CMSC 32900 Final Project Paper

Tina Oberoi (toberoi) and Nishchay Karle(nishchaykarle)

1 Introduction

Quantum computing has gained significant attention due to its potential to accelerate computing speed. It is based on principles of quantum theory such as superposition, entanglement, and interference, and aims to create computer technology that can accept input states representing multiple possible inputs and transform them into a corresponding output superposition. Quantum computation can impact every element of the superposition simultaneously, resulting in significant parallel data processing capacity even within a single piece of quantum hardware. This capability can bring substantial improvements in computational efficiency, including Shor's quantum algorithm for factoring large integers, Grover's algorithm for speeding up combinatorial searches, and quantum cryptography for secure communication. Therefore, quantum computing can effectively tackle some problems that classical computers struggle with, such as factorization.

Despite the potential of quantum computing, building a functional quantum computer is challenging due to the limitations of quantum hardware. To overcome these challenges, quantum computer simulation plays a crucial role in testing and exploring quantum computing. However, simulating quantum systems on classical computers can never replicate the full capabilities of a real physical quantum computer in polynomial time. To address this issue, the development of quantum computer simulators aims to make them faster, more reliable, capable of handling multiple qubits, and able to simulate quantum algorithms with higher performance. Typically, quantum computers are utilized as a coprocessor, similar to GPUs, to solve specific algorithms and return results to a host CPU. The initial quantum simulator was a sequential processor, which was later succeeded by parallel processors. The inaugural parallel quantum computing simulator was developed to function on a distributed parallel memory system consisting of 256 processors and utilized a message passing interface (MPI). This review article examines effective techniques for utilizing GPU-based parallelism to hasten quantum computer simulation.

In this work, we implement a simulator for Shor's quantum algorithm on GPUs to compute the prime factors of a few integers, utilizing the inherent parallelism of quantum system simulation that makes it well-suited for GPU-based implementations. We discuss effective methods for exploiting GPU-based parallelism to accelerate quantum computer simulation.

2 Shor's Algoirtm

2.1 Quantum Fourier Transform

The Quantum Fourier Transform (QFT) is a quantum algorithm that is used to perform a Fourier transform on quantum states. It is a quantum analog of the classical discrete Fourier transform, which is used to transform a sequence of complex numbers into another sequence of complex numbers.

The QFT is used in many quantum algorithms, including Shor's algorithm for factoring large numbers, and Grover's algorithm for searching an unsorted database. It is also used in quantum simulation and quantum chemistry.

The QFT takes as input a quantum state consisting of n qubits, where n is the number of qubits needed to represent the data to be transformed. The output is also a quantum state consisting of n qubits, which encodes the Fourier transform of the input state.

The Quantum Fourier Transform (QFT) is a unitary transformation that acts on a quantum state $|\psi\rangle$ consisting of n qubits, each of which can be in the state $|0\rangle$ or $|1\rangle$. The QFT maps this state to another state $|\phi\rangle$, which is a superposition of all possible states of n qubits.

The QFT can be defined as follows:

$$|\phi\rangle = 1/\sqrt{2^n} \sum_{k=0}^{2^n-1} e^{2\pi i \frac{mk}{2^n}} |k\rangle \qquad (1)$$

where m is an integer that determines the input state $|\psi\rangle$.

In this expression, $|k\rangle$ represents the binary representation of the integer k using n bits, and $e^{2\pi i \frac{mk}{2^n}}$ is a phase factor that depends on m and k.

The QFT can also be written in matrix form as:

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{2^n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(2^n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{2^n-1} & \omega_n^{2(2^n-1)} & \cdots & \omega_n^{(2^n-1)^2} \end{bmatrix}$$

where $\omega_n = e^{2\pi i/2^n}$ is an nth root of unity.

2.2 Algorithm

The key mathematical concepts involved:

- Modular arithmetic: In modular arithmetic, we perform operations (such as addition and multiplication) on integers using a modulus, which is another integer. For example, in modulo N arithmetic, we perform arithmetic operations on integers modulo N. Modular arithmetic is used extensively in Shor's algorithm to compute the modular exponentiation function $f(y) = a^y \bmod N$.

- Period finding: The key step in Shor's algorithm is to find the period of the function $f(y) = a^y \bmod N$, which is a periodic function. The period of the function is a positive integer r such that $a^r \bmod N = 1$. If we can efficiently compute the period of the function, then we can use it to factor N.

- Quantum Fourier Transform (QFT): The QFT is a quantum version of the classical discrete Fourier transform, which is used to transform a discrete sequence of numbers into a set of complex coefficients. The QFT is used in Shor's algorithm to efficiently compute the period of the function $f(y) = a^y \bmod N$. The QFT is implemented using a set of quantum gates, including Hadamard gates and controlled phase gates.

