# A Multiple GPU Implementation to Generate Prime Numbers

*CS-5600 Project Report*

Dilip Makwana, Kaustubh Shivdikar, Saurin Shah, Soumyadeep Sinha

## Introduction:

The impact of finding a large prime number is immense especially in the field of cryptography. Hence, it is important to find large prime numbers fast and with low cost (in terms of Hardware). In parallel programming the goal is to adjust algorithms in a way that will help us split the calculation into several computer, clusters or distributed systems to reach the result faster. However, CPUs have very limited number of cores so the Number of threads that a CPU can launch is limited. Hence, the algorithm developed will only be micro efficient.

Another parallelization method which is very popular is that GPU could be very helpful instead of CPU.
The ability of a GPU with 100+ cores to process thousands of software threads at a time is what we harvest. In this structure, calculations will accelerate computations by 406x over a CPU alone. The GPU achieves this parallelism on arrays of cores on a GPU with low power cores but also many processors.

## Method Used:

Sieve of Eratosthenes was used to achieve this goal. The Sieve of Eratosthenes is a method for finding all prime numbers up to a fixed number N.

**Gist of Algorithm:**

One iteration of Algorithm could be considered as operating on:
PL = Prime List = List of Prime numbers found till now. This PL is available to each GPUs (based on number of cores per GPU, with one Prime number per core)
Let $x$ be the largest number from the PL.
IL = Input List = numbers from $x+1$ till $x^2$ (because of prime property) … this IL will be split across GPUs (based on available memory per GPU)
Algorithm uses prior generated primes list, to find primes from the Input list IL. This newly found primes are added to PL to further expand Input list and find more primes.

## Kernel Revisions:

**V1:**
This was the first kernel designed with a focus on achieving correctness and getting a grasp of GPU concepts.

**V2:**
The earlier kernel was inefficient in terms of space. This kernel made a massive space optimization by using bit vectors and hence this kernel was a memory optimization. This increased the range upto which we could find primes.

**V3:**
This kernel made huge performance improvements over the previous one; speeding up the tasks done on the GPU by about 42% (1.42x speedup) as compared to V1.

**Obtained Results:**
We were able to calculate prime number as large as 999999937(for **$10^9$** ). In doing so we were able to have a speed up of over 406 as compared to CPU. When compared to Kernel 1, which was extremely inefficient in terms of space, Kernel 2, implementation used bit vectors to represent each number in the Input List. With the implementation of the bit vector our Kernel was able to save 62 times less space as required by the previous implementation of the Kernel. With Kernel 3, the focus was more on the time optimization. By increasing the number of threads, and allocating it to the number of input we get an instant speed up of over 42% as compared to Kernel 1.

## Division of Work:

We divided the code into modules so that each module could be worked and improved upon in parallel.

**Dilip**

- Primarily worked on file reading and writing of calculated prime numbers with error handling and checks for file.
- Also code to merge results from all the kernel threads and combining all primes at the end of iteration to write in the file.

**Kaustubh**

- Each GPU is controlled with CUDA API, Kaustubh worked on launching Pthreads per GPU and handled GPU kernel launches.
- Also ensured synchronization amongst the GPUs enabling coalescing of individual GPU outputs.

**Saurin**

- Memory handling for CPU and GPU.
- Allocating tasks to the available GPUs and launching the kernels.

**Soumyadeep**

- Kernel version implementations and improvements with the focus on space and time optimization.

