README

Özgün Özerk

June 2020

1 Description of The Problem And The Need For HPC

In this project, Fast Fourier Transform of Homomorphic Encryption will be optimized to run in parallel. Homomorphic Encryption is a special type of encryption where data manipulation, such as addition and multiplication can be calculated on the encrypted data. Say, an entity wants to send its data to a server but they are worried about data privacy. For that reason, encrypted version of that data is sent to server. Encrypted version of the data will be manipulated on server and the result will be sent in an encrypted format to entity. Result will be decrypted by the entity and the actual result will be received. During this whole cycle, server had no information about the plain data since it was encrypted, which made this process privacy preserving. Granting us this privacy, the reason of Homomorphic Encryption is not being very common at the moment is, it is still very slow compared to other encryption algorithms. The aim of this project is to increase it's speed with parallelism, via both reducing it's latency and increasing it's throughput.

The two algorithms we are focusing on are: NTT, and iNTT(inverse NTT). These two functions are sharing the same algorithmic structure and the complexity. The major difference between them are, their iterations in the loops and mathematical operations are reversed. Both of the algorithms will be parallelized. They are same from the parallelization aspect (their structure is the same), only one of them will be inspected and analyzed in this report to avoid being repetitive.

What is NTT?

Very briefly, NTT stands for Number Theoretic Transform. It's very similar to Fourier Transform. There are various versions of NTT, the version to be inspected and implemented in this project will be Kyber (N=256, q=7681 and N=1024, q=132120577). The usual bottleneck for encryption algorithms are (especially for the post-quantum ones) matrix/polynomial multiplications. NTT can be seen as a tool to ease this multiplication process. Mathematical details of NTT will not be discussed in here, since it's out of scope of parallelism. To put shortly, NTT is the bottleneck of Homomorphic Encryption, and without delving into details, it can be seen as a black box to be parallelized.

2 Description of The Solution and Comparison With Existing Work

The main-goal of this project is to accelerate the Homomorphic Encryption in GPU. For that, two aspects can be utilized: latency and throughput. GPU's structure is very convenient for handling throughput. This project consists of 2 steps (CPU and GPU parallelization). Since the ultimate goal will be the GPU parallelization, the first step (CPU part) can be seen as an intermediate step. It's been decided to inspect and improve latency in the CPU part solely, which fits the CPU structure better. And throughout the process, discovering the details of the algorithm. Then applying the gathered information on the GPU along with the throughput optimization.
There are some existing works on this issue, aiming the speed-up the fully Homomorphic Encryption with GPU:

- The AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data with GPUs
- Efficient GPGPU implementation of the leveled fully homomorphic encryption scheme YASHE

In the figure below (YASHE implementation), cuFFT took 0,3 ms. This result is worse than the current implementation exists in the CISEC group (by Can Elgezen), timing of which is 0,13. Additionally, NTT(ms) in CPU took 2,16 ms. In the sequential implementation of our code, it took only

Degree	cuFFT (ms)	NTT (ms)
1024	0,3	2,16
2048	0,53	2,06
4096	0,9	4,61
8192	1,71	9,05

Figure 1: YASHE implementation

0,323. These differences may due to the hardware specifications. In our implementation, we have managed to reduce NTT function in GPU to 0.036 ms. Detailed inspection and analysis of these performances will be done later in the report.

3 Description of The Model

There are specific parameters for different setups in homomorphic encryption. Explanation of the variables will be too mathematical. Instead, a very basic big-picture will be given in here.

- f: NTT will be applied on this array, this can be seen as the data to be encrypted. Elements of this array are randomly generated between 0 and q.
- N: size of the array that NTT will be applied on
- q: the modulus number
- zetas: an array (size of N), it's generated with a root(psi) with taking it's powers to N (with modulo q) in every step
- zetas_inverse: generated the same way with zetas, only difference being, with the inverse of N in modulo q, instead of N.

Given N, q, and root(psi), all of these variables are generated easily from the python document which will be supplied in the repository.
N, q, and root(psi) are taken from the CISEC group's SEAL Parameters (SEAL is the library for Homomorphic Encryption, supported by Microsoft)

4 System Specifications

CPU:
Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10 GHz **X32**
GPU:
Nvidia GeForce GTX 980 DDR5
RAM:
193335 MB
Operating System:
Ubuntu 16.04.6 LTS
Cuda Version:
10.0
GCC Version:
7.3.0
OpenMP Version:
4.5

5 Technical Description

5.1 Base code

Below is the initial python implementation of the NTT function:

```
length = N//2
    k = 1
    j = 0
    while length >= 1:
        start = 0
        while start < N:
            omega = zetas[k]
            k += 1
            for j in range(start, start+length):
                t = (omega*f[j+length])%q
                f[j+length] = (f[j] - t)%q
                f[j] = (f[j] + t)%q
            start = (j+1) + length
        length = length//2
```

5.2 Complexity of the Algorithm

The running cost of our sequential Encryption function is almost the same as sequential Decryption one. Going through the *sequential_ntt*(), the outer for() will iterate for $log_2 N$ times. The iteration number of code blocked inside first two outer for() loops is:

$$1 + 2 + 4 + \ldots + 2^{log_2(N)-1}$$

Two-inner loops together, will be doing N/2 times operation in total. So, the total complexity of our sequential encryption function is:

$$\in O(N/2 * log_2(N))$$

5.3 1st Step: CPU Parallelization (on Gandalf)