The steps involved in Shor's algorithm, explained in terms of the mathematical concepts listed above:

1. Choose a large composite integer N that needs to be factored, and choose a random integer a between 1 and $N-1$.

2. Determine the greatest common divisor (GCD) of a and N. If the GCD is greater than 1, then we have found a nontrivial factor of N, and we can stop the algorithm.

3. Create a quantum superposition of all possible values of x, where x is the unknown period of the function $f(y) = a^y \bmod N$. This can be done by applying the Hadamard transform to a set of n qubits, where n is the number of bits needed to represent N.

4. Apply the modular exponentiation function $f(y) = a^y \bmod N$ to the superposition of qubits, which creates an entangled state that depends on the unknown period x. This step involves performing modular arithmetic to compute the value of $a^y \bmod N$ for each value of y in the superposition.

5. Apply the QFT to the entangled state to efficiently compute the period of the function $f(y) = a^y \bmod N$. The QFT converts the superposition of values of y into a superposition of values of the period r, which can be measured to obtain an approximation of the period.

6. Use the value of r to determine the factors of N. If r is even, then we can compute $a^{r/2} \pm 1 \bmod N$, which gives the factors of N with high probability. This step uses the mathematical concept of modular arithmetic to compute the factors of N from the period r.

3 Approach to accelerating Shor's Factorization Algorithm on GPUs

1. Accelerating Shor's factorization algorithm on GPUs (Savran, 2018)

   In this paper, it is assumed that in typical implementations of Shor's algorithm, almost 97% of the processing time is spent on calculating the Quantum Fourier Transform (QFT). To address this, a GPU kernel function

was developed to perform QFT computations from the ground up. The quantum system's evolution-related calculations are performed by numerous threads inside the GPU. Each thread is responsible for computing an individual element of the output vector. This strategy ensures that threads can access memory coherently, improving the overall performance of the algorithm.

2. Exploiting GPU-based Parallelism for Quantum Computer Simulation: A Survey (Sengthai Heng, 2020)

   In this paper, it explains that since the Shor algorithm has three parts. The first applies the Hadamard transformation to n -qubits. The second finds the reminder x of N , where N is a binary number to be factorized. The last gate performs a quantum Fourier transformation; the output is n qubits. (Savran, 2018) showed how to accelerate the Shor algorithm with a focus on quantum Fourier transformation (QFT) performed by a Hadamard gate and controlled phase operators. The simulation architecture of quantum computers with multiple GPUs is regarded as similar to that of a single GPU. The CPU is responsible for preparing and transferring data to the GPUs, invoking the kernel function, and performing synchronization when the data is returned from the GPUs. Each GPU is assigned to perform a gate, with different coefficients that are synchronized at the GPU level. First, the qubit gate and state are initialized and copied to the GPUs. Then, a data distribution method configures all devices and changes non-local qubits to local qubits within the same group in a P2P manner. As a result, these non-local qubits exhibit improved performance as local qubits. Each quantum gate is computed by invoking a kernel, and no communication occurs between GPUs. Following the kernelization, the multiple GPUs are synchronized. Synchronization is not required if all coefficients are local. The computed data is transferred to the host, and data synchronization ensures accuracy. The computation ends when there are no more gates to process.

3. Quantum Computer Simulation on GPU Cluster Incorporating Data Locality (zhe, 2017)

   The GPUs are organized into nodes, and each node has a rank that is used to allocate processes. The architecture distinguishes between local qubits, inside-node nonlocal qubits, and outside-node nonlocal qubits. Performing a quantum gate on a nonlocal qubit requires communication between GPUs, and the speed of communication varies between inside-node and outside-node nonlocal qubits. This paper also describes a method for reducing communication in GPU clusters for quantum computer simulation. The method involves swapping k local qubits with k nonlocal qubits, with the other qubits remaining unchanged. The swapping process is performed in three steps: copying data from devices to hosts, reordering and packaging the data, and distributing data from hosts to devices. Two schemes for step two are proposed, with Scheme B being more efficient due to the elimination of the induction variable. The method is designed to enable GPU acceleration of the major computation steps and can be executed in parallel.

4 Shors Algorithm Implementation

4.1 Shors Algorithm on Qiskit

We implemented the Shor's algorithm using Qiksit. The circuit is implemented as below. (ibm) (qis)

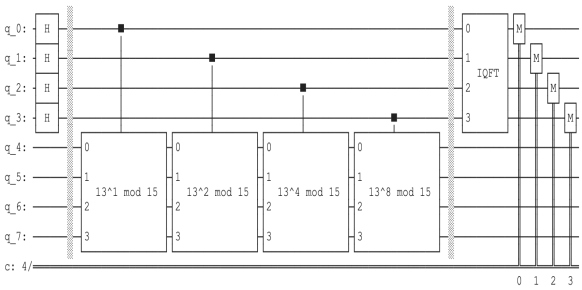

Figure 1: Shors circuit diagram qiskit

```
simulator = Aer.get_backend('qasm_simulator')

counts = execute(mycircuit, backend=simulator,
shots=1000).result().get_counts(mycircuit)
```

The results obtained from the qiskit qasm simulator code are presented in Fig4.