**Team as a whole :**

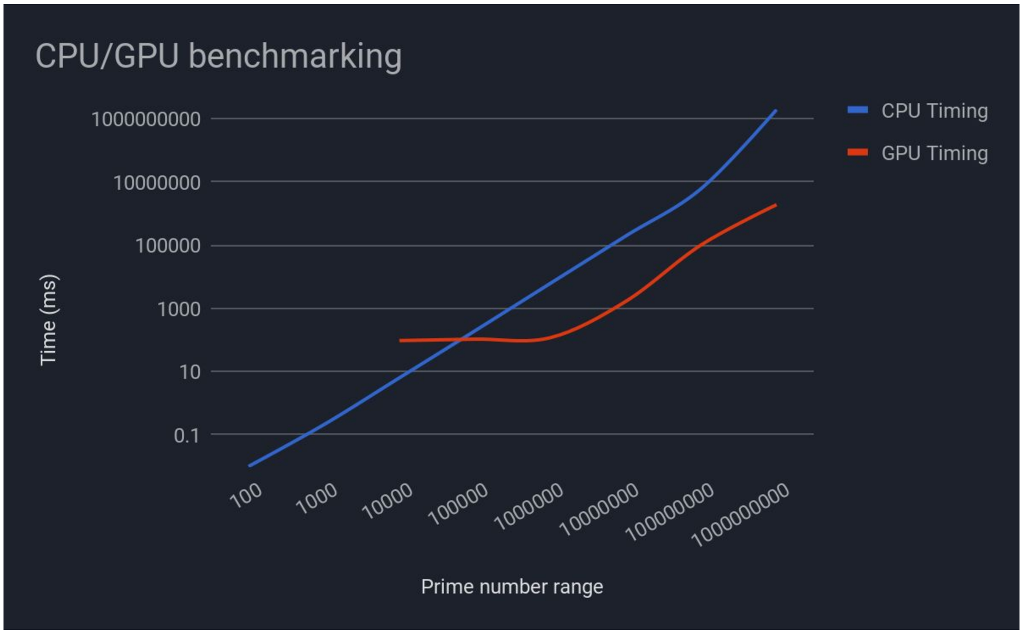
- Brainstorming the overall program flow and design
- GPU Kernel analysis and debugging
- GPU Multithreaded debugging and profiling

## Future Work:

The code we designed is capable of running on multiple GPUs across nodes when integrated with the implementation of MPI. The current cluster is capable of handling Infiniband network and thus MPI implementation is a suitable future work for our purpose of implementing program on multiple node architecture.

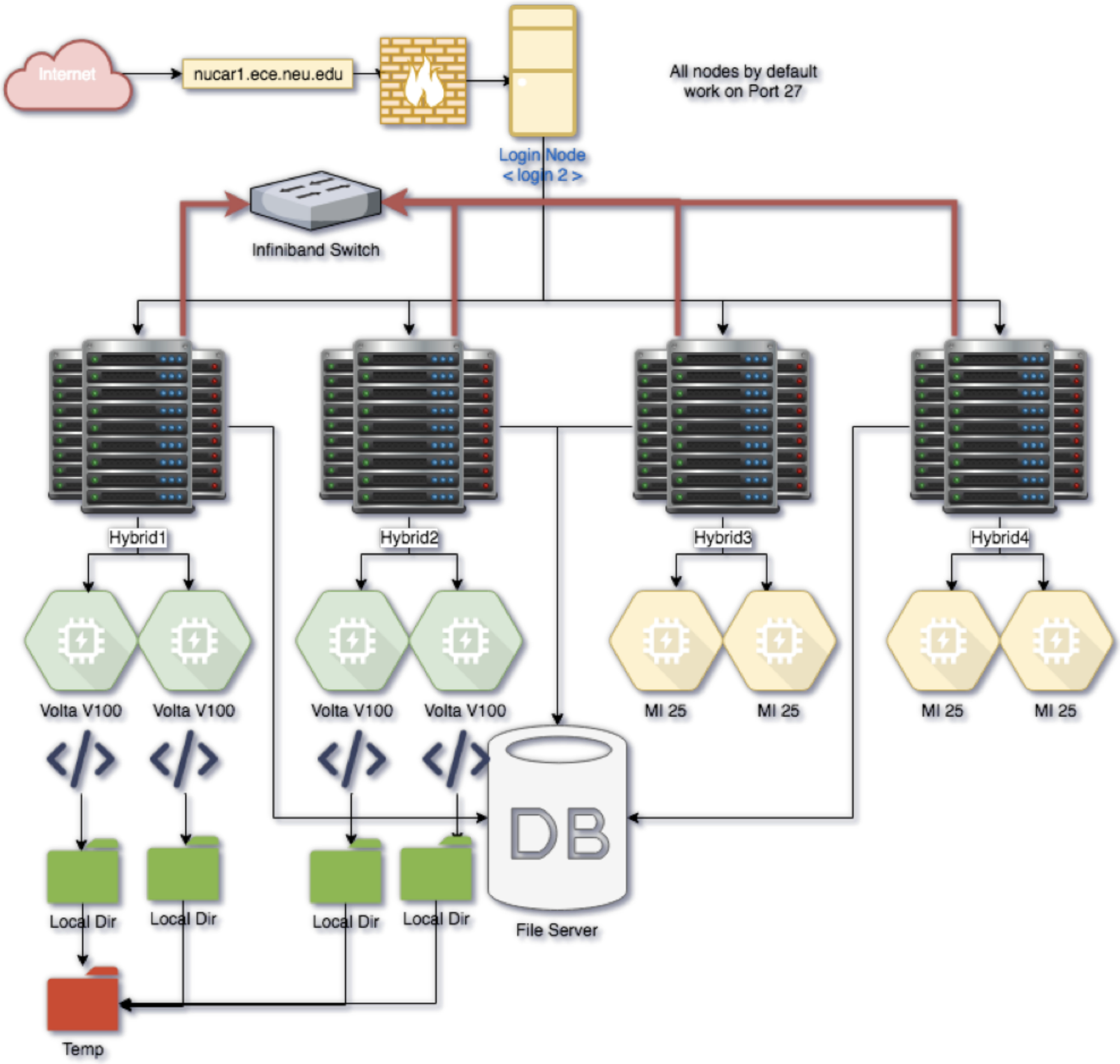
## APPENDIX:

