GPUHELIB AND DISTRIBUTEDHELIB: DISTRIBUTED COMPUTING VARIANTS OF HELIB, A HOMOMORPHIC ENCRYPTION LIBRARY

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Ethan Frame

June 2015

COMMITTEE MEMBERSHIP

TITLE:	GPUHElib and DistributedHElib: Distributed Computing Variants of HElib, a Homomorphic Encryption Library
AUTHOR:	Ethan Frame
DATE SUBMITTED:	June 2015
COMMITTEE CHAIR:	Zachary N J Peterson, Ph.D. Assistant Professor of Computer Science
COMMITTEE MEMBER:	John Clements, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	Robert Easton, Ph.D. Professor of Mathematics

ABSTRACT

GPUHElib and DistributedHElib: Distributed Computing Variants of HElib, a Homomorphic Encryption Library

Ethan Frame

Homomorphic Encryption, an encryption scheme only developed in the last five years, allows for arbitrary operations to be performed on encrypted data. Using this scheme, a user can encrypt data, and send it to an online service. The online service can then perform an operation on the data and generate an encrypted result. This encrypted result is then sent back to the user, who decrypts it. This decryption produces the same data as if the operation performed by the online service had been performed on the unencrypted data. This is revolutionary because it allows for users to rely on online services, even untrusted online services, to perform operations on their data, without the online service gaining any knowledge from their data.

A prominent implementation of homomorphic encryption is HElib. While one is able to perform homomorphic encryption with this library, there are problems with it. It, like all other homomorphic encryption libraries, is slow relative to other encryption systems. Thus there is a need to speed it up. Because homomorphic encryption will be deployed on online services, many of them distributed systems, it is natural to modify HElib to utilize some of the tools that are available on them in an attempt to speed up run times. Thus two modified libraries were designed: GPUHElib, which utilizes a GPU, and DistributedHElib, which utilizes a distributed computing design. These designs were then tested against the original library to see if they provided any speed up.

ACKNOWLEDGMENTS

Thanks to:

- Andrew Guenther, for uploading this template
- Dr. Easton and Dr. Peterson for proofreading drafts of this document and giving valuable feedback.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In the last five years the design and development of a new encryption scheme that could enhance the level of security on the internet has exploded. That encryption scheme is known as Homomorphic Encryption.

First conceived over thirty years ago, and finally designed in 2009 by Craig Gentry, homomorphic encryption is a revolutionary encryption scheme because it allows for computation to be performed on encrypted data. This means that a user can encrypt their data, and send it to a service. That service can then perform an operation, and send the encrypted result back. Upon decryption by the user, the result received will be exactly the same as if the data had not been encrypted at all, and the operation had been computed on the unencrypted data. The added benefit of this system however, is that the user data was never known by the service, thus the user can be assured that their information remains secret. By putting this encryption scheme on online services, user data can be passed from online service to online service without the user being worried of their information being known.

A few implementations of homomorphic encryption have been design in recent years. One of the most prominent is HElib. This library however, like all homomorphic encryption libraries, is not currently in use because it suffers from slow run times. Thus, there is a desire to speed it up.

Because the target audience for these schemes is online services (many of which are designed as distributed systems), it makes since to try and modify HElib to take advantage of these systems. Thus two modified libraries were designed in the hope they would perform better than the original library. GPUHElib, which utilizes a GPU; and DistributedHElib, which uses a distributed computing design, were both designed in the hope that they would provide run time improvements over HElib.

Unfortunately, these modified libraries fail to provide any speedup, as we will show later. These designs exhibit the same pitfalls that other distributed systems do. Mainly memory transfer speeds are too slow, which cause huge slowdowns compared to the original unmodified library. However there could be hope for them, given further work.

CHAPTER 2

BACKGROUND

2.1 Homomorphic Encryption

Homomorphic encryption is a form of encryption that allows computations to be performed on ciphertexts, thus generating an encrypted result, which when decrypted matches the result of operations performed on the plaintext. For example, the numbers 4 and 5 could be encrypted to `A` and `B`. Let `C` = `A` + `B`. When `C` is decrypted, its value will be 9. So common operations, like addition and multiplication, can be performed on encrypted data, and produce the same result as if the data was not encrypted in the first place. This is a desired feature in encryption schemes, because it allows encrypted data to be passed from online service to online service, each service performing operations on the data, without the online service knowing what the data is.

Currently, for an online service to perform an operation for a user, the online service must know what the data is. Thus any online service will know what data a user is giving them. However with homomorphic encryption, an online service (even an untrusted on-

line service), can perform operations on user data, without the user being worried about their data being known or exposed.

A fully homomorphic encryption (FHE) scheme has long been sought after [14]. For a scheme to be fully homomorphic, any arbitrary operation must be able to be executed, and still produce the correct results. It may seem like many operations need to be supported if any arbitrary operation can be performed, but in reality a fully homomorphic scheme need only support both addition and multiplication, as every other operation can be derived from those two. Partially homomorphic encryption schemes, schemes that only support one operation, have been known to exist since the 1970s. A few schemes that are known to be partially homomorphic are Unpadded RSA, ElGamal, Goldwasser-Micali, Benaloh, and Paillier. All of these schemes only support one operation, either addition or multiplication, but not both. It took more than 30 years before a fully homomorphic scheme was designed by Craig Gentry.

2.1.1 Gentry's Design

Craig Gentry proposed the first fully homomorphic encryption scheme in 2009 utilizing lattice-based cryptography [6]. His scheme supported both addition and multiplication, from which any arbitrary operation could be derived.

Gentry started by designing what was called a somewhat homomorphic encryption (SWHE) scheme. A SWHE scheme supports arbitrary operations, but could only com-

pute a limited number of operations. This is because the scheme uses a noise factor when representing a ciphertext. When an operation is performed the noise in the representation grows. If the noise grows too large, then the ciphertext becomes incorrect when decrypted. Gentry then took this SWHE scheme and added bootstrapping, meaning it could evaluate its own decryption circuit. This bootstrapping procedure allowed for the ciphertexts to be "refreshed", where the noise would be decreased, thus allowing for more operations to take place. Finally Gentry proved that a bootstrappable SWHE scheme can be made into a FHE scheme, by continually performing the bootstrapping procedure when the noise reaches a certain limit.

Gentry's scheme was the first construction of a FHE scheme, however impractical when implemented [7], because the ciphertext size and computation time increased sharply as the security level increased. It could take upwards of 30 minutes to compute operations on a single bit, for large security levels. Numerous improvements [17, 15, 16, 8] were offered to try to optimized the solution, however new techniques were required to produce a much more efficient scheme, which created a second generation of designs.

2.1.2 Second Generation Designs

Many researchers worked to create a second generation of FHE schemes [4, 1, 13, 9]. These new schemes relied on the Learning with Errors (LWE) problem and all featured much slower growth of the noise factor compared to Gentry's original design. There were

two optimizations found during this time that lead to a breakthrough design: ciphertext packing and modulus switching.

Ciphertext packing [2] is a technique that allows for multiple plaintexts to be encrypted and placed into a single ciphertext. With these new schemes based on the LWE problem, the ciphertexts must be large in order to satisfy security concerns. These large ciphertexts cause the operations being performed to be slow. When a single plaintext encrypts to a single large ciphertext, then it is evident that there will be a cost to efficiency. By packing the ciphertexts, allowing a vector of plaintexts to be encrypted into a single ciphertext, the cost to efficiency is almost negated. These ciphertexts can then be operated on component-wise in a SIMD (single instruction, multiple data) fashion.

Modulus switching [5] is a technique that allows the noise present in a ciphertext to be decreased without performing the bootstrapping operation. Each ciphertext in these schemes is relative to a modulus, `p`. Given a ciphertext `c` mod `p`, one can transform it into a ciphertext `c'` mod `p'`, which will have a lower noise factor, without knowing the secret key. By using modulus switching after every multiplication (the operation responsible for the largest noise increase) and by choosing the moduli carefully, the noise factor after multiplication will be unchanged. This technique allowed the largest modulus to grow linearly with multiplicative depth, which was a large improvement over previous systems.

These techniques were discovered separately by different researchers, before they were combined to create a single scheme, the Brakerski-Gentry-Vaikuntanathan (BGV)

scheme, which allowed for a FHE scheme that did not need bootstrapping.

2.1.3 FHE without Bootstrapping

The BGV scheme [3] combined the two techniques described above to create a FHE scheme that did not need a bootstrapping operation in order to perform arbitrary operations. This was a breakthrough because the bootstrapping operation was the one that cost the most run time in other similar schemes. Even though the scheme does not need bootstrapping, the authors do have a bootstrapping procedure for this scheme, but it is used as an optimization, not as a necessary component in making this scheme fully homomorphic. This scheme also has a SIMD design, which will allow for the possibility of speedup, seen in later chapters. This is the scheme that HElib is built off of.

2.2 HElib

HElib [11, 12, 10] is an open source implementation of the BGV scheme. It was developed by Shai Halevi and Victor Shoup in 2014, and is designed to be a very low-level library intended for research purposes.

The intention of our work is to enhance the run time when performing operations in this library. Thus the creation, encryption, and decryption of plaintexts and ciphertexts was not examined, as those operations will likely occur on the users end, not on the online service's end. The design of the addition and multiplication operations are the areas fo-

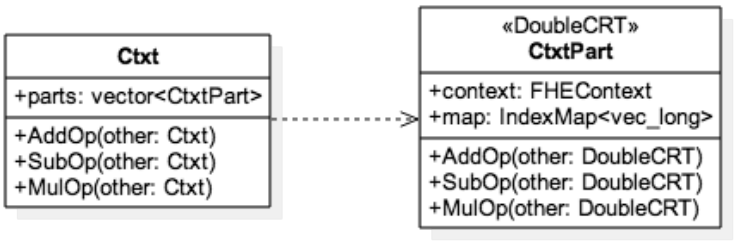

Figure 2.1: HElib Type Hierarchy

cused on for this work. Before addressing the changes present in the modified libraries, it is necessary to understand the current serial implementation of HElib.

2.2.1 HElib Design

Let $*$ be the operation being performed (here $*$ could stand for any of the operations, all of them are handled similarly) and A and B be the ciphertexts being operated on. The execution of the operation $A = A * B$ requires a few steps.

A and B are stored as `Ctxt` objects in HElib. Figure 2.1 shows the type hierarchy for a `Ctxt` object. `Ctxt` objects have one important variable, `parts`, a vector containing multiple `CtxtParts`. These `parts` constitute the ciphertext. The operations supported by `Ctxt` are the addition, subtraction, and multiplication of two `Ctxt` objects. Each of these operations use `parts` during the execution of the operation, thus the operations in `CtxtPart` are called.

`CtxtPart` is an extension of the class `DoubleCRT`, which is where the operations are implemented. Listing 2.1 is an excerpt from the function that performs the operations.

Listing 2.1: Add, Sub and Mul operations of two DoubleCRT objects

```
...
const IndexSet& s = map.getIndexSet();
long phim = context.zMStar.getPhiM();

for (long i = s.first(); i <= s.last(); i = s.next(i)) {
    long pi = context.ithPrime(i);
    vec_long& row = map[i];
    const vec_long& other_row = (*other_map)[i];

    for (long j = 0; j < phim; j++) {
      row[j] = fun.apply(row[j], other_row[j], pi);
    }
}
...
```

As the index set is iterated over, the i^{th} prime is extracted along with the i^{th} row from the maps. Even though the map is accessed like an array, it is an unordered map, with the array access syntax for convenience. These rows are then iterated over, applying the operation to each element. This is where the SIMD design is occurring, a double `for` loop to add, subtract or multiply two vectors together. This is where the modifications and possible run time speedups can occur by using distributed system techniques.

2.3 Distributed Systems

Distributed systems is a field of computer science that deals with computer systems performing concurrent computation to achieve a goal. These systems can be as tight as a

single computer running multiple threads or as loose as a group (cluster) of computers (nodes) connected via a message passing interface all over the world. What these systems have in common is that they are all connected and working to achieve a single goal.

The benefit of using a distributed computing system is the possibility for concurrent computation. Non-distributed computing systems are limited to only serial execution of programs. This means that if, for example, the system were tasked with adding two vectors together, it would have to loop over all entries one after the other and add each individually. For large vectors, this could be time consuming. Each individual operation is independent, and can therefore be performed at the same time. This is the purpose of a distributed system. By partitioning the data, and assigning a portion of the work to each node in the system, concurrent computation can be performed. For large vectors, this may result in a speedup in the run time, because of the concurrent execution.

The caveat of using a distributed computing system is the possibility for large overhead times, that can slow down computation. Time can be lost because of the added operations needed to facilitate the concurrent computation. The data must be partitioned and sent to the node that is performing the operation, which can cost time, if the means of transfer between nodes is slow, because in the original serial design, this did not occur. So for large vectors, it may be faster to perform the operation because of the concurrent execution, however in order to be able to perform the concurrent execution, some setup is required, that might cause the overall run time to be slower than the original serial execu-

tion. This is the trade off when working with distributed systems. Much work is done to reduce the amount of overhead in distributed systems, but they all suffer from it.

Distributed systems can be classified into two categories: parallel computing or distributed computing. A parallel computing system allows for shared memory, meaning that the processors all have access to a common memory which can be used to exchange information between processes. A common example of parallel computing is a graphics processing unit (GPU), because it is a single piece of hardware that has a common memory with many processors operating at once. In distributed computing systems, each processor has private memory and a message passing interface is used to exchange information between processes. A common example of this is the internet. Each computer attached to the internet has its own private memory and uses message passing to communicate with other machines on the internet. A common message passing interface is OpenMPI, which allows for the creation and running of a distributed computing system.

2.3.1 Parallel Computing on GPU

GPUs were designed to manipulate images for output on a display. Because of this, their design was such that they would perform operations in parallel over every pixel needing to be manipulated. Each pixel, or piece of data, was given its own compute core, that could be executed concurrently with every other compute core, which is how parallel execution was achieved. Expanding beyond manipulating images, one can see that this ap-

proach to computation can be applied to any circuit where the input is discrete and the same operation is applied to each piece of the input.

Going back to the example given earlier, each element in the vectors can be assigned to a separate core. Then each core can be executed simultaneously, and the result will be generated. For large vectors, a loop's run time would increase as the size of the vector increased. However, by using a GPU, the run time would be the same for any size vector (because the operation being performed was always the same, and because all the cores were executed at the same time). It makes no difference if 20 or 20,000 cores were executed, they would both take the same amount of time. This has allowed GPUs to be used for many more purposes than just manipulating images, and in recent years to help speedup the run times of SIMD algorithms.

2.3.2 Distributed Computing with OpenMPI

Distributed computing systems are a cluster of machines all linked through a message passing interface. One would use a distributed computing system when computation can not be performed on a single system alone because either the input is too large or the computation will take too long. By partitioning the input and assigning each node a portion of the work, the task can be completed, where before it could not.

A widely used message passing interface is OpenMPI. OpenMPI allows for the creation of a distributed cluster with a single call, and provides functions to send and receive

data from other nodes in the cluster easily. This allows for the partitioning and scheduling of work on a distributed system to be easily achieved.

Again addressing the example given earlier, depending on the number of nodes in the cluster, the vector can be partitioned so that each node gets roughly the same amount of elements. Each node can then compute the addition operation on only the portion of the vectors it is assigned. This allows for the concurrent execution of the addition operation, thus decreasing the run time compared to the serial computation. Distributed computing systems have become the design used by online services in recent years because they can service multiple customers at a single time, which is a desired feature when working on the internet.

CHAPTER 3

GPUHELIB DESIGN

HElib is one of the only implementations of homomorphic encryption. It suffers (like all implementations of homomorphic encryption) from slow run times compared to other standard schemes. Thus there is a desire to improve HElib by speeding it up. Because HElib is meant to be deployed on online services, it is natural to try an utilized hardware available on them. Thus the idea for GPUHElib.

GPUHElib is a variant of HElib, which attempts to speed up the run time of HElib by utilizing a GPU to parallelize operations. GPUs are often used to speed up computation where a single instruction or operation is performed on multiple pieces of data. GPUs are ideal for these types of designs because they allow for many compute cores to be run simultaneously, each performing the same operation. HElib utilizes a single instruction, multiple data (SIMD) design, however is single threaded. Meaning that, while being designed so a single operation occurs over multiple data pieces, the library is not efficiently utilizing the design to best effect. The hope then of adding GPU functionality to the library is to thus take advantage of this design, by utilizing hardware that will best handle

the SIMD nature of the scheme.

There are three phases when executing operations on a GPU. First the memory is copied from the host(CPU) to the device(GPU). Then a program(kernel) is created, which performs the operation on the data in the device's memory. Finally, the data is copied back from the device to the host, upon which the host continues execution. The first and third phases are discussed further in Section 3.1, and the kernel designs are addressed in Section 3.2. Furthermore, these three phases can be parallelized to achieve the fastest speedup, discussed in Section 3.3.

3.1 Memory Mapping

In order for the GPU to execute a kernel, it must have the data in a 1D vector. This requires that the data be mapped from its current storage model into a 1D vector. There are two pieces of data that are required to be mapped when performing an operation: the data that the operation will be performed on, and the moduli.

3.1.1 Mapping from CPU to GPU

Currently the data is stored as shown in Figure 3.1. The `map` contains vectors or rows, each of these rows are arrays of 64-bit integers. This structure is a non-contiguous 1D vector that needs to be mapped to a contiguous 1D vector. Thus the rows must be copied into a new vector, which the GPU will then operate on. Each successive row is concate-

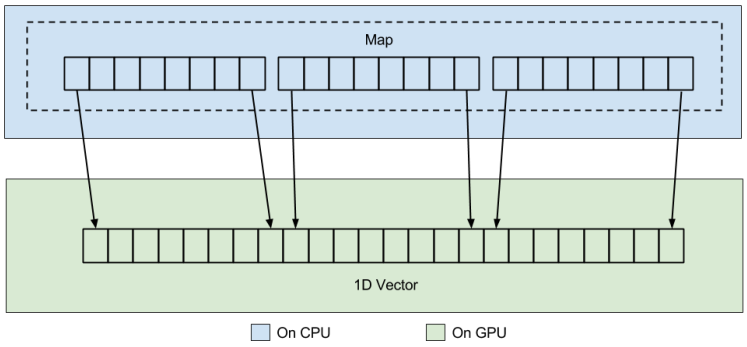

Figure 3.1: Data Mapping from CPU to GPU

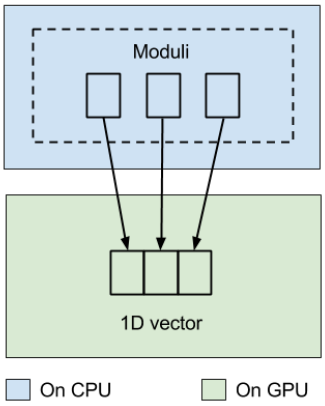

Figure 3.2: Moduli Mapping from CPU to GPU

nated to the preceding rows, thus creating a 1D vector.

Similarly the moduli are being stored as individual elements. In order to be used during execution on the GPU, they must also be mapped to a 1D vector. Figure 3.2 shows the current storage model for the moduli. Each successive modulus is concatenated to the preceding moduli, thus creating a 1D vector.

3.1.2 GPU Vector Management

Naively creating and freeing vectors on the GPU when the operation is run slows down run times because allocating and freeing memory on the GPU takes time. Creating a few

vectors at the beginning of execution, and maintaining them throughout the programs lifetime, allows for the most efficient memory usage, and greatest speedup.

