Performance Analysis of Prime Number Generation over CPU and GPU

Aashish Thyagarajan
Northeastern University
thyagarajan.a@husky.neu.edu

Harman Preet Singh
Northeastern University
singh.harm@husky.neu.edu

Vipul Sharma
Northeastern University
sharma.vip@husky.neu.edu

INTRODUCTION

Why Prime Number Generation?

Prime numbers are of utmost importance when it comes to security and cryptography. Most cryptography systems work by using prime factors of large numbers. If someone were to find a way to generate prime numbers faster, then everyone's security would be compromised. With the upcoming quantum technology on the rise, there can come a time where primality testing might not take as much time as we thought it would. Hence, we must work to get new prime numbers. An example for the security that utilizes large prime numbers would be the RSA encryption algorithm. The larger the prime that the algorithm uses, the more secure it becomes.

Why are we doing this analysis?

When we compare CPUs and GPUs, we know that GPUs perform better when it comes to parallelized tasks while the CPUs work better at serialized tasks. CPUs usually have 4 - 8 fast and flexible cores while GPUs have thousands of relatively simple cores, so even though CPUs might have few but powerful cores, GPUs have have thousands of less powerful cores. Comparatively, the CPU is much smarter than the GPU, while the CPU can perform high level and a wider range of tasks, the GPU can only do a fraction of those operations but at an incredible speed.

We wanted to test the performance of the CPUs to GPUs when it comes to calculating prime numbers over a range of numbers, and find a range where we can say that calculations at the CPU level might be more efficient than on GPUs.

THE SYSTEM USED

CPU systems

The CPU specifications that we used were the servers in the CCIS lab. The specific model was the Intel Xeon ES2630 running at 2.4GHZ. We can run upto 32 threads on it at a time. We ssh into the CCIS servers and run our code on it.

GPU systems

We used the GPUs that reside in the Discovery Lab at NEU. They are the Tesla K20m having 706MHz Cores with 5GB DDR5 SDRAM. We would ssh into the Discovery Lab and run our program over the GPUs there.

OUR TESTS

CPU generation of Primes

Our algorithm, written in C++, to generate prime numbers on the CPU was the brute force method. Where we specify a range of numbers starting from 2 to 18446744073709551615, which is the maximum value that can be stored for unsigned long long int datatype. For each number in the given range, we would divide it from 2 upto the square root of that number and if the number is found divisible, then it is not a prime, else it is. It is a straightforward brute force method to calculate if a number is prime. The numbers found to be prime are then stored into a file during calculations, before moving to the next number.

CPU generation of Primes with threads

This version of the code is different in the approach that, while we still take in a range of numbers from 2 to 18446744073709551615, we also specify over how many threads we want the algorithm to run on. Once the number of threads is specified, our next step is to divide the given range of numbers equally among all the threads. The threads then each perform the brute force method to assess if the number is prime or not. Each thread creates a file that has the thread number at the end of the file name and write the prime numbers to those files. Each thread on completion gives back the time that it took to complete the given task. We felt that the we must take the time taken by the last thread created as the total time as this thread would have the largest numbers and would hence take the most time.

There is another version of the threaded program, where we divide the amount of numbers that each thread will have to perform the primality testing on based on the ratio of threads there are to the amount of numbers in the given range. The first thread will perform the most work, the second lesser and so on based on the number of threads. So, this will reduce the load on the last thread which gets the largest numbers to do the test on.

GPU generation of Primes

To write a program on GPU we used CUDA protocol which is an NVidia product. CUDA is consisted of several C, Fortran programming libraries that is able to be ran on GPU directly. In GPU, we name each core as Device and our codes should be written in C using CUDA to be ran on CPU as Host and after that, send the instruction pointer of CUDA to GPU to start code execution on both Host and Device simultaneously.

Our code achieves multithreading by usage of functions like scatter, gather, map, etc. We were able to test our code on out laptop with GTX 1050 (w/ 4GB DDR5 VRAM) and on a GPU on discovery cluster. The largest prime number in the range of 10^8 which we were able to generate using the GPU was: 99999989.

