# Paillier Encryption Acceleration via GPU Programming


**Andrea Schmidt, Computer Science**

Mentor: Dr. Tingting Chen

Kellogg Honors College Capstone Project


## Problem

Today, massive amounts of information are stored digitally. This includes sensitive information, such as credit card information, personal health data, etc., all of which can be vulnerable to attack, and thus needs to be encrypted. In this case, we are using the Paillier encryption algorithm. Paillier is a additive homomorphic cryptosystem. It relies on heavy multiplication and modulation with a very large factor n = p*q, where p and q are prime numbers. The more data being encrypted, the less efficient the program becomes. Specifically, the multiplication of the exceedingly large numbers takes a significant amount of work. We attempt to remedy this issue by accelerating the encryption via GPU programming.

## Approach

**GPU-Accelerated Programming with CUDA**

The Graphics Processing Unit, GPU, is composed of hundreds of cores that can handle thousands of threads simultaneously, in contrast to the CPU (central processing unit), which is composed of just few cores with lots of cache memory that can handle a few software threads at a time**.** Thus, we will be using CUDA, a parallel processing platform and API, to integrate GPU programming into our Paillier implementation. With CUDA, we can make function calls from the CPU sequential code (the host) to the GPU device code for more computation-intensive functions **(Figure 1).** As stated before, Paillier utilizes very large numbers that are too large for primitive data types, meaning greater than 32-bits. Because of this, we use a BigInteger class to represent these, which has its own multiplication function. We developed a multiplication function for excessively large numbers that will run in parallel on the GPU.

The GPU multiplication function was implemented in a using C++, for sequential code, and CUDA , for the parallel code. To represent the large integers, we used a CUDA vector type, ulonglong4, which would store four long integers which, when concatenated, would represent our entire large integer. The values of our unsigned large integers were initialized as random in the sequential CPU code. After this, they were passed to another CPU function which allocated the memory on the device, copied the variables from the host onto the device, and called the multiplication kernel.

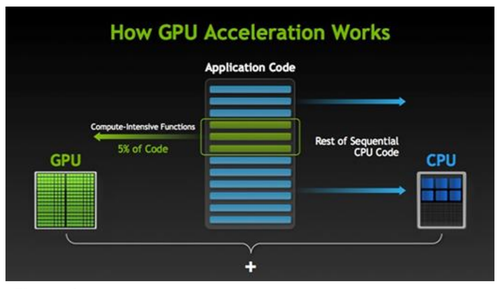


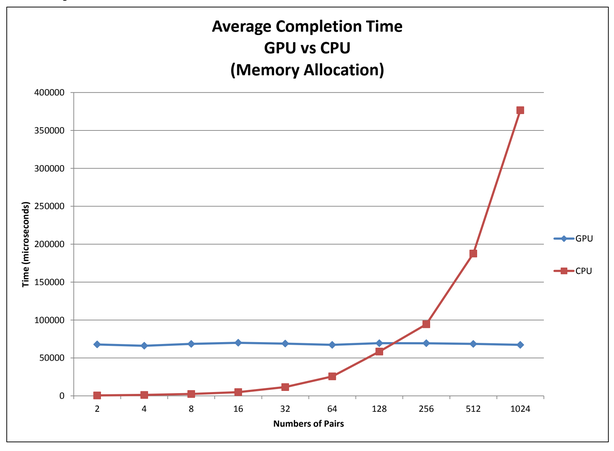

**Figure 1**

The multiplication kernel initialized the amount of blocks that would be run on the device, and then called the actual CUDA multiplication function on the device. The multiplication itself was implemented via PTX virtual instruction set. After the device code finished the multiplication, it returned to the CPU allocation function and copied the result from the device to the host and deallocated the space initially allocated on the device.

