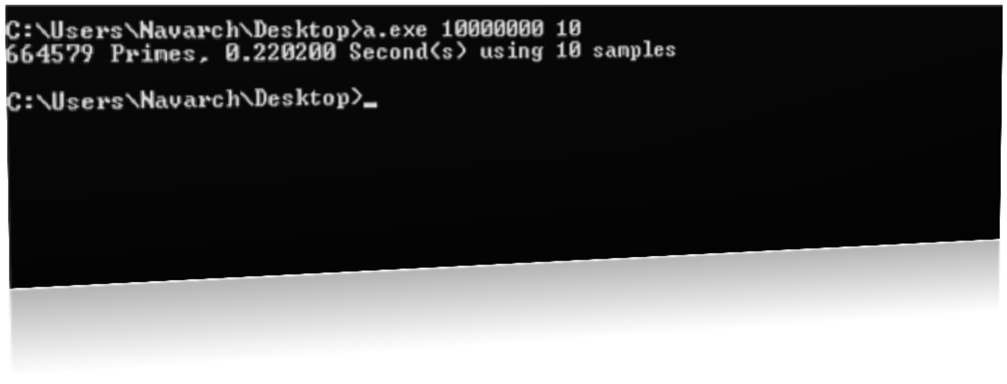
Praise be to Allah

Abstract

Starting with a simple sequential implementation of Sieve of Eratosthenes in a shared memory processor, I tried to implement the idea on a GeForce 635m GPU. Although I didn't achieve any speed-up in comparison to the sequential code, I optimized the parallel implementation so that I could achieve 2.36X speed up.

Sequential Implementation of Sieve of Eratosthenes

In this implementation, starting with prime numbers less than $\sqrt{N}$, where N is the upper-bound of our interval, we eliminate their multiples using a method called fast marking. Fast marking is used to omit the division stage which takes a huge time from the processor. Actually we can have a Wheel Factorization step before Sieving, but as I didn't implement the Wheel Factorization in the GPU version, I commented the code so not to use it. For ease of use you can send command-line arguments and for a more accurate profiling you can dictate the number of samples you want to run so providing a better average as wall clock time.



```
C:\Users\Navarch\Desktop>a.exe 10000000 10
664579 Primes, 0.220200 Second(s) using 10 samples

C:\Users\Navarch\Desktop>_
```

If you consider to run the Wheel Factorization, it's worthwhile to mention it's been proved that a modulo 210 (2*3*5*7) wheel provides the best performance for inputs as large as10^7.

In the main implementation you won't see any further optimizations. The Pseudo-Code of an optimized version of the main code can be found in Appendix A. All codes provided and more has been implemented but duo to changes they are not currently available.

There are two famous optimized versions of this Sieve. One of them is called Sieve of Atkins, It's mostly optimized for memory usage. The other one is named segmented sieve of Eratosthenes which is optimized for multithreaded implementations as well as for distributed computations. Its OpenMP implementation can be found in Appendix A.

Parallel Implementation of Eratosthenes Sieve

The idea behind the parallel implementation is the Sieve of Eratosthenes itself. At the beginning primes below $\sqrt{N}$ are calculate sequentially in host, then we have a kernel launch for each prime to mark its multiples.

All of the estimated elapsed times include the necessary time for data transfer.

Input size = 10^7, **Block Size** = (256,1,1) , **Grid** = (Input size/Block Size + Input size % Block Size, 1 , 1)

Memory Padding has been used to avoid the overhead of array bound checking which brings divergence in warps.

1- **The naive implementation : 4.227s**
2- **Using Pinned Memory to speed up data transfer stage : 4.141s**
3- **Using Zero-Copy Memory to avoid data transfer : 4.469s**
4- **Zero-Copy Memory removal + Using pinned memory + Elimination of even numbers marking : 2.206s**
5- **Using bool array instead of int array to mark numbers : 2.096s**
6- **Increasing Occupancy by doubling #Blocks and dividing #BlockSize by two : 1.790s**

To avoid the 'if' statement in the kernel, which causes divergence I implemented the code in a divergence free form, but it turned to be slower (4.761).

A % B = $\lfloor A/B \rfloor * B$

[Tid] = {[Tid] || (Tid%Prime==0)} = ~(~(1 & (((2*Tid+1)%Prime)==0)) & ~(1 & [Tid]))

Furthermore passing more than one prime number to be checked simultaneously in kernel, showed a significant speed up (about 1.6X, 4 primes simultaneously), I implemented it but then changed the code. If we assume it still speeds up the code, then the final elapsed time can be 1.12s which is a 4.25X speed up. The code will run better if we consider that for n > 3 primes can be written as 6k +/- 1, I didn't do a lot duo to time limitations.

