Chia VDF Competition: Largest Discriminant Break

30 Dec, 2018

Akashnil Dutta

Prerequisite

Please read the Binary Quadratic Forms document [1] for background on class groups.

Baby Step Giant Step algorithm

My submission is based on the baby-step-giant-step algorithm, which is provided with the sample submission. First, I'll describe a special case of the baby-step-giant-step algorithm. Then I'll discuss a series of small and big improvements in the following sections.

For a given discriminant $-8k+1$, suppose we know that the order is in the range $(a, a+b)$ where $a >> b$. We take an arbitrary element $g = (2, 1, k)$. Let $u = \lfloor \sqrt{b} \rfloor$. We can calculate $g, g^2, g^3, \cdots g^u$ and store them in a hash table. Let $f = g^u$ Then we can calculate $h = g^a$ using repeated sqauring, and find $h * f, h * f^2 \cdots h * f^u$. One of these values must match up with $g, g^2, g^3, \cdots g^u$ which are stored in the hash table. Hence the Baby Step Giant Step algorithm takes time $O(\sqrt{b})$ time to find order within the range $(a, a + b)$.

Using hashing and symmetry

When b is very large one of the limitations is the amount of memory to store the values $g, g^2, g^3, \cdots g^u$. To be most efficient, we only store a 64 bit hash of each element $g^i = (a, b, c)$ and the index i. When a hash collision is found later, the element g^i is recalculated to verify equality.

One more thing to note is that $g^{-i} = (a, -b, c)$ whenever $g^i = (a, b, c)$. So we can get more elements in the baby steps for free by mapping $g^i \to \text{hash}(a, |b|, c)$. Furthermore, we know that the order is odd, so only odd powers of g will suffice in the baby steps. Suppose the order is known to be in range $(2a - 4b, 2a + 4b)$ In this case we can store the hashes of $u = \sqrt{b}$ elements $g, g^3, g^5, \cdots g^{2u-1}$ in the baby steps and only search for $g^{2a}, g^{2a} * g^{\pm 4u}, g^{2a} * g^{\pm 8u}, \cdots, g^{2a} * g^{\pm 4u^2}$. Using these symmetries allow us to search 8 times larger range of orders while calculating only 1.5 times the number of elements ($3u$ rather than $2u$).

Approximating the order

In general we know that the order of the class-group is $O(\sqrt{p})$ where p is the absolute value of the discriminant. However for specific values of p, the order can vary a lot. In this section we'll describe an way to (very quickly) approximate the order with high accuracy.

Rather than finding equalities in the classgroup, another alternative way to find its order is to count the number of reduced forms (a, b, c). Recall that a reduced form satisfies $c > a > |b|$. We can count the number of triples (a, b, c) for each value of a satisfying $b^2 - 4ac = -p$. In this case $b^2 \equiv -p \mod a$, in other words, the discriminant must be a quadratic residue modulo a. Suppose a is a prime-power, then there are exactly two values of b for which $b^2 \equiv -p \mod a$, one of them is b and the other $-b$. If $-p$ isn't a quadratic residue mod a, then there is no value of b. If a is composite, then $-p$ must be a residue for each prime dividing a. The number of different values of b modulo a is 2^k where k is the number of distinct prime factors of a (this follows from the chinese remainder theorem).

The results in the above paragraph hold as long as (a, b, c) is reduced. If b is obtained from a by taking the appropriate residue, (a, b, c) is always reduced if $a < \sqrt{p}/2$, sometimes reduced if $a < \sqrt{p/3}$ and never reduced if $a > \sqrt{p/3}$. This follows from

$$a < c = \frac{p + b^2}{4a} \iff a < \sqrt{p + b^2}/2$$

For a moment let's ignore the complicated case when $\sqrt{p}/2 < a < \sqrt{p/3}$. For a prime q, let $F_p(q) = 2$ if $-p$ is a quadratic residue $\mod q$, else $F_p(q) = 0$, for composite q, let $F_p(q) = 1$. We wish to find

$$\mathrm{Ord}(p) \approx \sum_{n=2}^{\sqrt{p}/2} \prod_{q|n} F_p(q)$$

Now $\sqrt{p}$ is very large so we cannot evaluate the sum directly. We need an way to approximate this. An alternative way to estimate the sum is to count the effect of each prime q on the sum. Start with setting each term of the sum equal to 1. If $-p$ is a residue mod q, every qth term in the sum will be multiplied by 2, the overall sum will be multiplied by approximately $1 + 1/q$ (assuming every q-th term is statistically similar to the other terms on average). If $-p$ is a non-residue modulo q, then every q-th term will be equal to zero, the overall sum will be multiplied by approximately $1 - 1/q$.

Using the heuristic arguments of the above paragraph we arrive at the approximate formula

$$\mathrm{Ord}(p) \approx \frac{\sqrt{p}}{2} \prod_{q=2}^{k(?)} \left(1 + \frac{F_p(q) - 1}{q}\right) = \mathrm{Aprx}_k(p)$$

The more terms are used in the product, the more accurate the approximation gets. Note that it is very cheap to find $F_p(q)$, i.e. determine when u is a quadratic residue mod q:

$$\left(\frac{u}{q}\right) \equiv u^{\left(\frac{q-1}{2}\right)} \mod q$$

Here the LHS denotes Legendre symbol.

The formula isn't very rigorous, this works surprisingly well in practice. Although I haven't proven it, I'm quite confident that the following converges

$$\lim_{p\to\infty} \lim_{k\to\infty} \frac{\mathrm{Ord}(p)}{\mathrm{Aprx}_k(p)} = \lambda \approx 1.0471975 \text{ (experimentally)}$$

This value is not equal to 1 because we ignored the cases $\sqrt{p}/2 < a < \sqrt{p/3}$ earlier. Assuming that quadratic residues are distributed uniformly randomly for different primes, they will statistically average out to some constant for large values of p. From experiments, the speed of the convergence is found to be about $O(1/\sqrt{k})$ error for large values of p and k.