**Performance Graph:**



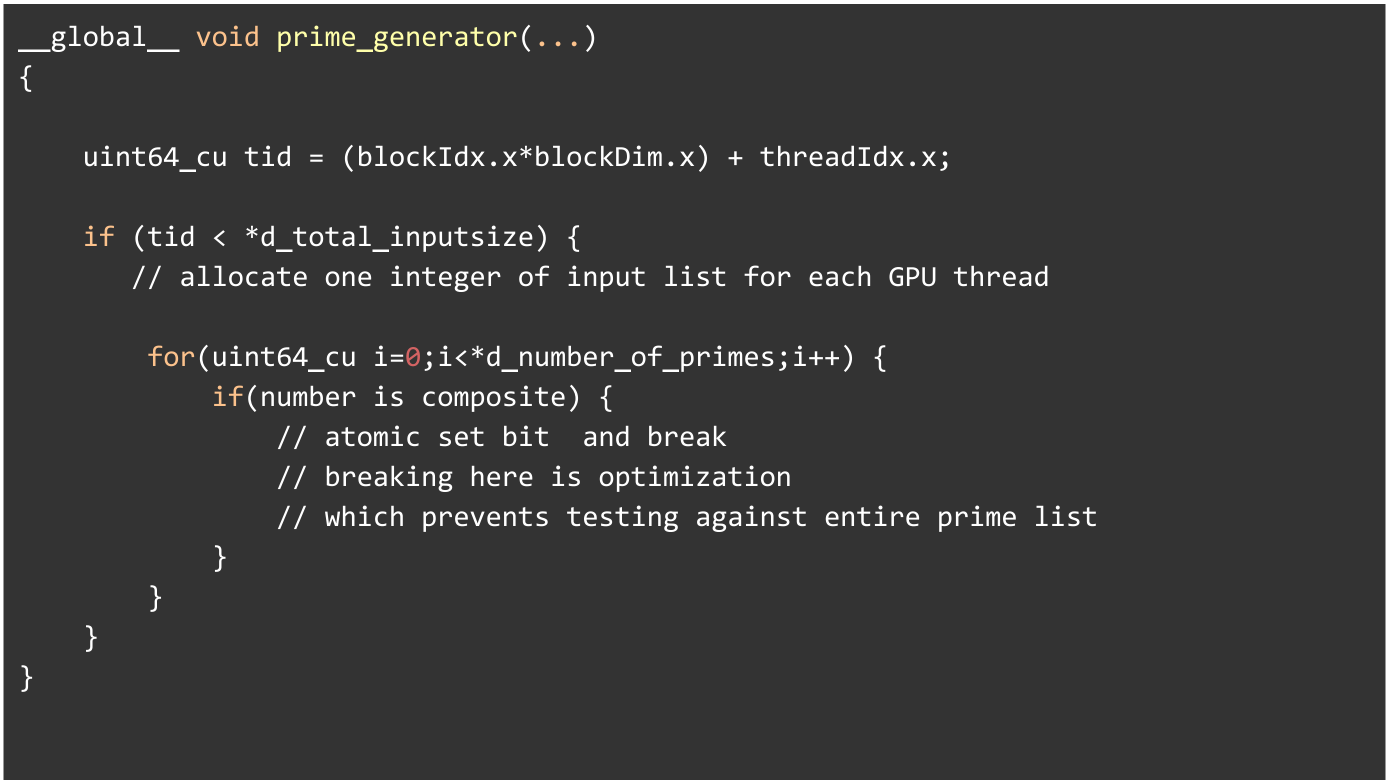
Above graph represents benchmark we obtained for CPU and GPU calculation. As we observer, apart from the initial range where CPU outperforms GPU, towards the larger ranges GPU is able to parallelize large input ranges and give faster results.

