CMPE 755 Final Project

Large Prime Number Discovery Using CUDA

Andy Belle-Isle
Submitted: December 9th, 2021

Lab Section: 01
Instructor: Sonia Lopez Alarcon
TA: Eri Montano

Lecture Section: 01
Professor: Sonia Lopez Alarcon

By submitting this report, you attest that you neither have given nor have received any assistance (including writing, collecting data, plotting figures, tables or graphs, or using previous student reports as a reference), and you further acknowledge that giving or receiving such assistance will result in a failing grade for this course.

Your Signature: _____

Abstract

Cryptography has become an important element of everyday life and is used across a wide array of fields. At the center of many popular cryptosystems, prime numbers are the basis of their security. Many cryptosystems use prime numbers for text hashing, and encryption key generation, both public and private. Since non-prime numbers have a chance of creating the same modulo output when factored (even extremely large numbers) it can lead to hashing collisions and encryption key weaknesses. Modern cryptography schemes, such as RSA, require prime numbers that are at least 2048-bits long, and the suggested length is even moving towards 2592-bits. In order for an encrypted piece of data to remain secured, a different prime number should be used for every set of transactions. Doing this means that if a single encryption key is discovered/leaked, a malicious party can only decrypt few messages. This leads to an issue with large data providers in which thousands, or even millions, of prime numbers must be generated in rapid succession for upcoming data transfers. This is a very computationally expensive process. The goal of this project is to explore the use of GPUs, specifically using NVIDIA's CUDA APIs, to generate large prime numbers rapidly in a way that CPUs cannot.

Procedure

There are many algorithms that can be used to test for primality; however, this project will only focus on a specific one. The algorithm in question is the Miller-Robin primality test. The Miller-Robin test will discover whether a number has a high probability of being prime, while being significantly faster than checking all possible factors of said number. However, it cannot guarantee the number is prime; therefore, multiple Miller-Rabin trials are run for each prime candidate to ensure a high probabilty of correctness. The Miller-Robin primality test is implemented using the pseudocode shown in Listing 1.

The Miller-Rabin test is based on the Fermat primality test, which states that for any prime number 'p', and any number 'a' that is not divisible by 'p' then: '$a^p - 1 \mod p = 1$'. This theorem will always return '1' for a prime number; however, it may sometimes return '1' for composite numbers. To ensure that a number is prime, the test should be performed multiple times with different values of 'a'. If the result is '1' for every value of 'a' that was tested, the prime 'p' has a *very* high probability of being prime.

Miller-Rabin was chosen for this project because it has a time complexity that is polynomial, meaning that once a prime candidate grows to a very large size, the time complexity of the text won't grow as much as a standard trial division which grows exponentially with the digits of the number.

```
function miller_rabin (prime, trials)
    if (prime < 2) return false
    if (prime != 2 and is_even(prime)) return false

    #Place prime into the following form 2^c * d = prime - 1
    d = prime - 1 #The odd factor
    c = 0         #Twos factor of prime
    while(is_even(d)) // Factor the twos out of d
        d >>= 1
        c += 1

    #Trial Loop
    while(trials -= 1)
        a = rand(2 to prime-2)
        x = a^d % prime #powmod
        if (x == 1 or x == prime - 1)
            continue
        for (q = 1, q < c, q++)
            x = x^2 % prime #powmod
            if (x == prime - 1)
                continue trial_loop
        return NOT_PRIME
    return PRIME
```

Listing 1: Miller-Rabin Pseudocode

Multi-Precision Arithmetic

Given that the numbers used for cryptography prime numbers are very large, to the tune of at least 1024 bits, multi-precision arithmetic must be used. Multiprecision (MP) numbers and arithmetic are the combination of multiple smaller numbers to store a much larger number than each of the smaller numbers can hold. Using a 1024-bit number as an example, on a 32-bit system, sixteen, 32-bit integers (digits) are used to store the 1024-bit number; with the least significant digit (LSD) storing the values 0 to 232 -1 and the second LSD storing 232 to 264-1, etc... While it sounds trivial, multiprecision arithmetic is very difficult to implement since all hardware operations must be performed by a software implementation. The issue with implementing multi-precision mathematics for a system like CUDA is the handling of carry results from operations. It's possible to generate a carry result in nearly all arithmetic operations, and these results must be added onto the next operation in order to maintain proper results. In the case of adding and subtracting, the carry is as simple as a single bit, which can be added onto the addition/subtraction operation on the next digit of the large number. Larger operations such as multiplication can generate carries that are as large as the number itself, e.g.: a 32 multiplication generates a 32-bit result (low) and a 32-bit carry (high). This means that for every multiplication that occurs, double the original size must be available for storage, and this carry must be propagated across all digits of the MP number. The propagation of carries has to happen with every digit, and each digit relies on the previous digit's result, meaning that these operations have to happen in serial for the most part. Certain operations, such as multiplication can occur partially in parallel, since each digit of the multiplicand can be handled by a different thread; however the semi-products must be added together serially since the carries are propagated through every digit, just like with addition and subtraction. With division operations, each division row relies on the result of the previous row, meaning that even though each row could possibly be done partially in parallel (the multiplication operation occurs here), the final result would still have to be serially processed.