These results are averaged over 10 interations.

SNo.	N	Time (sec)
1	15	25
2	77	64
3	155	108

```
Value :: 10
Guesses :: (1, 3)

Value :: 14
Guesses :: (1, 3)

Value :: 12
Guesses :: (5, 3)

Value :: 0
Guesses :: (1, 15)

Value :: 8
Guesses :: (1, 15)
```

Figure 2: Shors algo factorization results qiskit

5 Parallelising Shors Algorithm using OpenMp

Inorder to leverage OpenMp library, we implemented the shor algorithm code in c++. We chose c++ since it provides a wide range of STL libraries. We leveraged these STL libraries for implementing the structure needed like the registers, states etc.(cla) The code for reference is present here : Shor Simulation

```
class Register
{

private:
        state_map states;

public:
    unsigned int num_qubits;
    Register(unsigned int num_qubits);
    Register(Register &r);
    double probability(string state);
    char measure(unsigned int qubit);

    void Hadamard(unsigned int qubit);
    void ControlledNot(
        unsigned int control_qubit,
        unsigned int target_qubit);

    void ControlledPhaseShift(
        unsigned int control_qubit,
        unsigned int target_qubit,
        double theta);

    void Swap(
        unsigned int qubit1,
        unsigned int qubit2);

    void apply_function(
        function<string(string)> f);
};
```

All the experiments are being done on the linux peanut cluster with specifications.

```
Model: Intel(R) Xeon(R) Silver 4216
CPU Freq: 2.10GHz
CPU(s): 64
Thread(s) per core: 2
```

From the inspiration of the different research papers, we realised IQFT step in shor's takes maxiumum duration. Thus we decided to parallelise the IQFT part of the code.

Here are a few reasons why QFT is a preferred choice for parallelization(Hervé Yviquel, 2022) (Haoqiang Jin, 2011) (Joseph Huber, 2011):

1. Simultaneous computation:
   The QFT allows for the simultaneous computation of multiple Fourier coefficients in parallel. This is because the QFT operates on a superposition of states, and the transformation can be applied to each state in parallel, leading to a potential speedup in computation.

2. Efficient representation of Fourier coefficients:
   The QFT provides an efficient representation of the Fourier coefficients by mapping them onto the amplitudes of the quantum states. This representation allows for the manipulation and processing of the Fourier coefficients in a compact and parallelizable manner.

```
void IQFT()
{
for (
    unsigned int i = 0;
    i < floor((end - start) / 2.0);
    i++)

    reg->Swap(start + i, end - i - 1);

for (
        int j = int(end) - 1;
        j >= int(start);
        j--)
{
#ifdef OPENMP
#pragma omp parallel for num_threads(n)
#endif
    for (
```

```
            int k = int(end) - j - 1;
            k >= 1;
            k--)
    {
        reg->ControlledPhaseShift(
                (unsigned int)(j + k),
                (unsigned int)j,
                -pi / double(1 << k));
    }
    reg->Hadamard((unsigned int)j);
}
}
```

We experimented with different num of threads and schedule OpenMp process for the speedup. (Ferretti and Santangelo, 2018) The best speedup was achieved on $schedule(static)$ $num\_threads(128)$.

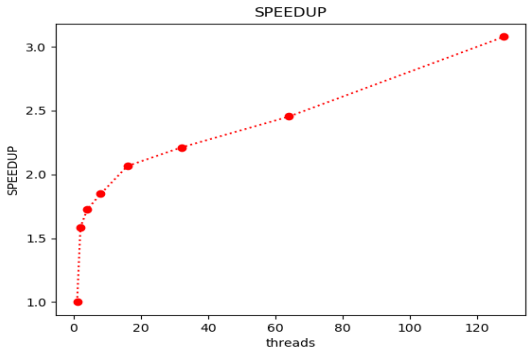

Figure 3: Speedup with different threads

This is the max number of threads on cluster $linux :: peanut$. As we increase the number of threads with the $log(N)$ (num of bits in N) more speedup can be achieved.

6 Results

Since we choose a random a everytime, it is necessary to run the algorithm multiple times and rely on the average time. Thus the serial and parallel versions both are run 20 iterations and the results of time taken and speedup are averaged over 20 iterations.

