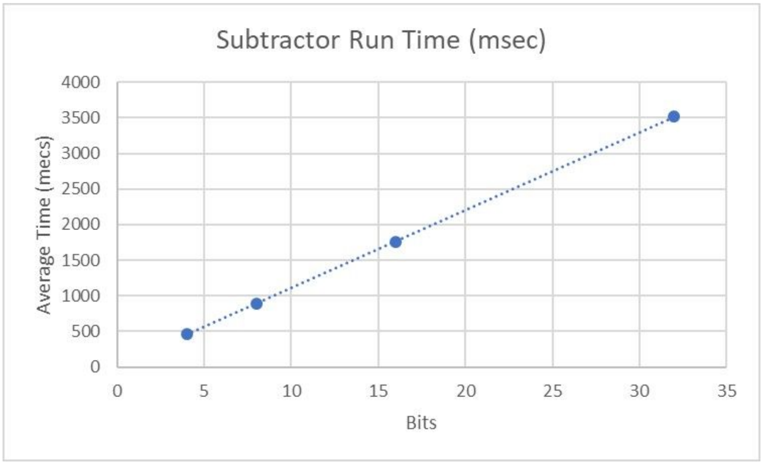


Figure 3.16. Subtractor Runtime

The subtractor timing was almost identical to that of our `Carry Select Adder` function, being only slightly slower by an average of 30 msec. This is unsurprising, as the `SUB` function makes a call to `Carry Select Adder` for the addition mentioned above. The additional time is likely due to the inverting of the subtractor, performed before the addition.

Unsurprisingly, the best way to speed up our `SUB` function would be to optimize our `Carry Select Adder`. The majority of operation time for the subtraction is performing the addition, so speeding it up will yield massive increases in speed for the function.

3.4 Multipliers

The two multipliers we designed were an array multiplier and a shift add multiplier. Although they have different designs, each multiplier utilizes at least one of our adders. Parallelization will be the main difference between the two we will demonstrate.

3.4.1 Array Multiplier

An array multiplier works by implementing a basic add and shift algorithm. Each partial product is found by multiplying the multiplicand with each multiplier bit. These products are shifted depending on their multiplier bit order. A full adder is then used to find the sum of all of the partial products, as described in [Kor08]. This architecture requires *n*-1 adders when the multiplier length is n. The design of an array multiplier is shown below in Figure 3.17.

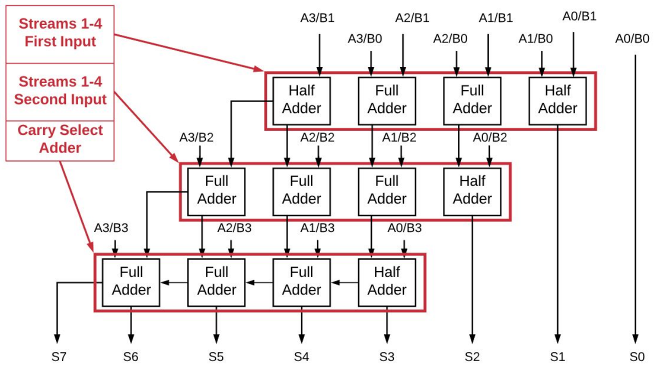


Figure 3.17: Array Multiplier Circuit Design.

In this array design, there are no long chains of dependencies, which allows us to easily break it up into streams and parallelize it. As one can see above, the first row is broken up into streams one through four in the case of a 4-bit multiplier. Once these blocks are completed, those same four streams are reassigned to the second row of blocks. This final row is calculated using our `Carry Select Adder` function mentioned above to find the output. If the inputs to this were larger than 4 bits, the design would just be expanded.

In Figure 3.18 below, testing data for the array multiplier is shown. As one can see, the results are fairly linear from 4 to 32 bits. Since we only used our half and full adders here, the issues with parallelization were not as evident. The main issue with this design is the dependencies on the previous row of adds. For example, the output S2 cannot be found until the output of A2/B0 and A1/B1 is found.

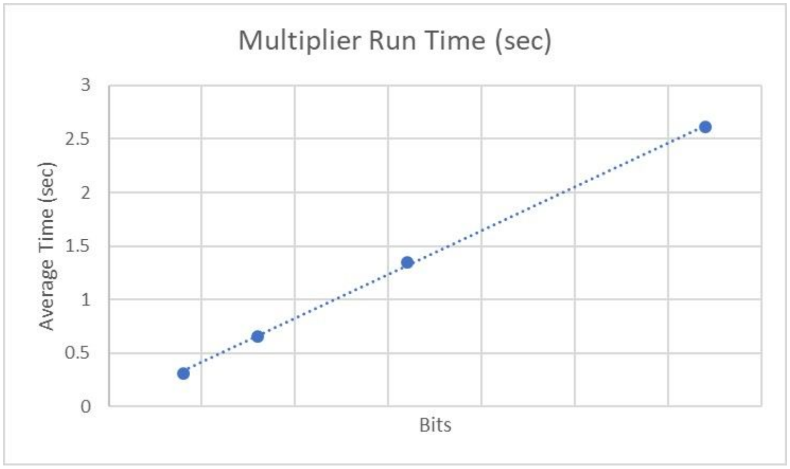


Figure 3.18: Multiplier Runtime.

3.4.2 Shift Add Multiplier

A shift add multiplier works by adding the multiplicand “X” to itself “Y” times, where Y is the multiplier. This is similar to how a person would multiply by hand with pen and paper. To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results. The final product would then be found by adding all of the intermediate products together. Reference [Waw13] explains in the case of binary multiplication; since the digits are 0 and 1, each step of the multiplication is simple. If the multiplier digit is 1, a copy of the multiplicand (1 × multiplicand) is placed in the proper positions; if the multiplier digit is 0, a number of 0 digits (0 × multiplicand) are placed in the proper positions. See Figure 3.19 for an example where we multiply a Multiplicand of 1000 (8) with a multiplier of 1001 (9).

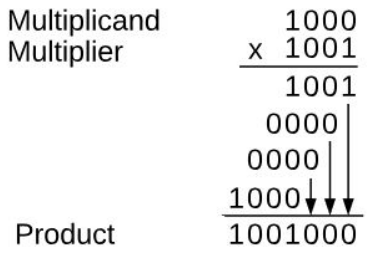


Figure 3.19: Example of a Shift-Add algorithm.

