513 Prime Finding

Hongyang Yu (167008944)

Yuzhuo Li (166009044)

1. Theoretical Part

Algorithm:

1. Simple Sieve of Eratosthenes:
Find primes with simple Sieve of Eratosthenes for the first sqrt(n) numbers.

2. Segment Sieve of Eratosthenes:
The rest n-sqrt(n) numbers are divided into sqrt(n) segments. Each segment consist of (n-sqrt(n))/sqrt(n) numbers. Then we do Sieve of Eratosthenes for all segments concurrently based on the information we got from the first sqrt(n) numbers.

The Segment Sieve of Eratosthenes can be done because in order to find all primes before n, we only need to know the primes in the first sqrt(n) numbers.

2. Practical Part

1.1 Simple Sieve of Eratosthenes

The first kernel function called basicPrime. It is used to find primes for the first sqrt(n) numbers. We improve the Sieve of Eratosthenes we discussed in the class. The time complexity of our algorithm is O(sqrt(n)) with n processors, while that of the algorithm we discussed in the class is O(n) with sqrt(n) processors. Since we only need to process sqrt(n) numbers in this step, the running time is O(sqrt(sqrt(n)) with ceiling(sqrt(n)) processors.

Explain the code:

Using external for loop to call basicPrime :

```
for (int i = 2; i * i < maxRoot; i++) {
    basicPrime<<<dimGrid, dimBlock>>>(d_Primes, maxRoot, i, i * i);
}
```

In basicPrime:

```
    __global__ static void basicPrime(char *primes, uint64_t max, int base, int start){
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        if (index == 0 || index == 1) {
            primes[0] = 1;
            primes[1] = 1;
            return;
        }
        if (index >= start && primes[index] == 0 && index % base == 0)
                primes[index] = 1;
    }
```

Give primes[0] and primes[1] to 1 and index is between 0 and sqrt(n). if index is the multiplication of the base number, it's not prime and set it to 1.
The running time of this part is sqrt(sqrt(n)) with sqrt(n) processors.

1.2 Segment Sieve of Eratosthenes

In this step, the rest n-sqrt(n) numbers will be divided into sqrt(n) segments. Then we do Sieve of Eratosthenes for all segments concurrently based on the primes got from the first step. The Segment Sieve of Eratosthenes can be done because in order to find all primes before n, we only need to know the primes in the first sqrt(n) numbers.

Then call the kernel function:

```
segmentSieve<<<dimGrid, dimBlock>>>(d_Primes, max, maxRoot);
```

In segmentSieve:

```
 __global__ static void segmentSieve(char *primes, uint64_t max, uint64_t maxRoot){
    int index = blockIdx.x * blockDim.x + threadIdx.x + 1;
    if (index > 0) {
        int low = index * maxRoot;
        if (low > max) return;
        int high = low + maxRoot;
        if (high > max) high = max;
        for (int i = 0; i < maxRoot; i++) {
            if (primes[i] == 0) {
                int loLim = (low / i) * i;
                if (loLim < low)
                    loLim += i;
                for (int j = loLim; j < high; j += i)
```

```
                    primes[j] = 1;
                }
            }
        }
    }
```

We consider that the rest n-sqrt(n) numbers can be divided into sqrt(n) segments and every segment consist of (n-sqrt(n))/sqrt(n) numbers. Variable low is the start of the segment, high is the end. Check if the number is multiple of the primes we got from last step by basicPrime() function.

In this function, the running time for this step is O(sqrt(n)loglog(sqrt(n))), and the maximum of input n is 2^20.

3. Conclusion

The totak running time of our algorithm is
sqrt(sqrt(n)) + sqrt(n)loglog(sqrt(n)) = O(sqrt(n)loglog(sqrt(n))
with ceiling(sqrt(n)) processors.