Solving Discrete Logarithms in Smooth-Order Groups with CUDA[1]

Ryan Henry Ian Goldberg

Cheriton School of Computer Science
University of Waterloo
Waterloo ON Canada N2L 3G1

{rhenry,iang}@cs.uwaterloo.ca

Abstract

This paper chronicles our experiences using CUDA to implement a parallelized variant of Pollard's rho algorithm to solve discrete logarithms in groups with cryptographically large moduli but smooth order using commodity GPUs. We first discuss some key design constraints imposed by modern GPU architectures and the CUDA framework, and then explain how we were able to implement efficient arbitrary-precision modular multiplication within these constraints. Our implementation can execute roughly 51.9 million 768-bit modular multiplications per second — or a whopping 840 million 192-bit modular multiplications per second — on a single Nvidia Tesla M2050 GPU card, which is a notable improvement over all previous results on comparable hardware. We leverage this fast modular multiplication in our implementation of the parallel rho algorithm, which can solve discrete logarithms modulo a 1536-bit RSA number with a 2^{55}-smooth totient in less than two minutes. We conclude the paper by discussing implications to discrete logarithm-based cryptosystems, and by pointing out how efficient implementations of parallel rho (or related algorithms) lead to trapdoor discrete logarithm groups; we also point out two potential cryptographic applications for the latter. Our code is written in C for CUDA and PTX; it is open source and freely available for download online.

1 Introduction

Over the past several decades, the algorithms and symbolic computation research communities have made considerable advances with respect to state-of-the-art algorithms for solving many number-theoretic problems of interest. At the same time, Moore's law has ensured steady speed increases in the computing devices on which these algorithms run. The results have been astounding: modern symbolic computation packages, such as Maple[2] and Mathematica[3], accept arbitrary-precision operands and can solve a plethora of useful problems very efficiently. Nonetheless, much work remains; there exist many fundamental problems for which no efficient (i.e., polynomial-time) algorithm is known. While there is considerable interest in expanding the range of problems that a modern symbolic computation toolkit can solve efficiently, it turns out that there are also practical advantages to having some problems remain intractable, specifically when the "inverse problem" is efficient to compute. In particular, such *one-way functions* give rise to public-key cryptosystems, such as the ones that protect our everyday online transactions.

In practice, nearly all public key cryptosystems derive their security guarantees from assumptions about (possibly relaxations of) one of the following three types of presumed 'hard' problems: those related to

[1] The latest version of this paper is available as CACR Tech Report 2012-02, http://cacr.uwaterloo.ca/techreports/2012/cacr2012-02.pdf.

[2] http://www.maplesoft.com/

[3] https://www.wolfram.com/mathematica/

factoring large integers, those related to computing discrete logarithms, or those related to solving certain computational problems on integer lattices. This paper is concerned with a variant of the second problem on this list; i.e., that of computing discrete logarithms when the modulus is large but has smooth totient (more precisely, it focuses on computing discrete logarithms in the multiplicative group of units modulo N when the group order $\varphi(N)$ is B-smooth for some $B \ll N$; that is, when all of the prime factors of $\varphi(N)$ are less than B). In particular, we explore the extent to which one can leverage the massive, yet cost-effective, parallelism provided by modern *general-purpose graphics processing units* (GP GPUs) to solve discrete logarithms (with respect to certain special moduli) that would otherwise be impractical to solve using CPUs alone. We also discuss the implications of being able to solve such discrete logarithms for existing discrete logarithm-based cryptosystems, and demonstrate how this ability gives rise to a useful cryptographic primitive called a *trapdoor discrete logarithm group*.

Outline. Before commencing our foray into parallel programming on GPUs with CUDA and PTX in §3, we first overview related work in the literature in §1.1, and then briefly touch on some mathematical preliminaries, including discrete logarithms and the parallel rho method for computing them, in §2. The key to an efficient realization of parallel rho on GPUs is fast modular multiplication; therefore, we devote much of §4 to the implementation details of efficient arbitrary-precision modular multiplication in CUDA, including various optimizations that lead to dramatic performance improvements over a naive implementation. §4 also discusses how we tailored the parallel rho method to run well within the hardware constraints of CUDA GPUs. A performance evaluation of our implementation appears in §5; based on these empirical observations, as well as some theoretical analysis, §6.1 discusses the implications for deployed discrete-logarithm-based cryptosystems, while §6.2 concludes that GPU-based implementations of parallel rho are sufficient to realize trapdoor discrete logarithm groups, and suggests some possible cryptographic applications. §7 wraps up with a brief summary and a pointer to our open source implementation.

1.1 Related work

In recent years, there has been considerable interest in using commodity graphics processing units (GPUs) to perform highly parallelized computations at a low cost, especially for use in speeding up (and attacking) public key cryptosystems. Because GPUs are particularly well suited to solving systems of linear equations, it should be unsurprising that several high-speed implementations of lattice-based cryptosystems have successfully employed them. For example, Hermans et al. [19] implemented NTRUEncrypt — the encryption function for the NTRU cryptosystem — in CUDA and ran it on an Nvidia GTX 280 graphics card at a record-breaking throughput of 200,000 encryptions per second with a 256-bit security level. Aguilar et al. [1] ported their single-server lattice-based private information retrieval (PIR) scheme to run on GPUs; in all of their experiments, Aguilar et al. observed an 8x–9x improvement in throughput when compared to throughput on a system composed of similarly priced CPUs. Several other research groups have obtained promising results using GPUs to perform modular exponentiations [15, 17, 27, 28], an operation that forms the basis of many number-theoretic public key cryptosystems. Harrison and Waldron [17], for example, report a 4x throughput increase using GPUs instead of comparably priced CPUs for the special case of computing 1024-bit modular exponentiations. More recently, Neves and Araujo [28] obtained similar positive results by implementing arbitrary-precision modular exponentiations in CUDA. The present paper focuses on leveraging GPUs to do the inverse of modular exponentiation; i.e., to solve instances of one variant of the so-called discrete logarithm problem.