To implement a shift-add multiplier into cuFHE, we made the design decision to require the lengths of both the multiplicand and the multiplier to be the same, and have this length be provided as a parameter passed into the function. We made this decision based on our prior experience with cuFHE functions, noting that inputs of unspecified length cause impractically slow runtimes. We implemented two FOR loops, one nested within the other. The outer loop iterated through the multiplier, while the inner loop iterated through the multiplicand, performing an AND function on the two values and storing them in a temporary array, one for each loop of the multiplier. In effect, each iteration of the outer loop represents the multiplication of the multiplicand by one of the digits of the multiplier. Once all bits of the multiplier have been iterated through, we use our CPA function to add all of the temporary arrays together, giving us the final result of the multiplication.

While designing the shift add multiplier, we realized it would be much slower than the array multiplier because of the design requirements. Although there are fewer dependencies than the array multiplier, since our adder was not optimized, we quickly realized this would have a large impact on its efficiency. Doing multiple n bit adds took too long due to the adder not being correctly parallelized. Therefore, a shift-add multiplier would be impractical slow.

3.5 Division

Our non-restoring division function implements the use of a carry select subtraction unit. In hardware, non-restoring division circuits usually use a shift register to shift the dividend and subtract or add the divisor [Gal07]. In code, we implemented this in the opposite way, by sliding the divisor past the fixed dividend. Since we do not know whether the remainders are positive or negative, both are calculated, and a multiplexer is used to select between them.

Our divider function is designed as an array, somewhat similar to our array multiplier mentioned above. Figure 3.20 below shows how a 4-bit division would be set up using this design. Each block is a controlled add/subtract (CAS). Depending on whether the T input is 0 or 1, the CAS block becomes either an addition or subtraction, respectively. This then creates a critical path that propagates throughout the entire array, as shown in red below. Because of the serial nature of this design, it is difficult to break up into smaller blocks to parallelize it. Our method of optimizing this design was to improve our `Carry Select Adder` and subtraction functions because of their heavy utilization in this function.

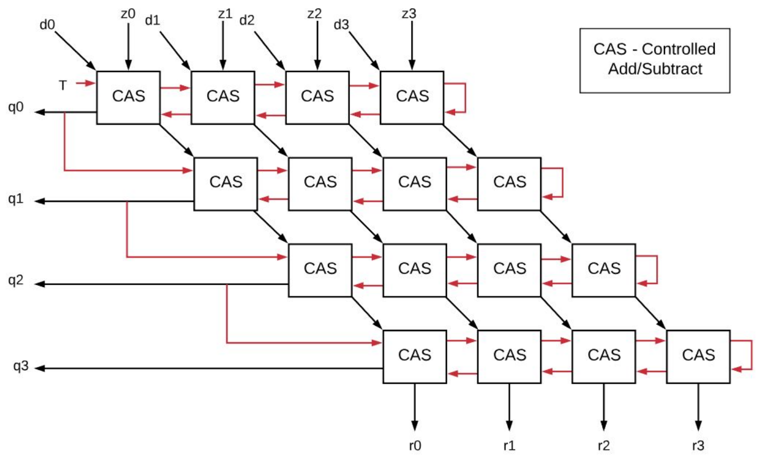


Figure 3.20. Array Divider Diagram.

In Figure 3.21 below, the average runtime of our divider is depicted. As you can see, the runtime starts at about 3.5 seconds for division between two 4-bit numbers, and increases to about 230 seconds for division between two 32-bit numbers. As previously mentioned with our other functions, this inefficiency is due to the parallelization issues we encountered. Since our divider utilizes the `Carry Select Adder`, the inefficiency of the adder leads to a greater inefficiency of our divider. A 32-bit add using the `Carry Select Adder` take about 3.5 seconds. At a minimum, this add function will be called four times in the divider.

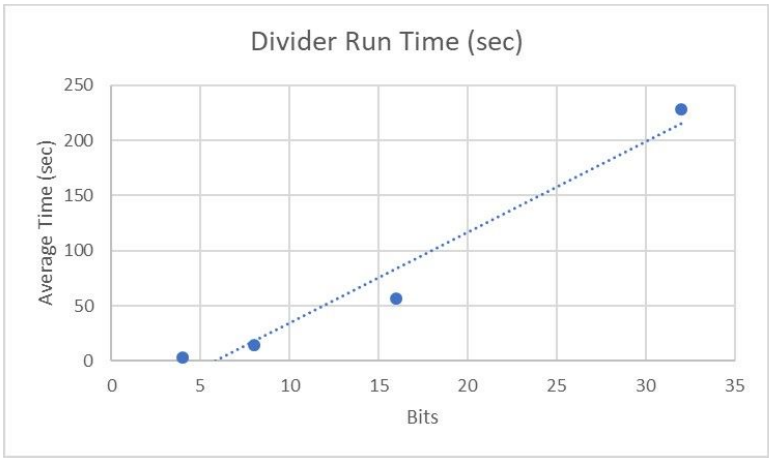


Figure 3.21: Divider Runtime.

Chapter 4. Floating Point Functions

This section discusses the design and testing of our floating point functions. We were able to make progress on a new multiplexer and floating point addition function, but were unable to achieve a fully operational function. The serial nature of many critical steps in floating point operations leads to inherently inefficient code design, especially when dealing with ciphertext. The approach we took with floating point operations differs greatly from our approach to integer operations, as noted in the following section.

4.1 Multiplexer (MUX)

When beginning our design phase of floating point operations, we began by creating a larger multiplexer which would be needed specifically for our addition function. With all the shifting needed for a floating point addition function, which we will discuss in the next section, we found the need to create a 10-1 `MUX`. This would be used to go through all of the possible shifts and then choose the correct one. We were able to simplify the design of this 10-1 `MUX` by breaking it up into multiple 2-1 `MUX`s which we already had available. Through this, we found it to be simpler to expand our design to a 16-1 `MUX` and pad the input to 16 total arrays to choose from. We will select this correct array beginning with the most significant selector bit. We were able to test this design in Python using ciphertext, using a total of 60 gate operations for a 16-1 `MUX`. We tested the multiplexer first with only 8 inputs, then with the full 16 inputs. The results of these tests are shown below.

Number of inputs	Total time	Time per gate
8	91 sec	3.25 sec
16	186 sec	3.1 sec

Table 4.1: 16-1 `MUX` Runtime.

Keep in mind these tests were run on a CPU, so no parallelization was implemented. The few issues we ran into while designing this multiplexer were easily resolved because all the testing was done in plaintext. It became evident to us that although our integer functions were simple enough to test in ciphertext, our floating point functions have a much more complex design. Because of this, testing must be done in plaintext and then converted to ciphertext, otherwise the root of many of our issues will never be known.

