**Analysis and optimization of prime number detection in parallel with CUDA**

JD Kilgallin, Eric Fernandes

Computer Science Department

University of Akron

**Introduction**

A prime number is a positive integer with exactly two divisors – the unit ‘1’ and the integer itself – with other numbers termed “composite”. This definition excludes the number 1 itself, as well as every even number except 2, every multiple of 3 except 3 itself, and so on. It can be easily proven that there are infinitely many prime numbers, though there is no known straightforward procedure for finding primes in a given range. Identification of large prime numbers is an important task, however, with critical applications in cryptography and other fields. The precise complexity of identifying prime numbers is not known but is believed to be solvable in polynomial time. In this paper, we present an implementation and analysis of a prime number detection program written in C using the CUDA multiprocessing framework.

The techniques employed in the search for prime numbers have been incrementally improved throughout history. Techniques from ancient Greece remain applicable today, such as the Sieve of Eratosthenes. This is essentially an extension of the definition above, where one simply lists the integers, then crosses out the multiples of 2, then of 3 since it is the next number that is not crossed out, then of 5 as the next number remaining, and so on. Any number remaining must be prime. Such millennia-old techniques underlie newer approaches, which generally further rely on modern number-theoretic results and probabilistic methods. For example, modular arithmetic can be used to rule out large classes of integers immediately. That is, all prime numbers must be of the form "6k +/- 1" for some k, as numbers congruent to 0, 2, 3, or 4 mod 6 are divisible by either 2 or 3. Further, they must all be of the form “30k + r”, with r in the set “1, 7, 11, 13, 17, 19, 23, 29” as elements from any other congruence class must be divisible by 2, 3, or 5. A more rigorous test can be devised by considering the congruence class mod 2\*3\*5\*7=210, and so on. Considering the possible congruence classes with respect to the product of the first k primes can quickly rule out large numbers of candidates as composite. This is, however, essentially nothing but a modern phrasing of the Sieve of Eratosthenes.

Other tests rely on probabilistic methods, such as Fermat's test or Miller-Rabin primality testing which can be used to further eliminate most composite numbers quickly. To conclusively prove that a number is prime, however, a basic brute force technique of testing divisibility by smaller prime numbers can be used. The set of primes to be tested as divisors of a candidate prime 'n' encompasses the range [2, sqrt(n)], as if k > sqrt(n) divides n then n/k < sqrt(n) also divides n.

At many stages of this process, the search for prime numbers can be parallelized effectively. First, for a given range [u,v] to search with p processors, processor 'i' can examine integers of the form 6k +/- 1 in the range [(v-u)/(p) \* i, (v-u)/(p) \* (i + 1)) and apply a probabilistic algorithm to generate candidate primes in this range. A final candidate list, then, can be split between processors for confirmation. For a single candidate, too, the process of testing divisibility by a set of integers can be split by divisor.

**Methodology**

The CUDA programming environment, which offers C++ extensions for parallel processing on an NVIDIA graphics card, is suitable for exploring an implementation of parallel primality testing. Ideally, such a program would work as follows:

Input: A range u, v, of integers in which to search for primes

Return: A set P of primes in the range [u, v]

Begin:

1. In parallel on all processors, generate a set of candidate primes of the form 6k +/- 1, that furthermore are not divisible by any known “small” primes.
2. Again in parallel, for each remaining candidate employ a heuristic test such as Fermat’s test or the Miller-Rabin test to rule out composites.
3. Again in parallel, for a given remaining candidate prime, parallelize the test for divisibility against all additional primes in the range [2, sqrt(n)]

In this project, we have implemented and examined a variety of primality testing algorithms and work-division strategies for effective parallel detection of primes. For each approach, we computed properties such as the speedup and efficiency of the technique for various test cases. We verified correctness against published lists of known primes and a reference brute-force primality testing implementation.

The algorithm we came to rely on for the best performance is the Fermat primality testing algorithm. This algorithm relies on Fermat’s “Little Theorem” that if p is prime, then for any *a*, *ap-1≡1.* The algorithm requires selecting various values of *a* at random and raising them to the power of *p-1*. If the result is not 1, then *p* is not prime. However, since the converse of Fermat’s theorem is not true, several values of *a* must be chosen since, if the result is 1, *p* still may not be prime. For certain (fairly rare) numbers called Carmichael numbers, this formula will always evaluate to 1 and so more rigorous testing must be used to eliminate these numbers. We are especially able to optimize the Fermat primality test by using a fast modular exponentiation algorithm that relies on the number being represented in binary. This provides more than ten times the calculation speed of a straightforward modular exponentiation algorithm, as it runs in time proportional to O(log2(p)) rather than O(p). The overall running time also scales linearly with the choice of the number of random values to start with (we found that 5 was sufficient to avoid any false-positive results) and the number of candidates for *p* each thread is to examine.

**Test cases**

Below, we give a table summarizing results from several executions of the following configuration, run on the tesla server: First, we launch *t* threads for each of the first two steps above, further splitting step 1 into two kernel function calls – one to generate candidates and one to filter out multiples of the first 25 known primes. We choose an integer *s* at which to begin our prime search and the size *c* of the interval for which to generate primes. With the list of primes generated, another block of *t* threads will filter out multiples of the small primes, and then a third block will apply the Fermat primality test. The number of primes detected and running time involved is provided as a function of *t*, *s,* and *c.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| t | s | c | Primes found | Time (ms) | Speedup |
| 1 | 6\*107 | 614,402 | 17,117 | 7850 | 1.00 |
| 32 | 6\*107 | 614,402 | 17,117 | 1033 | 7.60 |
| 320 | 6\*107 | 614,402 | 17,117 | 118 | 66.5 |
| 3200 | 6\*107 | 614,402 | 17,117 | 1 | 7850 |
| 1 | 6\*1011 | 614,402 | 15,331 | 8076 | 1.00 |
| 32 | 6\*1011 | 614,402 | 15,331 | 1136 | 7.11 |
| 320 | 6\*1011 | 614,402 | 15,331 | 129 | 62.6 |
| 3200 | 6\*1011 | 614,402 | 15,331 | 1 | 8076 |

Table 1: timing results for various execution configurations

Since each thread is able to proceed independently (no shared reads or writes), and all threads must wait between each kernel launch for the slowest thread to finish, it is not surprising that the time is reduced dramatically when splitting the same interval over more threads. It is also not surprising that, in the smaller range of numbers starting at 6\*107, more primes are found in a shorter amount of time. Also note that the speedup factors are very similar for the same number of threads between the two groups.

https://primes.utm.edu/lists/small/millions/

https://primes.utm.edu/howmany.html#table

wolfram alpha