A Multiple GPU Implementation to Generate Prime Numbers

CS-5600 Project Proposal

Dilip Makwana, Kaustubh Shivdikar, Saurin Shah, Soumyadeep Sinha

MOTIVATION

1. Many modern encryption algorithms depend on the factorization of very large primes which are hard to find, recognizing patterns in Prime numbers would revolutionize this industry.
2. The current research entails: Improvement over current paper in a way that multiple GPU (and nodes if possible) are not used but we will use.
3. Such a database of large prime numbers when made available to everyone, will allow other researchers to expand on their work.
4. This implementation of Prime numbers calculation would help us get a firm grip over CUDA as a programming language and a good understanding of writing multi-threaded programs.

NOVELTY

1. Previous work has extensively explored the field of working with primes over a **single GPU.**
2. The same has not been done yet with multiple GPUs. A major reason for such an approach was the **latencies of data transfer over PCIe.**
3. Now with the latest Tesla microarchitecture we believe the data **transfer cost can be successfully amortized.**
4. We also plan to implement a smart **GPU scheduler** on the CPU for our use case.

POTENTIAL DIFFICULTIES / LEARNING CURVE

The work involved in this project would be exhaustive covering multiple fields:

1. It involves writing the Prime numbers calculation algorithm in NVIDIA's proprietary **CUDA** Programming language.
2. GPU Kernel launches and memory sharing across GPUs will involve writing **Pthread programs**
3. Since the program would be run over Northeastern's Discovery Cluster, we will have to familiarize with **SLURM job scheduling**.
4. Since we are dealing with Speedups and runtimes are of the essence here, for which we will make use of profiling tools like
    a. **Gprof** (C program profiling tool)
    b. **Nvprof** (NVIDIA Profiling tool for CUDA program)
5. Our multi-gpu program is an excellent example of massively parallel implementation of 'Sieve of Eratosthenes'. Such a program would require ability to debug multi-threaded programs at such a scale.
6. **Storing** the huge prime numbers list and input numbers list requires efficient management of memory.
7. **Distribution of workload** into segments by the CPU to pass on to each GPU is in itself a challenge as it is a function dependent on number of cores per GPU and the speed and compute capability of every GPU.

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**)

Now for this iteration , we have PL and we have to find primes from IL. This is one iteration. We get the Output List (OL) from this,

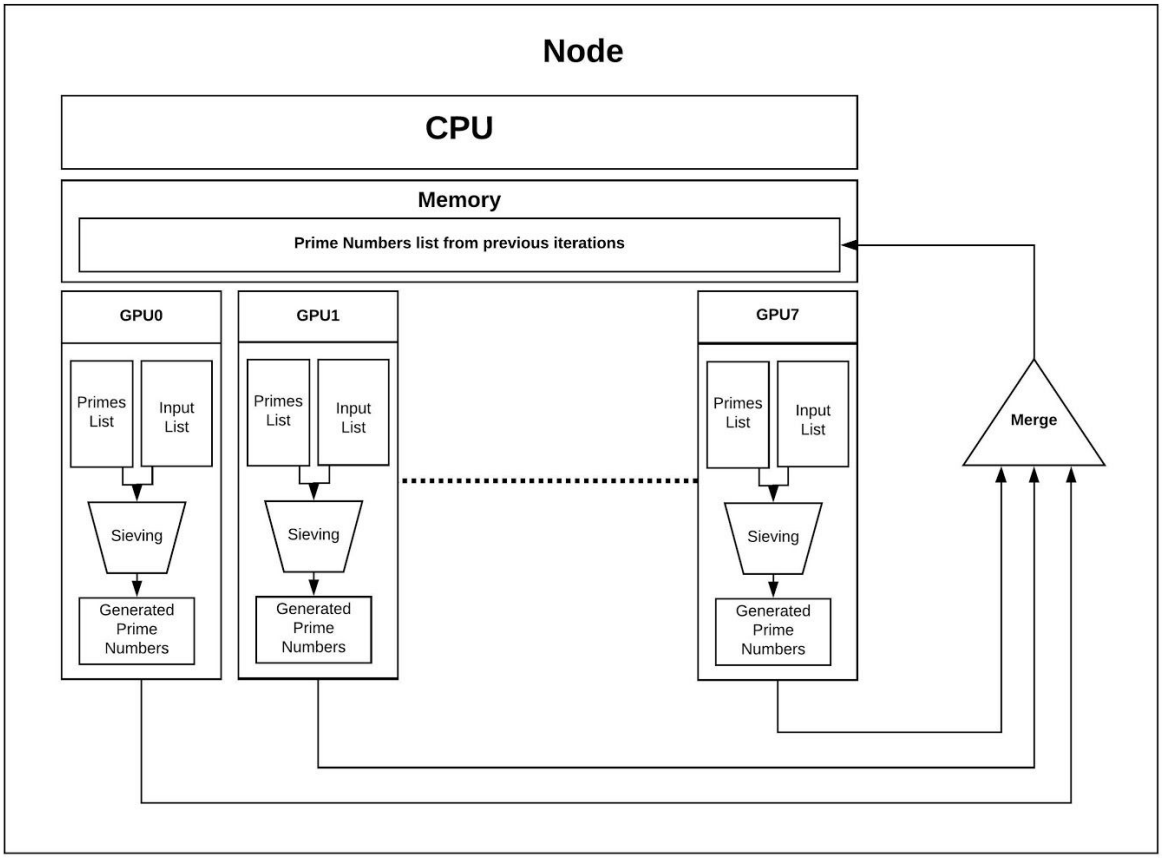
OL = Output List = prime numbers found in the given section of IL using given PL.