GPU generation of Primes over a cluster

We started our implementation for GPU cluster programming by gathering the GPUs allocated to the program and then dividing our workload among them, using them as multi-threaded cpu model. The algorithms seeding started with CPU giving us numbers till 10, and then launching pthread for dividing work over the GPUs in parallel. The blocking/failure point for us in the implementation was during the aggregation of the result. The division of when mapping the numbers over the GPU and then gathering the result back. Due to high level of concurrency among the thread indexes and the threads, the output often was corrupted.

RESULTS WE ACHIEVED

CPU vs CPU with threads comparison

Here we have given the comparison of the time taken by the CPU to the version where the CPU utilizes threads. As described above, there are two versions of the threaded CPU implementation with the difference in the amount of numbers each thread processes. Threads_V1 has equal amount of numbers in each thread and Threads_V2 calculates a ratio of the total numbers in the given range to the number of threads being used and gives the first thread the highest amount and decreases it for each of the other threads.

The values shown in the columns for both the threaded versions are those of the thread that took the most time.

Range	CPU		Threads_V1	
	ms	s	ms	s
$2-10^4$	7	0	5	0
$2-10^5$	83	0	34	0
$2-10^6$	934	0	355	0
$2-10^7$	21950	21	7592	7
$2-10^8$	571018	571	201414	201

Table 1. CPU vs Threads_V1 with 4 threads

Range	CPU		Threads_V1	
	ms	s	ms	s
$2-10^4$	7	0	15	0
$2-10^5$	83	0	24	0
$2-10^6$	934	0	214	0
$2-10^7$	21950	21	4180	4
$2-10^8$	571018	571	111710	111

Table 2. CPU vs Threads_V1 with 8 threads

Range	CPU		Threads_V2	
	ms	s	ms	s
$2-10^4$	7	0	4	0
$2-10^5$	83	0	34	0
$2-10^6$	934	0	420	0
$2-10^7$	21950	21	7277	7
$2-10^8$	571018	571	190334	190

Table 3. CPU vs Threads_V2 with 4 threads

Range	CPU		Threads_V2	
	ms	s	ms	s
$2-10^4$	7	0	4	0
$2-10^5$	83	0	25	0
$2-10^6$	934	0	272	0
$2-10^7$	21950	21	4212	4
$2-10^8$	571018	571	109731	109

Table 4. CPU vs Threads_V2 with 8 threads

Range	Threads_V1		Threads_V2	
	ms	s	ms	s
$2-10^4$	5	0	4	0
$2-10^5$	34	0	34	0
$2-10^6$	355	0	420	0
$2-10^7$	7592	7	7277	7
$2-10^8$	201414	201	190334	190

Table 5. Threads_V1 vs Threads_V2 with 4 threads

Range	Threads_V1		Threads_V2	
	ms	s	ms	s
$2-10^4$	15	0	4	0
$2-10^5$	24	0	25	0
$2-10^6$	214	0	272	0
$2-10^7$	4180	4	4212	4
$2-10^8$	111710	111	109731	109

Table 6. Threads_V1 vs Threads_V2 with 8 threads

CPU vs GPU comparison

Below are the performance comparisons of the CPU to the GPU performance. The comparisons show the performance of the Xeon GPU, when run against a single CPU, a CPU that was threaded with 4 and 8 threads and a modified version of the threaded CPU with 4 and 8 threads.

Range	CPU		GPU	
	ms	s	ms	s
$2-10^4$	7	0	14.38	0
$2-10^5$	83	0	121.55	0
$2-10^6$	934	0	1501.5	1.5
$2-10^7$	21950	21	6506.61	6.50
$2-10^8$	571018	571	89816.42	89.8

Table 7. CPU vs GPU

Range	Threads_V1		GPU	
	ms	s	ms	s
$2-10^4$	15	0	14.38	0
$2-10^5$	24	0	121.55	0
$2-10^6$	214	0	1501.5	1.5
$2-10^7$	4180	4	6506.61	6.50
$2-10^8$	111710	111	89816.42	89.8

Table 9. Threads_V1 with 8 threads vs GPU

Range	Threads_V1		GPU	
	ms	s	ms	s
$2-10^4$	5	0	14.38	0
$2-10^5$	34	0	121.55	0
$2-10^6$	355	0	1501.5	1.5
$2-10^7$	7592	7	6506.61	6.50
$2-10^8$	201414	201	89816.42	89.8