4.2 Addition

The floating point addition algorithm we used is far more complex than any of our integer addition functions previously discussed. In the following section, we will discuss how we dealt with the difficulties of normalizing, shifting, and rounding encrypted bits.

4.2.1 Normalization

We began with floating points by first understanding normalization. We worked with the IEEE 754 half-precision binary floating-point format which designates bits 0 through 9 for the mantissa (fraction), bits 10 through 14 for the exponent, and bit 15 for the sign. The sign bit will be a zero to denote a positive floating point, and one to denote a negative. The exponent bits include bias, which offset the exponent by 15. For example, the representation for minimum and maximum exponents for an IEEE 754 half-precision floating point are shown below.

$$\text{Minimum} = 00001 - 01111 = 2^{-14}$$
$$\text{Maximum} = 11110 - 01111 = 2^{15}$$

As you can see, to represent an exponent of -14 the exponent bits must represent 1 to account for the bias of 15. Similarly, for an exponent of 15, the five exponent bits must represent 30 because of the bias. Finally, the mantissa represents the actual floating point value. Although the mantissa takes up bits 0 through 9, there is an implied value of 1 placed before the mantissa bits. Therefore, if the mantissa is 0000000000 it really represents 1.0000000000. A full example of the IEEE 754 half-precision floating point is shown below in Figure 4.1.

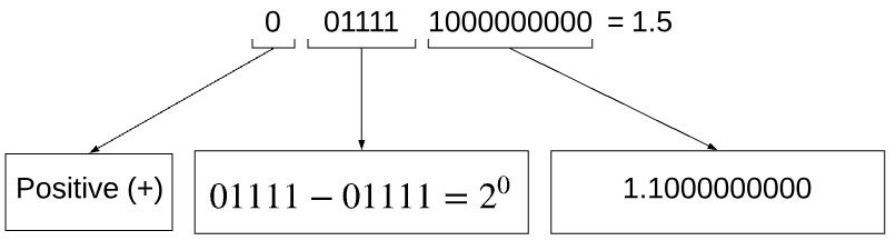
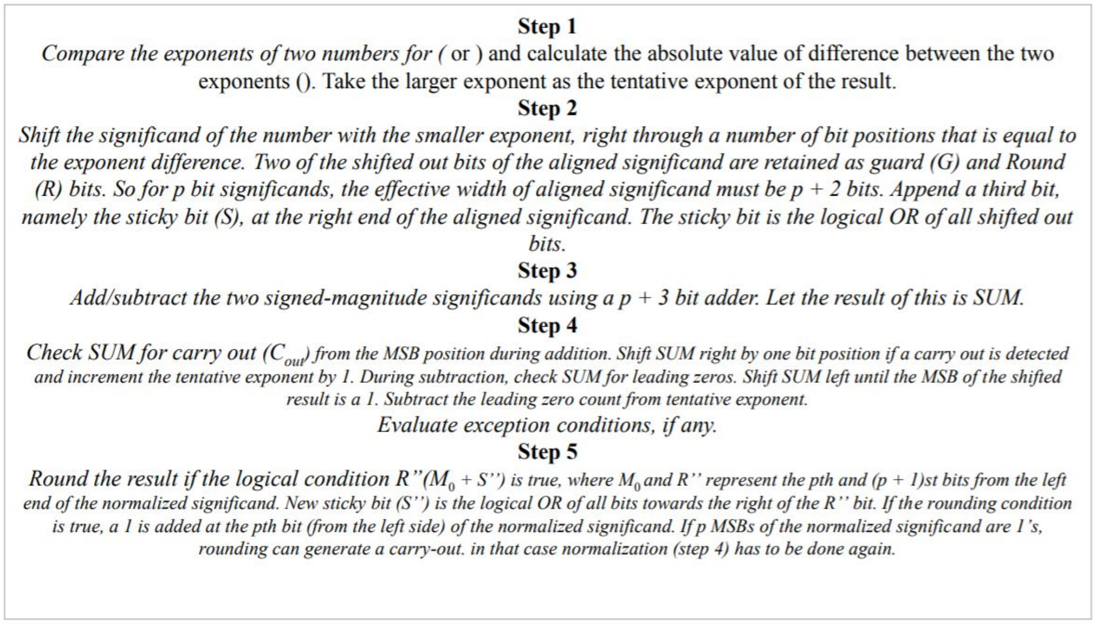


Figure 4.1: IEEE 754 Half-Precision Floating Point.

In our floating point addition function we are assuming all floating point values passed into the function are already normalized, and the function itself re-normalizes the final floating point sum after the addition. Once we complete our normalizing function, we can implement that with our addition function to allow it to accept denormalized numbers as inputs.

4.2.2 Addition Algorithm

Figure 4.2 below explains the floating point addition algorithm we followed when designing this function. For the first step, we use our integer subtraction function to find the difference in exponents of the two inputs, a and b. This one subtraction accomplishes all the goals of step one, finding which input has the larger exponent and the difference in the exponents. If the difference is negative, we know that b has the larger exponent. If the number is positive, a has the larger exponent. The larger exponent is copied and stored into a tentative variable for use later in step four.



Step 1

Compare the exponents of two numbers for (or) and calculate the absolute value of difference between the two exponents (). Take the larger exponent as the tentative exponent of the result.

Step 2

Shift the significand of the number with the smaller exponent, right through a number of bit positions that is equal to the exponent difference. Two of the shifted out bits of the aligned significand are retained as guard (G) and Round (R) bits. So for p bit significands, the effective width of aligned significand must be p + 2 bits. Append a third bit, namely the sticky bit (S), at the right end of the aligned significand. The sticky bit is the logical OR of all shifted out bits.

Step 3

Add/subtract the two signed-magnitude significands using a p + 3 bit adder. Let the result of this is SUM.

Step 4

Check SUM for carry out (C_{out}) from the MSB position during addition. Shift SUM right by one bit position if a carry out is detected and increment the tentative exponent by 1. During subtraction, check SUM for leading zeros. Shift SUM left until the MSB of the shifted result is a 1. Subtract the leading zero count from tentative exponent.

Evaluate exception conditions, if any.

Step 5

Round the result if the logical condition R"(M_0 + S'') is true, where M_0 and R'' represent the pth and (p + 1)st bits from the left end of the normalized significand. New sticky bit (S'') is the logical OR of all bits towards the right of the R'' bit. If the rounding condition is true, a 1 is added at the pth bit (from the left side) of the normalized significand. If p MSBs of the normalized significand are 1's, rounding can generate a carry-out. in that case normalization (step 4) has to be done again.

Figure 4.2: Floating Point Addition Algorithm.