Four vectors are created and maintained throughout the lifetime of the program. The first is a vector of size $num\_rows \times size\_of\_row$. This is the 1D vector that the data from the first `DoubleCRT` object is being copied into. The second is another vector of size $num\_rows \times size\_of\_row$, which holds the data that is being copied from the other `DoubleCRT` object. The third is a vector of size num_rows. This vector holds the moduli. The last vector is also of size num_rows, and is used when computing over a single `DoubleCRT` object and a constant number. The constant number, `mod` the moduli, is stored in this last vector. The sizes, num_rows and $size\_of\_row$, rarely change, thus there will be little memory reallocation occurring, and reallocation will only occur when the vector is too small, not when it is larger than needed. The function that handles initializing and reallocation of these vectors is found in Appendix B.1.

3.2 Overflow Considerations

Arithmetic overflow occurs when the result of an arithmetic operation is greater than the magnitude of the storage location that is being used to store the value. For example, using 4-bit unsigned integers, the largest possible value that can be stored is $2^4 - 1 = 15$. Thus when adding $12 + 14 = 26$, an overflow error will occur, because 26 requires 5 bits to store.

The type of the data being operated on in HElib is 64-bit signed integers. Meaning they can take values from $-(2^{63})$ to $2^{63}-1$. However, these values will never be negative, thus the actual range of these values is 0 to $2^{63}-1$. The largest modulus could be the largest possible value, $2^{63}-1$ and the largest value being operated on could be $2^{63}-2$. The reason the largest value being operated on is one less than the largest value, $2^{63}-1$, is because the value has to be smaller than the modulus, thus $(2^{63}-1)-1 = 2^{63}-2$. In CUDA, the language used on NVIDIA GPUs, (the language this implementation is written in) the largest variable type has a length of 64 bits. The type that can contain the largest value is `uint64_t`, unsigned 64-bit integers. This type can store values from 0 to $2^{64}-1$.

Six operations were designed, as they are the most commonly used operations. Addition, subtraction, and multiplication of a `DoubleCRT` with another `DoubleCRT` and addition, subtraction, and multiplication of a `DoubleCRT` with a constant number. Further considerations for overflow prevention for each operation is discussed in the following sections.

3.2.1 Addition Overflow Considerations

When considering the overflow prevention for addition, it is necessary to compute what the largest value could possibly be. As noted above, the largest a value could be is $2^{63}-2$ and the largest modulus is $2^{63}-1$. Performing the addition operation on the possible

values,

$$(2^{63}-2)+(2^{63}-2)=2^{64}-4 \tag{3.1}$$

shows that the number $2^{64}-4$ needs to be computed, before the modulus operation takes place. This number is outside the range for signed 64-bit integers, but not for unsigned 64-bit integers. Thus the original numbers must be cast to unsigned 64-bit integers (which could cause problems if any of the numbers were negative, but since the values are always positive, there is no problem). After the addition operation takes place, the modulus operation brings the result back down to the range of signed integers, because the modulus is in the range of signed integers. The result is then finally cast back to a signed integer, and the operation is complete. The kernels for both addition operations, between two `DoubleCRT` objects and between a `DoubleCRT` object and a constant are in Appendix A.1.

3.2.2 Subtraction Overflow Considerations

Similar to addition, it is necessary to compute the worst case scenario when considering overflow prevention for subtraction. Again, the largest a value could be is $2^{63}-2$ and the largest modulus is $2^{63}-1$. There are two scenarios to consider for subtraction: the first number being $2^{63}-2$ and the second number being 0 and the first number being 0 and the second number being $2^{63}-2$. Performing the subtraction operation for both scenarios,

$$(2^{63}-2)-0=2^{63}-2 \tag{3.2}$$

$$0 - (2^{63} - 2) = -(2^{63} - 2) \tag{3.3}$$

shows that all the computations can be completed using signed integers, because all numbers in the above equations are within the range of signed integers. Thus there does not need to be any step taken to prevent overflow for this design.

With this design, to ensure the resultant value is greater than 0, a check is made after the subtraction operation takes place, to determine if the result is less than 0. If so, the modulus is then added to the value, which will result in the value being larger than 0. This check will cause a branch to occur in the GPU, which could slow down run time. Alternatively, instead of performing a check to determine if the result is less than 0, the modulus can be added to the first value, then the second value is subtracted (shown in the below equation), which is a different approach to the subtraction operation. Equation 3.3 is redefined below, with this procedure applied.

$$(0 + (2^{63} - 1)) - (2^{63} - 2) = (2^{63} - 1) - (2^{63} - 2) = 1 \tag{3.4}$$

Now when the modulus operation occurs, the correct result is found, however none of the values throughout the calculation were ever negative. Thus there is no need to perform a check, avoiding the branch and possible slow down of the run time. This method however does result in an overflow issue, because the modulus is being added to the first value. Below is this approach applied to Equation 3.2.

$$((2^{63} - 2) + (2^{63} - 1)) - 0 = (2^{64} - 3) - 0 = (2^{64} - 3) \tag{3.5}$$

This equation generates the value $2^{64} - 3$, which is too large for the signed integer range. Thus the same procedure used for addition is performed here, to ensure overflow prevention. The original numbers are cast to unsigned 64-bit integers. The operation detailed above takes place, before the modulus operation brings the result back down to the signed integer range. The result is then cast back to a signed integer, and completes the operation. The kernels for both subtraction operations, between two `DoubleCRT` objects and between a `DoubleCRT` object and a constant are in Appendix A.2.

3.2.3 Multiplication Overflow Considerations

Multiplication presents many more problems compared to addition and subtraction. Again the largest value possible is $2^{63} - 2$, with the largest modulus being $2^{63} - 1$. Performing the multiplication operation on these values,

$$(2^{63} - 2) * (2^{63} - 2) = 2^{126} - 2^{65} + 4 \tag{3.6}$$

shows that very large numbers must be generated when performing the multiplication operation. Where addition and subtraction results could fit in a possible data type (unsigned 64-bit integer), these values will not fit in any data type available in CUDA. Therefore an algorithm must be used, which will break the original numbers into smaller pieces. These pieces will then be used during intermediary steps to generate other values; that, when combined back together will result in the correct answer, without ever generating a value that cannot fit in the GPU. The algorithm that is used is Karatsuba's algorithm.

Karatsuba's Algorithm

$$x = x_1 B^m + x_0$$

$$y = y_1 B^m + y_0$$

$$\begin{aligned} z_2 &= x_1 y_1 \\ z_1 &= x_1 y_0 + x_0 y_1 \\ z_0 &= x_0 y_0 \end{aligned} \tag{3.7}$$

$$\begin{aligned} xy &= (x_1 B^m + x_0)(y_1 B^m + y_0) \\ &= z_2 B^{2m} + z_1 B^m + z_0 \end{aligned}$$

Equation 3.7 shows Karatsuba's algorithm in general. The values being multiplied are x and y. They are broken into pieces x_1, x_0 and y_1, y_0 respectively. These pieces are then used to create z_2, z_1, and z_0, which are finally combined with the base number, B^m, to generate the original result. For this case, $B = 2$ and $m = 32$. These values will ensure that operations performed throughout the execution of this algorithm never become greater than the maximum 64-bit unsigned integer value. Each of these will be examined in-depth below to see this.

When $x = 2^{63} - 2$, the variables x_1 and x_0 have values $x_1 = 2^{31} - 1$ and $x_0 = 2^{32} - 2$. The exact same values are assigned to y_1 and y_0 since $y = x = 2^{63} - 2$.

Computing z_2,

$$\begin{aligned} z_2 &= x_1 y_1 \\ &= (2^{31}-1)(2^{31}-1) \\ &= 2^{62} - 2^{32} + 1 \end{aligned} \tag{3.8}$$

shows that z_2 can fit inside a signed 64-bit integers, since the largest possible value is less than $2^{63}-1$ (the largest value possible for signed 64-bit integers).

Computing z_1,

$$\begin{aligned} z_1 &= x_1 y_0 + x_0 y_1 \\ &= 2[(2^{31}-1)(2^{32}-2)] \\ &= 2[2^{63} - 2^{33} + 2] \\ &= 2^{64} - 2^{34} + 4 \end{aligned} \tag{3.9}$$

shows the intermediate pieces can be computed using signed 64-bit integers, however, the addition operation causes the result to be in the range of unsigned 64-bit integers. Thus this calculation requires casting the pieces to unsigned 64-bit integers, carrying out the operation to calculate z_1, then performing the modulus operation to bring z_1 back down to the signed 64-bit integer space.

Computing z_0,

$$\begin{aligned} z_0 &= x_0 y_0 \\ &= (2^{32}-2)(2^{32}-2) \\ &= 2^{64} - 2^{34} - 4 \end{aligned} \tag{3.10}$$

shows that z_0 must be calculated using unsigned 64-bit integers, since the result is too large for signed 64-bit integers. Thus this calculation also requires casting the pieces to unsigned 64-bit integers, performing the multiplication, before calculating the modulus to bring the result back into the signed 64-bit integer range.

Now that each of the intermediate pieces have been addressed, the final piece needs consideration. Computing xy,

$$\begin{aligned} xy &= z_2 B^{2m} + z_1 B^m + z_0 \\ &= (2^{62} - 2^{32} + 1)(2^{64}) \ + \ (2^{63} - 2^{34} + 5)(2^{32}) \ + \ (2^{63} - 2^{34} - 3) \end{aligned} \tag{3.11}$$

shows there will be problems computing $z_2 B^{2m}$ and $z_1 B^m$. However these can be dealt with deterministically. By performing a loop, multiplying z_2 and z_1 by 2, 64 and 32 times respectively, performing the modulus operation after every multiplication, one can obtain the correct value, without exceeding the unsigned 64-bit integer limit. Final modulus operations are applied to each addition, finally resulting in the correct value. This value is cast back to a signed 64-bit integer, and the operation is complete. The kernels for both multiplication operations, between two `DoubleCRT` objects and between a `DoubleCRT` object and a constant, are in Appendix A.3.

3.3 Pipelining

As mentioned at the beginning of this chapter, there are three phases when performing operations on a GPU. First, the data is copied from the host to the device, then the kernel is

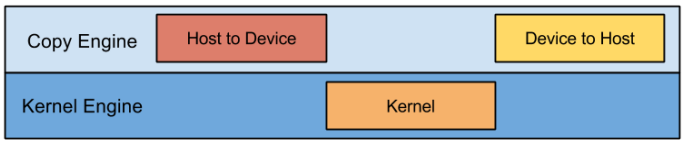

Figure 3.3: Serial GPU Execution

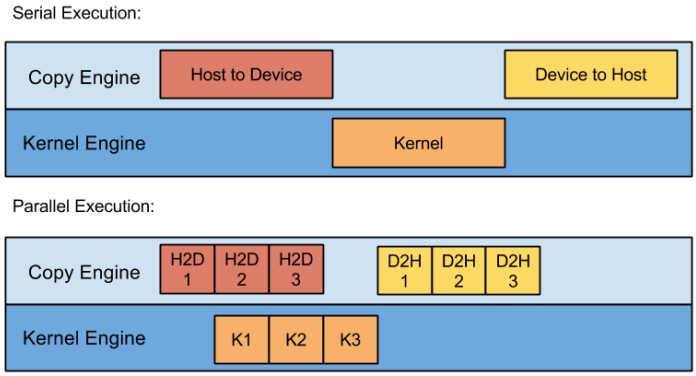

Figure 3.4: Concurrent GPU Execution with 3 Streams

executed, and finally the data is copied back from the device to the host. The copy operations are handled by the copy engine, a processor that specifically deals with copying data back and forth from the host and device. Kernels are executed by the kernel engine. This operation is serial, meaning that all the data is copied from the host to device, before the kernel is executed. And the memory is not copied back, until all of the kernels have finished executing. This is illustrated in Figure 3.3. CUDA allows for these operations to be parallelized, using streams.

3.3.1 CUDA Streams

CUDA streams are a sequence of operations that execute in order on a GPU. Operations in different streams can execute concurrently, and be interleaved. Figure 3.4 shows this process applied to the same operations shown in Figure 3.3. One can see that this allows for speedup, because the copy to/from host to device can be executed while the kernels are being executed. Streams are useful for parallelizing the computation, however the order in which the operations are dispatched also plays a role.

3.3.2 Overlapping Kernel Execution

There are two ways of dispatching GPU operations, all at once or by batching similar operations together.

All At Once Method

The first approach launches all the operations at once, shown in Listing 3.1.

Listing 3.1: Operations launched all at once

```
for (int i = 0; i < nStreams; i++) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,
                    cudaMemcpyHostToDevice, stream[i]);
    kernel<<<streamSize/blockSize, blockSize,
                0, stream[i]>>>(d_a, offset);
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
                    cudaMemcpyDeviceToHost, stream[i]);
```

```
}
```

All three phases are launched successively on the same stream, before the next stream's operations are launched.

Batching Method

The second approach is to launch similar operations together, instead of all at once. This is show in Listing 3.2.

