Optimization of Primality Testing Methods by GPU Evolutionary Search

Steve Worley*
Worley Laboratories

Abstract

Modern fast primality testing uses a combination of Strong Probable Prime (SPRP) rejection tests. We find more powerful combinations by intensive search of the vast domain of SPRP test configurations. Evolutionary guidance using previous promising results boosts search speed. We implement the entire search on the GPU with the CUDA programming language resulting in 65-time speedup over a CPU search. This project has already found a test an order of magnitude more powerful than the best previously known.

1 Primality Tests by Strong Probable Primes

How do you determine if a given n is prime? Countless practical and diverse problems ask this, ranging from hash table design, cryptography, Rabin-Karp string searching, and randomized task scheduling. Simple trial division with the integers from 2 to $\sqrt{n}$ is a classic and well known method, but as $O(\sqrt{n})$, faster methods are much better for n larger than 5000 or so. In practice the fastest computational method for larger n is Strong Probable Prime testing.[1]

By Fermat's Little Theorem, for an odd prime n, the number 1 has only two square roots modulo n, 1 and $n-1$. So for prime n and any integer $a > 1$ the square root of $a^{n-1} \bmod n$ is 1 or $n-1$. We can use this fact to identify many composite numbers by merely choosing any a, substituting n into the equation and checking the modular remainder. If the remainder is not 1 or n-1, we can stop since we now know n is composite. However this does *not* yet guarantee that all passed values are prime.

We can make the test even better at quickly identifying composites by noting that if $(n-1)/2$ is even, we can take another square root and get yet another rejection test.

We remove out the factors of 2 to form $n-1 = d \cdot 2^s$ where d is odd and s is non negative. n is called a strong probable prime base a (an a-SPRP) if either $a^d \equiv 1 \bmod n$ or $(a^d)^{2^r} \equiv n-1 \bmod n$ for some non-negative r less than s.

All integers $n > 1$ which fail this test are composites. Integers that pass *probably* are primes. For $a = 2$, the 2-SPRP test is rarely incorrect, and in fact the first composite n undetected by the 2-SPRP test is $n = 2047 = 23 \cdot 89$. The first missed by the 3-SPRP test is $n = 121 = 11 \cdot 11$. The first miss by the 5-SPRP test is $n = 781 = 11 \cdot 71$.

SPRP tests are very fast in practice using standard methods for modular powers (also used in RSA encryption). The computation is $O(\log^3 n)$ and is *independent* of the choice of a.

Since an SPRP test can miss an occasional composite, testing can be repeated with two or more independent SPRP bases to give better and better confidence that a value repeatedly passed is indeed prime. Miller and Rabin made this a practical *guaranteed* primality test by finding an explicit pair of SPRP bases, which combined are pre-verified to correctly identify primality for as large n as possible. For example, the smallest composite n which is *both* a 2-SPRP and a 3-SPRP is 1,373,653. This immediately forms a guaranteed primality test for any $n < 1,373,653$. Such a (now deterministic) test is known as the Miller Rabin primality test. Other researchers realized this was a very useful tool in practice, and significant attempts were made to discover the best values of bases a to maximize primality testing ranges beyond 1,373,653. In 1980 Carl Pomerance[2] explored the SPRP tests and shared the first high speed prime testing code, used extensively for years. All tests using two SPRP evaluations require the same amount of computation, so interest grew in discovering the SPRP bases that maximize the range of n that the tests apply to. In 1993, a paper by Gerhard Jaeschke showed that primes from the much wider range of up to 9,080,191 could be classified with just two tests (a 31-SPRP and 73-SPRP). In 2004 Zhang and Tang improved the theory of the search domain. Finally in 2005, Greathouse and Livingstone used computation on several computers for a period of months to find the currently best known published results: a combination of 2-SPRP and 299417-SPRP gives guaranteed primality results for all $n < 19,471,033$.[3]

2 Searching for better SPRP combinations

The speed of SPRP tests do not vary by a, so it is completely acceptable to choose whatever values work best. A practical limit keeps $a < 2^{32}$ just for ease of representation and use of native processor math operations.