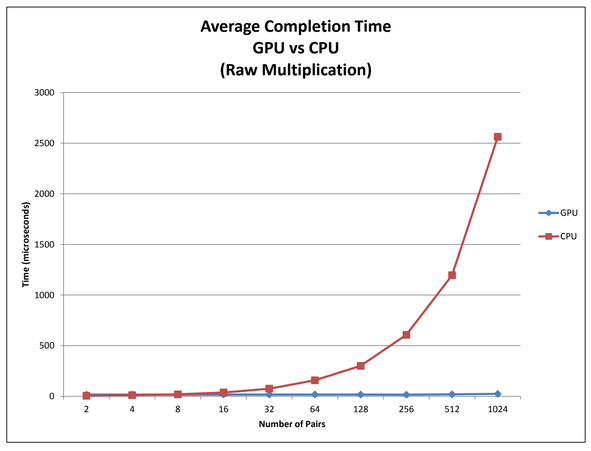
## Data and Analysis

In order to test the code, we ran it with increasing amount of pairs of large integers to be multiplied. We recorded the completion times of the code both including the memory allocation and the raw multiplication only. After running each of these through 100 iterations, we took the average and used those values to present our data.

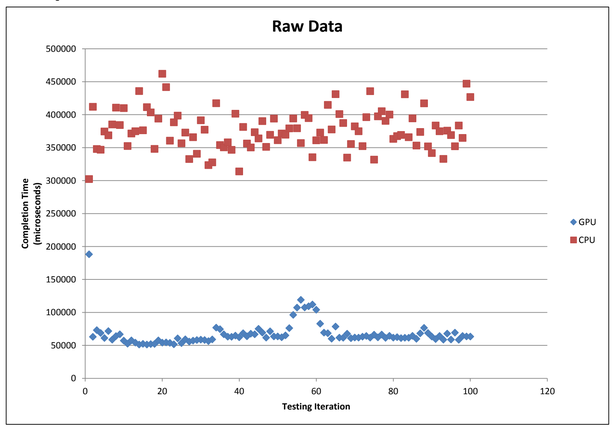
**Graph 1**



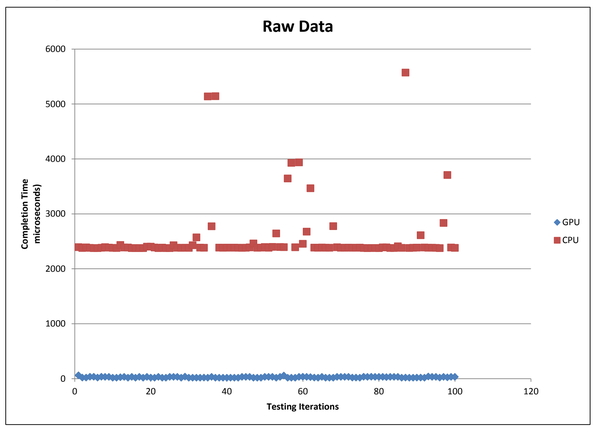

**Graph 2**




**Graph 3**




**Graph 4**




**Graph 1** – Completion time of GPU and CPU multiplication function vs the amount of pairs of large integers being multiplied. This analysis includes the time to allocate memory for the variables.

**Graph 2** – Completion time of GPU and CPU multiplication function vs the amount of pairs of large integers being multiplied. This analysis **only** includes the pure multiplication

**Graph 3** – Completion time of GPU and CPU multiplication function vs the testing iterations (1-100) for 1024 pair multiplication. This analysis includes the time to allocate memory for the variables.