Listing 3.2: Operations batched

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,
                  cudaMemcpyHostToDevice, stream[i]);
}
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  kernel<<<streamSize/blockSize, blockSize,
           0, stream[i]>>>(d_a, offset);
}
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
                  cudaMemcpyDeviceToHost, stream[i]);
}
```

In this second approach, all the copy from host to device operations are launched on their respective streams, then the kernels are executed, and finally the device to host copies are dispatched.

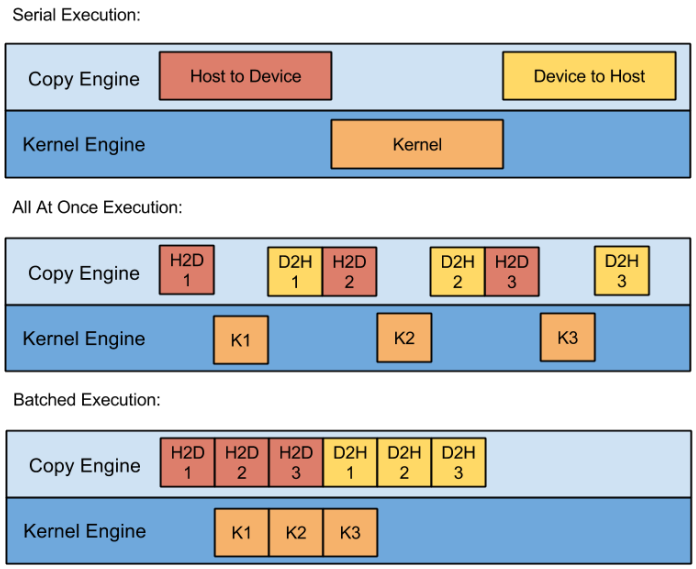

Figure 3.5: GPU 2-Way Pipelining

Both of these methods will produce the same result, as they are executing the same commands. However, depending on the hardware being used, one method might achieve a better speedup over the other.

3.3.3 2-Way Pipelining

Older GPU hardware only have a single copy engine and a single kernel execution engine. Figure 3.5 shows both methods of kernel execution compared to serial execution. As one can see, both provide a speed up, however the batching method provides a greater speedup over the all at once method. This is because the all at once method launched the second copy from host to device after the first copy from device to host. Because there is only one copy engine, and the engine executes operations in the order they were launched, the copy engine must execute the copy from device to host, before the next host to device

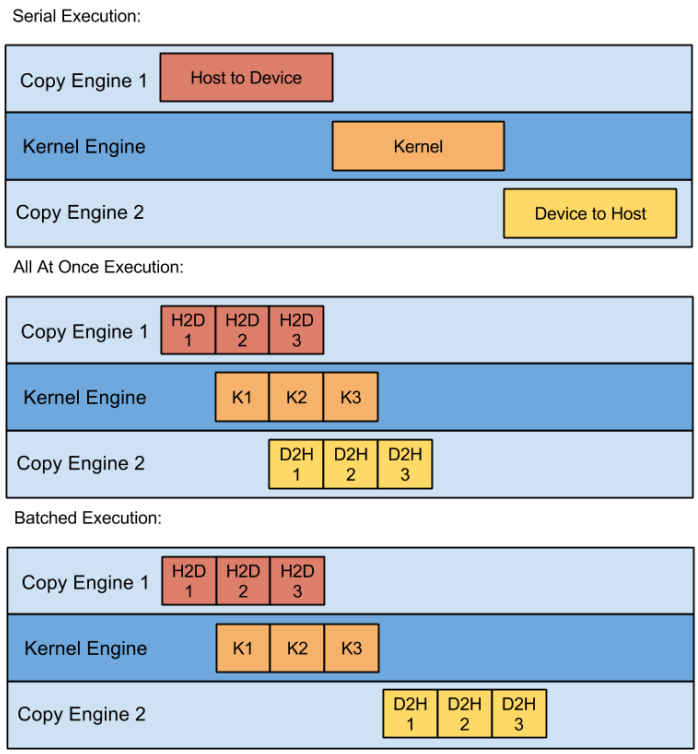

Figure 3.6: GPU 3-Way Pipelining

copy can occur. Using the second method though, the copies from host to device are all launched before the copies from device to host, so they call all execute one after the other. This allows for the most efficient pipelining, and the greatest speedup on hardware that only has one of each engine.

3.3.4 3-Way Pipelining

Newer GPU hardware have both a copy from host to device engine and a copy from device to host engine, as well as the kernel engine. The two methods under this new hardware configuration are shown in Figure 3.6. Here one can see that again both methods provide an overall speedup, however for this case, the all at once method achieves the

greatest speedup. Even though the first device to host copy was issued before the second host to device copy in the all at once method, the second host to device copy can be executed earlier than it because of the two copy engines. This is what is allowing the all at once execution method to be faster than the batching method. The batching method is suffering from a design choice in the scheduler of the GPU. The GPU tries to execute the kernels concurrently, and in doing so, delays a signal telling the device to host engine to starting copying until all the kernels have finished execution.

3.3.5 Stream Management

Naively creating and destroying streams as they are needed causes avoidable run time slowdown. By managing the available active streams, one can achieve the most efficient design. For any instance of operation, the number of active streams is *num_rows* in the `map`. This number will rarely change, thus initially creating *num_rows* amount of streams, and only creating more streams when there are not enough, efficiently maintains the fewest amount of needed streams at all times. Appendix C.1 contains the functions used to manage the streams.

CHAPTER 4

DISTRIBUTEDHELIB DESIGN

HElib is a prominent implementation of homomorphic encryption. However, it suffers (like other implementations of homomorphic encryption) from slow run times. Therefore, it must be speed up, before it will be used anywhere. HElib is meant to be deployed on online services, most of which are designed as distributed systems. Thus there was the idea to utilize the distributed system design present on these online services to facilitate the run time speed up.

DistributedHElib attempts to speed up the run time of HElib by utilizing a distributed system design. This is done by utilizing a cluster of compute nodes to parallelize operations. A cluster of nodes is a connected group of machines that communicate with each other to achieve a common task. By delegating separate work to each machine, work can be split between many machines, instead of a single machine handling the entire work load. By doing this, run times can decrease, especially if each node is working on completing an independent task, which helps complete an overall global task.

HElib utilizes a single instruction, multiple data (SIMD) design, meaning that a sin-

gle instruction or operation is performed over many pieces of independent data. Unfortunately, HElib does not take advantage of this design, because it is only single-threaded. Meaning that while the operation has the potential to be concurrently computed, it is being serial computed. With a distributed design then, the hope is to split the data up, send it to compute nodes, have those nodes perform the operations, and send the results back. Having each node work simultaneously on separate pieces of the data can allow for a speedup in run time compared to only having a single machine work on all the data.

For our design design, a master-slave architecture was chosen. This means there will be one node, the dispatcher node, controlling the other nodes, the compute nodes. The dispatcher node will be running HElib and when needed, will assign work to the compute nodes.

There are a few phases when executing operations in a distributed computing environment. First the cluster must be setup, and nodes must be designated as compute or dispatcher nodes. Then the dispatcher node must assign work to the compute nodes. As part of assigning work, the dispatcher node must partition the data, and send the pieces to the respective compute nodes. The compute nodes must then perform the operations, and send the data back. Finally the dispatcher node collects all the results and stores them, before returning to regular execution. The cluster setup and work assignment phases are discussed in Section 4.1. The partitioning of the data is discussed in Section 4.2. Finally the methods by which the data are transmitted between nodes is discussed in Section 4.3.

4.1 Node Cluster Setup

Upon start up of a cluster of nodes, each must be assigned a job. For this design a master-slave architecture is used. This means that one node must be designated the master node, which will be the dispatcher node, and the others are designated slave nodes, or compute nodes. The dispatcher node is the node responsible for running the serial portion of HE-lib, and distributing the data to the worker nodes when a distributed part of computation is reached. The compute nodes just wait for instructions from the dispatcher node, and act accordingly when given tasks.

When starting a cluster with OpenMPI, the distributed computing communication interface, each node is assigned a number, starting at 0 through $num\_nodes - 1$. Node 0 becomes the dispatcher node, and the others are compute nodes. The dispatcher node then returns to normal program execution, while the compute nodes wait for messages from the dispatcher node.

When the dispatcher node reaches a point of execution that is meant to be distributed, it partitions the data (discussed in Section 4.2) and assigns each compute node a piece of the data to operate on. The manner in which the dispatcher node chooses what data the compute node will operate on is examined next.

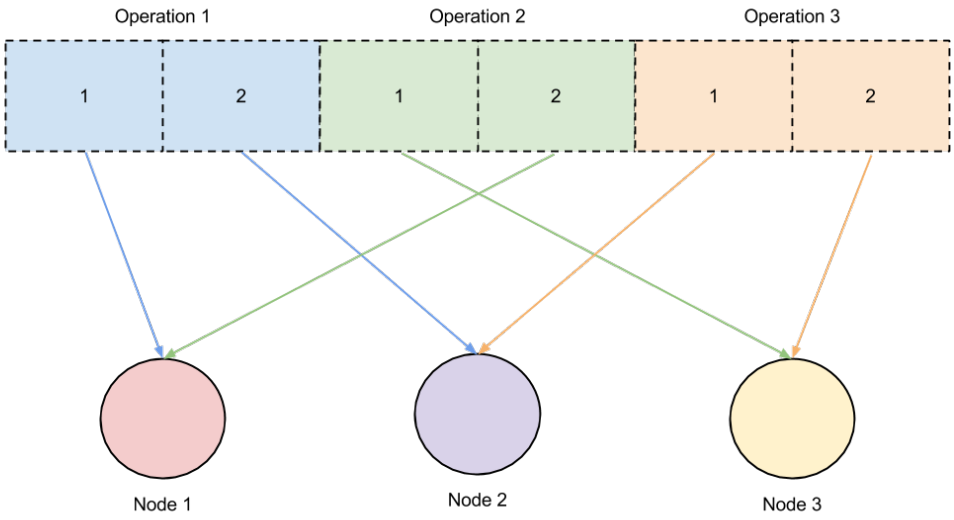

Figure 4.1: Rolling Round Robin Example with More Nodes than Data Pieces

4.1.1 Work Assignment

After the data has been partitioned, it must be assigned to a compute node to be operated on. The scheme used to choose the next compute node for a piece of data is round-robin scheduling. This scheme allows for an even or almost even distribution of the data across all the compute nodes. Having an even or almost even distribution means the lowest run times and best efficiency. There are two cases to consider when determining if a distribution scheme is efficient: when there are more compute nodes than data pieces, and when there are fewer compute nodes than data pieces.

For the first case, with the round-robin scheduling, this means that some nodes will not be working, while others are. For this design a rolling round-robin design is used. Meaning for any operations the next node to be assigned work will always be the node assigned work the longest time ago. This node has the highest probability of being free

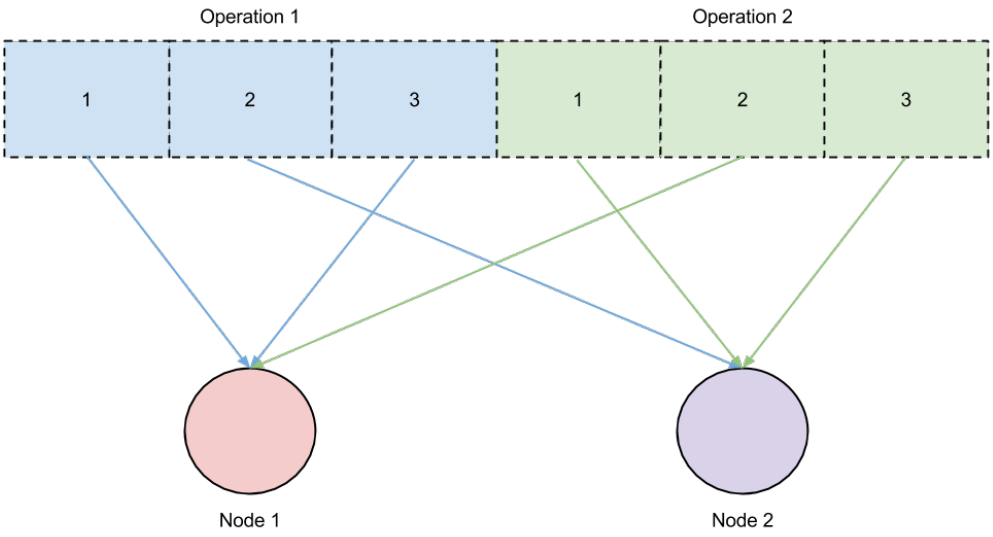

Figure 4.2: Rolling Round Robin Example with Less Nodes than Data Pieces

and ready to receive more work, compared to all the others. Figure 4.1 shows an example of rolling round-robin scheduling applied to this case.

For the second case, with the round-robin scheduling, some nodes will have multiple pieces of data to operate on. By using the round-robin scheduling though, the amount of work done by each compute node should be about equal, and thus evenly spread over the compute nodes. If work was unequally proportioned to a single compute node, compared to others, then the dispatcher node might have to wait longer for the results before continuing normal execution. This way the greatest run time and efficiency is achieved. Figure 4.2 shows an example of rolling round-robin scheduling applied to this case.

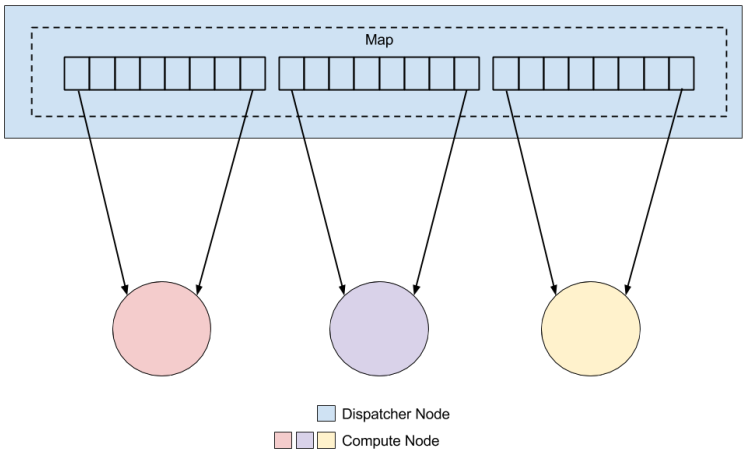

Figure 4.3: Data Mapping from Dispatcher to Compute Nodes

4.2 Memory Mapping

For compute nodes to execute, they must have a portion of the data to work on. This requires the dispatcher node to partition the data from its current storage model into pieces before they are sent to the compute nodes to be worked on. There are two pieces of data that need to be mapped: the data that the operation is being performed on, and the moduli.

4.2.1 Mapping from Dispatcher Node to Compute Nodes

Currently data is stored as shown in Figure 4.3. The `map` contains vectors or rows, each of these rows are arrays of 64-bit integers. The rows present a great partitioning point. Splitting the data up by these rows, and sending individual rows to each compute node to be operated on is the logical splitting point because there will always be about the same number of rows during execution, whereas the size of the rows might change often. Also, splitting rows would cause more communication between the nodes, which could slow

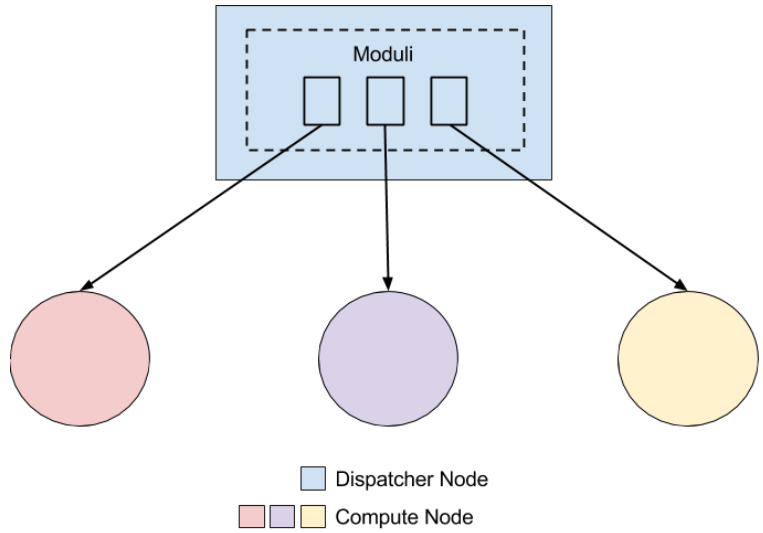

Figure 4.4: Moduli Mapping from Dispatcher to Compute Nodes

down run times.