We could try to find a test with an even higher n limit by picking two a values and testing every composite integer until both SPRP tests fail and hoping we exceed our previous best known n. Unfortunately while the SPRP test is very fast, we need to evaluate the test for millions of composite value to determine the minimum composite which fails, and hope that first failure is larger than the best known test's range. Such a brute force algorithm is straightforward to implement but on a CPU requires several seconds to test a pair of SPRP bases over the several million test values. If we restrict $a < 2^{32}$ for efficiency, there are 2^{63} possible pairs of bases to choose from, far too many to explore exhaustively.

The distribution and behavior of prime number distribution is a huge topic in number theory, and the behavior of SPRP composite failures is similarly chaotic. The best theoretical analysis has been by Zhang and Tang, who found limits on minimum n but no useful guidance on how to find sets of tests which are especially efficient. Therefore, finding better SPRP tests is a task of raw power and empirical heuristics.

Several obvious search strategies were initially explored. One was a simple brute force enumeration of pairs in a deterministic order based on minimizing their sum. Another enumerated (2,x) for all $x < 2^{32}$, then proceeded to (3,x) and so on. A third randomly sampled the whole domain.

However, the most effective algorithm found was a simple progressive evolutionary test, subjecting test bases to a successively more difficult set of challenges and allowing only the best-performing to continue. The most promising base values pass each test and there-

*e-mail: sw@worley.com

[1] Much of the mathematical description for this section draws from the excellent Prime Pages website at primes.utm.edu. This website also inspired this work when I was searching for an isPrime() function.

[2] Developer of the factoring method to crack RSA-129 decryption!

[3] Errors have been found in some of Greathouse's higher order results, but his SPRP tests for two bases are correct.

fore even more computation is invested in them for another round of search. This strategy performed exceedingly well, perhaps because the amount of effort spent on each value is proportional to how well it proved itself on previous rounds.

The search algorithm starts by picking a random initial base value. The value is exhaustively tested as a partner with all values from 2 to 10,000,000, taking approximately 10 seconds on a NVidia GTX280. The fitness measure to maximize is the value of the first composite n which the pair of values both misidentify as a prime. The best result of that search is tested against a predetermined threshold to allow only about 30% of test values to pass to a more intensive second round of 25M additional tests. This cycle of survival-of-the-fittest continues with five rounds, each with an increasing number of tests but also an increasingly difficult fitness threshold to cross. The final state is an ultimate exhaustive test of all $2^{32} - 1$ partner values. Less than 0.1% of test values are fit enough to reach the final exhaustive stage.

3 GPU Implementation

GPUs have enormous computational horsepower compared to CPUs, making them attractive for a compute intensive task like SPRP searching. Even though GPUs are actually faster at floating point than integer, the vast number of GPU subprocessors makes up for any weakness.

We implement the search using the CUDA programming enviroment for NVidia GPUs. The very C-like language made prototyping and implementation straightforward.

The SPRP's evaluation is quite straightforward as a classic modular power computation, very similar to one used for RSA encryption. We have an advantage in that our values of a and n are under 2^{32} and we can use many native CPU math operations instead of using a generalized "bigint" library.

Most of the SPRP algorithm's time is simply spent doing modular powers, such as $a^b \bmod c$. This is a classic computer science homework problem, and it is often solved by reduction by noting that if b is even then $a^b \bmod c = (a^{b/2} \bmod c)^2 \bmod c$, and if b is odd, then $a^b \bmod c = a \cdot (a^{\lfloor b/2 \rfloor} \bmod c)^2 \bmod c$.