# Our Supercomputer



All nodes by default work on Port 27

## Time and Space Optimized GPU Kernel (Final version of GPU Kernel)



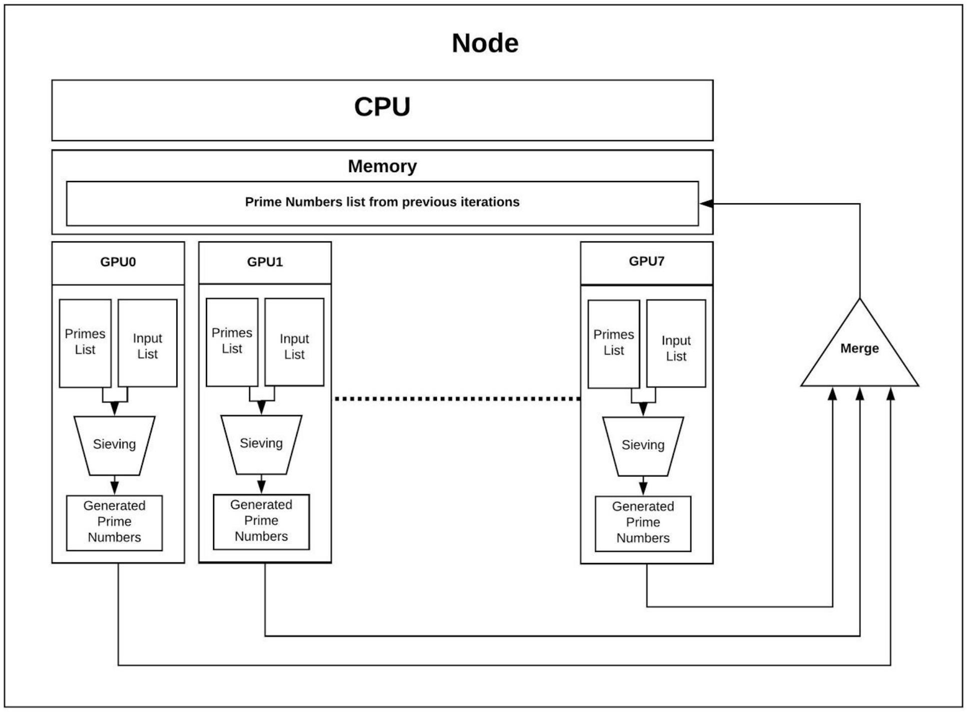
```
__global__ void prime_generator(...)
{

    uint64_cu tid = (blockIdx.x*blockDim.x) + threadIdx.x;

    if (tid < *d_total_inputsize) {
       // allocate one integer of input list for each GPU thread

        for(uint64_cu i=0;i<*d_number_of_primes;i++) {
            if(number is composite) {
                // atomic set bit  and break
                // breaking here is optimization
                // which prevents testing against entire prime list
            }
        }
    }
}
```

CS5600 PROJECT PROPOSED SYSTEM DIAGRAM



**Proposed Model for Calculation of Prime Numbers**

## REFERENCES:

1. A New High Performance GPU-based Approach to Prime Numbers Generation - Amin Nezarat, M. Mahdi Raja, GH. Dastghaybifard