For step two, we shift the small exponent number right equal to the difference between the exponents. To do this, we used the multiplexer we explained above. If *ns* is the number of bits in the exponent, and *n* is the total number of bits of the input, then this `MUX` has a minimum depth of *2+ns* with maximum width $n*2^{ns}$. To limit the width to *n* bits, the `MUX` would be depth $(2^{ns})/n(ceiling) + 2^{ns+1}-1$. To put this in perspective, the maximum depth option would take about 8.58 microseconds with a half precision floating point number. Even using the AWS GPU with 80 SMs, we would not be able to do a half precision floating point using the minimum depth multiplexer method. This multiplexer process is slow and is something to be investigated for ways to either optimize or replace it in the future. Step three is simply an addition of the transformed inputs using our existing `Carry Select Adder`. Step four requires us to normalize the sum of the two floating point values, as well as evaluate exception conditions such as

Overflow, Underflow, and dividing by zero. To normalize, we created a function called `roundNormalize`. First it creates a temporary copy of the sum and shifts the mantissa right one bit. Then it adds a value of one to the exponent of this temporary value. Finally, it selects between this modified sum and the original sum using a multiplexer. For our design we decided to skip evaluating exception conditions, as our run time was already very slow and checking for these conditions would take up significant time. Step 5 is the rounding step, where we use the Sticky, Round, and Guard bits to determine if rounding up is necessary or not. This process is discussed in Section 4.2.2. It is important to note that we did not include a final check for if all bits in the significand are 1, in which case there would be a carry out and the normalization process would have to be repeated. In order to implement this, we would have to create a ciphertext copy of the final value and have it go through the normalization process again in order to select which final value is correct with a multiplexer. Because of the relative infrequency of this scenario occurring, we decided to exclude this check from the design knowing the consequences it would have on total run time.

4.2.3 Shifting

The main issue we faced during the construction of this function was shifting the bits to the right, which is needed for step two of the addition algorithm. After finding the difference between the exponents, step two calls for the smaller input to shift its mantissa to the right by the difference in the exponents. Normally this would not be difficult, but because we are working with homomorphically encrypted data we cannot see the value of the difference because it is encrypted. The way to get around this issue is to find the value of the mantissa for every possible shift, and then use a multiplexer to select the correct output. Instead of finding 2^5 different shift possibilities, we can simply shift 0 through 10 times. Since shifting 10 or more times results in a mantissa of all zeros, any shift more than 10 will result in the same output. The number of iterations would of course be different for a full precision or double precision number, but the concept would remain the same.

First, we created a basic shift function that took in an input value, the length of the value, the number of times to shift the value, and an array to store the shifted value in. We utilized the `Copy` function we made earlier to copy the value of the input into the correct index in the output array. The `Shift` function we designed shifts a 10-bit input to the right a desired number of bits. The output array is 13-bits because this one includes the Guard, Round, and Sticky bits. The three extra bits are necessary to use later in the floating point addition function and will be explained in the next section. Since we are shifting to the right, and into a larger array, the output is offset by 3 bits due to the previously mentioned Guard, Round and Sticky bits. On a zero bit shift, the 9th bit of the input will become the 12th bit of the output, the 8th bit of the input will become 11th bit of the output, and so on. This is illustrated below in Figure 4.3 below. If it were a 1-bit shift, the 9th bit of the input would become the 11th bit of the output. Any spaces that are left empty after the shift are then filled with zeros. Instead of padding on 3 extra bits to the input so the input and output are the same size, we were able to bypass this step by offsetting the shift by three bits. As a result, it may seem like it is shifting to the left, when it is actually shifting all of the inputs to the right. This was easily solved when we made the decision to have the function take the parameter `n`, or the length of the input. Once the shift register was complete, we used a for-loop that looped from 0 through 10.

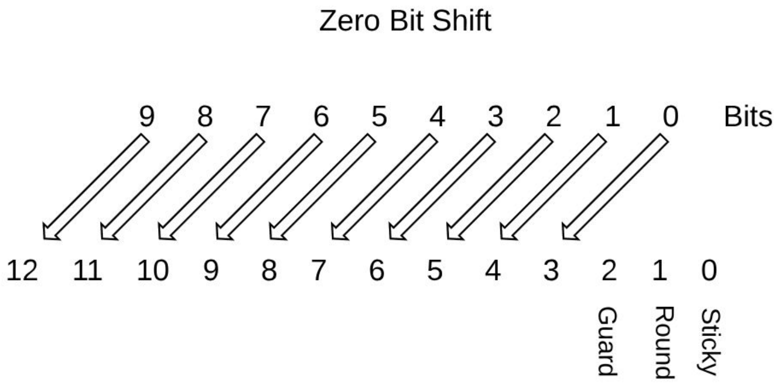


Figure 4.3. Zero Bit Shift Diagram

4.2.4 Guard, Round, Sticky Bits

Step two is where the rounding aspect of floating point numbers started to come into play. In our algorithm, each input mantissa gets a pad of three extra least significant bits before they are added together in step three. These are for the Guard, Round, and Sticky bits, respectively. Figure 4.4 below shows the position of these bits. For the input with the larger exponents, these bits are all zero, as no rounding is needed. However, for the smaller input, the Guard and Round values are filled with the last two shifted out bits. Sticky is the value of all bits shifted out with an OR gate applied to them, excluding the Guard and Round bits. After Step three, these bits are used to determine if the shifted input should be rounded up or not. Getting the Guard and Round bit was simply done by using the `Copy` function on the corresponding value in the input. The Sticky, however, would require multiple gates to be used, which would be highly inefficient for 10 different shifts. To solve this, instead of applying an OR gate to every shifted-out bit, we just applied the OR gate to the current Sticky bit with the Sticky bit from the iteration before. In effect, we cut down to 10 total OR gates instead of 54 spread out through each of the 10 shift iterations. This is an 81% increase in speed.

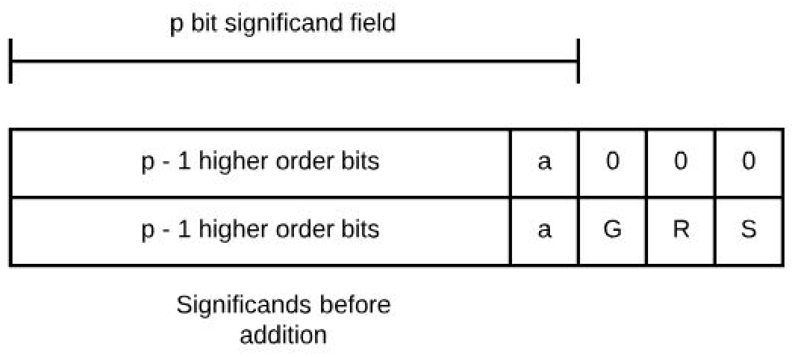


Figure 4.4: Visualization of 3 Rounding Bits.