Similarly, the moduli are being stored as individual elements. Because each moduli is assigned to each row, and the rows are being assigned to a single compute node, each moduli must only be sent to the specific compute node that the row it corresponds to is on. Figure 4.4 shows the mapping process. Each modulus is only sent to the compute node that its corresponding row is sent to.

4.2.2 Compute Node Vector Management

Naively creating and freeing buffers on the compute nodes that will receive the data from the dispatcher node slows down run times. By creating a few buffers, and maintaining them throughout the programs lifetime, the most efficient memory usage is achieved, along with the greatest speedup.

Two buffers are created and maintained throughout the lifetime of the program. Both buffers are of size $size\_of\_row$. These are the buffers that the rows will be copied into on the compute nodes. The size, $size\_of\_row$, rarely changes, thus there will be little memory reallocation occurring. Also, reallocation will only occur when the buffer is too small, not when it is larger than needed. The buffer will only ever grow, not shrink, thus cutting down on the occurrences of reallocation needing to be performed. The function that handles initializing and reallocation of these buffers is found in Appendix B.2.

4.3 Concurrency

Concurrency for a distributed system means that each node is executing operations simultaneously. To achieve concurrency in this system, the dispatcher node must be able to assign work, and not have to wait for a response before assigning more work, and the compute nodes must be able to perform computations at the same time. For all of these operations to happened concurrently, the communication between the nodes must be non-blocking.

4.3.1 Non-Blocking Send and Receive with OpenMPI

Blocking send and receive functions requires the data to be completely sent or received before continuing execution. This means if a node were to call `receive`, it would wait until it received data, before continuing execution. This can both be beneficial and detri-

mental, depending on the needs of the design. Non-blocking send and receive functions however schedule requests, and then continue execution. These requests will later be filled, but execution can continue. A request can be generated and even if the data has not finished sending or been completely received, execution can continue. This request can then be tested, and once the data has been sent or received, the request will be filled. Both methods have their uses, discussed next.

Send and Receive on Compute Nodes

For the compute nodes, blocking send and receive functions are used, because it is necessary for the buffer holding the result of the operation to be sent back to the dispatcher node, before the next operation occurs. If the next operation occurred before the dispatcher node received the data, the buffer could be cleared or overwritten, and data sent back would be incorrect.

Send and Receive on Dispatcher Node

For the dispatcher node, non-blocking send and receive functions are used. This allows the dispatcher node to continue assigning work, even if the compute nodes have not fully received their assignment. Also, the dispatcher node will use non-blocking receive functions in order to receive data back as quickly as possible. If the dispatcher node used blocking receive functions, then it could be waiting on a node that is taking a long time to perform an operation, causing other compute nodes to wait that might have already fin-

ished computation, and may have pending computation that they could be moving onto.

Only using non-blocking receive functions could cause problems for the dispatcher node, if, for example, two operations are performed, and the second requires the results from the first. Without any mechanisms to ensure the result to the first operation is received, before the execution of the second operation, unknown results can be computed. Therefore there is a need for a syncing mechanism.

4.3.2 Syncing

A syncing mechanism will cause the dispatcher node to wait for all pending requests to be completed before continuing execution. In this way, it can be ensured that a single operation is fully completed before moving onto other operations. To keep track of these requests, a queue is used. When a new request is created, through either a call to `send` or `receive`, it is added to the end of the queue. When the `sync` function is called, each of these is tested to see if they have completed. If a request has been completed, then it is removed from the queue, if not, the function moves onto the next in the queue, and checks it. The function only returns when all the pending requests have been filled, when the queue is empty. The `sync` function can be found in Appendix C.2. The `sync` function is performed after every operation(addition, subtraction, or multiplication), to ensure the operation has completed, before moving onto the next operation.

CHAPTER 5

EVALUATION

The evaluation of this work has two objectives. First, to make sure that the modified designs still produce correct output. Meaning that the result of an operation when decrypted is the same for both the modified and unmodified versions of the library. Secondly, to profile each design and compare run times of the modified libraries to the unmodified library. These time comparisons will occur at multiple levels to best understand each design, and how they compare to the serial version.

Distributed systems achieve the best efficiency when working on large inputs, not small ones. This is because there is some overhead associated with setting up and distributing work. For GPU designs, this overhead time happens when transferring the data to and from the GPU. For distributed computing designs, the overhead time is also in the memory transfers like the GPU, but instead of transferring to the GPU, the memory transfers are between machines. This overhead costs time, that when working with small inputs, usually takes even longer than the operation to complete. Thus the distributed design causes a run time slow down compared to the serial version for small input sizes. How-

ever, when working with large amounts of data, the time saved to complete the operation is so great, that the overhead costs are worth it. This characterization of distributed systems is prevalent for these designs, which will be seen later. For these encryption systems, large input sizes occur when *size_of_row* is large. For any distributed system, large has a variable meaning that is relative to the system being examined. It could be anywhere from a few hundred thousand to a few billion. For these libraries, large is defined to be above a few hundred thousand. As discussed in the design chapters, *size_of_row* is the length of the vectors in `map`. To best demonstrate the effect *size_of_row* has on the system, *size_of_row* is steadily increased during the testing, so its effect on each design can be seen and compared to the original design.

To evaluate these designs a few profiling tools were used, discussed in Section 5.1. The test environments used for each design are detailed in Section 5.2. The results for GPUHElib are examined in Section 5.3, followed by the results for DistributedHElib in Section 5.4. Finally some conclusions are drawn regrading both of these designs in Section 5.5

5.1 Evaluation Tools

The following tools were used to evaluate the correctness of the modified libraries and record run times at various levels in the library.

5.1.1 Test Program

A test program was created based on testing programs provided with the original unmodified version of HElib. This new test program first sets up the ciphertexts that are used during computation. Two ciphertexts are created, both are the integers, 0 to $num\_slots\_in\_plaintext$.

Three operations are reported on in this document, of the six implemented. They are addition, subtraction, and multiplication of one ciphertext with another ciphertext. The other three operations supported, addition, subtraction, and multiplication of a ciphertext and a number, were not reported, as preliminary tests showed that they displayed similar results to their ciphertext-only counterparts. It was decided that for conciseness and to avoid repetition, that they would be left out of this document.

The program requires that the user pass in the $size\_of\_row$ they would like to use, which as discussed above, will be incremented during testing. The test program then performs each operation, checking the decrypted result after each operation to make sure that the results are correct, before moving on. Timing blocks were placed around each operation to record the overall time it took to perform the operation. The timers were printed out after each operation. Lower level timers, discussed below, were reset after each operation, so the lower level times reported with each operation were only times recorded for that operation.

This test program was compiled twice for both designs, first linking against the un-

modified library, and second against the modified library. This produced two executables, that were then run to generate the results described below.

5.1.2 HElib Timing Functions

The standard version of HElib provides fine grained timing functions that can be placed anywhere throughout the library. To utilize these functions is simple. First a timer is created. Upon the creation of the timer, it is started. A call to `stop` is made in code when the timer should stop. A timer can be started and stopped multiple times, and the average time will be recorded. The timers are stored in a map, and can be reset if needed. This setup allows for fine grained measurement of functions and detailed profiling of run times.

Each design, serial HElib, GPUHElib, and DistributedHElib required the timers be placed at some similar places, for comparison purposes, and some distinct places, in order to assess the efficiency of each unique design.

Serial HElib Timer Placement

There were two levels at which timers were placed, with each successive level more fine grained. The first level was in the test program, described above. This is the circuit level. The other timer was placed at the function level, inside the function that performed the operations in `DoubleCRT`. This function was where the double `for` loop was located that

both of the distributed designs are trying to improve upon.

GPUHElib Timer Placement