Algorithm overview

The algorithm proceeds in 4 stages.

1. For a given discriminant size, we spend some time generating many prime discriminant candidates. We approximately find the order of each (using only minuscle amount of time, it's easy to approximate the order to 1%). We select for 3 good candidates by keeping the ones with smallest approximate order, and discard the rest.

2. In the second stage, we find the approximate order using many primes (about $p^{1/5}$ of them). In this stage it is possible to narrow down the order within a factor of 1 ± 10^{-6} when $p \approx 10^{50}$. I'm assuming this is $O(p^{-1/10})$.

3. In the 3rd stage, we have the order narrowed down to $(a \pm b)$ where $a = O(p^{1/2})$ and $b = O(p^{2/5})$. Then we calculate $O(p^{1/5})$ baby steps in the third stage and populate the hash table in memory.

4. In the 4th stage, we keep searching with giant steps until we find hash equal to a baby step, in which case we verify if we indeed found the order. This step will go on until the order is found, in practice this takes $O(p^{1/5})$ steps as well.

In stage 1, 2, 3, we have freedom to choose how much time we'll spend. If we explore for more candidates in stage 1, we'll get a smallest order, in stage 2, if we search with more primes, we get a better approximation, hence faster search in stage 4. If we generate more baby steps in stage 3, we also need to search less in stage 4. They are approximately inversely proportional. So the amount of time spent in each stage was chosen to be approximately equal for best performance. However stage (3) is also limited by memory available. So for extremely large discriminants, (> 170 bits) we have to limit how many baby steps we calculate. Depending on how many CPUs are available for parallelism, the relative time spent in each phase should be changed too. Hence the parameters need to be set correctly for the target hardware for optimal performance.

If my assumptions are correct, the overall complexity of the algorithm is $O(p^{1/5})$ time and space.

Optimizing class-group computations

In our algorithm the main step that take a lot of time is the composition algorithm in the class-group, which is used in stage 3 and 4. We have implemented this part in C++, and it is possible to parallely generate many classgroup elements. We manage the parallelism using seperate C++ processes from a python script by dividing the work into appropriately sized chunks.

An interesting observation is that the classgroup composition algorithm becomes simpler for the special case $(a, b, c) * (2, 1, k)$. Let $\beta = (b - 1)/2$.

If β odd, a odd, or β even, c odd:

$$a' = 2a, b' = b + 2a, c' = \frac{a + b + c}{2}$$

If β odd, a even:

$$a' = a/2, b' = b, c' = 2c$$

If β even, c even:

$$a' = 2a, b' = b, c' = c/2$$

This must be followed by the usual reduction steps. In this case there can be at most one or two reduction steps because the coefficients only grow by a small amount.

Another simplification is the classgroup composition algorithm when $(a_1, b_1, c_1)*(a_2, b_2, c_2)$ satisfy $\gcd(a_1, a_2) = 1$. In this case all we have to do is solve the system of linear congruences

$$\begin{cases} b \equiv b_1 \mod a_1 \\ b \equiv b_2 \mod a_2 \end{cases}$$

This has a unique solution mod $a_1 a_2$ using chinese remainder theorem. The solution is of the form $b_1 + k a_1$ where $k \equiv a_1^{-1}(b_2 - b_1) \mod a_2$. $a_1^{-1} \mod a_2$ can be quickly found using the extended GCD algorithm. After $a = a_1 a_2$ and b is calculated, c can be found using the discriminant formula. Then the resulting form must be reduced.

Because the classgroup multiplication is simplified when $\gcd(a_1, a_2) = 1$, we deliberately choose the number of baby steps u so that $g^{4u} = (a, b, c)$ has a prime. Then the giant steps are all multiplications by $g^{4u} = (a, b, c)$ which become faster because a is prime.

Faster Reduction algorithm

After multiplication or squaring in the classgroup, the form must be reduced. This takes up a lot of time. Recall that the standard reduction algorithm is the following ([2] Section 3.1)

- Start with (a,b,c). Calculate $\delta = \lfloor \frac{b+c}{2c} \rfloor$
- Set $(a, b, c) := (c, -b + 2c\delta, a - b\delta + c\delta^2)$
- if $a > c$ repeat from step 1.

If a, b, c are large integers, step (2) takes up a lot of time, multiplying and adding large integers. An interesting observation is that even with approximate values of (a, b, c) the reduction algorithm can proceed in several steps. We just need to keep track of the coefficients by which (a, b, c) are getting multiplied, and apply them all at once.

Instead of working with (a, b, c), we find the high order 64 bits of each a, b and c, apply the reduction steps until $a \leq c$. When one stage terminates, we multiply all the matrices that acted on (a, b, c) (Find the formulas necessary in [2] Section 2), apply the resulting matrix on the true (full precision) vector (a, b, c), repeat until reduced.

This is the only significant improvement applied in my submission to track 1 as well. In that case this is even more significant because the reduction steps take up most of the time for 2048 bit squaring. The number of reduction steps are reduced from 200 to 25 stages by using this improvement. The reduction takes 5x less time, which is now 60% of the overall runtime, GCD still takes the same amount of time, which is now 35% of total. The overall runtime is improved about 3x.

References

[1] Lipa Long. *Binary Quadratic Forms.*

`https://github.com/Chia-Network/vdf-competition/blob/master/classgroups.pdf`

[2] Nicole Miller. *The Structure of the Class Group of Imaginary Quadratic Fields.*

`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.590.2666&rep=rep1&type=pdf`