Chapter 5. Synchronization

Though parallel execution can decrease computation time significantly, it introduces the possibility of data corruption, or hazards. These hazards arise from the combination of concurrent streams of execution and the data shared between them. They usually result in indeterminate behavior; when they are present, the result of a computation is not guaranteed to be correct. The methods of avoiding hazards all involve synchronization. Unfortunately, synchronization is often synonymous with waiting, which means it potentially slows down execution. A significant portion of our effort was put toward using synchronization effectively and efficiently in order to create functions that produce the correct result with the smallest possible latency.

5.1 Data Hazards

Data hazards occur when shared data is modified by two concurrent processes. If the data is modified before it was expected to be, it can cause corruption. When not accounted for, data hazards cause an indeterminate result. An indeterminate function cannot be guaranteed to produce the correct answer, so it is useless. There are three types of data hazard: read after write (RAW), write after write (WAW), and write after read (WAR). Each of these names represent the type and order of occurrence of two operations on a single piece of data. The risk is that the operations will occur out of order. It is as if the two streams of execution are racing, so data hazards are also called race conditions.

A RAW hazard is the only one which indicates a true data dependency. It occurs when one operation is meant to modify, or write, a piece of data before another reads it. The output of the first operation is used as the input to the next. Thus, the operation which reads the data is dependent on the result of the first operation. This is the type of hazard which occurs in our `Cpa` function. The carry chain is a series of RAW hazards: the carry out of each full adder becomes the carry in of the next. If the second full adder runs before the first completes, it will operate on an uninitialized ciphertext. An uninitialized ciphertext has a random value, either 1 or 0. If its value happens to be the same as the carry out, the dependent full adder will produce a correct result. It is equally likely, though, that the value will be incorrect, and the dependent full adder will produce an incorrect result. As the chain of dependencies lengthens, the likelihood of an error corrupting the result increases.

A WAW hazard does not indicate a true dependency, but it can still cause corruption. It occurs when one operation is meant to modify a piece of data before another. One place this hazard occurs in our integer functions is when using ciphertexts to store temporary results, such as in our multiplexer. The multiplexer stores the partial results of the `And` operations in two arrays of ciphertexts. If we were to issue two `Mux` operations at the same time with the same temporary ciphertexts, a WAW hazard would occur. Each multiplexer would try to place its partial results into the same arrays. We would expect that the first one which is issued will do so first, but that is not implicitly guaranteed. Factors such as slight differences in speed between streams and the number of operations already queued can cause the second `Mux` to run before the first. When the first multiplexer runs, placing its temporary results in the arrays, it will overwrite the expected values. This will cause the second `Mux` to have a potentially incorrect result.

WAR hazards, like WAW hazards, do not indicate a true data dependency. They occur when an operation is meant to read a piece of data before another operation writes to it. Also similar to WAW hazards, this occurs most often in our functions when using shared temporary ciphertexts. The example of a WAW hazard above can easily be applied to WAR hazards. Since neither of these types are true data dependencies, they can be avoided by eliminating shared data. The issue with this solution, though, is that it increases the memory required for storing temporary data. In our divider, for instance, we reuse a *2n* bit array to store the result of each subtraction. If we were to instead create arrays for each temporary result, we would need *n* arrays of size *2n*: a total of $2n^2$ ciphertexts. This would be massively inefficient.

We cannot eliminate RAW hazards, as all arithmetic functions have some data dependency. We could eliminate the WAW and WAR hazards by removing all shared data, but this is impractical due to limited memory resources. The only remaining factor we can control is the parallel streams of execution. We need a way to resolve all three hazards by removing race conditions. This solution is called synchronization.

5.2 Host Synchronization

It is important to consider two types of synchronization: host and device. The host (CPU) has a single thread of execution. It dispatches intensive computations to the device (GPU) where they run on any one of a set of streams. The device implicitly synchronizes to the host, because it must receive instruction to operate. In other words, it cannot get ahead of the host. The host, however, can issue operations to the device asynchronously before previous ones have completed. It is therefore necessary for the host to be able to synchronize with the device and wait for the result of computations being performed. Finally, the streams within the device can operate independently of one another. This creates a need for synchronization between streams.

When we started the project, the `cuFHE` library contained only two simple methods of synchronization. The simplest is the `Synchronize` function. This function blocks the host until all work on the device is complete. The `Synchronize` function can be used after work is issued to the device so that the result ciphertexts are ready to be copied back to the host or computed on further. This function is simple to use but has significant drawbacks when considering performance. If, for example, the host uses the `Synchronize` function to wait for the result of a short operation after a long running operation has been issued to the device, it must wait for both operations to complete. This limits flexibility in designing parallel applications and reduces the opportunity to take advantage of the resources on the device.

The other synchronization method involves streams and the order of operations issued to the device. Every GPU has some number of streams which operate independently of each other. If two operations are issued to the same stream, the second must wait for the first to finish. This happens independently of the host, so more operations can be issued concurrently. Once issued, operations are queued on the device and the host is free to perform other work. For straightforward operations this is a more powerful method of synchronization. The limitations of this method arise, though, when synchronization between streams is required. Most gate operations, for example, take two input arguments. Ideally, both inputs would be computed in parallel on different streams. Since there is no guarantee that one will finish before the other, though, there is no way to determine which stream the dependent operation should be placed in. This is known as a race condition.

There are numerous other methods of synchronization available using the CUDA platform. One such method is using `cudaStreamSynchronize`. This function blocks the host until a single stream is done computing events. For some applications, this provides a significant advantage over the `Synchronize` method, but it still causes the host to synchronize with the device. In order to build more complex operations on top of the gate functions, we require a method of synchronizing between streams without affecting the host.

5.3 Device Synchronization

The `Synchronize` method provided by the cuFHE library and the `cudaStreamSynchronize` method from the CUDA API each have their use cases. Common to both is that they operate on the host. After those functions are called, the host thread blocks until some or all prior work issued to the GPU is complete. This prevents any additional work from being issued to the device, which slows execution. There are many cases in which we need some work issued to the GPU to wait for some other work which is also being processed on the GPU. Threads of execution on the GPU are called streams. What we need is a way of synchronizing between streams on the GPU, without blocking the CPU thread.

The CUDA platform provides a primitive called `cudaEvent_t`. These structures can be assigned to a stream as a sort of flag using the `cudaEventRecord` function. This function marks an event as incomplete. The event is only be marked complete after all prior computational work issued in the given stream has been executed by the GPU. Future work issued to the same stream will not modify the event. This note is important for our use case and is discussed further below.