For GPUHElib, there are three levels at which timers were placed. As described above, the first timer was placed at the circuit level in the test program. This records the time that the entire operation took. The second level of timing was the function level. At this level a timer was placed in the function that is performing the operations in `DoubleCRT`. These first two timer levels allow for comparison against the serial version, as the serial version also has timers placed at these levels. The third and lowest level is the phase level. Four timers are placed at this level to record the setup (vector and stream creation), phase one (host to device memory transfer), phase two (GPU computation), and phase three (device to host memory transfer) times. The timers at this level allow for the times gathered at level two to be broken down even further, and allow each phase of level two to be examined closer.

DistributedHElib Timer Placement

For DistributedHElib, there are three levels at which timers were placed. The first at the circuit level in the test program. The second at the function level, in `DoubleCRT`, where the operation is being performed. These first two timer levels are necessary for the comparison against the serial version, as the serial version also has timers at these two levels. The third consists of two timers: the distribute (where the job assignment and data parti-

tioning happens) and the wait (where the sync function is called, and the dispatcher node is waiting for the compute nodes to finish and send back their results). This third level allows for a breakdown of the function level for further analysis.

5.2 Testing Environment

Both systems required unique testing environments that had the capabilities needed for each design. Both variants used machines with 64-bit Intel Core 2 Duo CPUs running at 3.0 Ghz with about 4 Gb of DDR2, 800 Mhz RAM.

5.2.1 GPU Testing Environment

GPUHElib was tested on a machine with a NVIDIA Quadro NVS 290 GPU, which has 256 MB of RAM. Also this particular GPU only has one copy engine. Thus the tests reported here are using the 2-Way Pipelining design discussed in Section 3.3. CUDA version 6.5 was used.

5.2.2 Distributed Computing Testing Environment

DistributedHElib was tested on a cluster of machines all connected through Ethernet. OpenMPI version 1.8.5 was used. Three cluster configurations were used, one with 4 nodes, one with 8 nodes, and one with 16 nodes.

5.3 GPUHElib Evaluation Results

As discussed earlier in this chapter, GPUHElib has three levels of timing information begin recorded. The first, and highest level, is the circuit level, where the high level operation is being computed. The second level is the function level, inside `DoubleCRT` where the `parts` are being operated on. And the lowest level is the phase level, where timing results are recorded for all four phases of the operation. The timing results for each of these levels is discussed in more detail in the following sections.

5.3.1 GPUHElib Circuit Level Run Times

Table 5.1 and Table 5.2 display the run times for serial HElib and GPUHElib tests respectively. Both tests were run with inputs sizes starting at 1,000 and increasing until 400,000.

Figure 5.1 visualizes these times as the slowdown of GPUHElib over serial HElib. A value of 1 means that the serial version and the GPU version had the same run time. Above 1 means that the GPU design has a slower run time, and below 1 means that the GPU has a faster run time.

One can see in Figure 5.1 that for the smaller input sizes, the run times for the GPU are much larger than the serial version for addition and subtraction. They started off taking about 72 times as long to complete, compared to the serial version. The multiplication operation also starts off taking longer, however only about 2.2 times as long. The run

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	1.400E-05	1.080E-04	2.545E-03	5.396E-03	5.354E-03	1.053E-02
Sub	1.400E-05	1.300E-04	2.532E-03	5.304E-03	5.366E-03	1.075E-02
Mul	4.962E-03	1.036E-01	1.157	2.648	7.217	1.232E+01

Table 5.1: Serial HElib circuit level run times (in seconds)

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	1.050E-03	1.675E-03	8.972E-03	1.646E-02	1.671E-02	3.704E-02
Sub	1.012E-03	1.805E-03	9.023E-03	1.683E-02	1.682E-02	3.700E-02
Mul	1.106E-02	1.187E-01	1.309	2.868	7.393	9.333

Table 5.2: GPUHElib circuit level run times (in seconds)

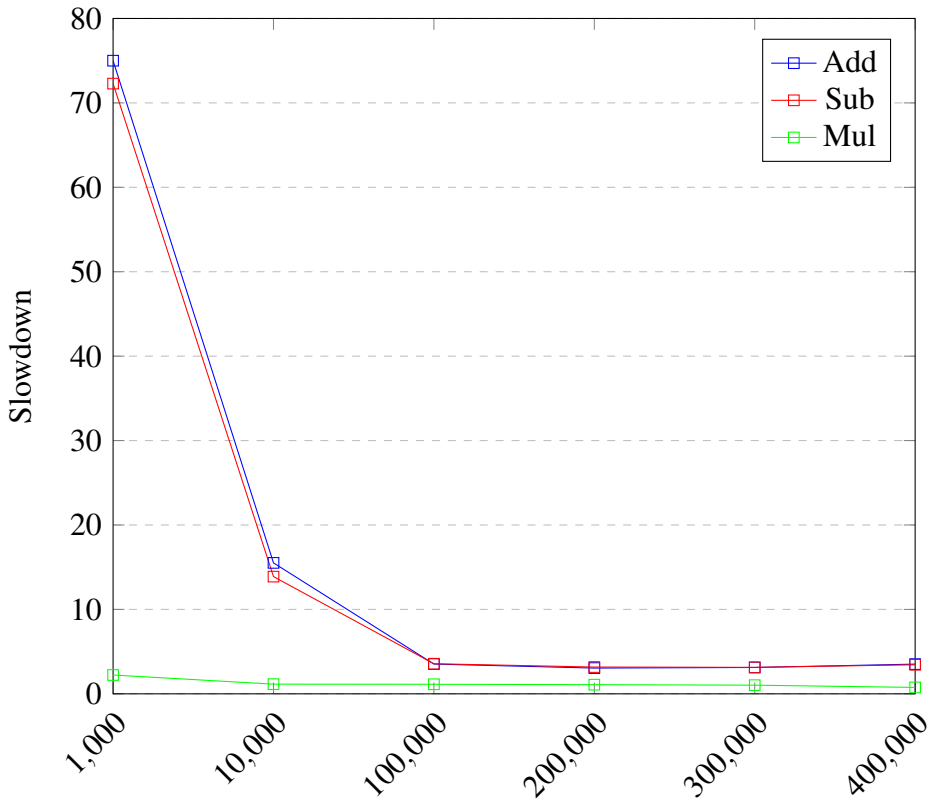

Figure 5.1: Run Time Comparison at Circuit Level

times for serial HElib and GPUHElib get closer and closer, as the inputs sizes approach 400,000. The addition and subtraction circuit run times minimize at about 3 times as long, for the 300,000 size input. However for the 400,000 size input, the times go in the opposite direction desired, becoming about 3.5 times as long. The multiplication circuit actually has the best results, with the 400,000 test taking about .75 times the serial version. The multiplication circuit took about 3/4 the time to complete in GPUHElib compared to the serial version. While this result might look good, further analysis of the lower level tests show that this was probably not caused by the usage of the GPU, but by other operations computed during the multiplication operation being faster. The function level run times will be examined next.

5.3.2 GPUHElib Function Level Run Times

Table 5.3 and Table 5.4 display the run times for the serial HElib and GPUHElib tests respectively. As noted before, both tests were run with input sizes ranging from 1,000 to 400,000.

Figure 5.2, Figure 5.3, and Figure 5.4 show the comparisons between the run times for each of the operations at the function level. Also displayed in the figures is the run time slow down of the GPU variant compared to the serial version. For example, in Figure 5.2, the 104.2x above 1,000 means that the GPU variant took 104.2 times longer to complete compared to the serial version.

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	5.000E-06	4.825E-05	1.007E-03	2.648E-03	2.653E-03	5.466E-03
Sub	5.000E-06	6.150E-05	1.259E-03	2.644E-03	2.674E-03	5.366E-03
Mul	2.900E-05	2.830E-04	2.879E-03	5.863E-03	5.856E-03	1.176E-02

Table 5.3: Serial HElib function level run times (in seconds)

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	5.210E-04	8.298E-04	4.392E-03	8.257E-03	8.346E-03	1.864E-02
Sub	5.020E-04	8.950E-04	4.498E-03	8.398E-03	8.395E-03	1.848E-02
Mul	5.302E-04	1.006E-03	6.599E-03	1.273E-02	1.276E-02	2.687E-02

Table 5.4: GPUHElib function level run times (in seconds)

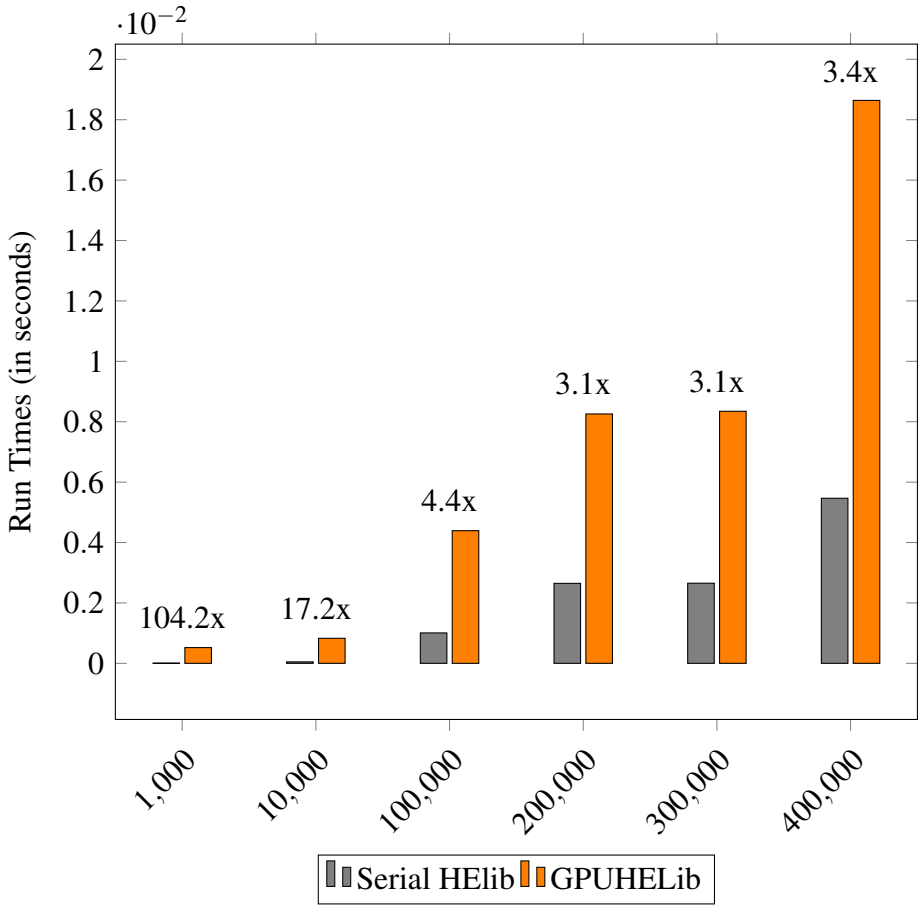

Figure 5.2: Add Run Times Comparison at Function Level

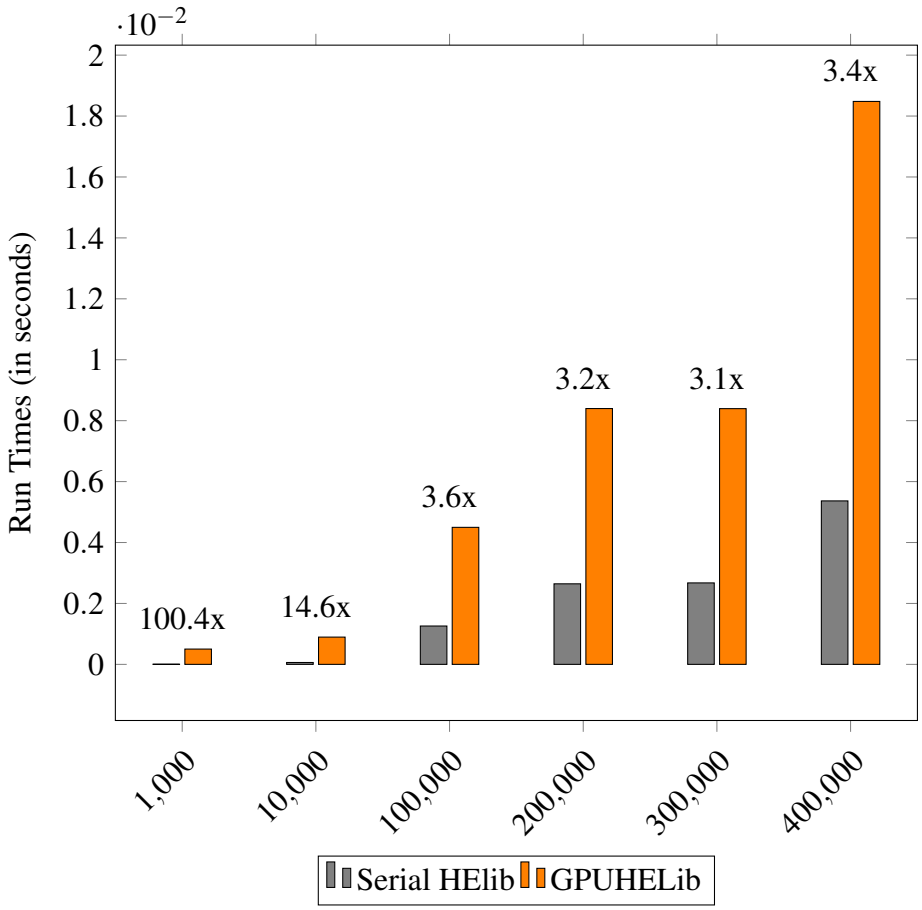

Figure 5.3: Sub Run Times Comparison at Function Level

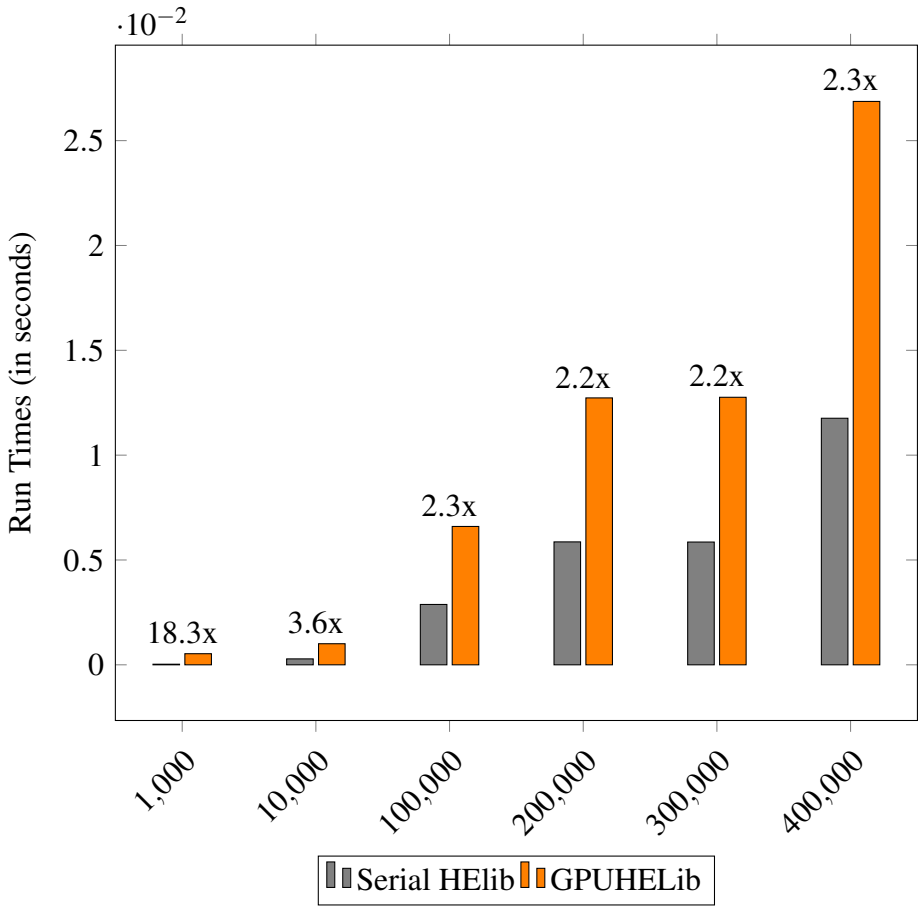

Figure 5.4: Mul Run Times Comparison at Function Level

All these times again reiterate that for the smaller input sizes, the run times for the GPU variant are vastly larger, almost 100x for addition and subtraction, and about 18x for multiplication. As the input sizes increase, the run times get closer and closer, but minimize at about 3.1x for addition and subtraction and at about 2x for multiplication. Again one can see that for the 400,000 input size, the slow downs increase for all the operations compared to the previous input size, 300,000, going from 3.2x to 3.4x and 3.5x respectively and from 2.2x to 2.3x. The results for the multiplication times make it clear that the speedup seen at the circuit level must be caused by other factors than the GPU implementation of multiplication. These times show that multiplication behaves exactly like the other operations in terms of run time patterns. The cause for these slow downs is evident after examining the recorded times at the phase level.

5.3.3 GPUHElib Phase Level Run Times

Table 5.5, Table 5.6, and Table 5.7 all display the phase level run times for each operation respectively. The four phases are as follows: setup (vector and stream creation), phase 1 (host to device memory copy), phase 2 (operation on GPU), and phase 3 (device to host memory copy). These times have been split between two plots for each operation. One group of plots focuses on the overall run time of serial HElib compared to the setup and phase 2 times recorded. These are the "Operation" plots, Figure 5.5, Figure 5.7, and Figure 5.9. The other group of plots focus on the overhead phases, phase 1 and phase 3, of

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Setup	2.420E-04	2.465E-04	2.848E-04	3.013E-04	3.015E-04	3.133E-04
Phase 1	9.500E-05	2.568E-04	2.137E-03	4.289E-03	4.345E-03	1.164E-02
Phase 2	4.425E-05	3.350E-05	1.283E-04	2.423E-04	2.420E-04	3.728E-04
Phase 3	1.373E-04	2.900E-04	1.837E-03	3.420E-03	3.454E-03	6.308E-03

Table 5.5: GPUHElib Add phase level run times (in seconds)

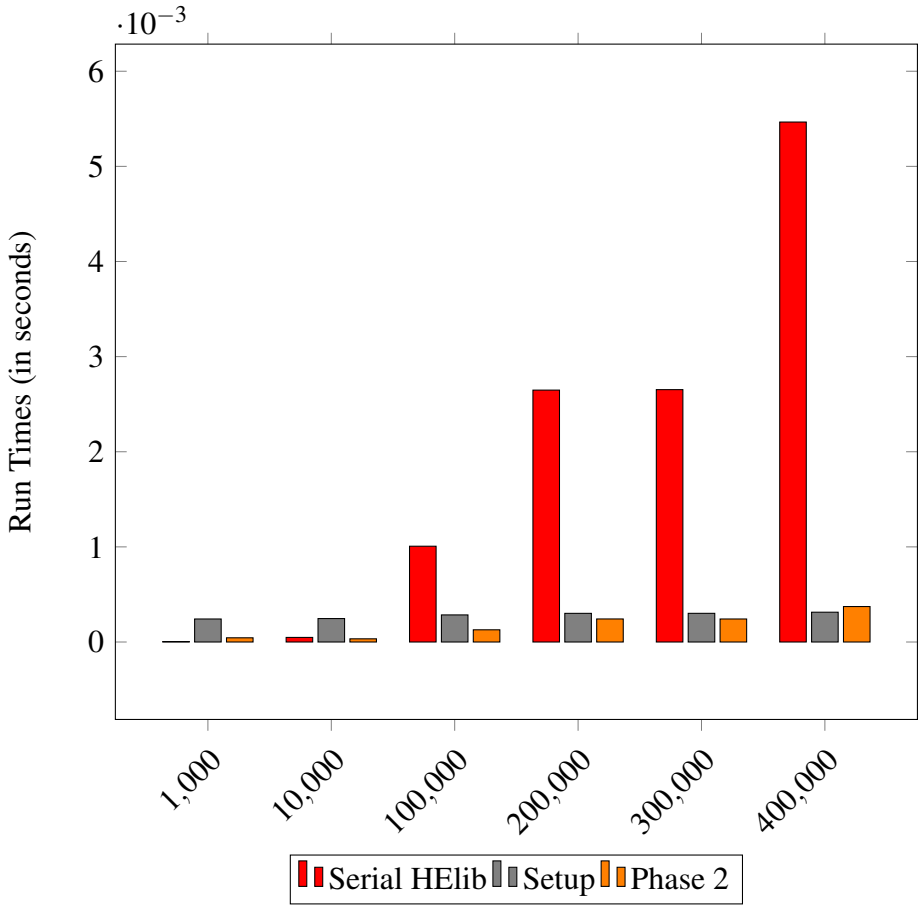

Figure 5.5: Add Phase Level Run Times Comparison - Operation

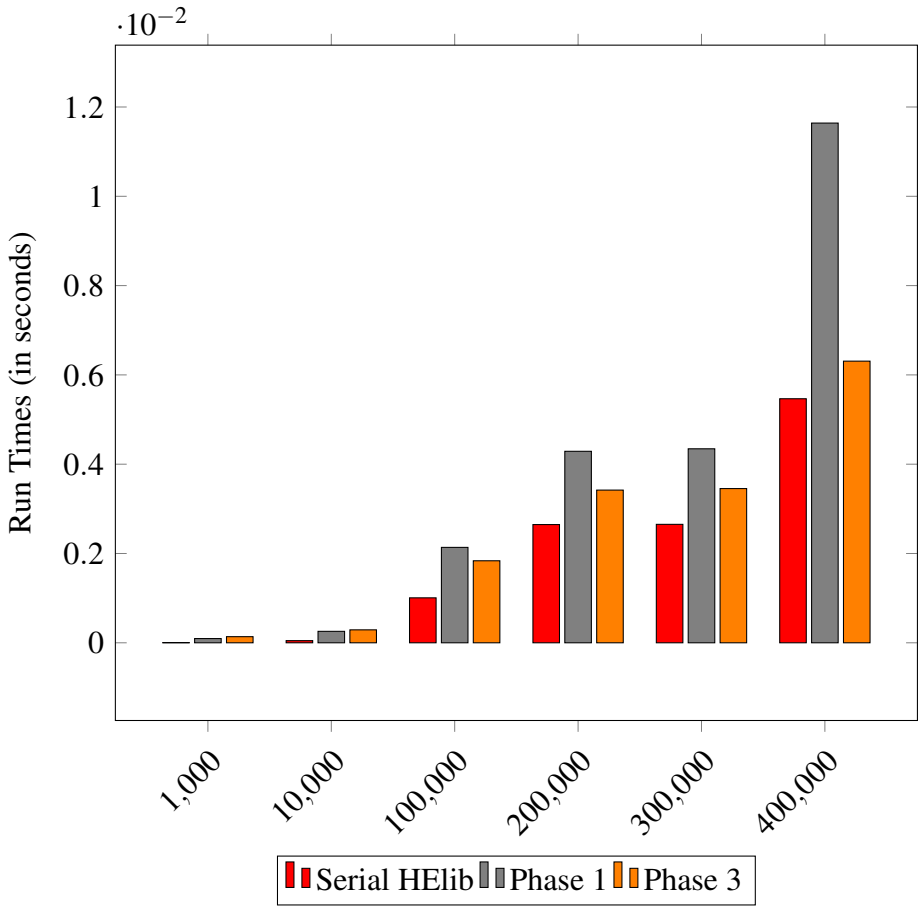

Figure 5.6: Add Phase Level Run Times Comparison - Memory

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Setup	2.455E-04	2.830E-04	3.035E-04	3.065E-04	3.045E-04	3.050E-04
Phase 1	9.000E-05	2.535E-04	2.262E-03	4.377E-03	4.377E-03	1.152E-02
Phase 2	3.400E-05	4.200E-05	1.280E-04	2.420E-04	2.425E-04	3.745E-04
Phase 3	1.275E-04	3.120E-04	1.799E-03	3.466E-03	3.465E-03	6.275E-03

Table 5.6: GPUHElib Sub phase level run times (in seconds)

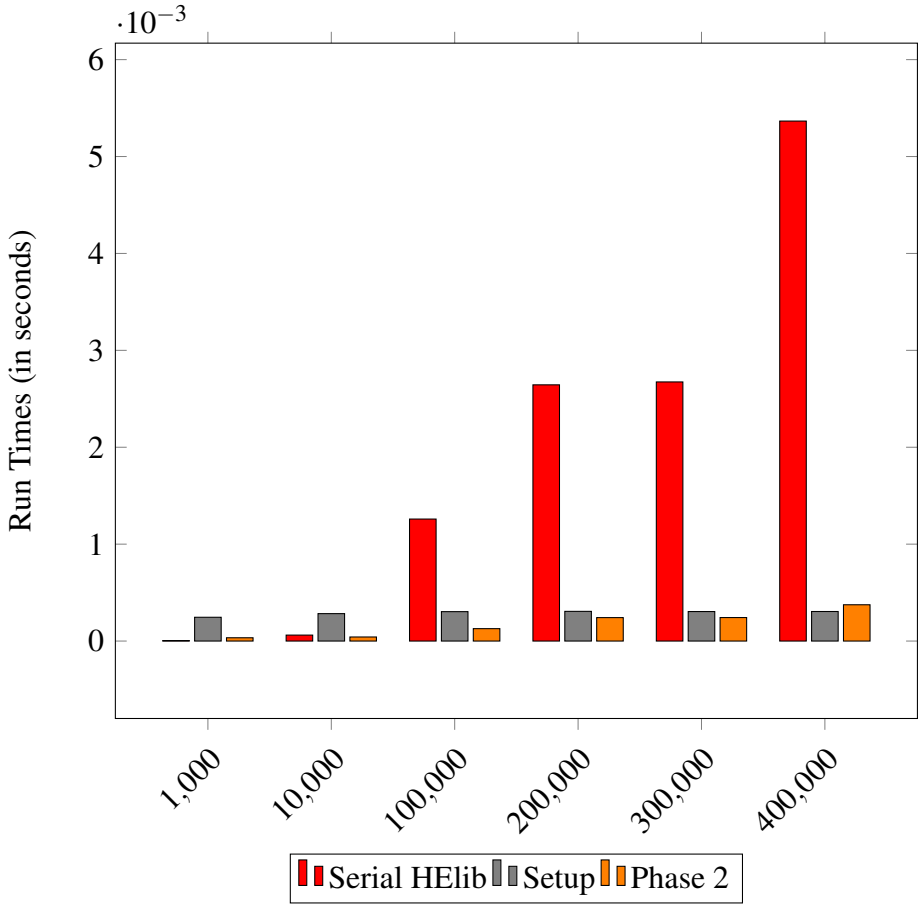

Figure 5.7: Sub Phase Level Run Times Comparison - Operation

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Setup	2.405E-04	2.447E-04	3.027E-04	3.200E-04	3.157E-04	3.207E-04
Phase 1	7.933E-05	1.853E-04	1.845E-03	3.955E-03	3.997E-03	1.029E-02
Phase 2	3.000E-05	3.183E-05	1.247E-04	2.398E-04	2.353E-04	3.508E-04
Phase 3	1.772E-04	5.418E-04	4.323E-03	8.213E-03	8.212E-03	1.591E-02

Table 5.7: GPUHElib Mul phase level run times (in seconds)

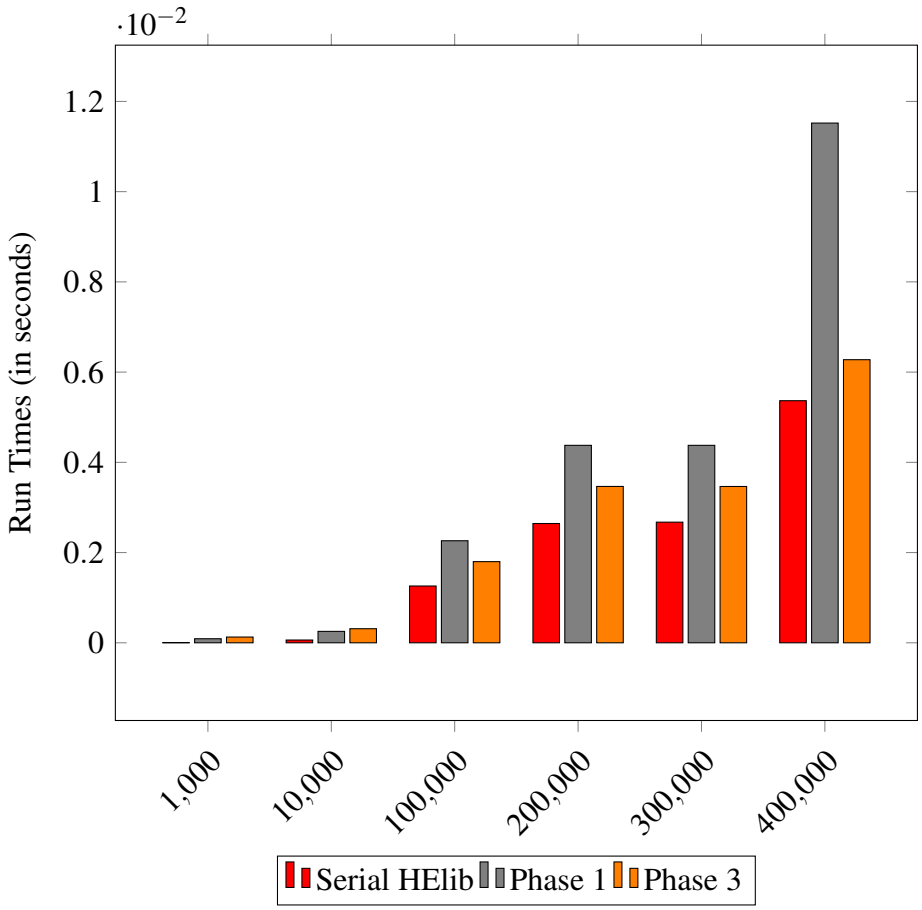

Figure 5.8: Sub Phase Level Run Times Comparison - Memory

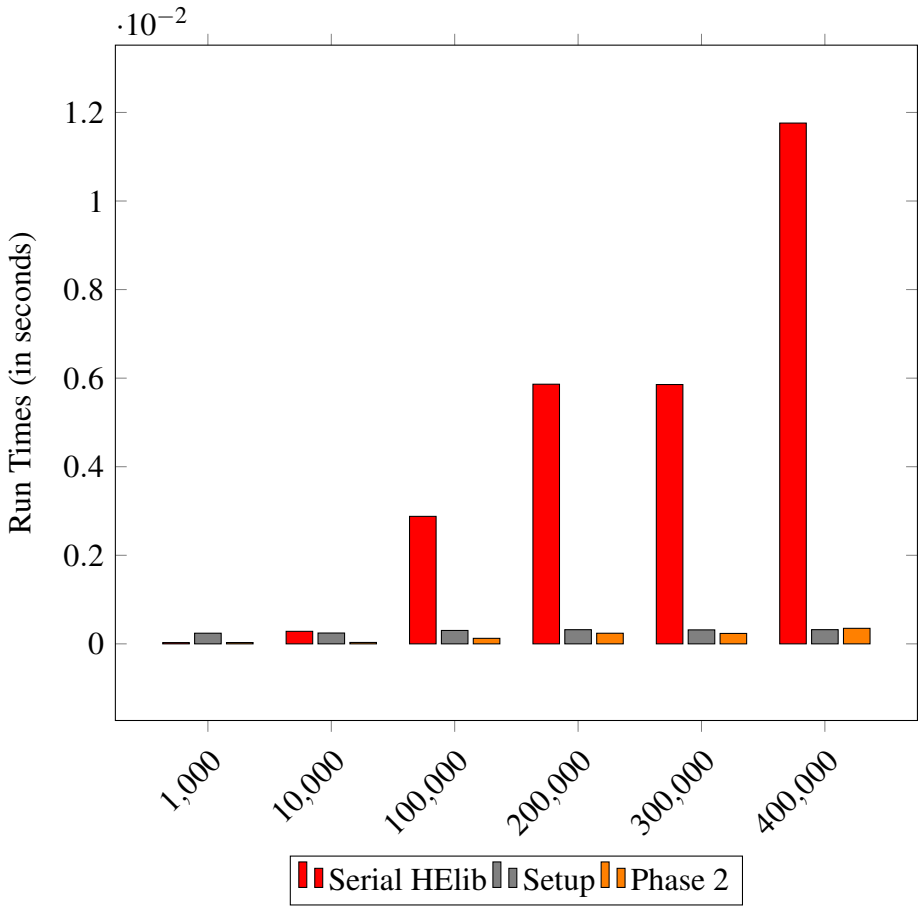

Figure 5.9: Mul Phase Level Run Times Comparison - Operation

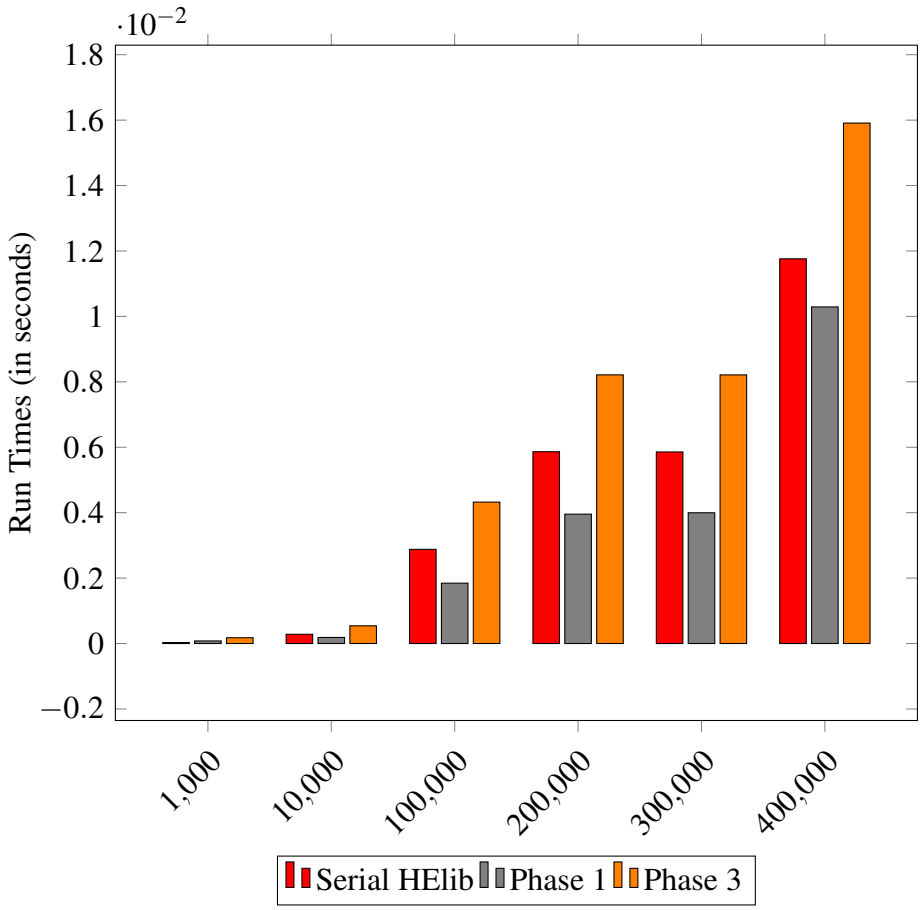

Figure 5.10: Mul Phase Level Run Times Comparison - Memory

GPUHElib compared to the overall run time of serial HElib for each operation and are denoted “Memory”. These are Figure 5.6, Figure 5.8, and Figure 5.10.

The “Operation” plots show the design is working as intended and as all GPU designs are ideally suppose to work. The reliance on the GPU to perform the computation drastically reduces the run time for those phases. Furthermore, as the input size increases largely, the run times for setup and phase 2 only increase slightly.

The setup phase always took about the same amount of time no matter the input size, only varying by about 8E-05 from the smallest input to the largest. Phase 2 times steadily increased, however not at the rapid pace of the serial HElib versions. This is what is expected when using a GPU to perform operations.