Cuda Implementation

Since many of the operations had to be done in serial anyway, all operations were implemented to run on a single processing device (thread) in a serial manner. This meant that while the operations themselves could not be parallelized, multiple operations could occur simultaneously across multiple threads. This is the basis in which the CUDA code is written. For this implementation, each MP number is stored in BCD form, meaning that each base-10 digit is stored as a separate value. Each base-10 digit is stored in an 8-bit value. The reasoning behind this is two-fold:

1. It's easier to implement and understand since humans do math in base-10
2. It allows for large carry results (multiplication) to not be lost. Since each base-10 digit only requires the 4 least significant bits, the most 4 most significant bits are used when multiplication occurs, so no data is lost/overflowed.

The downside of this approach is also significant:

1. Storing each MP number takes **significantly** more memory to store than its bitwise counterpart. Since only 10 of the possible 127 positive values are stored in each 8-bit digit, the storage efficiency is **at best** 7.87% per byte. As an example: a 2048-bit number requires 64 bytes of information, and 128 bytes if using multiplication operations. Storing 2048-bits in base-10 BCD requires at least 617 bytes, and 1234 bytes if multiplication is used. Plus a few additional bytes for keeping track of sign, and digit depth.

2. Math is less efficient. Performing math on BCD is a lot less efficient than on standard data because of the increased number of digits. Again, using 2048-bits as an example. A standard 2048-bit number using 32-bit digits would require 64 digits to represent, meaning that operations would only require 64 iterations for carry propagation. With a 2048-bit BCD number, the 617 digits to represent the number would require 617 iterations of carry propagation.

Standard implementations of every basic arithmetic operation were implemented, addition, subtraction, multiplication and division. The specifics of how each was implemented will be left out of this report, but the implementation of each was the "long" version that's traditionally used to find results by hand. A few optimizations were made to each operation to make the execution faster, one common was the number of digits stored in each MP number. While each MP number had the ability to store nearly infinite (memory dependent) digits, not every digit would be used if the number was smaller, e.g.: adding 'two' to a larger number. To speed up computations, the number of digits being actively utilized for each number was kept track of so fewer iterations could be used for mathematical operations if possible.

In order to implement Miller-Rabin a few extra functions had to exist for use with MP numbers. Namely, this included a modulus function, and a `powmod` function. The modulus function was written by slightly modifying the division function to return the remainder as well as the quotient, and just throwing away the quotient. The `powmod` function is a compound operation that performs the following calculation: $x^y \mod p$. The issue with the `powmod` function is its inefficient computational speed when using a standard exponent followed by a modulus operation. Knowing this, the `powmod` function was implemented using a different method known as right-to-left binary exponentiation. The pseudocode for this implementation is shown in Listing 2.

To avoid dividing by two, an additional bit shift function was created to "shift" the BCD numbers right one bit. Since the BCD numbers couldn't be shifted right traditionally, each digit (starting at the MSD) was shifted one bit to the right, and if the least-significant bit was '1' before the shift, a value of '10' was added to the next less significant digit. The outcome of this shift, while less efficient than a standard bit shift, was much more efficient than running a divide operation.