There is another function, `cudaStreamEventWait`, which blocks a given stream, and only the given stream, until the given event has been completed. In other words, any future work submitted to the same stream will wait for some prior work in a different stream. According to the CUDA documentation [CUDA1], this synchronization is performed efficiently on the GPU when it is possible to do so. Though it does not give any further information about when it is not possible, we have not come across any such conditions in our testing. We speculate that one such situation would be when calling cudaStreamEventWait on a stream associated with one GPU and an event associated with another. We believe that this situation can be avoided for most use cases. For situations in which the synchronization is done inefficiently through the host, the result will still be correct, but computation may be slowed significantly.

We added two fields to the Ctxt structure called revent_ and wevent_. When using the GPU, the event fields point to cudaEvent_t structures. These events are used to synchronize access to ciphertexts between streams. There are two events for each of the two types of operation which are performed on Ctxt objects: the revent_ for reading and the wevent_ for writing. We could have implemented a single event for each ciphertext, but that would limit our flexibility in designing complex functions. Specifically, this would prevent multiple functions from reading a ciphertext concurrently.

A ciphertext is read wherever it is the input to a gate or copy operation. Likewise, a ciphertext is written to wherever it is the output of a gate or copy operation. The purpose of the revent_ and wevent_ fields is to capture these read and write operations, respectively. Thus, we added calls to cudaEventRecord after all gate and copy operations. Each call takes the given stream, st, and an event associated with one of the ciphertexts. For ciphertexts which are inputs to the gate, the revent_ is recorded. For ciphertexts which are outputs of the gate, the wevent_ is recorded.

Recall that writing to memory can cause two types of data hazards: WAR and WAW. Thus, we want to ensure that all reading and writing operations on a given ciphertext are complete before starting a new write operation. We accomplish this by modifying the beginning of the gate and copy operations in the cuFHE library to use the cudaStreamEventWait function. This function is called on the revent_ associated with each input ciphertext in order to prevent WAR hazards. Similarly, it is called on the wevent_ associated with each output ciphertext in order to prevent WAW hazards.

An example WAW hazard is shown in Figure 4.2. Two operations, Xor and Copy, are going to write to the same ciphertext, z. Note that the output is the first item in parentheses. The Xor is issued first on stream 0: it has already waited for all input and output ciphertext events, and it has been recorded into the input and output ciphertext events. It is still executing when the call to Copy is issued on stream 1. Before the copy can occur on the GPU, though, stream 1 must wait on the input and output ciphertext events. Since the call to Xor is recorded on z.wevent_, the actual call to cudaMemCpy waits. Without this synchronization, it would be impossible to predict which function would complete first; the result would be indeterminate. If the Copy operation finished first, the expected result would be overwritten, with a potentially incorrect result.

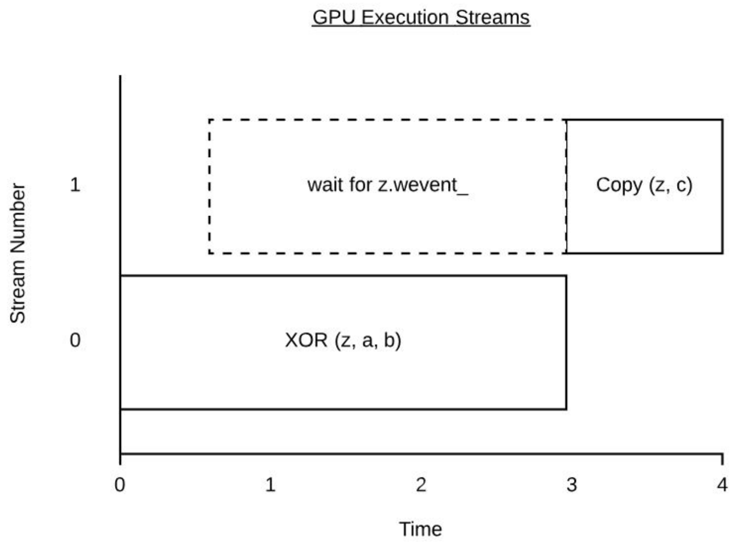


Figure 4.2 WAW hazard example

The final type of hazard we had to design for is the RAW hazard. Before a ciphertext can be read, the stream of execution must wait for any prior work which might modify its data. We, again, modified the gate and copy functions. Before the kernel call in each function, `cudaStreamEventWait` is called on the `wevent_` object associated with each input `Ctxt` object and the given stream, `st`. This ensures that the gate kernel or copy API call can read the inputs without the risk of another stream modifying them and causing corruption.

An example of a RAW hazard is shown in Figure 4.3. In this case, the input to `Copy` must wait for the output of `Not`. The call to `cudaEventStreamWait` at the beginning of `Copy` causes it to wait until the `Not` has completed and placed its result in b. Without this synchronization mechanism, the `Copy` operation would start at time 1. Since the `Not` operation is incomplete, the b ciphertext will not contain the expected value. If the ciphertext had been used previously, it would still hold the previous value. If it was just created, it may be uninitialized. We have observed that, in the `cuFHE` library, ciphertexts have random values until they are initialized with the `Encrypt` function or written to by a gate function. Either scenario will potentially lead to an incorrect result. The result would be indeterminate.

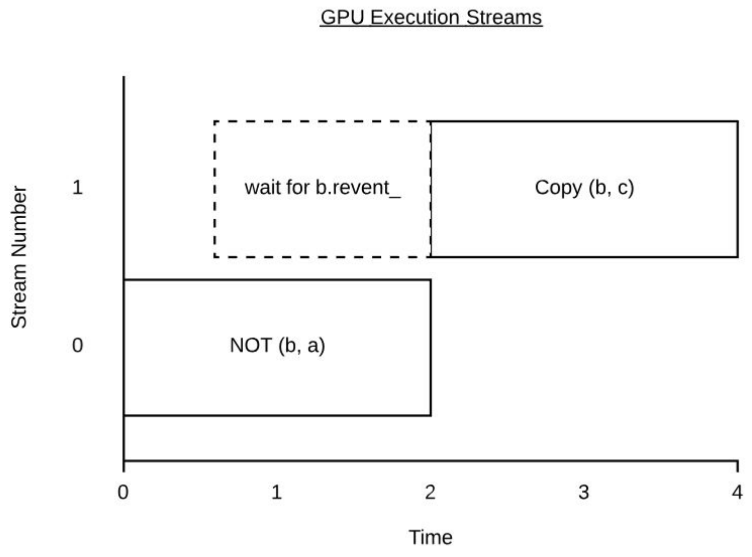


Figure 4.3 RAW hazard example

One detail which is crucial to the success of this synchronization approach is that all CUDA API calls are made from a single host thread. Though there may be concurrent streams on the GPU, the `cuFHE` library is designed to use a single thread on the CPU. This single threaded nature enables us to control the order of execution of the CUDA kernel launches and API calls discussed above. The single thread of execution progresses linearly through our functions. Each kernel launch and copy API call is preceded and succeeded by corresponding calls to `cudaStreamWaitEvent` and `cudaEventRecord`, respectively. This guarantees that all three types of data hazard are prevented.