As first, the conversion from while loops to for loops have been done:

```
for(int length = N/2; length >= 2; length/= 2)
{
    for(int start = 0; start < N; start += length * 2){
        // ...
    }
    // ...
}
```

Although the implementation was correct, the parallelization of the above code did not give the correct results. The problem was emerged from the dependency on "k". It is being incremented in order, but the threads are being run in random order. For each thread, the value of "k" has to be determined in advance. Below is the c++ version of the whole code, with the rewritten for loops with removed dependencies:

```
for (int length = N / 2; length >= 2; length /= 2)
{
    int k_start = N / (length * 2);
    for (int i = 0; i < k_start; i++)
    {
        int omega = zetas[k_start + i];
        int f_start = i * length * 2;
        int f_end = f_start + length;

        for (int j = f_start; j < f_end; j++)
        {
            int t = (omega * f[j + length]) % q;
            f[j + length] = (f[j] - t + q) % q;
            f[j] = (f[j] + t) % q;
        }
    }
}
```

5.3.1 Analysis/Profiling of the Parallel Code

After the test with the given parameters above, it's ensured that the code is working as it supposed to be. For the CPU part, the focus was on the latency, being able to test our trials more efficiently, N is dramatically increased to

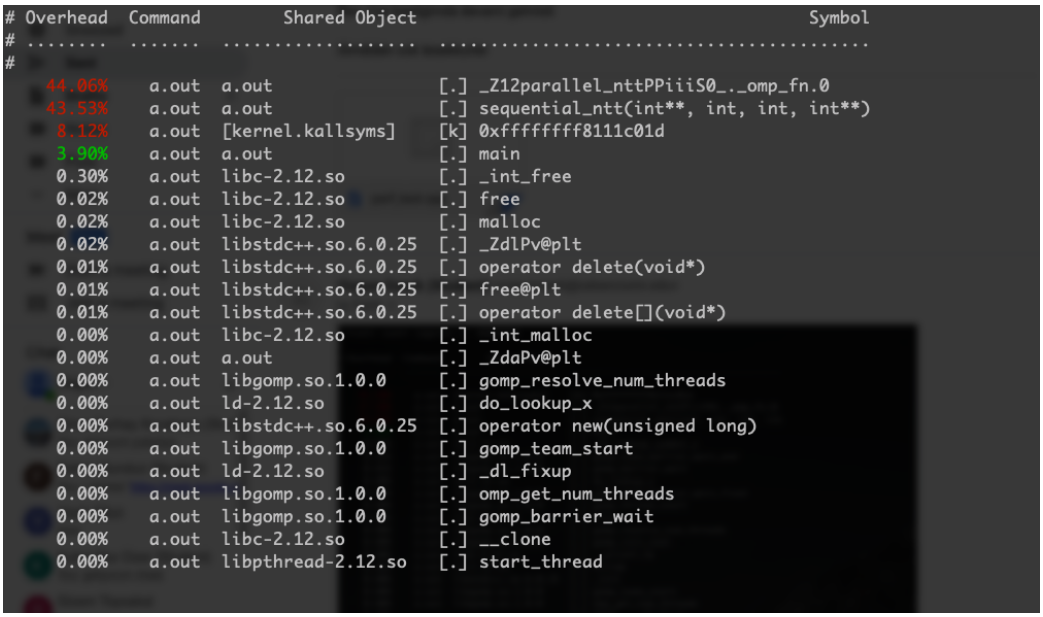
$$2^{25}$$

The code we are trying to parallelize is the NTT and inverse NTT algorithm snippets. Very briefly, they are consisting of 3 loops and some basic mathematical operations on an array. Without complicating the things too much, and reducing the overhead via keeping the things as simple as possible, "#pragma omp parallel for" have been tried on the all applicable loops. The most-outer for loop cannot be parallelized, since the steps are dependent upon the previous step, and if parallelized, resulting in wrong answers. Also, parallelization of the most-inner loop reduced the speed gain, simply because the speed gain was not bigger than the overhead it's creating. Below is the profiling of the most efficient parallel version (a more detailed inspection will be discussed below for the main overhead detection. For the sake of not being repetitive, it's not replicated in here).

```
# Event count (approx.): 56859846
#
# Overhead  Command        Shared Object
# ........  .......  ...................  ....................................
#
    45.50%    a.out  [kernel.kallsyms]    [k] 0xffffffff8111c01d
    27.50%    a.out  a.out                [.] _Z12parallel_nttPiiiPKi._omp_fn.0
    24.95%    a.out  a.out                [.] sequential_ntt(int*, int, int, int
     2.01%    a.out  a.out                [.] main
     0.02%    a.out  ld-2.12.so           [.] _dl_lookup_symbol_x
     0.01%    a.out  libgomp.so.1.0.0     [.] gomp_team_barrier_wait_end
     0.01%    a.out  libgomp.so.1.0.0     [.] gomp_barrier_wait
     0.00%    a.out  ld-2.12.so           [.] do_lookup_x
     0.00%    a.out  libgomp.so.1.0.0     [.] gomp_team_barrier_wait_final
     0.00%    a.out  libgomp.so.1.0.0     [.] gomp_thread_start
     0.00%    a.out  libc-2.12.so         [.] free
     0.00%    a.out  libgomp.so.1.0.0     [.] gomp_resolve_num_threads
     0.00%    a.out  libgomp.so.1.0.0     [.] gomp_init_task
     0.00%    a.out  libc-2.12.so         [.] __printf_fp
     0.00%    a.out  ld-2.12.so           [.] strcmp
     0.00%    a.out  libstdc++.so.6.0.25  [.] _init
     0.00%    a.out  libgomp.so.1.0.0     [.] gomp_team_start
     0.00%    a.out  libgomp.so.1.0.0     [.] omp_get_num_threads
     0.00%    a.out  a.out                [.] GOMP_parallel@plt
:
```

Figure 2: Profiling of the best version

And below is the profiling of the version with the padding (size 4). The reasoning for choice of padding will be discussed below.