Miller-Rabin Implementation

With all of these operations possible for MP numbers, performing the Miller-Rabin primality test with MP numbers could be done by porting the Pseudocode in Listing 1 to using multi-

```
function powmod (base, exponent mod)
    result = 1
    while (exponent != 0)
        if (is_odd(exponent))
            tmp = base * result
            result = tmp % mod
        tmp = base * base
        base = tmp % mod

        exponent >>= 1
```

Listing 2: `powmod` pseudocode implementation

precision operations instead. Since the CUDA implementation required many operations to occur in parallel for efficient execution, each CUDA thread within a block performed a "trial" of Miller-Rabin. Considering there are 32 CUDA threads per warp, the block size was always a multiple of 32; however, this block size could be changed dynamically at compile time. Since there were 32 threads per Miller-Rabin test, thirty-two primality trials were performed on each prime candidate. Every block within the GPU shared a prime candidate, which was stored in shared memory for all threads to access. Each of the random numbers used for a trial was generated on the CPU and sent to the GPU at the start of the program. The CUDA threads "treated" their own trial number to fit within the required range, from 2 to the prime number minus 2. The execution diagram of the CUDA code per block is illustrated in Figure 1.

Certain operations were only performed on a single CUDA thread because it was a block-wide operation that was only completed once. During this process all other threads were stalled until the thread finished. The most significant portion of this code was the factoring of the value 'prime - 1' into '$2^c * d$', and preliminary prime checks. These basic preliminary checks were used to speed up operation by avoiding the costly Miller-Rabin trials that contained `powmod` calls. The primality checks that occurred were testing if the prime candidate was less than 1 (including negative) or even, in which case the number was known to be composite. If the number was either '2' or '3', it was known to be prime. If any of these tests returned, that result of that prime test could be written and the next prime could be tested. To determine the primality of a number based on all thread trials, a shared 'flag' variable was used in each block, with the initial value of the flag set to '0'. If any thread completed a Miller-Rabin test and had the result of *composite*, it would set the value of the flag to '1' using an atomic operation. This way, once all threads were complete, if the value of the flag was not '0', then the value was not a prime. The GPU notified the CPU if a number was prime by setting all non-prime candidates to zero, and keeping the value of all prime number candidates. This meant that once the CUDA code completed, the list of candidates could be sent back to the CPU, and the CPU would know which values were prime, since all others were '0'.