A side effect of this property is that, in our updated `cuFHE` library, each gate or copy function called on a single ciphertext will occur on the device in the same order that they are issued from the host. This holds true regardless of which stream, or streams, they are issued to. This is does not necessarily help or hinder use of the library, but we found it helpful when designing our integer functions. There are many cases in which we reuse ciphertexts to store temporary results. Even if two function calls share temporary ciphertexts, neither one will interfere with the correctness of the other. The function which is called second will potentially block certain streams until the temporary variables used by the first function are available. Using the NVIDIA Visual Profiler (NVVP), we were able to visualize these delays and allocate additional temporary ciphertexts where possible.

These modifications to the `cuFHE` library not only help prevent data hazards for our integer functions, they also do so for end users of the gate functions. Synchronization between streams happens automatically, so the only remaining concern is how to best allocate streams to maximize concurrency. We have optimized our integer functions for any number of provided input streams, so the user can focus on applying them optimally to a specific computational workload.

5.4 Multithreading

When our functions became limited by our current synchronization methods, we looked into implementing multithreading to help alleviate the problem and optimize our functions. Multithreading is done at a processor level to improve area and energy efficiency [Kis07]. It used thread level parallelism to increase the utilization of a single core. When the memory access of a program looks to advance, it must reference random memory access (RAM). In this case, the cycle of RAM is close to ten times slower than that of a process, and the processor has no choice but to wait. An advantage of this is possibly faster runtimes by utilizing unused computing resources. One disadvantage of multithreading is the multiple threads can interfere with each other when sharing hardware resources. This can lead to longer overall runtimes in some functions.

We performed testing with CPU threading to enable simpler synchronization on the Tesla V100. We first created a new thread for each functional block (i.e. full adder). This allowed us to call the blocking synchronize function while continuing to execute in other functional blocks. This partly alleviated the issue we were having with the synchronize function, which puts a hold on the CPU adding additional function blocks when the synchronize function is called. We tested this using the C++ thread class as well as the `async` function, which also uses the thread class with some additional functionality. Both methods added overhead which was an order of magnitude greater than the latency of a gate function. Therefore, it does not make sense to create a new thread for a functional block with a small number of gates.

After initial implementation, it was decided multithreading is impractical for use within our functions. The runtimes would remain the same, if not longer, because of the time it takes for the main thread to create other threads. A possible alternative we looked into is to somehow have the main thread create separate threads before the function itself is run. Then the buffer time between thread creations would be eliminated, and the runtime would be reduced.

5.5 Unintended Synchronization

Some amount of synchronization is required to prevent hazards, but it should be avoided as much as possible. Synchronization implies a thread of execution is waiting for something, and always incurs some amount of overhead. This is undesirable when trying to maximize concurrency. The CUDA platform provides several methods and structures to perform synchronization intentionally. Of course, we use many functions which have other purposes such as memory allocation. Due to the limitations of the hardware, though, some of these functions cause the host to synchronize with the device. These functions, in addition to their primary application, also act similarly to the `Synchronize` function. This includes all of the drawbacks mentioned above.

One group of functions which cause this synchronization are memory allocations. To maximize the speed of memory transfer between the host and device, the `cuFHE` library uses pinned memory on the host. When memory is allocated on the device using `cudaMalloc` or pinned memory is allocated on the host using `cudaMallocHost`, the host must synchronize to the device. This is an issue when allocating memory between operations. It is often useful or even necessary to allocate ciphertexts to hold temporary results. For example, the multiplexer requires an array in which to store the result of the negation of the select bit. It would be possible to allocate these temporary ciphertexts at the beginning of the program and pass them to the functions as arguments. This is less than ideal, though, as it requires the user to know the required number of ciphertexts at the beginning of the program.

Another function with synchronization as a side effect is `cudaEventCreate`. As mentioned above, we used `cudaEvent_t` structures to control dependency between streams. When creating temporary ciphertexts, we must also create events. This requires use of the `cudaEventCreate` function. Since the event is used by the device, it follows that it requires memory to be allocated. Since all device memory allocations cause the host to synchronize with the device, so too does the `cudaEventCreate` function. This suggests, though, that the solution to synchronization in the `cudaEventCreate` function will be similar to that for the memory allocation functions.

Our solution to unintended synchronization is based on a similar system from the similar `cuHE` library, also developed at Vernam Labs. The idea behind it is to provide an interface between the homomorphic program and the device. This interface does the job of the GPU when allocating and freeing memory. A large allocation of host and device memory is made at the beginning of the program and freed at the end. During execution, calls to memory allocation functions instead go to the memory management interface, which returns pointers within the block of pre-allocated memory. This does not require the host to synchronize to the device entirely, while minimizing the work for the program developer. The only additional responsibility is to call an initialization function which instructs the memory management interface to pre-allocate memory. We believe this is the most reasonable compromise between performance and usability.

Unfortunately, there are a few complications with the memory management system which prevented us from implementing it fully. The primary issue is the difference in timing between the host and device. At the beginning of a function such as the multiplexer, which requires temporary ciphertexts to hold partial results, we would like to allocate memory and events. This can be done trivially, by constructing the temporary ciphertexts with the allocator enabled. We need the allocated memory to last for the entire time of execution on the GPU, though, and therein lies the problem. The host function returns before execution has finished, or perhaps even begun, on the device. Thus, the memory will be released and potentially reassigned to another ciphertext, causing corruption. We investigated some possible solutions to this problem which are briefly discussed in the Future Work Chapter.

The solution we arrived at in place of the memory management unit is to allocate a large array of ciphertexts in advance and pass them to the functions. This places more work on the user of the library, as he must calculate in advance the number of ciphertexts which will be required. This calculation is performed based on the function, the size of the inputs, and the number of streams available. In general, larger inputs and available streams require more temporary ciphertexts. Ultimately, this solution should be replaced with the previously mentioned memory management system for maximum ease of use.