These are the desired results when working with a GPU. The offloading of work onto the GPU allows for the operation portions of the work to drastically decrease in run time. Of course, these results do not characterize the overall recorded times for GPUHElib compared to serial HElib. Therefore something else must be going wrong, causing the run times to be longer than the their serial counterparts.

The “Memory” plots show where this design fails. The amount of time needed to move the data back and forth from the GPU is immense. The times for phase 1 and phase 3 are always larger than the entire run time of the serial version for every operation across all input sizes, except for the multiplication operation, where the phase 1 times are actually less than the overall run time of the serial version, but not by much. These times are

very disappointing, as they are the reason this design is performing so poorly. Luckily, these times could be lower, given better hardware and possible future work, which could make GPUHElib a viable option. If the problem was in the design or if the run times for the setup or phase 1 were worse, then the total design would not have any hope of being used. But because they are in the memory transfer phases, there is still hope that this design could become viable with further work.

5.4 DistrubtedHElib Evaluation Results

As discussed earlier in this chapter, DistributedHElib has three levels of timing information begin recorded. The highest level is the circuit level in the test program. The next is at the function level, inside `DoubleCRT`. The third and lowest level is the distribute and wait level. This level has two timers which measure the distribute time, the time necessary for the dispatcher node to assign work and partition the data, and the wait time, the time the dispatcher node is waiting for the compute nodes to finish their work and return the results. Additionally three cluster sizes, 4, 8, and 16 nodes, were used during testing The timing results for each of these levels is discussed in more detail below.

5.4.1 DistributedHElib Circuit Level Run Times

Table 5.8, Table 5.9, Table 5.10, and Table 5.11 display the run times for serial HElib and DistributedHElib tests. Both tests were run with inputs sizes starting at 1,000 and increas-

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	1.400E-05	1.070E-04	2.525E-03	5.347E-03	5.313E-03	1.048E-02
Sub	1.400E-05	1.250E-04	2.491E-03	5.308E-03	5.244E-03	1.080E-02
Mul	4.996E-03	1.030E-01	1.151	2.644	7.286	1.202E+01

Table 5.8: Serial HElib circuit level run times (in seconds)

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	1.457E-03	7.496E-03	8.933E-02	1.783E-01	1.550E-01	3.455E-01
Sub	1.544E-03	7.479E-03	7.773E-02	1.666E-01	1.647E-01	3.203E-01
Mul	1.254E-02	1.371E-01	1.559	3.444	8.350	1.009E+01

Table 5.9: DistributedHElib circuit level run times (in seconds) on 4 nodes

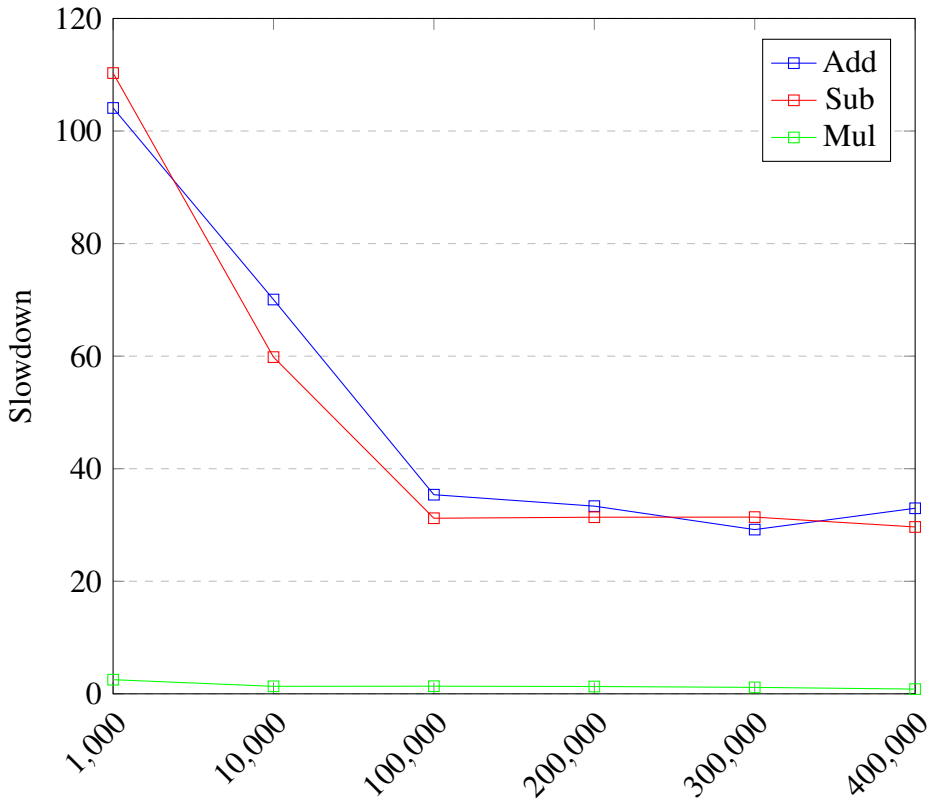

Figure 5.11: Run Time Comparison at Circuit Level on 4 Nodes

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	1.712E-03	9.161E-03	1.076E-01	2.228E-01	2.266E-01	4.540E-01
Sub	1.791E-03	9.219E-03	1.117E-01	2.113E-01	2.139E-01	4.560E-01
Mul	1.350E-02	1.454E-01	1.666	3.689	8.589	1.136E+01

Table 5.10: DistributedHElib circuit level run times (in seconds) on 8 nodes

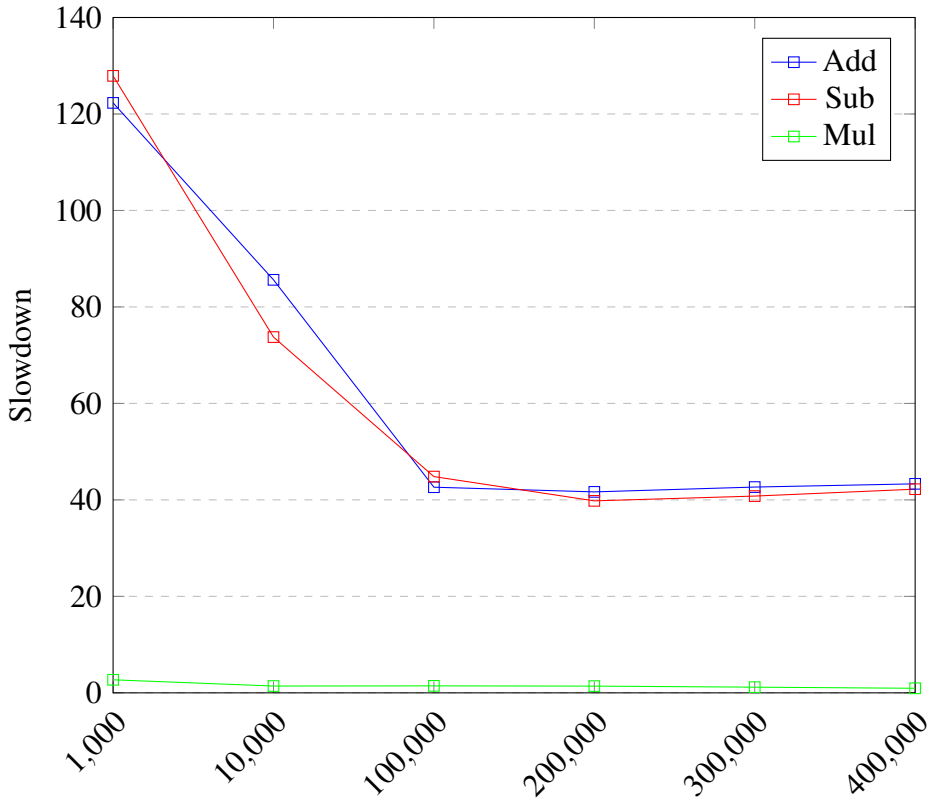

Figure 5.12: Run Time Comparison at Circuit Level on 8 Nodes

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	1.772E-03	9.960E-03	1.187E-01	2.391E-01	2.390E-01	4.911E-01
Sub	1.850E-03	9.741E-03	1.194E-01	2.398E-01	2.364E-01	4.744E-01
Mul	2.484E-02	1.623E-01	1.717	3.811	8.664	1.135E+01

Table 5.11: DistributedHElib circuit level run times (in seconds) on 16 nodes

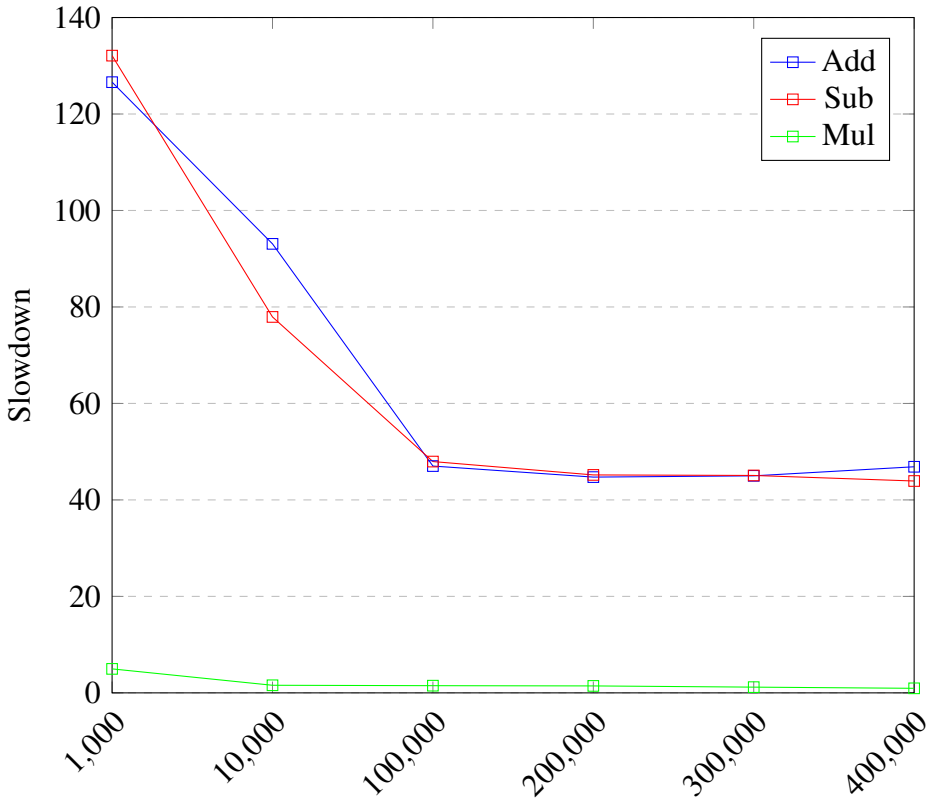

Figure 5.13: Run Time Comparison at Circuit Level on 16 Nodes

ing until 400,000. The clusters sizes were 4, 8, and 16 nodes.

Figure 5.11, Figure 5.12, and Figure 5.13 visualize these times as the slowdown of DistributedHElib over serial HElib. A value of 1 means that the serial version and the distributed version had the same run time. Above 1 means that the distributed design has a slower run time, and below 1 means that the distributed version has a faster run time.

One can see in these figures that for the smaller input sizes, the run times for addition and subtraction across all cluster sizes are much larger for the distributed design compared to the serial design. These operations take over 100 times as long to complete as their serial counterparts. The multiplication operation also takes longer, however only about 2.5 to 5 times as long depending on the number of nodes in the cluster.

As the input sizes increase, the addition and subtraction operation slowdowns decrease, until they plateau at around 35, 40, and 45 times as long for the 4, 8, and 16 node clusters respectively. Once they reach these slowdowns, they bounce around, but never continue on the downward trajectory they had for the first few input size increases. The multiplication operation on the other hand always decreases as the input size increases. As the input sizes increase, the multiplication slowdowns approach 1, and at the 400,000 size input, all three cluster sizes dive below 1. The cluster size of 4 has the best results, having about a .84 slowdown, with the other two sizes, 8 and 16, having about a .94. These results mean, for the 400,000 input size, the distributed variant of HElib had faster run times than the serial version across all cluster sizes. Similar to the GPU results, while

these times look good, further examination of the lower level tests show that this speed up is probably not happening because of the distributed design, but because of other factors. Next the function level times are examined.

5.4.2 DistributedHElib Function Level Run Times

Table 5.12, Table 5.13, Table 5.14, and Table 5.15 display the run times at the function level for serial HElib and DistributedHElib on 4, 8, and 16 nodes.

Figure 5.14, Figure 5.15, and Figure 5.16 show the comparisons between the run times for each of the operations at the function level across all variants and cluster sizes. Also displayed in the figures is the average run time slow down, across all cluster sizes, of the distributed variant compared to the serial version. So, for example, in Figure 5.14, the 168.7x above 1,000 means that the distributed variant took 168.7 times longer to complete compared to the serial version.

Again for the smaller inputs, these figures show that the distributed variant takes much longer to complete compared to the serial version. For addition and subtraction, the operations take about 170 times as long, and for multiplication, about 25 times as long. As the input sizes increase, the slow downs do decline, however level out around the 200,000 size input. The addition and subtraction operations level out at about 40x, and the multiplication operation levels out at about 16x. Again one can see that the run times plateau for the addition and subtraction operations, just as they did at the circuit level. The results

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	5.000E-06	4.800E-05	9.873E-04	2.588E-03	2.611E-03	5.457E-03
Sub	5.000E-06	5.850E-05	1.238E-03	2.646E-03	2.614E-03	5.391E-03
Mul	2.967E-05	2.838E-04	2.887E-03	5.845E-03	5.850E-03	1.183E-02

Table 5.12: Serial HElib function level run times (in seconds)

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	7.423E-04	3.604E-03	4.898E-02	8.503E-02	8.048E-02	1.764E-01
Sub	7.695E-04	3.735E-03	3.886E-02	8.329E-02	8.233E-02	1.601E-01
Mul	7.148E-04	3.298E-03	3.517E-02	7.759E-02	7.771E-02	1.482E-01

Table 5.13: DistributedHElib function level run times (in seconds) on 4 nodes

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	8.698E-04	4.427E-03	5.489E-02	1.124E-01	1.125E-01	2.186E-01
Sub	8.930E-04	4.427E-03	5.583E-02	1.056E-01	1.069E-01	2.280E-01
Mul	8.055E-04	4.032E-03	4.830E-02	9.798E-02	9.804E-02	1.951E-01

Table 5.14: DistributedHElib function level run times (in seconds) on 8 nodes

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	9.180E-04	7.439E-03	5.794E-02	1.166E-01	1.152E-01	2.316E-01
Sub	9.220E-04	4.866E-03	5.971E-02	1.199E-01	1.182E-01	2.372E-01
Mul	8.373E-04	4.498E-03	5.036E-02	1.025E-01	1.054E-01	2.126E-01

Table 5.15: DistributedHElib function level run times (in seconds) on 16 nodes

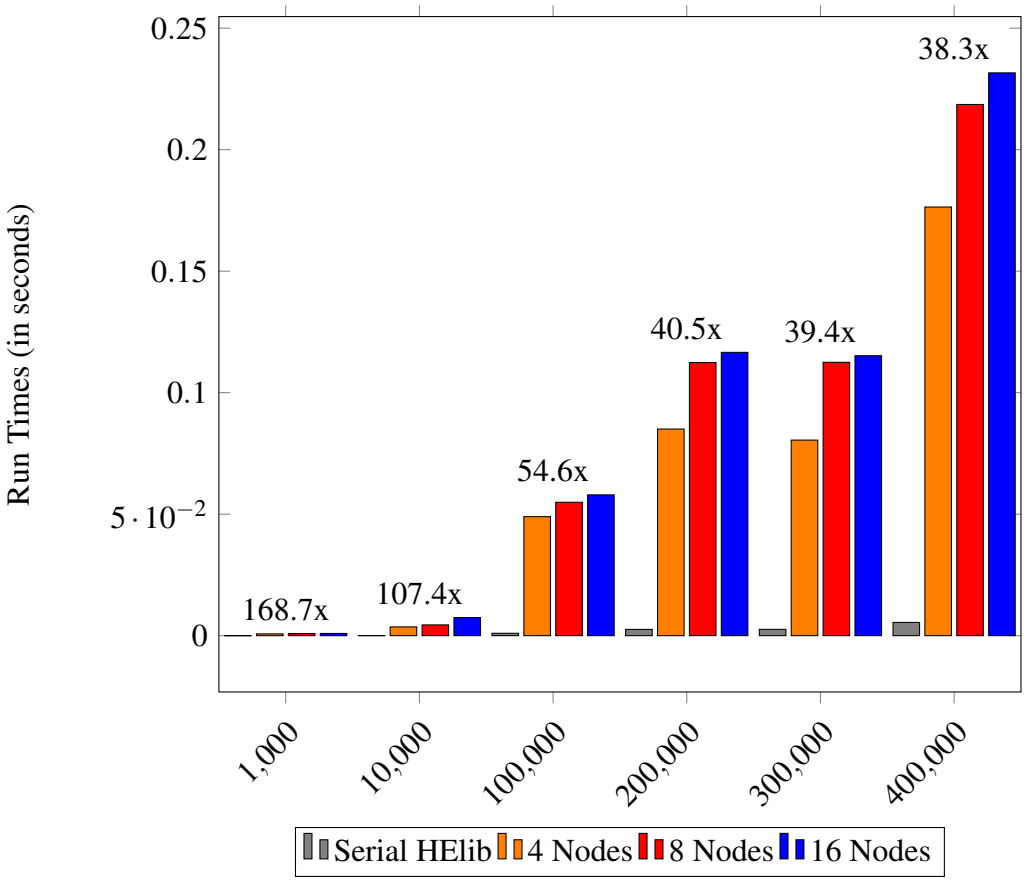

Figure 5.14: Add Run Times Comparison at Function Level

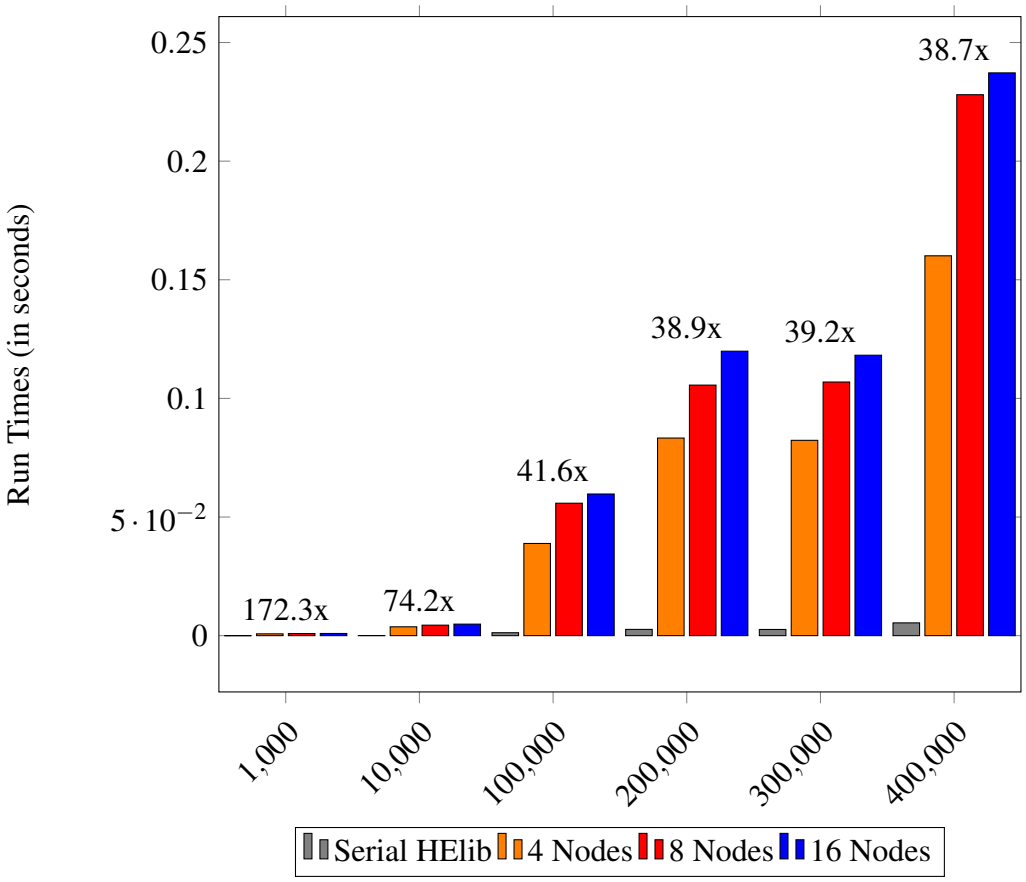

Figure 5.15: Sub Run Times Comparison at Function Level

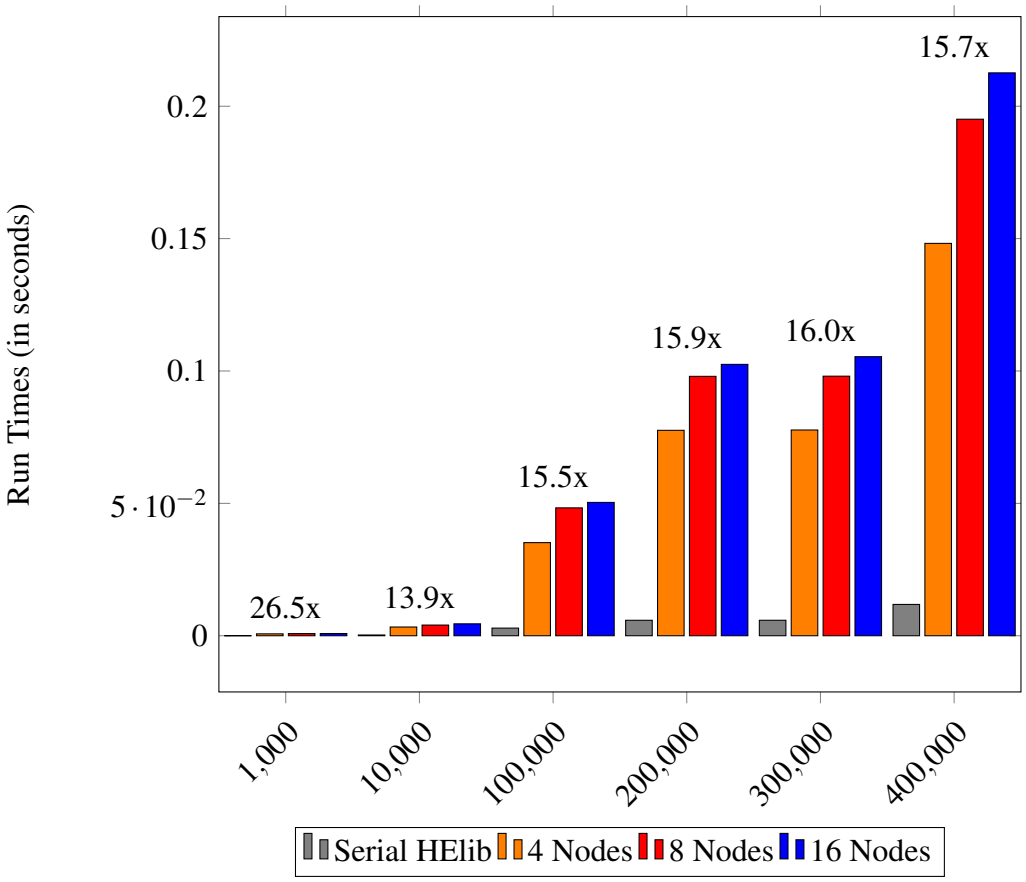

Figure 5.16: Mul Run Times Comparison at Function Level

for the multiplication operation at this level show it also has this characteristic, whereas at the circuit level, this characteristic was not observed. Thus the results observed at the circuit level must not be the result of the distributed design, but something else. By looking at the distribute and wait times at level 3, one can understand why these results are occurring.

5.4.3 DistributedHElib Distribute and Wait Run Times

Table 5.16, Table 5.18, and Table 5.20 display the distribute run times for each operation across the three cluster sizes. Table 5.17, Table 5.19, and Table 5.21 display the sync run times for each operation across the three cluster sizes. These times have been split into two plots for each operation. One group of plots focuses on the distribute times (the time it took to partition the data and assign the work) and compares them to the overall run time of the serial version. These are the "Distribute" plots, Figure 5.17, Figure 5.19, and Figure 5.21. The second group of plots display the sync time (the time the compute nodes took to receive the data, compute the results, and send the data back to the dispatcher node) compared to the overall run time for the serial design. Figure 5.18, Figure 5.20, and Figure 5.22 display these results, and are the "Sync" plots.

By looking at the "Distribute" plots, one can see that the partitioning of data and assignment of work times across all clusters sizes and operations remains constant even when the input sizes are increased. This looks good, but remember that this part of the

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	2.240E-04	4.638E-04	2.373E-04	2.500E-04	2.438E-04	2.698E-04
Sub	2.405E-04	5.700E-04	2.470E-04	2.760E-04	2.995E-04	2.340E-04
Mul	2.015E-04	4.190E-04	2.150E-04	2.585E-04	2.668E-04	2.700E-04

Table 5.16: DistributedHElib distribute run times (in seconds) on 4 nodes

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	5.158E-04	3.137E-03	4.874E-02	8.478E-02	8.023E-02	1.761E-01
Sub	5.260E-04	3.162E-03	3.861E-02	8.301E-02	8.203E-02	1.599E-01