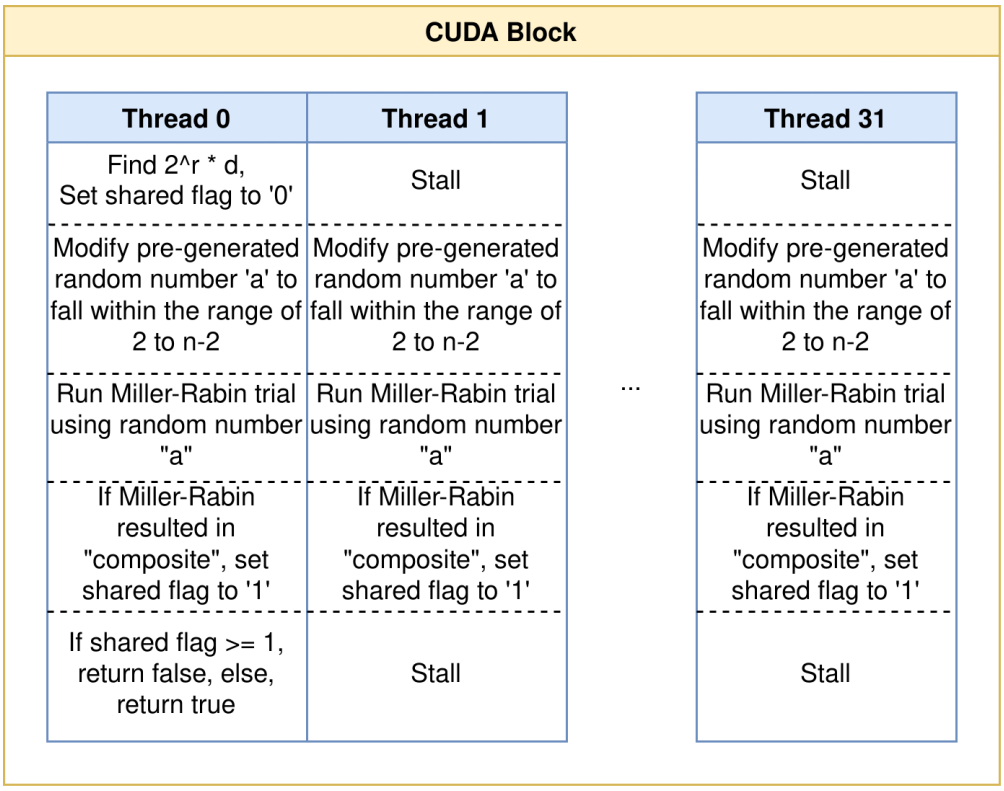

Figure 1: Block Execution Diagram

One of the issues encountered while processing large prime numbers was the lack of stack space per GPU thread. It was found that for any MP number, the storage required to run Miller-Rabin was four times the size of the number. This is because in the `powmod` function, the result of a multiplication was then multiplied by itself. The original multiplication doubled the storage size, and the second multiplication doubled that. This meant that for a 2048-bit number, the 617 digits actually required at least 2468 digits (bytes) of storage. In certain MP operations, there are two or three temporary MP variables to store intermediate results. With sizes this large, the stack size of each CUDA thread was overwhelmed. To fix this problem, a multi-precision stack type was created. This stack was allocated in main GPU memory, and was accessed by each thread for certain computations. Each thread had access to its own stack, and each stack had a depth of 15 MP values. This way, multi-precision operations that required temporary values could "push" a value onto the stack by pointing an MP pointer to the current stack pointer index. After each "push" the stack pointer was incremented, and at the exit of each function, the stack pointer was subtracted to "pop" the values off the stack.

To benchmark the code, the same implementation was run on both the CPU and GPU, the only difference being that the CPU ran multiple trials of Miller-Rabin at once instead of splitting each trial across a different core. Both the CPU and GPU contained randomized numbers for both prime candidates and MR trial numbers. The number of trials chosen for each prime number was 32. This was because of the thread architecture mentioned

previously, but also because of the accuracy that 32 trails gives. Each Miller-Rabin trial has a 25% chance to miss a non-prime for each trial, so with 32 trials, the chance of misclassifying a composite as prime was $0.25^{32} \approx 5.42 \times 10^{-20}$. The size of the primes chosen for this test were 512-bits (154 digits), and 64-bits (20 digits). The reason 512-bit primes were chosen over larger primes such as 1024 or 2048-bit primes is due to the limited amount of memory available on the GPU. A random 1024-bit number has around a 1/9000 chance of being a prime number, and 9000 trials with 1236-digit numbers required 6 GB of memory, which was too much for the GPU being tested, since around 2 GB of the 8 GB of GPU memory was used for other programs running on the testing computer.

Results

For the 512-bit test, 1280 prime candidates were chosen for testing on both the CPU and GPU, the results of which are shown in Figure 2, and Figure 3. The CPU tests were performed on an i7-4930k (overclocked to 4.3 GHz) and the GPU operations were performed on a GTX 1070, with 8 GB of VRAM. Table 1 shows the average results of multiple 512-bit primality tests. Each trial performed primality testing of 1280 different random numbers.

Table 1: 512-bit Prime Finding Test Results

Device	Average Time (s)	Average Primes Found
CPU	1735.88	**5**/1280
GPU	913.72	**5**/1280

```
Memory Usage statistics:
 - MP Stack: 10 KB
 - Primes:   785 KB
Generating prime attempts…
Running kernel…
722310576538616141395682226079162495664228301115465153497206232673330782512745283998458480023581816804267651391793558596728178451262780677006197473344126383746910747287761258535989709629995707791719697034880378540422333833722016185080588717763827250663958489097705773747615918645803419257178160657223107610736822440309586691967791559085481385541582752311739401878776436764330610887111404656933722367164797758066790850417020594943759120027123994100243156225822188655041187932310964789238356297054778089792955142740665623542279646777640063590774052726073210196863740348907996514798697940159758344589391280134723784479639740189177536793053424278522795661672789562351678112869812081879158339207652941752058162562869785830782447861959374292649837027701999135831227184725083952835680141430159915273321346933136814120273280391820173407212548159764703043082366603966897242069235615939467696376662263218614557010726862853481269618161871052371730552772811169585751030365813918163430641050120625820888660151969422418714726988483791559564872715278049441331100471726604283784403207153489514827038014590526204782182220282694517430099244984552882226257436197052171297687043443786303076887704011341301035591969722049859649043407559672432740914131381524924515286451950033753266596650262614434090387491986522174731867249682382322281371124631235333057146476820048406730196539264599203842744771613406629512872873757602282359521649406750157334702674815240302688628495411464057416917662692060044884357613119062676332591719913414041732243349828451612903126487Found 6 primes out of 1280 numbers tested.
Elapsed Time: 987.036s
```

Figure 3: GPU Results for 512-bit prime test

These images show demo outputs of the Miller-Rabin programs with 512-bit primes and 1280 total prime candidates. After execution, the values of all the discovered prime numbers were

printed to the console for viewing. Memory statistics of each program was also printed at the beginning of the execution to show the memory impact each program had on the computer. On average, with 512-bit prime numbers, the GPU provided a 1.9 times speedup over the CPU.