# Overhead	Command	Shared Object	Symbol
44.06%	a.out	a.out	[.] _Z12parallel_nttPPiiiS0_._omp_fn.0
43.53%	a.out	a.out	[.] sequential_ntt(int**, int, int, int**)
8.12%	a.out	[kernel.kallsyms]	[k] 0xffffffff8111c01d
3.90%	a.out	a.out	[.] main
0.30%	a.out	libc-2.12.so	[.] _int_free
0.02%	a.out	libc-2.12.so	[.] free
0.02%	a.out	libc-2.12.so	[.] malloc
0.02%	a.out	libstdc++.so.6.0.25	[.] _ZdlPv@plt
0.01%	a.out	libstdc++.so.6.0.25	[.] operator delete(void*)
0.01%	a.out	libstdc++.so.6.0.25	[.] free@plt
0.01%	a.out	libstdc++.so.6.0.25	[.] operator delete[](void*)
0.00%	a.out	libc-2.12.so	[.] _int_malloc
0.00%	a.out	a.out	[.] _ZdaPv@plt
0.00%	a.out	libgomp.so.1.0.0	[.] gomp_resolve_num_threads
0.00%	a.out	ld-2.12.so	[.] do_lookup_x
0.00%	a.out	libstdc++.so.6.0.25	[.] operator new(unsigned long)
0.00%	a.out	libgomp.so.1.0.0	[.] gomp_team_start
0.00%	a.out	ld-2.12.so	[.] _dl_fixup
0.00%	a.out	libgomp.so.1.0.0	[.] omp_get_num_threads
0.00%	a.out	libgomp.so.1.0.0	[.] gomp_barrier_wait
0.00%	a.out	libc-2.12.so	[.] __clone
0.00%	a.out	libpthread-2.12.so	[.] start_thread

Figure 3: Profiling of the version with the padding (size 4)

5.3.2 Main Overheads

In the code we are trying to parallelize, there are 3 loops. As the first step, the while loops have been converted to for loops. Expectedly, this was not enough for us to start parallelization. After the dependencies have been discovered, the for loops have been rewritten with the removal of dependencies. Although, the dependency in the first (most-outer) for loop couldn't have been eliminated. The steps of this for loop are dependent upon the previous iterations. Hence, sequential sections of this code are creating some overhead. To mitigate this disadvantage, the other loops that can be parallelized have been intensely focused for parallelization purposes.

When the terminal command "perf" utilized, it came to light that, the current best version has 78% cache-miss rate. To reduce our cache-miss rate, **Padding** was the first thing to come our minds.

Here is the more detailed summary of the performance of the current best version:

- Sequential: 26.9853 seconds
- Parallel: 7.2796 seconds
- Speed-up: 3.70698
- Efficiency for 8 threads: 0.463373
- 46079.215730 task-clock - 1.660 CPUs utilized
- 124192126282 cycles - 2.695 GHz
- 69609437860 instructions - 0.56 insns per cycle
- 74234507 cache-references - 1.611 M/sec
- 58023009 cache-misses - 78.162 % of all cache refs

5.3.3 Padding

To reduce cache-miss rate, we have tried multiple padding sizes. Here are the cache-miss rates for some different padding sizes:

Padding Size	2	4	8	16	32
Cache-miss%	49	44	81	92	98

Table 1: Padding Size Comparison

```
void parallel_ntt(int* f, int N, int q, int* zetas)
{
    for (int length = N / 2; length >= 2; length /= 2)
    {
        int k_start = N / (length * 2);

#pragma omp parallel for default(shared)
        for (int i = 0; i < k_start; i++)
```

```
9           {
10              int omega = zetas[k_start + i][0];
11              int f_start = i * length * 2;
12              int f_end = f_start + length;
13
14              for (int j = f_start; j < f_end; j++)
15              {
16                  int t = (omega * f[j + length][0]) % q;
17                  f[j + length][0] = (f[j][0] - t + q) % q;
18                  f[j][0] = (f[j][0] + t) % q;
19              }
20          }
21      }
22  }
```

Although we could achieve lower cache-miss rates, we did not achieve more speed. Below is the detailed report for padding with size 4:

- Sequential: 48.9622 seconds
- Parallel: 22.7328 seconds
- Speed-up: 2.15382
- Efficiency for 8 threads: 0.269227
- 235567.567218 task-clock - 2.222 CPUs utilized
- 634893271597 cycles - 2.695 GHz
- 210688973272 instructions - 0.33 insns per cycle
- 6211325711 cache-references - 26.367 M/sec
- 2998526540 cache-misses - 48.275 % of all cache refs

To summarize: None of the padding versions granted us a better result than the no-padding version. Although we have 30% less cache-misses, we are referencing the cache 26 times more. Padding is not reducing the main over-head, quite the contrary, making it harder to access our data by complicating the data-structure and creating a massive over-head to our performance.