Mul	5.110E-04	2.875E-03	3.495E-02	7.733E-02	7.744E-02	1.479E-01

Table 5.17: DistributedHElib sync run times (in seconds) on 4 nodes

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	2.773E-04	6.288E-04	2.740E-04	2.978E-04	2.913E-04	3.055E-04
Sub	2.915E-04	7.465E-04	3.015E-04	2.950E-04	2.955E-04	3.085E-04
Mul	2.427E-04	5.405E-04	2.872E-04	3.077E-04	2.953E-04	3.198E-04

Table 5.18: DistributedHElib distribute run times (in seconds) on 8 nodes

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	5.903E-04	3.796E-03	5.461E-02	1.121E-01	1.122E-01	2.183E-01
Sub	5.940E-04	3.855E-03	5.553E-02	1.053E-01	1.066E-01	2.277E-01
Mul	5.608E-04	3.490E-03	4.801E-02	9.767E-02	9.774E-02	1.948E-01

Table 5.19: DistributedHElib sync run times (in seconds) on 8 nodes

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	2.985E-04	7.418E-04	2.923E-04	3.083E-04	3.515E-04	3.235E-04
Sub	3.130E-04	8.175E-04	3.270E-04	3.190E-04	3.270E-04	3.445E-04
Mul	2.548E-04	6.265E-04	2.853E-04	3.130E-04	3.157E-04	3.338E-04

Table 5.20: DistributedHElib distribute run times (in seconds) on 16 nodes

	size_of_row					
	1,000	10,000	100,000	200,000	300,000	400,000
Add	6.178E-04	6.692E-03	5.765E-02	1.163E-01	1.149E-01	2.313E-01
Sub	6.060E-04	4.045E-03	5.938E-02	1.196E-01	1.178E-01	2.368E-01
Mul	5.807E-04	3.869E-03	5.007E-02	1.022E-01	1.051E-01	2.123E-01

Table 5.21: DistributedHElib sync run times (in seconds) on 16 nodes

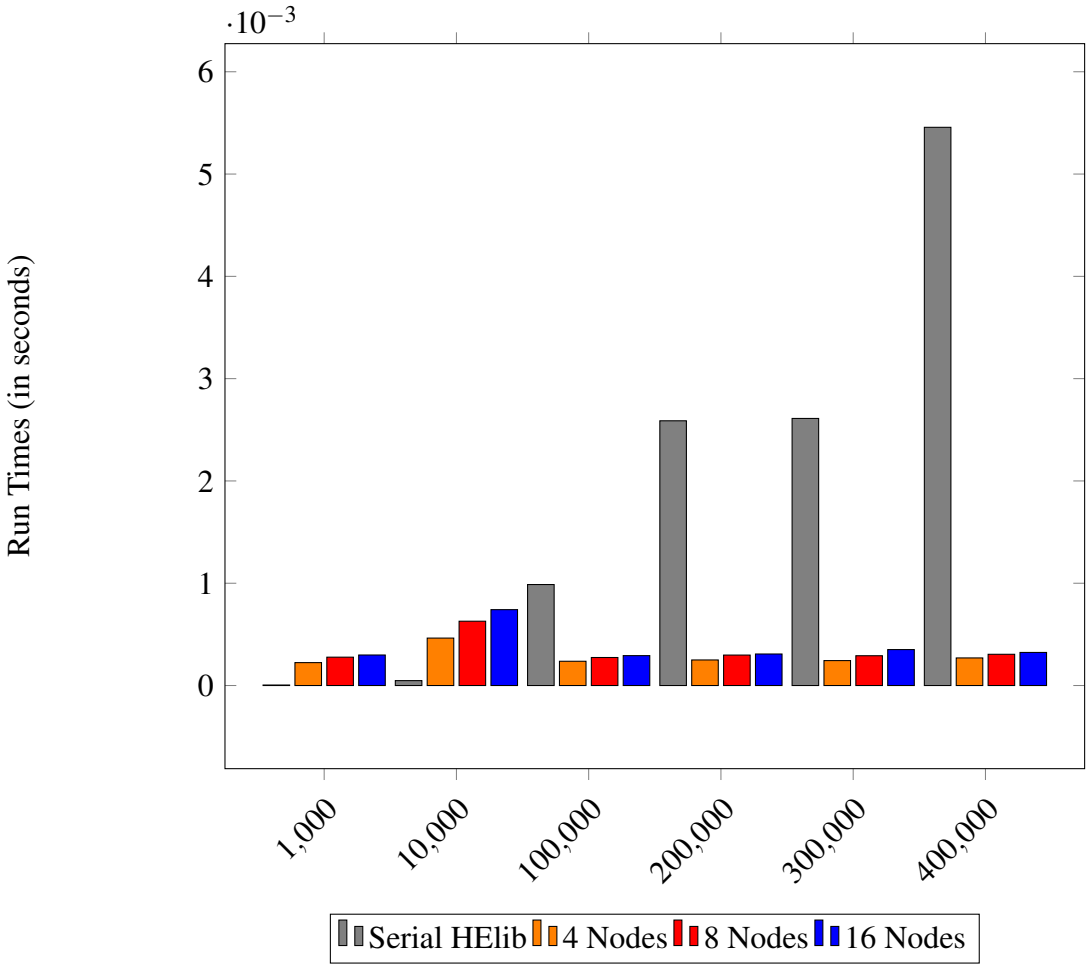

Figure 5.17: Add Third Level Run Times Comparison - Distribute

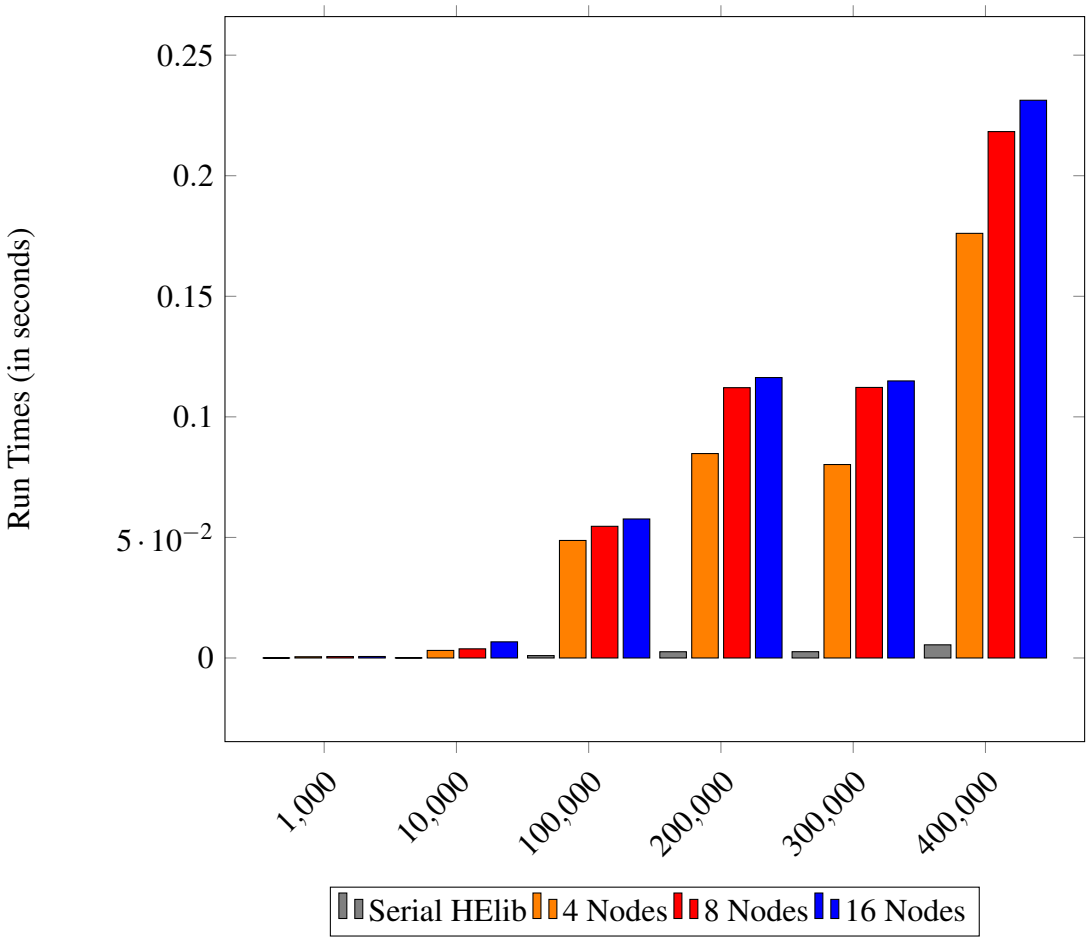

Figure 5.18: Add Third Level Run Times Comparison - Sync

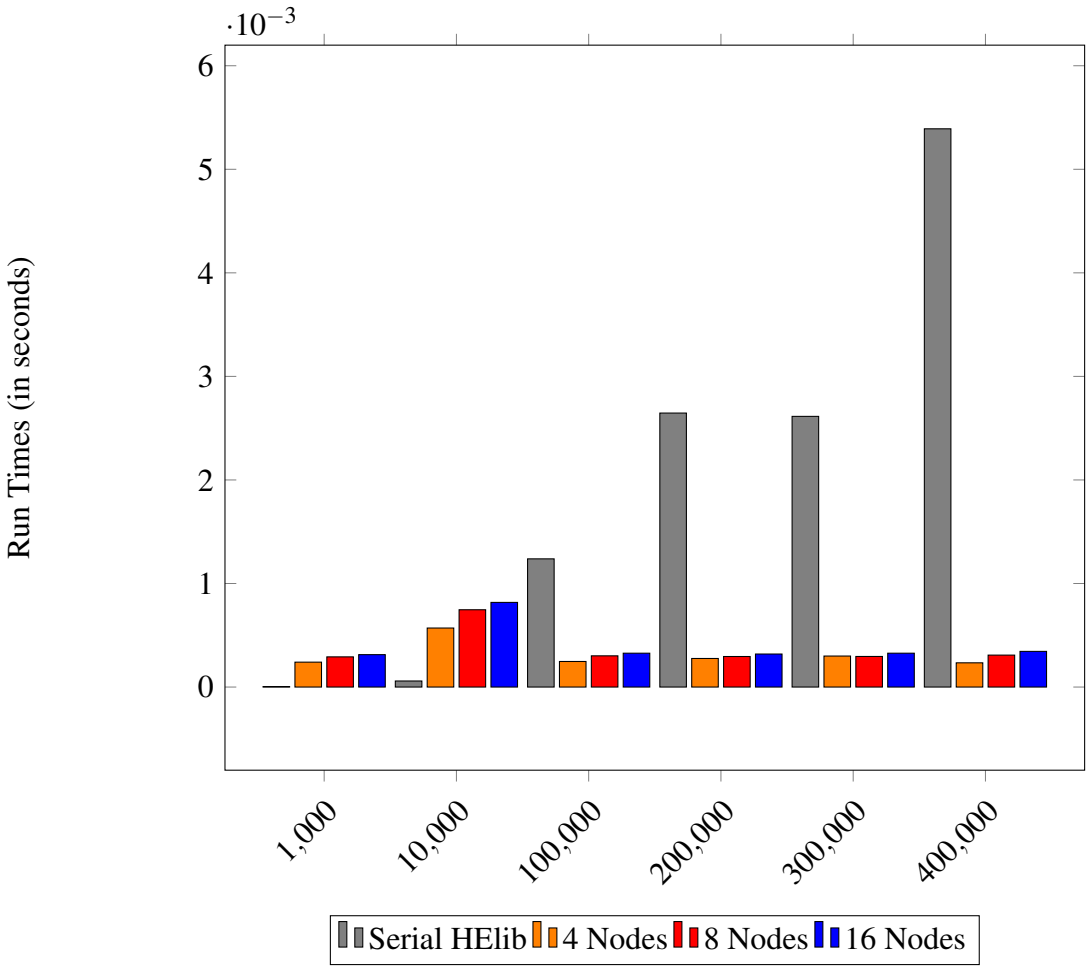

Figure 5.19: Sub Third Level Run Times Comparison - Distribute

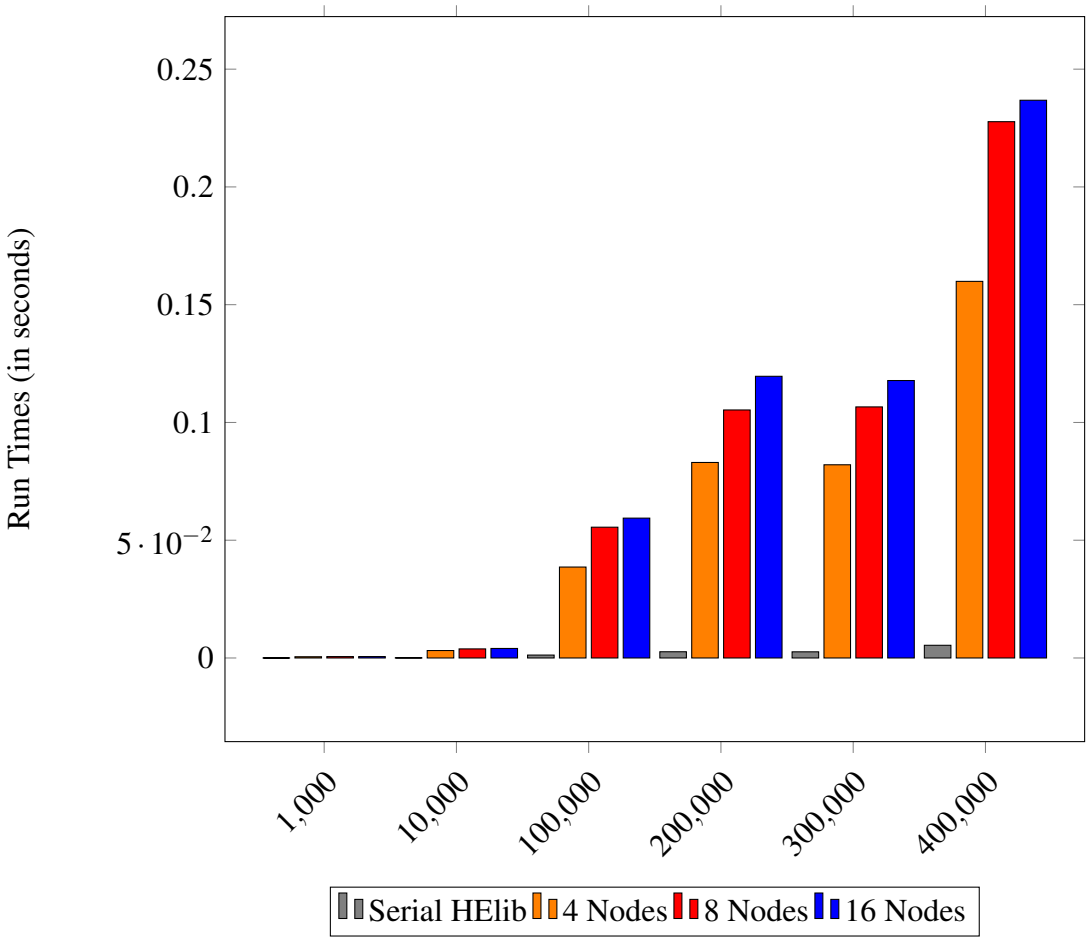

Figure 5.20: Sub Third Level Run Times Comparison - Sync

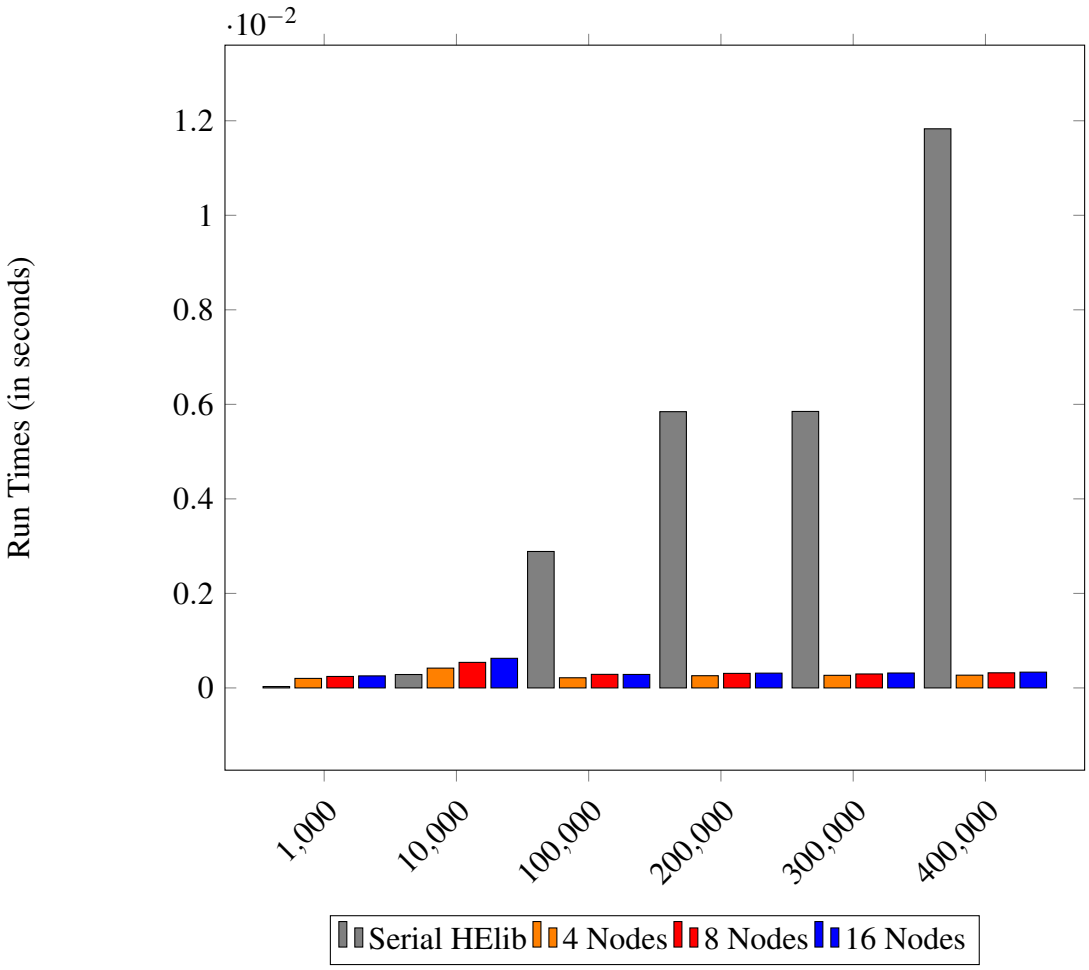

Figure 5.21: Mul Third Level Run Times Comparison - Distribute

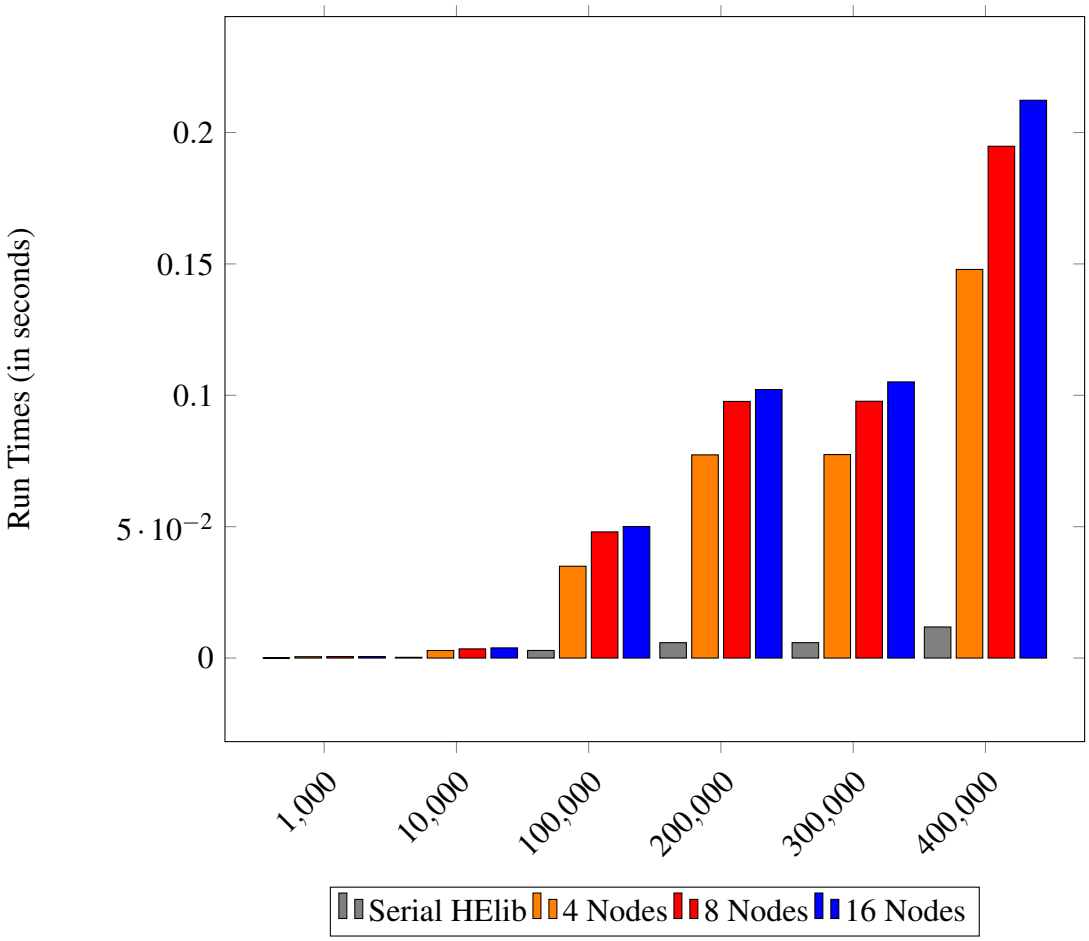

Figure 5.22: Mul Third Level Run Times Comparison - Sync

work only records the times it takes to partition the data and assign the work, not send the data to the compute nodes. Only non-blocking sends and receives are scheduled during this portion of the recording. The actual sending and receiving between the dispatcher and compute nodes happens mostly during the sync portion of the recording. This is because the non-blocking send and receive functions only schedule requests, which are later fulfilled in the background, during the sync part. So it is to be expected that these results are seen.

The "Sync" plots show where this design is failing. The amount of time needed to send the data over the network to the compute nodes, have them perform the operation and then send the data back is much greater than the serial times. This has to do with the network speed, which causes a bottleneck, and which is responsible for the plateau characteristic seen in the circuit and function level times. The network speeds are much too slow in this environment to allow for this design to be viable. With better speeds (ones which would not cause this bottleneck to happen) this design might provide better results. Additionally, future work to try and minimize the amount of data transfers between the dispatcher and compute nodes could also lead to better run times.

5.5 Evaluation Conclusions

As mentioned in the introduction of this chapter, when working with distributed systems, the overhead time, which comes from the memory transfer phases of each design, usu-

ally is the downfall of distributed systems. The same is true for these designs. GPUHElib suffered from slow transfer times between the CPU and GPU, which caused slow downs compared to the serial version. Similarly, DistributedHElib suffered from slow network speeds, which caused a bottleneck when transferring data between the dispatcher node and the compute nodes. With better transfer speeds, both of these designs might be viable, however under these circumstances, they are not.

CHAPTER 6

FUTURE WORK

6.1 GPUHElib Future Work

As seen in Chapter 5, GPUHElib failed to provide any speed up over serial HElib. This was because the memory transfer times between the CPU and GPU where much too great. Therefore all of the future work for GPUHElib is designed around trying to reduce these times.

6.1.1 Persistent Memory in GPU

To cut down on memory transfer times, it would be useful to keep all the memory that needs to be transferred in the current design, in the GPUs memory permanently. Thus when the operations need to be performed, the data does not need to be transferred from the CPU to the GPU, as the GPU already contains all the data that needs to be operated on. This would reduce the memory transfer times down to almost zero, thus making the run times only dependent on the operation times, which as seen in Section 5.3, were much

lower than the serial HElib run times.

6.1.2 Full Operation Implementation

Only the most used operations were implemented on the GPU. Because of this, the results from the GPU operations needed to be copied back to the CPU, so that other unsupported operation could take place. It would be beneficial then to implement all of the operations supported by serial HElib using the GPU. Thus the results would not need to be copied back, as all the operations that a user could perform would be supported on the GPU.

6.2 DistributedHElib Future Work

As seen in Chapter 5, DistributedHElib failed to provide any run time improvement over serial HElib. This was a result of the network speed being a bottleneck. Thus all future work for DistributedHElib would try to reduce amount of data sent over the network.

6.2.1 Distributed Memory on Compute Nodes

In order to rely less on the network to transfer data, one could design the system so that the data is partitioned onto compute nodes. Thus each compute node would be responsible for a piece that only they were responsible for. Then when operations occur, these compute nodes can perform the operation on their piece of the data, without needing to receive the data from the dispatcher node, as the data will already be present on them.

Thus data transfers would only happen during initial setup, and finalization, reducing the network traffic, and removing the network bottleneck.

6.2.2 Full Operation Implementation

Similar to the future work for GPUHElib, it would be useful to have all the operations supported on the compute nodes. Because they are not supported in the library as it is designed now, the results of operations must be sent back to the dispatcher node, so that it can perform one of these unsupported operations. With all operations supported however, the data would not need to be sent back to the dispatcher node to perform the operation, because the operation could be run on the compute nodes.

CHAPTER 7

CONCLUSIONS

This work was attempting to improve the run time of the homomorphic encryption library, HElib. By applying distribute system techniques, which would suit the intended deployment environment for HElib, we tried to improve the run times of operations being performed by HElib. Two libraries were designed: GPUHElib and DistributedHElib.

GPUHElib attempted to add GPU functionality to HElib, in order to cut down on the run times for the operations. The design of these operations on the GPU required memory mapping from the CPU to the GPU, overflow considerations for GPU operations, and pipeline techniques be applied. Unfortunately, as these tests showed, this design did not perform better than the serial version. This was due to the memory transfer times from CPU to GPU being much too large to facilitate a speedup, even though the operation times were much lower. With further work however, this design might become viable.

DistributedHElib applied distributed computing techniques in an attempt to speed up the run time of HElib. This design utilized a master-slave cluster architecture and non-block send and receive function to facilitate concurrent computation. Again as the tests

showed, this design failed to perform better than the serial version. This was because the network speed caused a bottleneck, and made the run times drastically slower than the serial version. Similarly, with future work, this design might become a viable option.

While both of these variants were unsuccessful, they have promise, and in the future might be useful in the design of faster homomorphic encryption libraries.

BIBLIOGRAPHY

[1] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. Cryptology ePrint Archive, Report 2012/078, 2012. http://eprint.iacr.org/.

[2] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *Public-Key Cryptography–PKC 2013*, pages 1–13. Springer, 2013.

[3] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. http://eprint.iacr.org/.

[4] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. Cryptology ePrint Archive, Report 2011/344, 2011. http://eprint.iacr.org/.

[5] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Advances in Cryptology–EUROCRYPT 2012*, pages 446–464. Springer, 2012.

[6] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[7] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. Cryptology ePrint Archive, Report 2010/520, 2010. http://eprint.iacr.org/.

[8] Craig Gentry and Shai Halevi. Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. Cryptology ePrint Archive, Report 2011/279, 2011. http://eprint.iacr.org/.

[9] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. Cryptology ePrint Archive, Report 2013/340, 2013. http://eprint.iacr.org/.

[10] Shai Halevi and Victor Shoup. Design and implementation of a homomorphic-encryption library. 2013.

[11] Shai Halevi and Victor Shoup. Algorithms in helib. Cryptology ePrint Archive, Report 2014/106, 2014. http://eprint.iacr.org/.

[12] Shai Halevi and Victor Shoup. Bootstrapping for helib. Cryptology ePrint Archive, Report 2014/873, 2014. http://eprint.iacr.org/.

[13] Adriana Lopez-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. Cryptology ePrint Archive, Report 2013/094, 2013. http://eprint.iacr.org/.

[14] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[15] Nigel P Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography–PKC 2010*, pages 420–443. Springer, 2010.

[16] N.P. Smart and F. Vercauteren. Fully homomorphic simd operations. Cryptology ePrint Archive, Report 2011/133, 2011. http://eprint.iacr.org/.

[17] Damien Stehle and Ron Steinfeld. Faster fully homomorphic encryption. Cryptology ePrint Archive, Report 2010/299, 2010. http://eprint.iacr.org/.

APPENDICES

APPENDIX A

KERNELS

A.1 Addition

A.1.1 Addition of two DoubleCRT objects

Listing A.1: Addition Kernel for Two DoubleCRT