The above reduction now just needs a way to compute $a \cdot b \bmod c$, which is complicated by the fact that $a \cdot b$ can easily overflow a 32-bit integer. Another reduction of the same style solves the problem. If b is even, then $a \cdot b \bmod c = 2a \cdot \frac{b}{2} \bmod c$ and if b is odd, then $a \cdot b \bmod c = a + 2a \cdot \lfloor \frac{b}{2} \rfloor \bmod c$. The final subproblem of simple modular addition of $a + b \bmod c$ is an easy addition and single overflow test.

Unrolling the reductions one step for even more efficient computation is another trick known as Montgomery Reduction. This uses a small table of precomputed values of $1, a, a^2, a^3 \bmod c$ to halve the number of steps needed. In practice we are able to significantly benefit from such a table lookup. An interesting implementation detail involves the use of CUDA shared memory for the lookup table. We give each thread in a block "ownership" of 4 words in shared memory for its own lookup table. These table entries are spaced 16 words apart in order to prevent a subtle memory efficiency problem known as bank conficts. By giving each thread its scratch memory spaced by 16, each thread has its own bank of shared memory and thus does not interfere with other thread's lookups. This subtle implementation detail improved performance 20%.

The CUDA kernel is designed to assign every thread one independent pair to test. A thread checks its pair, proceeding until it finds the first misclassified composite. We want kernel launches to remain in the millisecond range to keep the OS interactive. The compute proceeds by evaluating only a few thousand tests and checkpointing its progress (needing to store only two words) to resume in the next launch. Thus the continuous GPU search has little OS impact, allowing the compute to run unobtrusively for days or weeks.

The pause-and-restart design also allowed for automatic load balancing. If a thread is unlucky and tests a poor candidate, it can finish its work much earlier than another thread who needs to test more values, leaving the early thread idle. A simple test at kernel launch identifies idle threads and has them steal work from threads with work remaining. This load balancing was easy to implement by merely rearranging the saved work status storage. Load rebalancing boosted search throughput by 60% by retasking these idle threads. The design of CUDA automatically handles block-wise work assignment, so the software does not know nor care if the GPU is a small 16 core or large 240 core GPU; speed simply scales completely linearly with SP count.

All computation is done *entirely* on the GPU, leaving the CPU idle except as a polling kernel host. The simple polling loop keeps the CPU loaded by about 10%. This could likely be improved by queuing multiple asynchonous kernel launches. The original expectation was to have the CPU simultaneously run searches, but the raw speed of the GPU search makes any CPU contribution negligable and merely a distraction.

Since the evolutionary search begins with a random base choice, it is embarassingly trivial to support multi-GPU and even multi-machine evaluation. Each GPU uses its own random seed and each follows its own independent randomized evolution. Results don't need to be combined or intercommunicated except to occasionally manually note which GPU or machine has the best results so far. Unlike many distributed computations, there is little need to avoid work duplication given the huge search space, and the progressive results from each device is just a simple pair of numbers: the best results that GPU has found so far.

4 Results

On a 3.0 GHz Core2 PC with an NVidia GTX280, the GPU search implementation performs approximately 65 times faster than a CPU version. This makes sense since there is little memory transfer overhead and the SPRP search is computationally limited. Even our relatively weak 32-processor laptop GPU easily exceeds its CPU computation speed. Multiple randomized tests show that it's quite easy to find new *record breaking* SPRP results for $n \approx 80,000,000$ within an hour. Compare this to the previously best known test by the 2005 search by Greathouse and Livingstone, who found a best result of only $n < 19,471,033$.

After one week of constant search on two machines, the best SPRP test we have found so far is a combination of an 350-SPRP and a 3958281543-SPRP. This combination correctly identifies all primes for $n < 170,584,961$. This is an order of magnitude better than the best results known previously.

Further searching continues, and we plan to share the final results online. The identification of better test values will immediately allow various software libraries providing primality tests to give fast results over a wider range than was previously possible.

Although not presented here, the cases of higher order sets of 3 and 4 SPRP tests are now also being explored with a CPU/GPU hybrid. This is significantly complicated by the need for 128-bit integer math. This work is not yet complete but has also already found order of magnitude improvements in primality testing.