The same test was run again, but with 20 digits primes ($\tilde{6}$4-bits) and 25,600 total prime candidates tested. Table 2 shows the average results from these tests.

Table 2: 64-bit Prime Finding Test Results

Device	Average Time (s)	Average Primes Found
CPU	92.11	**721**/25600
GPU	913.72	**719**/25600

The results of the 64-bit prime number test shows that the GPU offered, on average, a 2.66 times speedup over the CPU. With Miller-Rabin, the suggested number of trials for a 64-bit number is 12, with each trial being a prime between 2 and 37 (2, 3, 5, 7, 11, 13, 17, 19, and 23). Due to the unique situation of the GPU architecture, it was more efficient to keep the number of trials at 32 since the execution would take just as long.

The results of these two test sets show where the strength of the GPU comes into play. Since the GPU tests all trials within Miller-Rabin at once, it's effectively taking as long as a single trial. However, since each GPU core is less performant than that of a CPU, the time it takes for each trial is much longer. This is overcome by providing more prime candidates to the GPU, which will give it a higher utilization for longer, allowing it to outperform the CPU. The performance gain of the GPU began to slow down as the numbers got larger, with this being due to extra thread divergence in numbers with more digits. The solutions to this are discussed in the following improvement discussion section.

Improvement Discussion

The limiting factor for GPU performance was the execution efficiency of the multi-precision (MP) implementation. This could be improved in a variety of ways:

1. Currently, the BCD storage format used to store the MP numbers is set up in base-10. While this makes it easier to understand and write, it is far less efficient than power-of-two bases. Changing the BCD to use a power-of-two base would more efficiently take advantage of the available bits in the BCD, and would make operations such as bit shifting much more efficient since bits could be manipulated quickly without the need for base-10 carry and remainder logic. This also means that for carry calculations in addition, subtraction, and multiplication operations, the modulo operator could be replaced with a simple bitwise-AND, vastly speeding up the execution of each cycle.

2. The storage efficiency and serial execution paths could be reduced by increasing the storage size of each digit. Moving from 8-bits per digit to 32-bits per digit would allow

for a base of 65536, solving the problem listed in #1, and would also reduce the number of digits required. Not only would this reduce the number of bytes required to store a large number, but it would also increase the efficiency of each calculation since there would be less digit iterations for arithmetic and carry propagation.

3. The most effective speedup would occur by not limiting the number of iterations performed for each arithmetic operation. For example, adding 2 and 32 would only require two iterations over the digits since the larger operand, 32, only requires two digits for storage. Initially this was done to increase the performance of single core environments, but in a many-core environment like CUDA in which each thread in the same warp needs to process the same instruction for maximum efficiency, it actually slows down the execution. This slow down is due to divergence that occurs when each thread has a slightly differently sized number. The current solution is placing common `__syncthread()` calls after every operation, but this is slow and can possibly hang the program if used carelessly. Instead of limiting the number of digits for each operation, every single operation should be performed on every digit of an MP number, not just the populated ones. While it would take longer for single-threaded environments, it would be *much faster* in CUDA since the threads wouldn't have to diverge at all. The performance impact of this would also be minimized if the two prior improvements were made first, which would improve the efficiency of each digit, and would reduce the total number of digits required.

4. Finally, a few software optimizations could be made to improve the performance. A few optimizations already exist, like checking for even numbers. Instead of running an expensive modulo operation, a simple 'even' function is used to check if the '0' bit of the least expensive digit is set, if so, that number is odd. Another optimization would be running powers of two through bit shifts instead of the power or multiplication functions like they currently are. The largest optimizations would happen within the division operation, which could be greatly improved to perform checking on whether a bit shift could be used instead of a pricey division operation. These optimizations however, could cause divergence to occur within CUDA threads, meaning that they should be used sparingly, and only after extensive performance testing.

Conclusion

In conclusion, the results were promising, with the GPU providing a sizable speed boost over the CPU for large prime number generation in modern cryptography systems. However, once multithreaded, the CPU would easily outperform the GPU. This is largely due to the thread divergences that happen within the CUDA code, forcing the GPU to run the code sequentially. The solution to this could be solved through improvements to the multi-precision operational flow, and would provide more efficient execution on the GPU with less divergence. Knowing this, it shows that GPUs can provide a viable option for increasing the efficiency in prime number generation for modern cryptosystems.