```
__global__ void vectorAddMod(long *vector_A,
                             long *vector_B,
                             long width,
                             long sub_width,
                             long offset) {
        long tid;
        tid = offset + blockIdx.x * blockDim.x + threadIdx.x;
        while ((tid-offset) < width) {
            // extract modulus from array
                uint64_t modulus =
                        (uint64_t)vector_moduli[tid / sub_width];
                // compute sum
                uint64_t sum =
                        (uint64_t)vector_A[tid] +
                        (uint64_t)vector_B[tid];
                // perform modulus operation, store result
                vector_A[tid] =
                    (long)(sum - modulus * (sum / modulus));
                tid += blockDim.x * gridDim.x;
```

```
    }
    return;
}
```

A.1.2 Addition of a DoubleCRT object and a constant

Listing A.2: Addition Kernel for a DoubleCRT and a constant

```
__global__ void vectorAddMod(long *vector_A,
                             long width,
                             long sub_width,
                             long offset) {
    long tid;
    tid = offset + blockIdx.x * blockDim.x + threadIdx.x;
    while ((tid-offset) < width) {
        // extract modulus from array
            uint64_t modulus =
                (uint64_t)vector_moduli[tid / sub_width];
            // compute sum
            uint64_t sum =
                (uint64_t)vector_A[tid] +
                (uint64_t)vector_ns[tid / sub_width];
            // perform modulus operation, store result
            vector_A[tid] =
                (long)(sum - modulus * (sum / modulus));
            tid += blockDim.x * gridDim.x;
    }
    return;
}
```

A.2 Subtraction

A.2.1 Subtraction of two DoubleCRT objects

Listing A.3: Subtraction Kernel for Two DoubleCRT

```
__global__ void vectorSubMod(long *vector_A,
                             long *vector_B,
                             long width,
                             long sub_width,
                             long offset) {
        long tid;
        tid = offset + blockIdx.x * blockDim.x + threadIdx.x;
        while ((tid-offset) < width) {
            // extract modulus from array
                uint64_t modulus =
                    (uint64_t)vector_moduli[tid / sub_width];
                // compute difference
                // add modulus to ensure result is > 0,
            // will be removed when mod by modulus
                uint64_t diff =
                    (uint64_t)vector_A[tid] +
                    modulus -
                    (uint64_t)vector_B[tid];

                // perform modulus operation, store result
                vector_A[tid] =
                    (long)(diff - modulus * (diff / modulus));
                tid += blockDim.x * gridDim.x;
        }
        return;
}
```

A.2.2 Subtraction of a DoubleCRT object and a constant

Listing A.4: Subtraction Kernel for a DoubleCRT and a constant

```
__global__ void vectorSubMod(long *vector_A,
                             long width,
```

```
                                long sub_width,
                                long offset) {
        long tid;
        tid = offset + blockIdx.x * blockDim.x + threadIdx.x;
        while ((tid-offset) < width) {
            // extract modulus from array
                uint64_t modulus =
                    (uint64_t)vector_moduli[tid / sub_width];
                // compuate difference
                // add modulus to ensure result is > 0,
                // will be removed when mod by modulus
                uint64_t diff =
                    (uint64_t)vector_A[tid] +
                    modulus -
                    (uint64_t)vector_ns[tid / sub_width];
                // perform modulus operation, store result
                vector_A[tid] =
                    (long)(diff - modulus * (diff / modulus));
                tid += blockDim.x * gridDim.x;
        }
        return;
}
```

A.3 Multiplication

A.3.1 Multiplication of two DoubleCRT objects

Listing A.5: Multiplication Kernel for Two DoubleCRT

```
__global__ void vectorMultMod(long *vector_A,
                                long *vector_B,
                                long width,
                                long sub_width,
                                long offset) {
```

```
long tid;
tid = offset + blockIdx.x * blockDim.x + threadIdx.x;
while ((tid-offset) < width) {
    // extract modulus from array
        uint64_t modulus =
            (uint64_t)vector_moduli[tid / sub_width];
        // compute intermediate values
        uint64_t a1 =
            (uint64_t)vector_A[tid] / 4294967296;
        uint64_t a2 =
            (uint64_t)vector_A[tid] - 4294967296 * a1;
        uint64_t b1 =
            (uint64_t)vector_B[tid] / 4294967296;
        uint64_t b2 =
            (uint64_t)vector_B[tid] - 4294967296 * b1;
// compute z0, perform modulus
        uint64_t z0 = a2 * b2;
        z0 = z0 - modulus * (z0 / modulus);
// compute part of z1
        uint64_t p12 = a1 * b2;
//compute part of z1
        uint64_t p21 = a2 * b1;
// compute z1, perform modulus
        uint64_t z1 = p12 + p21;
        z1 = z1 - modulus * (z1 / modulus);
        // compute z2
        uint64_t z2 = a1 * b1;
// loop to get z2*2^32 and z1*2^32
        int i;
        for(i=0; i<32; i++) {
                z1 = z1 * 2;
                if(z1 >= modulus) {
```

```
                        z1 =
                            z1 - modulus *
                            (z1 / modulus);
                    }
                    z2 = z2 * 2;
                    if(z2 >= modulus) {
                        z2 =
                            z2 - modulus *
                            (z2 / modulus);
                    }
                }
            // loop to get z2*2^32*2^32=z2*2^64
                for(i=i; i<64; i++) {
                    z2 = z2 * 2;
                    if(z2 >= modulus) {
                        z2 =
                            z2 - modulus *
                            (z2 / modulus);
                    }
                }
            // compute final value
                uint64_t z = z0 + z1 + z2;
                // perform modulus operation, store result
                vector_A[tid] =
                    (long)(z - modulus * (z / modulus));
                tid += blockDim.x * gridDim.x;
            }
            return;
}
```

A.3.2 Multiplication of a DoubleCRT object and a constant

Listing A.6: Multiplication Kernel for a DoubleCRT and a constant

```
__global__ void vectorMultMod(long *vector_A,
                              long width,
                              long sub_width,
                              long offset) {
    long tid;
    tid = offset + blockIdx.x * blockDim.x + threadIdx.x;
    while ((tid-offset) < width) {
        // extract modulus from array
            uint64_t modulus =
                (uint64_t)vector_moduli[tid / sub_width];
    //compute intermediate values
            uint64_t a1 =
                (uint64_t)vector_A[tid] / 4294967296;
            uint64_t a2 =
                (uint64_t)vector_A[tid] -
                4294967296 * a1;
            uint64_t b1 =
                (uint64_t)vector_ns[tid / sub_width] /
                4294967296;
            uint64_t b2 =
                (uint64_t)vector_ns[tid / sub_width] -
                4294967296 * b1;
    // compute z0, perform modulus
            uint64_t z0 = a2 * b2;
            z0 = z0 - modulus * (z0 / modulus);
    // compute part of z1
            uint64_t p12 = a1 * b2;
    // compute part of z1
            uint64_t p21 = a2 * b1;
    // compute z1, perform modulus
            uint64_t z1 = p12 + p21;
```

```
        z1 = z1 - modulus * (z1 / modulus);
        //compute z2
        uint64_t z2 = a1 * b1;
// loop to get z2*2^32 and z1*2^32
        int i;
        for(i=0; i<32; i++) {
                z1 = z1 * 2;
                if(z1 > modulus) {
                        z1 = z1 - modulus *
                            (z1 / modulus);
                }
                z2 = z2 * 2;
                if(z2 > modulus) {
                        z2 = z2 - modulus *
                            (z2 / modulus);
                }
        }
// loop to get z2*2^32*2^32=z2*2^64
        for(i=i; i<64; i++) {
                z2 = z2 * 2;
                if(z2 > modulus) {
                        z2 = z2 - modulus *
                            (z2 / modulus);
                }
        }
// compute final value
        uint64_t z = z0 + z1 + z2;
        // perform modulus operation, store result
        vector_A[tid] =
            (long)(z - modulus * (z / modulus));
        tid += blockDim.x * gridDim.x;
}
```

```
            return;
    }
```

APPENDIX B

VECTOR MANAGEMENT

B.1 Device Vector Management

Listing B.1: Device Vector Management

```
void init_vector(long **vector,
                 long size,
                 long num_elements,
                 long *length) {
    // perform initial allocation
        if(*vector == NULL) {
                 if(cudaSuccess !=
                     cudaMalloc((void **)vector, size))
                 {
                     GPU_error(__FILE__,
                         __LINE__,
                         cudaGetLastError());
                 }
                 *length = num_elements;
        // resize if vector is too small
        } else if (*length < num_elements) {
                 cudaFree(*vector);
                 if(cudaSuccess !=
                     cudaMalloc((void **)vector, size))
                 {
```

```
                GPU_error(__FILE__,
                    __LINE__,
                    cudaGetLastError());
            }
            *length = num_elements;
        }
}
```

B.2 Compute Node Buffer Management

Listing B.2: Compute Node Buffer Management

```
void init_vector(long **vector,
                 long *vector_length,
                 long need_length) {
    // perform the initial allocation
        if(*vector == NULL) {
            *vector =
                (long *) malloc(sizeof(long) *
                    need_length);
            *vector_length = need_length;
    // resize if buffer is too small
        } else if (*vector_length < need_length) {
            *vector =
                (long *)realloc(vector,
                    sizeof(long) * need_length);
            *vector_length = need_length;
        }
}
```

APPENDIX C

CONCURRENCY MANAGEMENT

C.1 Device Stream Management

Listing C.1: Device Stream Management

```
void create_streams(long need_num_streams) {
        for(long i = num_streams;
                i<need_num_streams; i++) {
                if(cudaSuccess !=
                    cudaStreamCreate(&stream[i]))
                {
                    GPU_error(__FILE__,
                        __LINE__,
                        cudaGetLastError());
                }
        }

        num_streams = need_num_streams;
}

void init_streams(long need_num_streams) {
        if(num_streams == 0) {
                stream = (cudaStream_t *)
                    malloc(need_num_streams
                        * sizeof(cudaStream_t));
```

```
            create_streams(need_num_streams);
        } else if (num_streams <
                need_num_streams) {
            stream =
                (cudaStream_t *)realloc(stream,
                                        need_num_streams *
                                        sizeof(cudaStream_t));
            create_streams(need_num_streams);
        }
}
```

C.2 Synchronization Management

Listing C.2: Synchronization Management

```
void sync() {
    // until the queue is empty
        while(!request_queue.empty()) {
            // get the first request
            // out of the queue
                MPI::Request request =
                    request_queue.front();
                request_queue.pop();

                // if unfinished, put back
                if(!request.Test()) {
                        request_queue.push(request);
                }
        }
}
```