Quadratic Sieve on GPUs

Bayley Wang

December 16, 2011

Abstract

High-performance factorization is a naturally parallelized task, lending itself well to the "stream processing" paradigm found on graphics processing units. The quadratic sieve algorithm by Pomerance shares many characteristics with the cutting-edge number-field sieve (smoothness, factor bases, and linear algebra over GF_2), in addition to being simpler and significantly easier to implement. Furthermore, the most computationally expensive part of the algorithm is simply a floating-point vector add, which GPU's can perform very quickly

1 Integer Factorization: History and Importance

Integer factorization is a famously hard problem. Not only are all known algorithms at best subexponential, the problem is not known to be NP-complete or even NP-hard. While factorization is a purely academic interest worthy of study, the main interest in the problem stems from the fact that the security of several important cryptosystems are based on the hardness of factoring. In particular, the RSA cryptosystem, which is the heart of Internet security (there, the symmetric, but slow, RSA is used to transfer keys for faster, asymmetric cryptosystems). RSA proceeds as follows. The **public key** is $n = pq$ and e such that $\gcd(n, e) = 1$, where p and q are primes (preferably large and of similar bit length for security). The **private key** consists of p, q, and d such that $de \equiv 1 \bmod (n)$. Now suppose Bob wants to send a message M to Alice. He looks up Alice's public key and sends $c = M^e \bmod n$. Alice recieves this encrypted message and computes $c^d = M^{de} = M \bmod n$. Clearly, RSA is broken if p and q are known (since then d could be computed). Curiously enough, it is not known whether breaking RSA is equivalent to finding p and q.

"Semiprimes" of the form RSA keys take are the hardest numbers to factor. In their case, factoring means "finding one nontrivial factor of the number", and indeed, we define the general problem in the same way. Generally, this is enough to factor n with at most logarithmic overhead, in the case n is the product of several similarly sized primes (there are faster algorithms for locating small prime factors, e.g. Lenstra's elliptic curves and Pollard rho).

Up until the 1980's, factorization was very hard - a combination of limited resources and the lack of good subexponential algorithms made even 70-bit

numbers unfactorable. The revolution came in 1981 with Dixon's "random squares" factorization algorithm, which used a trick dating back to Fermat. Instead of trying to find p and q explcitly, we try to find X and Y such that $X^2 \equiv Y^2 \bmod n$. This then hopefully gives us a factorization of n via $\gcd(X - Y, n)$ and $\gcd(X + Y, n)$.

Dixon's algorithm uses a few key concepts that are present in later algorithms. In particular, it uses a **factor base**, which is a set of small primes F_B, and the trick of finding "smooth numbers" - numbers whose prime factors lie entirely within the factor base. By finding values of x for which $y = x^2 \bmod n$ is smooth, we can find some subset of the y whose product is a square. To do this, we write y_i as the the product $\prod p_i^{e_i}$, where the p_i are primes in our factor base. Then, with each y_i we can associate a binary vector E_i corresponding to the list of $e_i \bmod 2$. If there are at least $|F_B| + 1$ such E_i, we can find a linear combination of them that are 0 mod 2, corresponding to a product of y_i which is a square. This product is Y^2; X^2 is simply the product of the corresponding x^2. We hope this congruence gives us a nontrivial factorization of n. If not, we find another such congruence via another linear algebra step, and try again.

The current factorization record is 768 bits, done with a custom implementation of the General Number Field Sieve at INRIA. Other notable factorizations include the factorization of several 512-bit keys belonging to Texas Instruments in the summer of 2009 via a distributed effort, RSA-155 (the first 512-bit semiprime factored) in 1999, and RSA-100 on a massively parallel vector machine.

The fastest (public) implementation of the GNFS can factor a 512-bit number in 70 days on a Athlon 64 X2; the fastes implementation of the quadratic sieve can factor a 90-digit number in 5 minutes on a quad core Intel i7-950.

2 The Quadratic Sieve

The quadratic sieve is a fast algorithm for factoring general integers. In particular, with current hardware and implementations it is the algorithm of choice for factoring numbers with up to 100 digits. While not the fastest algorithm in existence, the algorithm is fairly simple to implement and has many similarities with the Number Field Sieve, making it a good benchmark for GPU factorization.

The quadratic sieve extends Dixon's algorithm by a change of paradigm - instead of picking x and seeing whether $x^2 \bmod n$ is smooth, we use a sieving process to find which values of x result in $y(x) = x^2 - n$ that are divisible by some prime. Because of the polynomial nature of $y(x)$, once we find a base solution to $x^2 - n \equiv 0 \bmod p$, we've found them all.

We initialize an interval of integers we wish to sieve on. Then, for each prime in our factor base, we find the base solution to the above congruence, use that to find all solutions, and divide out each one by p. After doing this for all p, we simply look through the interval for values which are 1, and add those to our set of smooth numbers.

Once we have enough smooth values, the algorithm proceeds as in Dixon, and

we hopefully get a nontrivial factorization of n.
In practice, the division in the sieving can be replaced with subtracting $\log p$.
To improve performance, multiple polynomials can be used.

3 Graphics Processing Units and Stream Computing

Historically, graphics processing units were originally fixed-function units for accelerating the drawing of basic primitives. Later, they were extended to have hardware acceleration for z-buffering, clipping, shading, etc. The first major advance came with the release of DirectX 9.0 and OpenGL 2.0, both of which added support for shaders, small programs that could be executed at each pixel, vertex, or polygon. Using rather inelegant hacks, programmers devised ways to execute non-graphics code on graphics hardware. Examples included raytracers, radiosity, and, most notably, Folding@Home, a distributed protein-folding project.
In late 2006, Nvidia released the G80 GPU in the form of the GeForce 8800 Ultra/GTX/GTS. The G80 replaced the vector processors (usually 4 components wide) found on older GPU's with 128 single-component processors. From the graphics point of view, 128 identical processors improved the performance of the graphics pipeline, as they could now be reconfigured to act as pixel, vertex, or fragment shaders depending on the scene load. More importantly, from a general-purpose programming point of view, this eliminated the need to explicitly vectorize code. The release of G80 also coincided with increased vendor support for GPGPU, in the form of Nvidia's CUDA and, not long after, OpenCL, a open standard for parallel computing.
The GPU computing paradigm has been termed "stream computing". Stream computing indicates massively data-parallel and instruction-parallel tasks. In fact, existing toolkits enforce this - all device side code must be in a single function (the "kernel") whose only differing input parameter across threads is the thread index (which can be thought of, loosely, as the processor index).

4 Implementation Notes: QS on GPUs

The only other reference to quadratic sieves on the GPU parallelizes a multiple polynomial sieve by dispatching one polynomial per thread. Unfortunately, this technique does not scale infinitely - MPQS suffers from diminishing returns after a few polynomials.
For this project, I partitioned the work by dividing the interval into several smaller intervals. Good performance can be achieved if the subinterval length is a multiple of 32, since then the global memory reads are coalesced (GPUs achieve high bandwidth via parallel memory accesses on a very wide bus).
The core kernel does some partitioning precomputation, followed by a vector add of $\log p$ to a buffer that is initialized at 0, at the positions corresponding to

the solutions of $x^2 - n \equiv 0 \bmod p$. Another kernel does the precomputation of base solutions. Matrix math is currently handled using MSieve's Lanczos solver; a GPU matrix step will be implemented later on, as it is a lengthy, independent project of its own.