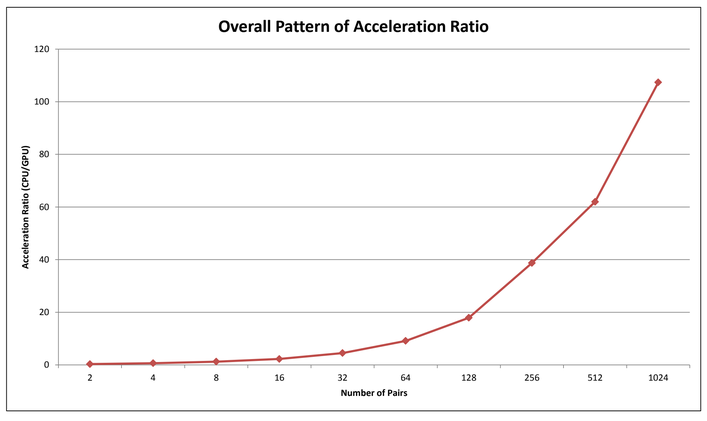
**Graph 4** – Completion time of GPU and CPU multiplication function vs the testing iterations (1-100) for 1024 pair multiplication. This analysis **only** includes the pure multiplication.

**Table 1 – Acceleration Ratios with Memory Allocation**

| Number of Pairs | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|---|
| Acceleration Ratio (CPU/GPU) | 0.00907 | 0.0185 | 0.03766 | 0.071104 | 0.16891 | 0.38217 | 0.83972 | 1.3595 | 2.73851 | 5.6007 |

**Table 2 – Acceleration Ratios with Multiplication Only**

| Number of Pairs | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|---|
| Acceleration Ratio (CPU/GPU) | 0.30992 | 0.63929 | 1.22975 | 2.26313 | 4.5042 | 9.13114 | 17.9094 | 38.7318 | 61.9682 | 107.3426 |




$T_{total}$: Total CPU time to run the Paillier encryption (microseconds)

Ratio: The average acceleration ratio of raw multiplication with 1024 pairs

$T_{mul}$: The CPU time to calculate one big integer multiplication with a 1024 bit key (microseconds)

m: the clear text length function

n: the modular length

Paillier Acceleration ratio = $$\frac{T_{total}}{T_{total}-\left(1-\frac{1}{Ratio}\right)*(m*4+n*4+1)*T_{mul}}$$

$$= \frac{104368000}{104368000-\left(1-\frac{1}{107.342}\right)*(1024*4+1024*4+1)*11554.88} = 9.88942$$

## Discussion

The performance of large integer arithmetic was shown to improve vastly via the GPU device code. The GPU performance is steady no matter the amount of pairs being multiplied because all of them run at the same time. In contrast, the completion time for the CPU BigInteger multiplication is shown to increase exponentially with the amount of pairs being multiplied. Because of this, the CPU functions out-perform the GPU function with smaller amounts of pairs, but at a certain point their completion times surpass the steady GPU completion times, and then continue to do so. This remains true when factoring in memory allocation time, although at a later rate. As can be seen in graphs 3 and 4, with large amounts of data the GPU function far outperforms the CPU function. This vast acceleration is also displayed when applying the multiplication acceleration ratio is applied to the overall runtime of Paillier encryption. By plugging these values in, we calculate an overall acceleration ratio of approximately 9.88942 for Paillier run with a 1024-bit key and a 1024-bit input length.

## Conclusion and Future Work

The results regarding large integer arithmetic were consistent with expectations. The more data being processed, the more efficient the GPU function was relative to the CPU function. This means that, theoretically, GPU programming can be very useful in implementations of algorithms that process large amounts of data with computationally intense functions. This remained apparent when calculating an overall acceleration ratio for Paillier encryption, which showed that the GPU-accelerated multiplication function could speed up the encryption by a factor of almost 10 for a 1024-bit key. I believe that further steps can be taken from here on out to optimize the use of this specific cryptosystem via GPU acceleration, such as the extensive key generation. Also, it could be useful to parallelize other mathematical functions of large integers, as they can be useful beyond encryption. GPU acceleration showed to be very useful when working with large amounts of data, and so its capabilities can be further explored in nearly any relevant field.

## Acknowledgments


I would like to thank my mentor, Dr. Tingting Chen, for all of her guidance; this project would not have been possible without her. I would also like to recognize Steve Jankly, who provided the Paillier encryption implementation and other supporting classes.