Now append OL to the PL of previous iteration and start next iteration and perform till certain max value (upto which we want to find prime) is reached in the input.

Example :
Iteration #1
$PL = [2, 3, 5, 7, 9]$
$IL = [11 \ldots 100]$
$OL = [11, 13, 17, ...97]$

Now in the next iteration,

Iteration #2
$PL = [2, 3, 5, 7, 9, 11, 13, 17, ...97]$ ….. Primes from previous iteration are appended to the PL
$IL = [101 \ldots 10000]$



Proposed Model for Calculation of Prime Numbers

MILESTONES

Week 1: Developing and implementing the algorithm for 1 CPU and 1 GPU (getting the entire flow right)
Week 2: IL split across more than one GPU
Week 3: PL split for more than one GPU iteration
Week 4: Deployment in Discovery Cluster and possible debugging and running the program for a week to get the final results.

GRADING MATRIX

A	Error free and fairly stable program running on Discovery cluster and Comparison of runtimes as well as space complexity with related work.
A-	Error free and fairly stable program without proper time and space optimization.
B+	If the goal itself has a fundamental flaw or we discover some flaw along the way such that we cannot find a possible solution but come up with a potential workaround.

B	If we could not overcome potential difficulties which prevents us from achieving the end goal.

DISADVANTAGES OF MULTI-NODE MODEL [FYI]

1. The algorithm we designed of splitting workload of prime number calculation into small batches and distributing it to GPUs can easily be extended to **Multiple GPUs across different nodes.**
2. An important factor to consider in this would then be the **time it takes to transfer data** across to multiple nodes.
3. This will include both, the time delay for network packets as well as the throughput of the underlying network fabric.
4. In this project we will work on the implementation over multiple GPUs in a single node. And based on our results predict if running over multiple nodes would help amortize the cost of data transportation.

REFERENCES:

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

IMPLEMENTATION

Kernel V1:

```
__global__ void prime_generator(...)
{
    int p = blockIdx.x * blockDim.x + threadIdx.x;
    int prime = d_prime_list[p];

        for(i in range 0 to Total_Input_List){
                        if(Number[i] % prime == 0)
                        {
                                d_input_list[i]=0;
                        }
        }
}
```

- Kernel 1 extremely inefficient implementation:
    - Entire input list was transferred.
    - Each prime numbers generated in previous iteration, was allocated one thread.
    - Each number took 64bits of space.

Kernel V2:

```
__global__ void prime_generator(...)
{
    uint64_cu tid = (blockIdx.x*blockDim.x) + threadIdx.x;

    if (tid < *d_number_of_primes) {
       // allocate one prime for each GPU thread
        uint64_cu primes=d_prime_list[tid];

        for(uint64_cu i=0;i<d_total_inputsize[0];i++) {
            // each number is mapped to one bit
            if(number is composite) {
                if(bit is not set earlier for composite){
                    // atomic set bit
                }
            }
        }
    }
}
```

- Kernel 2 better space optimization:
    - Better than Kernel 1: bit vector is used to better optimize the space
    - Input list was split and combined every iteration.
    - Runtime of the algorithm was Unsatisfactory

Kernel V3:

```
__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
            }
        }
    }
}
```

- Kernel 3 better time optimization:
    - Better than Kernel 2: each input number was allocated one thread.
    - Since break was there, we avoid reading entire primes list (example: once 100 was tested composite against divisibility by 2, we wont be testing divisibility back for 5)

GPU Timings / 3model:

10^4: Total GPU Time: 95.46 ms
10^6 GPU TIMES:117.78 ms
10^8 Total GPU Time: 11167.49 ms || 3 Model

GPU Timings/ 1 model:

10^4 : Total GPU Time: 1.59 ms
10^8 : Total GPU Time: 15866.01 ms