Table 8. Threads_V1 with 4 threads vs GPU

Range	Threads_V2		GPU	
	ms	s	ms	s
$2-10^4$	4	0	14.38	0
$2-10^5$	34	0	121.55	0
$2-10^6$	420	0	1501.5	1.5
$2-10^7$	7277	7	6506.61	6.50
$2-10^8$	190334	190	89816.42	89.8

Table 10. Threads_V2 with 4 threads vs GPU

Range	Threads_V2		GPU	
	ms	s	ms	s
$2-10^4$	4	0	14.38	0
$2-10^5$	25	0	121.55	0
$2-10^6$	272	0	1501.5	1.5
$2-10^7$	4212	4	6506.61	6.50
$2-10^8$	109731	111	89816.42	89.8

Table 11. Threads_V2 with 8 threads vs GPU

GRAPH REPRESENTATION

The below shows the performance comparison in a graph for CPU, CPU with 4 and 8 threads and the GPU for the generation of prime numbers over a range

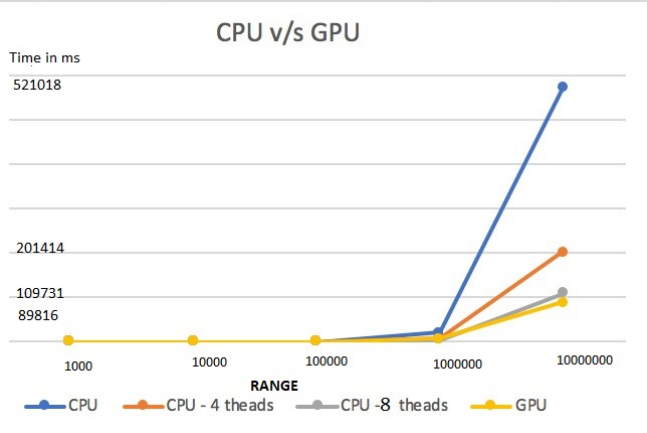

Figure 1. Proposed model

DISCUSSION ABOUT THE WORK

CPU implementation

By looking at the data we obtained, we can safely say that the CPU implementation with threads is much faster than a direct implementation which utilizes just a single CPU. When we compare the results for both the threaded versions, we can see that based how we are to divide the load for the threads to perform the task, there is difference in the time taken by the threads to complete. Although these are smaller numbers on which we test for primality, we can definitely say that for larger numbers, a threaded implementation would be a must. And if we were to use more threads there would definitely be a larger decrease in the time taken by the threads.

While the CPU is powerful, it works better for such tasks if we are to use threads and parallelize the task.

GPU implementation

The implementation of sieve of Eratosthenes through the CUDA programming is the heart of the project. We created the kernel (functions in CUDA programming) calcPrime. For initial seeding of the kernel's primelist we calculate the prime number up to 10 using the CPU which we feedback it to our algorithm to calculate the further calculations.

We create a grid and transfer the list to the GPU (device) using the cudaMemcpyHostToDevice parameter. The list is mapped over the x dimension and block of the CUDA core, and the result scattered back to fill the primelist array. The result is then transferred to the CPU (host) using the cudaMemcpyHostToHost parameter using cudaMemcpy and then processed to be saved into diskpart.txt file. The file is then read by the reader file to display and store all the prime numbers generated for verification and display it to the user.

CONCLUSION & FUTURE WORK

We were able to calculate prime number as large as 99999989 in the range 2 to 10^8. While doing so, we observed that CPUs without multithreading were way slower and took large amount of time to calculate the prime numbers. With multithreading, the CPUs were faster until 10^5 and then after that point, GPUs were significantly faster. Our implementation allows CUDA to use as many threads and this model should be faster with multiple GPUs too.