We obtained a max speedup of 70x (830 serial time, 12 parallel time) for larger $N = 9797$ value. And an average speedup of 3.5x for 4 digit values. See 4 and 5 for reference.

This comparison is against the serial c++ implementation on CPU

The speedup achieved as compared to the qiskit code is average $15x$ *for 2-3 digits.*

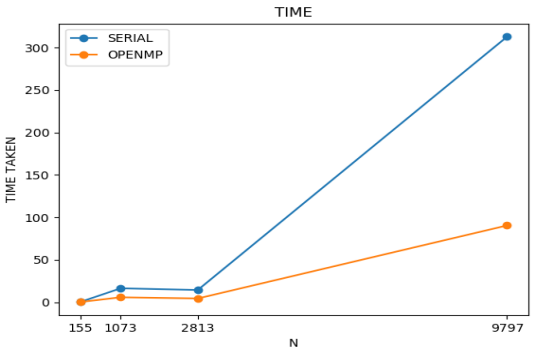

Figure 4: Time taken after parallelisation

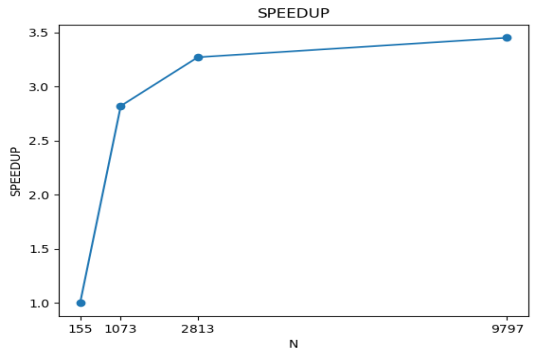

Figure 5: SpeedUp after parallelisation

7 Bottlenecks with the GPU implementation

1. Memory limitations: GPUs have limited memory compared to the amount of memory typically required for executing Shor's algorithm on a large number of qubits. As Shor's algorithm scales exponentially with the size of the input number to be factored, it often requires a significant number of qubits and associated memory. The limited memory capacity of GPUs can become a bottleneck, as it may restrict the size of the numbers that can be factored or require additional memory management techniques.

2. Classical pre- and post-processing: Shor's algorithm combines classical and quantum computations, requiring classical pre-processing and post-processing steps. While GPUs excel at parallel processing of large datasets, they may not be optimized for the classical computations involved in these steps. Consequently, the classical components of Shor's algorithm may become a bottleneck, limiting the overall performance of the algorithm on a GPU.

3. Quantum error correction: Shor's algorithm relies on the ability to perform error correction to mitigate the effects of noise and decoherence in quantum computations. However, implementing quantum error correction codes and techniques on a GPU can be complex and challenging. Error correction itself introduces additional overhead, which may further impact the performance of Shor's algorithm on a GPU.

8 Conclusion

From the results it can be observed that the speedup obtained is considerable for shors algorithm on GPU. This is not only due to parallelism but also because we leveraged the speedup C++ provides as a language and its STL libraries(inbuilt data structures like map) as compared to python (on which qiskit is based). But still there are some obstacles in speedup journey that include:

- The classical part yet has to be performed on CPU and is one of the performance bottlenecks.
- Openmp/CUDA and other libraries require implementation in C/C++ there is no existing support of quantum algorithms on C/C++. We created registers from scratch for our use case.
- CUDA and Thrust doesnt support stl librariers yet as a result speeding up other areas will be a challenge, as it will require removing stl's from code. These STL (like using map) provide speedup in the classical section of the algorithm.

In future we plan to use $simd$ instructions to speedup the classical part. Besides this we also plan to change our infrastructure code (registers, state, etc.) from map based architecture to vector based architecture. This will enable us to leverage CUDA on QFT and it would be interesting to know if CUDA can perform better than OpenMp and give higher speedups.

9 References

References

Ibm shors algorithm implementation.

Qiskit shors algorithm implementation.

Shors algorithm implementation.

2017. Quantum computer simulation on gpu cluster incorporating data locality.

Marco Ferretti and Luigi Santangelo. 2018. Hybrid openmp-mpi parallelism: Porting experiments from small to large clusters. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 297–301.

Dennis Jespersen Haoqiang Jin. 2011. High performance computing using mpi and openmp on multi-core parallel systems.

Emílio Francesquini Hervé Yviquel, Marcio Pereira. 2022. The openmp cluster programming model.

Giorgis Georgakoudis Joseph Huber, Melanie Cornelius. 2011. Efficient execution of openmp on gpus.

Demirci M. Yilmaz A. T. Savran, I. 2018. Accelerating shor's factorization algorithm on gpus.

Youngsun Han Sengthai Heng. 2020. Exploiting gpu-based parallelism for quantum computer simulation: A survey.