5.3.4 Task

With the release of OpenMP 3.0, concept of Tasking was introduced. Since the parallelization of the most-inner loop with simple "#pramga omp for" was giving more overhead than it's speed gain, we wanted to try if Tasks will be a better candidate for this purpose. The computation in the inner loop will be assigned to a thread by creating a task. The code below represents the first experiment with tasks.

```
1  void process(int *f, int q, int omega, int length, int i)
2  {
3      int f_start = i * length * 2;
4      int f_end = f_start + length;
5
6      for (int j = f_start; j < f_end; j++)
7      {
8          int t = (omega * f[j + length]) % q;
9          f[j + length] = (f[j] - t + q) % q;
10         f[j] = (f[j] + t) % q;
11     }
12 }
13
14 void parallel_task_ntt(int *f, int N, int q, int *zetas)
15 {
16     for (int length = N / 2; length >= 1; length /= 2)
17     {
18         int k_start = N / (length * 2);
19
20 #pragma omp parallel
21 #pragma omp single
22         {
23             for (int i = 0; i < k_start; i++)
24             {
25 #pragma omp task
26                 process(
27                     f, q, zetas[k_start + i],
28                     k_start, length, i
29                 );
30 }   }   }   }
```

Tasking can be useful in some cases, however task creation has an overhead. This design creates task for every loop iteration. As a result, millions of task will be created in large inputs. Due the huge overhead with millions of tasks, this approach performs very poor.

- Sequential: 26.9853 seconds
- Parallel Task: 823.022 seconds
- Speed-up: -
- Efficiency for Task 8 threads: 0.0034197

In order to reduce task creation overhead, task creation can be reduced. In the code below, a new task is created if variable *k_start* is greater than or equal to variable *length*.

```
void parallel_task_ntt(int *f, int N, int q, int *zetas){
    for (int length = N / 2; length >= 1; length /= 2){
        int k_start = N / (length * 2);

#pragma omp parallel
#pragma omp single
        {
            for (int i = 0; i < k_start; i++){
                if (k_start >= length){
#pragma omp task
                    process(
                        f, q, zetas[k_start + i],
                        k_start, length, i
                    );
                } else {
                    process(
                        f, q, zetas[k_start + i],
                        k_start, length, i
                    );
                }
            }
}   }   }
```

In comparison to first task design, this implementation works faster. Although, it is still slower than the sequential implementation.

- Sequential: 26.9853 seconds
- Parallel Task: 718.184 seconds
- Speed-up: 0.0313719047
- Efficiency for Task 8 threads: 0.00392148

Implementation above has a major drawback. Task creation is limited, however when *k_start* is large, there will be tasks with very large size. To overcome this drawback, a new task is created if variable *k_start* is less than or equal to variable *length*.

```
void parallel_task_ntt(int *f, int N, int q, int *zetas) {
    for (int length = N / 2; length >= 1; length /= 2) {
        int k_start = N / (length * 2);

#pragma omp parallel
#pragma omp single
        {
            for (int i = 0; i < k_start; i++){
                if (k_start <= length){
#pragma omp task
                    process(
                        f, q, zetas[k_start + i],
                        k_start, length, i
                    );
                } else {
                    process(
                        f, q, zetas[k_start + i],
                        k_start, length, i
                    );
                }
            }
}   }   }
```

In this implementation, task sizes will be small. Threads will complete tasks faster and task queue will be consumed faster in comparison to first two implementations.

- Sequential: 26.9853 seconds
- Parallel Task: 13.086 seconds
- Speed-up: 1.72184
- Efficiency for Task 8 threads: 0.21523

5.3.5 Pre-fetching

We also wanted to know how well we will do with prefetching, although prefetching is not something that will boost your performance two or three times. In the best case scenarios and where prefetching seems to logical it can give 5-20 percents performance boost. In our case it didn't help us much. On the contrary it made us a little bit slower. The reason for that as we predict is the accessing of array from multiple points although we also tried to prefetch from different portions of arrays. The other reason that we think may result in slowing is the overhead of computation. What do we mean by that is that iteration space and prefetch size may not be a multiple of prefetch so we have to make sure that we don't go beyond the bounds. Also some loop may have very low or zero iterations. All these means extra computation overheads which seems to be the slowing us. So lets make this clear with time performances.

- Sequential Without Prefetching: 2.39 seconds
- Sequential With Prefetching: 2.51 seconds

5.3.6 Vectorization and SIMD

We are dealing with a single array. Computations on that array is usually done with other integers or the other indices of that very array. Also, there is no multiplication/division/addition/subtraction operation that applied directly on the array that can be vectorized. Again, usage of a single array being the most important reason, vectorization and SIMD techniques seemed irrelevant and have not been tried.