Chapter 6. Future Work

Throughout our work on this project, some of our original goals were not met due to some of the unintended issues we encountered, such as synchronization. Additionally, we set some new goals based on how our vision for this project changed over time. This section explains some of the future work that can be done if a future team were to continue our project.

6.1 Memory Allocation Optimization

As mentioned above, a memory management unit will increase the ease of use of our new homomorphic functions. We have a nearly working implementation, but it can only be used for functions which are synchronous between the host and device. When using asynchronous kernels, the difference in timing between the host and device causes corruption. The requirement to be synchronous between host and device eliminates the intended advantage of the memory management unit.

A possible solution to this problem involved using host functions as callbacks. The host functions for allocating and deallocating memory from the management unit can be queued into the device streams. Before a set of functions requiring temporary ciphertexts runs on the device, a callback will run and allocate memory. Execution will resume until the temporary ciphertexts are no longer needed, at which point another queued callback will deallocate the associated memory.

We were unable to implement this, though, as callbacks queued on the device cannot invoke CUDA API functions. This is generally not an issue when calling dynamic allocation functions, except when the dynamic allocator has run out of memory. When insufficient memory was allocated, though, our allocator calls the `cudaMalloc` or `cudaMallocHost` function. This conflicts with the limitations of callback functions on the device.

This issue could be resolved by removing this functionality, but that would cause errors when the functions try to consume more memory than initially allocated by the user. Rather than running slowly while additional memory is allocated, the functions would halt altogether. The only way to resolve this would be to manually allocate more memory in advance. This would essentially leave the user doing the same calculations required by the existing system. We believe there is likely a better solution available, and it should be investigated fully before implementation.

6.2 Floating Point Operations

Our original goal was to develop both integer and floating point functions to add to `cuFHE`. Due mostly to the synchronization issues we encountered, only the integer functions were fully completed. As previously stated, we were able to make progress on floating point addition and a new multiplexer to go with it. We quickly realized while working on floating point operations how much more time-consuming they would be than integer operations. Similar to the integer operations, most of the floating point functions heavily utilize addition blocks as well. Once a floating point addition block can be developed, we believe all of the other operations should follow in line. Because we were unable to deliver our original goal of a library containing integer and floating point operations, we would encourage the next team to pick up the development of floating point operations. We think there is room for efficiency improvements on our floating point adder design, especially if a better method for bit shifting is found. Ideally, the library will include floating point addition, subtraction, multiplication, and division.

6.3 Optimizing Compiler

The final suggestion we have for the future of the `cuFHE` library is an optimizing compiler. This could operate at or just prior to run time. The concept is similar to a reorder buffer in a pipelined processor, in which the order of operations is swapped to prevent stalls in execution. The compiler would allocate streams as they become available to work on a queue of functions. This would essentially be performing automatically the same task which we have done by hand in designing the arithmetic functions. It could also potentially do the same for multiple calls to functions by the user. If, for example, the user designs an application which performs many independent additions, a sophisticated compiler could automatically use the carry propagate adder to perform them concurrently with a high occupancy. If, instead, the user designs an application which is highly serial, the compiler could use the maximally parallel implementations of each arithmetic function which minimize runtime.

As hardware become cheaper and more efficient, GPUs gain more streaming multiprocessors. We worked with devices that had 10, 30, and 80 streams. As this number grows, so too does the number of gate functions which can be computed in parallel. Eventually, a compiler could use extensive multiplexing to intelligently eliminate dependencies and utilize future hardware to its full potential. All of these optimizations are best implemented as rules in a compiler to make them adaptable. Rather than redesigning each function by hand every time a new generation of hardware arrives, they can simply be recompiled with new specifications. This will speed development of homomorphic applications, making them more practical with each new generation of hardware.

Bibliography

[Gre12] M. Green, "A very casual introduction to Fully Homomorphic Encryption," *Fundamentals in A Few Thoughts on Cryptographic Engineering,* 2012.

[Gre14] A. Greenberg, "Hacker Lexicon: What is Homomorphic Encryption?," *Security in Wired,* 2014.

[Gen09] C. Gentry, "A Fully Homomorphic Encryption Scheme," Ph.D. dissertation, Stanford University, 2009.

[ACE16] "Comparative Data." A.C.E Project. Administration and Cost of Elections, 2015. [Online]. Available: http://aceproject.org/epic-en

[Hen16] T. Henderson, F. Torija and A. Noakes, "Distributed Homomorphic Voting," Massachusetts Institute of Technology, 2016.

[Arc14] D. Archer *et al*, "Applications of Homomorphic Encryption," in *Crypto Standardization Workshop*, pp. 2-8, 2017.

[Koc14] O. Kocabas and T. Soyata, "Medical Data Analytics in the Cloud Using Homomorphic Encryption," in *Handbook of Research on Cloud Infrastructures for Big Data Analytics,* pp. 471-474, 2014.

[Mar18] J. Martindale, "What is a CPU?," *Computing in Digital Trends*, March 2018.

[Intel1] *Intel® Core™ i7-4900MQ Processor*, Product Specification, Intel Corporation, Available: https://ark.intel.com

[Kre09] K. Krewell, "What's the Difference Between a CPU and a GPU?," *Nvidia,* December 2009.

[Raj18] A. Raj, "Multiplexer Circuit and How it Works," *Digital in Circuit Digest,* July 2018.

[Che19] D. Chen, "Lecture 11: Adders," *Lecture Notes in Electrical Engineering,* Illinois University, 2019.

[Wan13] R. Wang, "Fast Addition -- Carry Lookahead," *Lecture Notes in Electrical Engineering,* Harvey Mudd College, 2013.

[Par13] S. Parmar and K. P. Singh, "Design of high speed hybrid carry select adder," in *IEEE International Advance Computing Conference (IACC),* 2013, pp. 1656-1663.

[Kor08] I. Koren, "High-speed multiplication," *Lecture Notes in Digital Computer Arithmetic,* University of Massachusetts, Amherst, 2008.

[Waw13] J. Wawrzynek, “Multipliers & Shifters,” *Lecture Notes in Digital Design,* University of California, Berkeley, 2013.

[Gal07] S. Galal and D. Pham, “Division Algorithms and Hardware Implementations,” *Lecture Notes in Advanced DSP Circuit Design,* The University of California, Los Angeles, 2007.

[CUDA1] “CUDA Runtime API,” *Event Management in CUDA Toolkit Documentation,* Nvidia, v10.1.105, March 2019.

[Kis07] K. D. Kissell, "Demystifying multithreading and multi-core: Tapping concurrency as a resource," *EE Times,* 2007.