From our experience with CUDA, we personally think that the CUDA programming model is a very nice framework well balanced on abstraction and expressing power, enough control for algorithm designers, and supported by hardware with exceptional performance (compared to other alternatives). We like the fact that the algorithm designer can manually manage cache (shared memory to global memory, and global memory to PC memory). For us, one of the key requirements for high performance computing on a many-core architecture is the ability to optimize against the memory hierarchy

Other Algorithms for CPU

For this analysis, we implemented the brute force method to test the primality of a number over the range. We could also try using more efficient algorithms for testing primality like the Sieve of Eratosthenes or if possible, even the Miller-Rabin primality test.

Our analysis only works upto the maximum value of the unsigned long long int data type, which is

18446744073709551615. This number is quite small compared to the prime numbers that are utilized by most encryption algorithms. So we could also extend this to analysis to numbers that are much greater than the current maximum value used, as they might provide better insight to the performance of CPUs and GPUs.

Other implementations for parallelization for CPU

We could also do this analysis by using other methods of parallelization to see which performs better like utilizing OpenMP and also trying other methods for load distribution among the threads. We could also extend the load distribution in such a way that each thread gets a certain amount of work to do, and once the thread execution is completed, we assign it some more work or create another thread, thereby whichever thread gets completed does not sit idle nor does all the workload fall only onto the thread that were created at the start. We could monitor the number of threads that were created and maintain that number by creating new threads only when another has completed execution.

Proposed model for generation of prime numbers with GPUs in future

Each GPU will have a prime numbers list and an input list which can be sieved and will generate numbers independently.

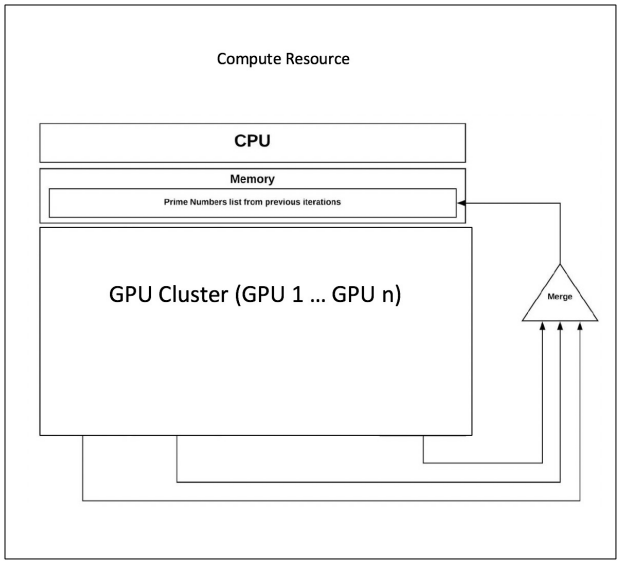

Figure 2. Proposed model

Narrowing range over which CPU might be faster over

Given enough time, and after extending the maximum value that can be evaluated, we might be able to narrow a range of data for which we could utilize the CPU with threads over utilizing a GPU for the same. This is quite important, as it would reduce the manhours that might be given towards learning the CUDA language and implementing a CPU implementation in a language that the programmer might be more proficient or comfortable with.

FINAL WORDS

We are a long way away from performing a complete analysis of the performance of prime number generation over CPUs and GPUs, but with this project, we are one step closer to doing so.

CUDA Programming Model is very vast. Significant amount of time needs to be devoted so as to learn and implement it efficiently. The numbers produced by us during this project can be significantly brought down with complete knowledge of the CUDA. CUDA programming model provides us with a very well balanced framework to perform such complicated tasks which can be achieved very effeciently by exploiting 1000s of cores on the GPUs. We are intrigued with the power of CUDA and will keep learning it in the near future to get better results and as Dr. Bill Dally, who is Chief Scientist and Senior Vice president of Research NVIDIA said **the future of the world is parallel. you know?**

ACKNOWLEDGMENTS

We would like to thank our Professor Dr. Mike Shah who taught us for the CS5600 Computer Systems course and guided us about how we could proceed and complete this project.

REFERENCES

- Why it is important to find largest prime numbers
- Why we should care about prime numbers
- Difference between CPU and GPU
- Difference between a CPU and GPU - 2
- Miller Rabin Primality test
- Sieve of Eratosthenes
- CUDA Programming Guide
- A New High Performance GPU-based Approach to Prime Numbers Generation - Amin Nezarat, M. Mahdi Raja, GH. Dastghaybifard