5.3.7 Scheduling

The design of our for loops is favorable for static scheduling. Workload is balanced. Different Chunk Sizes are tried to see if they are effective on our performance. Below are the results:

Chunk Size	Sequential	Parallel	Speed-up	Efficiency
No Chunks	28.1942s	4.5201s	4.991	0.623
1	26.5342s	5.8241s	3.864	0.483
2	26.2482s	6.4234s	3.500	0.437
4	27.5316s	7.2122s	3.116	0.389
8	26.9834s	8.0605s	2.788	0.348
16	28.480s	9.0765s	2.476	0.309
32	27.474s	9.8714s	2.276	0.284

Table 2: Chunk Size Comparison

	Sequential	Parallel	Speed-up	Efficiency
dynamic	26.2834s	18.0559s	1.2467	0.1558375
guided	26.5342s	4.5368s	4.94828	0.618535
auto	26.9834s	4.54829s	4.95563	0.619453

Table 3: Schedule Comparison

As expected, best performance was obtained when static is used with no chunk size was defined.

Estimated theoretical speed-up and scalability

# Threads	Sequential	Parallel	Speed-up	Efficiency
2	28.1942	16.2452	1.73554	0.867772
4	26.2471	10.555	2.48671	0.621677
8	26.9853	7.2796	3.70698	0.463373
16	28.5408	5.36437	5.32043	0.332527
32	28.7926	4.75673	6.05301	0.189157

Table 4: # Threads Comparison for N = 134217728, q = 23

Note that, N = 134217728, q = 23, and final degree=2 scenario is created for parallel efficiency testing only. One should keep in mind that these values for N and q are not applicable to real-life application, since CPU part was an intermediate step, they are picked just to discover our algorithm's scalability. And all of the testings above related to the CPU, are done with these parameters.

Our parallelization's efficiency drops significantly after 16 threads. At that point, the overhead becomes greater than the speed gain. Between thread amount 1, 2 and 4, the parallelism seems very scalable, but this scalability starts to drop from 4 threads. Especially between 8 to 16, although the thread amount doubles, the run-time shortens by 1/4.

Our estimation is, up to 8 threads, the performance gain will be beneficial (around 5 times faster compared to sequential), and it will be optimal around 4 threads with respect to scalability (3.2 times faster compared to sequential). This estimation will be more successful and noticeable on bigger inputs.

5.4 GPU Parallelization (on Nebula)

5.4.1 Code Design

Whilst trying to decrease the latency of the code in the CPU Part, many things have been considered as can be seen above. This process helped us a lot to see what's in front of us, and how we should deal with it. It's been discovered that, for the every iteration of the most-outer for loop, all elements of the zetas and f array are traversed. In GPU, we can afford seperating a thread for each element of f array.

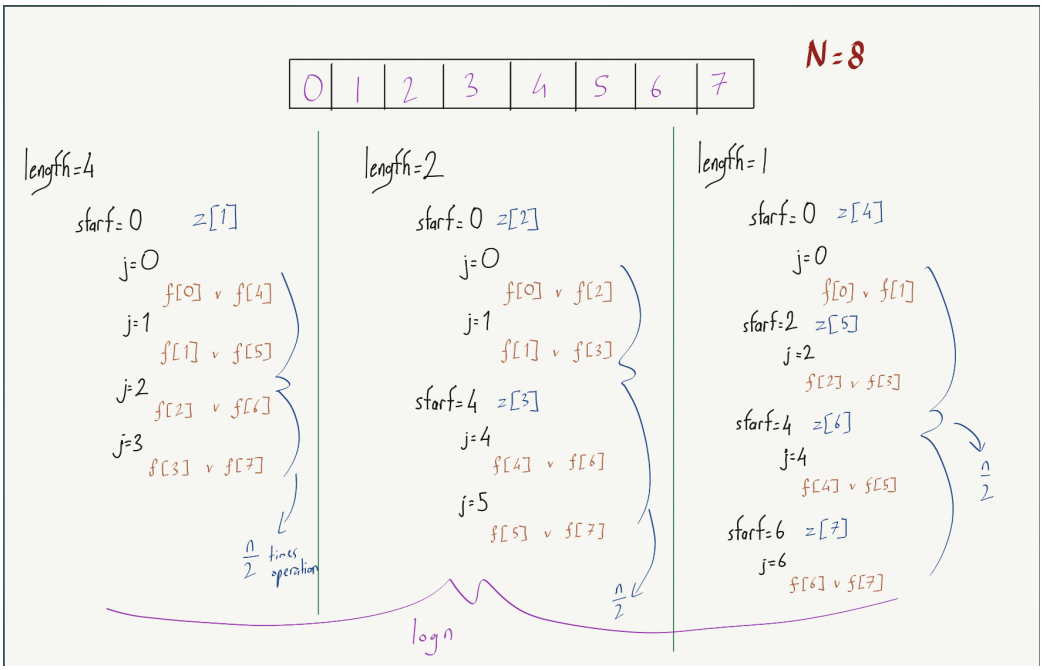


Figure 4: Demonstration of the algorithm as a sketch

This sketch is a very basic demonstration of what's going on. for example, "f[0] v f[1]" can be interpreted as a single code line, in which f[0] and f[1] will be read or altered. By assigning this code line to a thread, we can eliminate the inner for loops, thus only left with a single loop instead of 3.
For this, the mathematical pattern should be derived from the above demonstration.

```
1 if((thread_idx % length*2) < length)
```

Condition handles this mathematical pattern successfully. With the help of this condition, the NTT function can be rewritten in GPU like below:

```
for (int length = N / 2; length >= 1; length /= 2)
{
    int a = length * 2;
    if ((local_idx % a) < length)
    {
        int omega = shared_zetas[(N/a) + (local_idx/a)];
        int t = (((omega * shared_f[local_idx + length]) % q) +
        ↪  q) % q;
        shared_f[local_idx + length] = (((shared_f[local_idx] -
        ↪  t + q) % q) + q) % q;
        shared_f[local_idx] = (((shared_f[local_idx] + t) % q)
        ↪  + q) % q;
    }
    __syncthreads();
}
```