- **Total Speed up achieve during GPU implementation optimization: 2.36X**
- **Total Speed up in comparison to the sequential code: 0.11X**
- **If you elicit the data transfer time you will surely see better results, but I found it necessary not to do so.**

Appendix A

Sequential: Sieve Of Eratosthenes

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

typedef unsigned char BOOL;

int wheelPrimes[4] = {2,3,5,7};

void Primes(int n,BOOL *primes){
        int r = sqrt(n);
        for(int i = 2; i <= r; i+=1){
                if(primes[i] == TRUE){
                        for(int j = 2*i ; j <= n ; j += i){
                                primes[j] = FALSE;
                        }
                }
        }
}

/*
void WheelFactorization(BOOL *P){
        int m = 2*3*5*7;
        for(int i = 0; i < 4;i++){
                for(int j = 2*primes[i]; j < m; j+=primes[i]){
                        P[j] = FALSE;
                }
        }
}
/**/

int main(int argc, char *argv[]){
        int n = atoi(argv[1]);
        int s = atoi(argv[2]);
        int count;
        float average = 0;
        BOOL *primes;
        clock_t t;
        primes = (BOOL *) malloc(sizeof(BOOL)*n);
        for(int sample = 1; sample <= s ; sample += 1){
                memset(primes,TRUE,n);
                t = clock();
                //WheelFactorization(primes);
                Primes(n,primes);
                t = clock()-t;
                average = (((float)t/CLOCKS_PER_SEC) + average*(sample - 1))/sample;
        }
        count = 0;
        for(int i = 2; i <= n; i++)
                if(primes[i] == TRUE)
                        count += 1;
        printf("%d Primes, %f Second(s) using %d samples \n",count,average,s);
        free(primes);
        return 0;
}
```

Source: Distributed prime sieve in heterogeneous computer clusters, Carlos Costa, Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal

Pseudo-Code of an Optimized Sieve of Eratosthenes

```
Function isPrime(n)
if n=1 then return false
else
if n<4 then return true  //2 and 3 are prime
else
if  n mod 2=0 then return false
else
if n<9 then return true    //we have already excluded 4,6 and 8.
else
if n mod 3=0 then return false
else
r=floor( n) // n rounded to the greatest integer r so that r*r<=n
f=5
  while f<=r
if n mod f=0 then return false (and step out of the function)
if n mod(f+2)=0 then return false(and step out of the function)
f=f+6
endwhile
return true (in all other cases)
End Function
```

Source: ProjectEuler.net – Problem 7 - Find the 1001st prime

OpenMP Implementation of Segmented Sieve of Eratosthenes

```
#include <iostream>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <omp.h>

#define BLOCK_LOW(id,p,n)   ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n)  (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id,p,n)  (BLOCK_HIGH(id,p,n)-BLOCK_LOW(id,p,n)+1)

void usage(void){
	std::cout << "sieve <max_number> <thread_count>" << std::endl;
	std::cout << "<max_number> range between 2 and N." << std::endl;
	std::cout << "<thread_count> is the number of threads to use." << std::endl;
}

int main(int argc, char ** argv){
	if(argc != 3){
		std::cout << "Invalid number of arguments!" << std::endl;
		usage();
		return 0;
	}

	int range_max = atoi(argv[1]);
	int num_threads = atoi(argv[2]);

	if (range_max < 2){
		std::cout << "<max_number> Must be greater than or equal to 2." <<
std::endl;
		usage();
		return 0;
	}
	if(num_threads < 1){
		std::cout << "<thread_count> between 1 and <max_number>" << std::endl;
		usage();
		return 0;
	}

	if(num_threads > omp_get_max_threads()){
		num_threads = omp_get_max_threads();
	}

	int temp = (range_max - 1) / num_threads;
	if((1+temp) < (int) sqrt((double)range_max)){
		std::cout << "Too many threads!" << std::endl;
		std::cout << "thread should be greater equal than sqrt(n)." << std::endl;
		exit(1);
	}

	// Global count
	int count = 0;

	int prime_index = 0;

	// Global k
	int k = 2;

	int thread_id = 0;
	omp_set_num_threads(num_threads);
#pragma omp parallel for default(shared) private(thread_id)
	for(thread_id = 0; thread_id < num_threads; thread_id++){

		int low_value = 2 + BLOCK_LOW(thread_id, num_threads, range_max - 1);
		int block_size = BLOCK_SIZE(thread_id, num_threads, range_max - 1);