5.4.2 Complexity of the GPU Code

Most-outer loop was not altered, thus it's complexity is still the same

$$1 + 2 + 4 + \ldots + 2^{log_2(N)-1}$$

Two-inner loops together, will be doing N/2 times operation in total. But they are eliminated via assigning their job to threads, which will run at the same time, reducing their complexity to O(1). Which is (N/2) times faster than the previous one. The updated complexity is:

$$\in O(log_2(N))$$

5.4.3 Optimizations

To see if it's working or not, parameters of N=256 and q=7681 have been tried as in the CPU part. After ensuring everything was working as intended, a bigger instance have been selected, N=1024, q=132120577. The reason of N being 1024 is, it is the size of f array, which we are distributing to the threads. For better performance, utilization of shared memory is important,

and it can be realized only inside a block. Considering there can be only 1024 threads per block, an f array with 1024 elements is the upper-limit for this design per block.

In sum, this design can handle f array's to size 1024, with the complexity of logn. In addition to that, with the possibilities of GPU, it may be possible to work on more than one f array at a time. There is no dependency or shared variables between the blocks, this is a great benefit not be despised.

5.4.4 Performance Analysis

Array Amount	1	2	4	8	16	32
GPU	0.037	0.026	0.025	0.025	0.025	0.035

Array Amount	64	128	256	512	1024	2048
GPU	0.047	0.070	0.129	0.233	0.444	0.870

Array Amount	4096	8192	16384	32768	65536	
GPU	1.724	3.387	6.745	13.450	26.870	

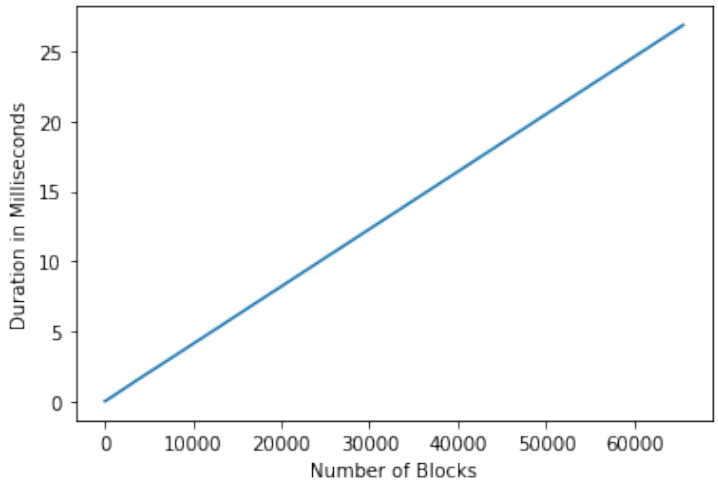


Figure 5: Blocknum = 1, 2, 4, 8 ... , 65536

Although it's known that the line is not linear (in the beginning, there is a slight decrease, then increase), the graph represented in here seems to be perfectly linear. This is not abnormal. This decrease happens between the values of Blocknum = 1 and 32. The interval of 1-32 not being visible on a scale of 1-65536 is not surprising (also consider there is only 6 data points for 1-32 interval). While the number of blocks being able to run in parallel can be large, it is still finite due to limited on-chip resources. If the number of blocks requested in a kernel launch exceeds that limit, any further blocks have to wait for earlier blocks to finish and free up their resources. However, the above graph is consistent with these information, it did not satisfy us, we wanted to dig deeper. Since at some point, there should have been a parallelization visible.

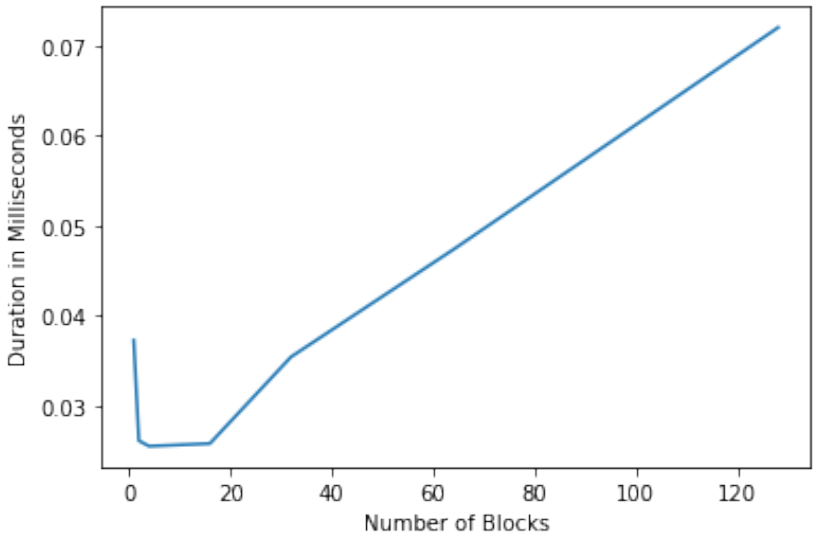


Figure 6: Blocknum = 1, 2, 4, 8 ... , 128

Now, the decrease at the beginning is visible. But instead of trying with the multitudes of 2, all discrete values have been tried in below:

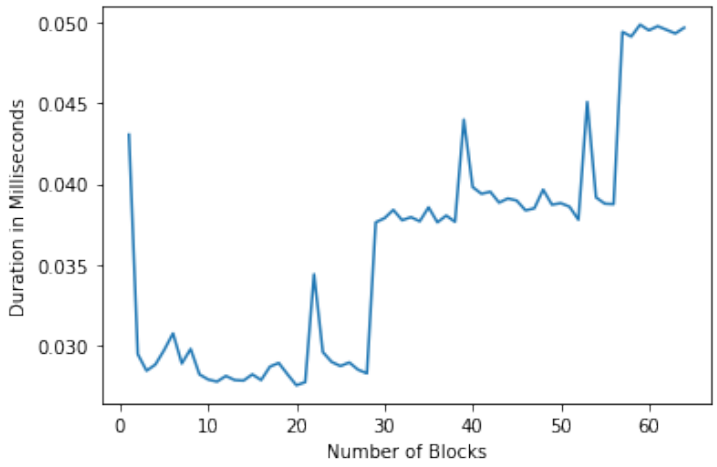


Figure 7: Blocknum = 1, 2, 3, 4 ... , 64

It was surprising to us that the plot is very much alike with a cardiac rhythm graph. The most remarkable highlight of this plot could be, the drastic increase points like Blocknum = 28, 56. From these points, the time increases permanently (after Blocknum = 28, it will always be bigger than when Blocknum = 28, same applies to 56).
We were curious about observing the same behaviour on a bigger scale:

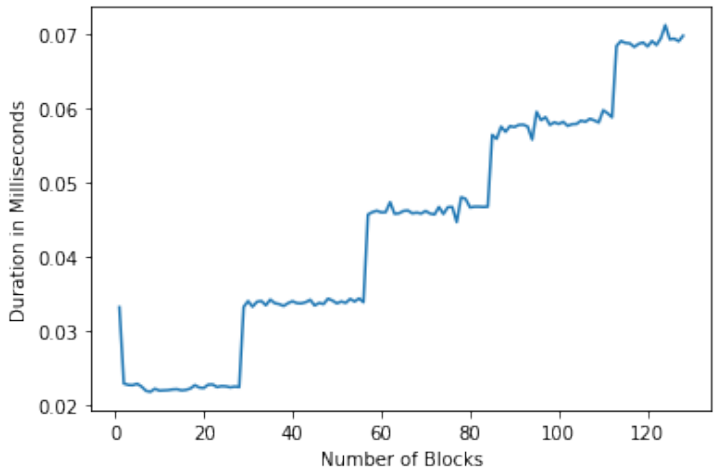


Figure 8: Blocknum = 1, 2, 3, 4 ... , 128

This time, the behaviour of the plot became more clear, it's can be liken to a stairway model. Again, all of the drastic increase points are multitudes of 28. In the code design, we couldn't catch something related to 28. That led us to turn our search into the hardware rather than the software. Inspecting the specifications of Nvidia GTX 980, we found out that it's bandwith is 224 GB/s, and 224 is 8x28. Remembering the parallelization amount related to the block number and limited GPU resources, it can be clearly seen that till 28 arrays, the code is fully parallelized, and increasing the block number (array amount) has no effect on the timing.

Comparison with sequential CPU

# Array	1	2	4	8	16	32
CPU(ms)	0.323	0.646	1.292	2.584	5.168	10.336
GPU(ms)	0.037	0.026	0.025	0.025	0.025	0.035
Speedup	8.671	24.769	50.722	101.06	200.62	292.04

# Array	64	128	256	512	1024	2048
CPU(ms)	20.672	41.344	82.688	165.37	330.75	661.50
GPU(ms)	0.047	0.070	0.129	0.233	0.444	0.870
Speedup	436.78	584.35	637.23	708.43	744.56	759.66

# Array	4096	8192	16384	32768	65536	
CPU(ms)	1323.0	2646.0	5292.0	10584	21168	
GPU(ms)	1.724	3.387	6.745	13.450	26.870	
Speedup	767.07	781.01	784.57	786.89	787.79	

The latency speedup can be observed at place where "Array Amount = 1", it is 8.67 for N=1024 and q=132120577. This improvement is definitely because of the new design which reduced the complexity of the problem.

As Array Amount grows bigger, it can be said that the speedup roughly converges into 800. There are few things that should be mentioned in this part. The obvious thing is, GPU can handle throughput fairly well, speedup

increase with more arrays is expected. Although, the second derivative of this speedup increment is negative (it's growth is getting smaller, and finally, it converges).

One limited resource is shared memory. CUDA 8.0 compatible Nvidia GPUs offer between 48 and 112 kilobytes of shared memory per streaming multiprocessor (SM). Other limited resources are registers and various per-warp resources in the scheduler. In our case, GTX 980 has 16 SMs with 96Kb of shared memory each, so at most 48 (16 x 3) blocks can run in parallel. Which explains in a great detail why GPU time did not increase till 32, and increased only a little between 32-64.

Comparison with parallel CPU

The parallelization of the sequential code for latency did not help much for small N's, which are the real-life application values that are going to use. Instead of latency, CPU parallelization can be used for throughput as well. Considering GPU version had achieved around 800 times speedup versus the sequential code, it would require roughly 800 cored CPU to reach that performance, which is impractical.