		// block of data initialized to zero
		char * marked = (char *) calloc(sizeof(char),block_size);

		if(marked == 0){
			std::cout << "Thread " << thread_id << " cannot allocate enough
memory." << std::endl;
			exit(1);
		}

		int first_index = 0;

		do{
			if (k > low_value){
				first_index = k - low_value + k;
			}else if(k*k > low_value){
				first_index = k * k - low_value;
			}else{
				if(low_value % k == 0)	first_index = 0;
				else	first_index = k - (low_value % k);
			}

			for(int i = first_index; i < block_size; i += k){
				marked[i] = 1;
			}
#pragma omp barrier
                    if(thread_id == 0){
                            while(marked[++prime_index]);
                            k = prime_index + 2;
                    }
#pragma omp barrier
            }while(k*k <= range_max);

            int local_count = 0;
            for(int i = 0; i < block_size; ++i){
                    if(marked[i] == 0){
                            ++local_count;
                    }
            }
            free(marked);
            marked = NULL;
#pragma omp atomic
            count += local_count;
    }

    std::cout << count << " Primes in " << std::endl;//((float)t)/CLOCKS_PER_SEC << "
Second(s)." << std::endl;

    return 0;
}
```

Source: Parallelization of the Sieve of Eratosthenes, Mário Cordeiro, Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, s/n 4200-465 Porto PORTUGAL

CUDA Implementation of Eratosthenes Sieve

```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <ctime>

#define BLOCK_SIZE 256

__global__ void Sieve(bool * d_inout, int Prime){
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    /*          With Divergence          */
    if(((2*tid+1)%Prime) == 0){
            d_inout[tid] = 1;
    }
    /**/
    /*      Without Divergence          *//*
    d_inout[tid] = !(!(1 & (((2*tid+1)%Prime)==0)) & !(1 & d_inout[tid]));
    /**/
}

int main(){
    bool * h_inout, * d_inout;

    //-------------- Taking the range from User --------------//
    int range = 0;
    std::cin >> range;
    int r = sqrt((double)range);
```

```
	//--------- Device launch configuration ---------//* Calculating it here to 
eleminate the time it takes from profiling as well as using its data for memory padding 
*/
	int block_size = BLOCK_SIZE;
	int num_blocks = range/block_size + ( range % block_size == 0 ? 0 : 1);
	dim3 globalDim(num_blocks/2 + num_blocks%2);
	dim3 blockDim(block_size);

	// ------ Start of Timing -------- //
	clock_t t = clock();

	//-------------- Alocating and Initializing memory in host & device ------------//
	cudaHostAlloc(&h_inout,sizeof(bool)*num_blocks*BLOCK_SIZE/2,cudaHostAllocDefault);
	cudaMalloc(&d_inout,num_blocks*BLOCK_SIZE*sizeof(bool)/2);
	cudaMemcpy(d_inout,h_inout,range*sizeof(bool)/2,cudaMemcpyHostToDevice);

	//-------------- Finding all prime numbers between 2 and sqrt(range) ------------- 
//
	bool * Integers = (bool *) calloc(sizeof(bool),r/2);
	int * Primes = (int *) malloc(sizeof(int)*r);
	int num_primes = 1;
	for(int i = 3; i*i < r; i+=2){
		if(Integers[i/2] == 0){
			for(int j = 3*i; j < r; j+=2*i){
				Integers[j/2] = 1;
			}
		}
	}

	// ------- Counting the nmber of primes below sqrt(range) ----------//
	for(int i = 3; i < r; i+=2)
		if(Integers[i/2] == 0)
			num_primes += 1;

	Primes = (int *) malloc(sizeof(int)*num_primes);
	Primes[0] = 2;
	for(int i = 3, j = 1; i < r; i+=2){
		if(Integers[i/2] == 0){
			Primes[j++] = i;
		}
	}

	for(int i = 1; i < num_primes ; i++){
		Sieve<<<globalDim,blockDim>>>(d_inout,Primes[i]);
	}

	cudaMemcpy(h_inout,d_inout,range*sizeof(bool)/2,cudaMemcpyDeviceToHost);

	// ------ End of Timing ------//
	cudaThreadSynchronize();
	t = clock() - t;

	int count = num_primes;
	for(int i = 3; i < range; i+=2){
		if(h_inout[i/2] == 0){
			count += 1;
		}
	}

	std::cout << count << " ,Primes in " << ((float)t)/CLOCKS_PER_SEC << " Second(s)."
<< std::endl;

	free(Integers);
	free(Primes);
	cudaFreeHost(h_inout);
	cudaFree(d_inout);

	return EXIT_SUCCESS;
}
```

Source: Fernando Silva, DCC, Faculdade de Ciências da Universidade do Porto