6 Conclusion and Insights

By parallelizing the Homomorphic Encryption's bottleneck (NTT) using HPC, we have achieved around 800 times speedup compared to the sequential CPU code design. The same level of speedup couldn't have been achieved with the CPU-Parallelization. In the end, the function to be implemented is not very complex, there is no recursion or if-else conditions, it consists of basic mathematic operations only. Although modulus operation is a costly one, it's not that hard for GPU cores. Considering the core amount difference between the CPU and GPU, this huge performance difference makes more sense. CPU cores are basically becoming an overkill for such task. Also, because of the fewer core count on CPU, CPU parallelization performance suffers more from the cache-misses.

Figure 9: ?

The length of CPU and GPU parts in this report might be misleading. Debugging process of GPU code was also very challenging. To our experience, CPU parallel debugging is a lot harder than normal debugging, and GPU code debugging is that much harder than the CPU parallel debugging. These previous experiences we have gathered from the CPU parallelization taught us to delve even in the smallest details of our code when designing it. If not, much more time is going to be wasted during the debugging. The sketch demonstration of the algorithm shared in GPU part, was the milestone for this project. In every detail we got lost in, we referred to it, and it cleared the fog on our problems, acting as a mind-map.

During performance measurement, another difficulty came to our path. At first, we thought our design is slower than the sequential CPU design, however, this was because we have misinterpreted the results. Timings of both CPU and GPU code are way smaller than seconds. The fact that we are not very accustomed to the micro/milliseconds, caused us to do false evaluations on the results. Correction of this mistake took 1.5 days for us. Although this was a very minor mistake, it's a great indicator that our attention and perception have depleted throughout the process.

Depending on the problem type to be dealt with, HPC might not grant us the best results. For example, if the problem consists of only sequential steps, it could be impossible to parallelize it, or if it includes some independent tasks, but only a few of them, CPU core count might be sufficient. If more parallelization than CPU core amount is possible, than GPU can be utilized better than CPU with the right code design.

For this, the architecture of the problem should be analyzed well, and a similar abstraction should be applied to the GPU by using grids-blocks-threads. To provide a concrete example, in this project, we were dealing with arrays that are 1 dimensional. But for the throughput, there could be many arrays to be processed. You can refer to the codes to verify that grids have not been used. The reason for that is, our design is 2 dimensional. 1st dimension being the length or arrays, second dimension being the amount of arrays. We already have threads in blocks, which can be interpreted as the elements of the arrays. Also, we can provide as much blocks as the amount of arrays. Including grids in this design, would just complicate the things for no good avail. Similarly, we could have utilized 3 dimensional placement

of threads in a block, but we didn't, because it was not needed. If our arrays were 3 dimensional arrays, that could have made sense. Considering the hardships of HPC, keeping the design as simple as possible, might be a highly preferable approach.

7 Future Work

There are still much to do for this project. For example, zetas array can be generated in the GPU, making the memory transfer to the GPU even less, also it will be faster than CPU zetas generation. Registers in GPU may be utilized better. Modulo operation being the bottleneck in this example, Barret function can replace the modulo, providing 20% faster modulo operation. Also, for 190-bit security, bigger numbers would be the subject. For operations on these big numbers, maybe new classes can be written with operation overloads.

Also, in C Plus Plus:

$$-5\%3 = -2$$

Although this is correct, the convention is

$$-5\%3 = 1$$

In this application, because of CPP's modulo behaviour, we had to apply modulo 2 times, which is also taking extra time. Application of Barret instead of modulo might also surpass that handicap. Further more, instead of sending constant variables (N and q) as parameters, they can be global variables for both CPU and GPU.

Last but not least, the code should be parameterized for different values of N. At the moment, the code design restricts us to work with N's smaller than 1025 (per block, at most 1024 threads are possible). Some slight differences on the design may increase the flexibility of this code, allowing it to process any number of N (with some cost to the efficiency). The goal when thinking on this new design should be, minimizing the penalty of cost to the speedup for big numbers of N.

8 How To Run

8.1 CPU code

Open a terminal in the respective folder, enter the command "gcc -fopenmp filename.cpp -lstdc++", after the compilation, enter the command "./a.out".

Run under gcc version 8

8.2 GPU Code

Open a terminal in the respective folder, enter the command "nvcc gpu_ntt.cu", after the compilation, enter the command "./a.out".

Run under cuda version 10

References

[1] P. Alves, D. Aranha, Efficient GPGPU implementation of the leveled fully homomorphic encryption scheme YASHE.

[2] Ahmad Al Badawi, Jin Chao, Jie Lin, Chan Fook Mun, Sim Jun Jie, Benjamin Hong Meng Tan, Xiao Nan, Khin Mi Mi Aung, and Vijay Ramaseshan Chandrasekhar. *The alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus* arXiv preprint arXiv:1811.00778, 2018.

[3] Kamara S., Raykova M. (2013) *Parallel Homomorphic Encryption*. In: Adams A.A., Brenner M., Smith M. (eds) Financial Cryptography and Data Security. FC 2013. Lecture Notes in Computer Science, vol 7862. Springer, Berlin, Heidelberg

[4] Hayward, R. and Chiang, C.-C. 2015. *Parallelizing fully homomorphic encryption for a cloud environment*. Journal of Applied Research and Technology. 13, 2 (Apr. 2015).

[5] https://developer.nvidia.com/hpc