Perhaps the most relevant prior work along these lines is that of Bailey et al. [2]; they describe their efforts to use several clusters of conventional computers, PlayStation 3 consoles, powerful graphics cards, and FPGAs to break the Certicom ECC2K-130 challenge [13]. (A more recent paper [4] further elaborates on how the team has optimized their implementation for efficient binary field arithmetic on commodity GPUs.)

As in the present work, Bailey et al. use the parallel rho method to solve discrete logarithms; however, our efforts differ in that the ECC2K-130 challenge involves solving discrete logarithms in an elliptic curve over a binary field, whereas this work considers the problem of solving discrete logarithms in a special class of multiplicative groups. The latter setting is quite different since it involves computing modular arithmetic with a very large modulus, rather than computing binary field elliptic curve arithmetic. At the time of writing, the team's efforts to break ECC2K-130 are still underway; the interested reader should consult `http://ecc-challenge.info/` for nearly real-time progress updates. Other groups [7, 8] have obtained positive results using game consoles — specifically, the Cell-based PlayStation 3 — to solve discrete logarithms over elliptic curves using the parallel rho method, but so far no other group has reported positive results using commodity hardware to solve discrete logarithms in the setting considered in this work.

The current state of the art with respect to fast modular multiplications on GPUs appears to be Bernstein et al.'s work on 'The Billion-Mulmod-Per-Second PC' [5]. The authors of that work managed to obtain an impressive 481 million 192-bit modular multiplications per second on an Nvidia GTX 295 graphics card. The Nvidia GTX 295 has 480 cores that each run at 1.2 GHz; thus, their implementation achieves a per-core throughput of about 1 million 192-bit modular multiplications per second, which is about one modular multiplication per 1200 clock pulses on each core. The experiments considered in this paper use two Nvidia Tesla M2050 cards, which each have 448 cores that run at 1.55 GHz. Our implementation computes roughly 840 million 192-bit modular multiplications per second on each one of these cards — a per-core throughput of about 1.875 million 192-bit modular multiplications per second, which is about one modular multiplication per 830 clock pulses on each core.[4] Several other groups have also implemented efficient modular multiplication in CUDA; unfortunately, the source code for most of these implementations is not publicly available, thus preventing their numerous "speed records" from being independently replicated or verified. To avoid such shortcomings in our own work, all of our source code is open source and freely available for download from `http://crysp.uwaterloo.ca/software/`.

2 Mathematical preliminaries

This section provides a terse overview of the discrete logarithm problem and Pollard's rho method [34] for computing discrete logarithms, as well as van Oorschot and Wiener's approach [40] to parallelizing Pollard's rho. It also briefly discusses Pollard's $p-1$ factoring algorithm [33], which will be relevant to the discussion in §6.2. We begin with a formal statement of the discrete logarithm problem.

Definition 1 (Discrete logarithm problem [25, §3.6]). *Given a finite, cyclic group $\mathbb{G}$ of order n, a generator g of $\mathbb{G}$, and an arbitrary group element $\alpha \in \mathbb{G}$, the **discrete logarithm problem** is to find the unique integer exponent x in the interval $[0, n-1]$ such that $g^x = \alpha$. The exponent x is the **discrete logarithm** of α with respect to g in $\mathbb{G}$.*

Our focus in this work is on solving discrete logarithms in cyclic subgroups $\mathbb{G}$ of the multiplicative group of units modulo N. The parallel rho method has fallen out of fashion for computing discrete logarithms in this setting, with more specialized algorithms such as index calculus being preferred [25, §3.6.5]; nonetheless, in the special case where the group order n has only small factors, Pollard's rho may dramatically outperform

[4] Fundamental differences exist between the Nvidia GTX 295 graphics cards that Bernstein et al. used and the Nvidia Tesla M2050s used in this work. The latter cards are *general-purpose* GPUs, rather than standard graphics cards like those in the GTX series; as such, they possess certain characteristics that make them more suitable for general-purpose computations (for example, computing modular multiplications). Most significantly, the cards in the Tesla series have more memory than those in the GTX series do, and the Tesla M2050 can perform true 32-bit multiplication in hardware, whereas the GTX 295 uses a sequence of 24-bit multiplications to simulate hardware support for 32-bit multiplications. Thus, comparing the relative performance of the two implementations requires a more nuanced approach than simply comparing per-core throughputs; we omit a more meaningful comparison, as such a comparison is not the goal of this work.

index calculus, and it is on this special case that we focus our attention. We do note, however, that Pollard's rho method *is* currently the standard technique for computing discrete logarithms on elliptic curves [4] and many standard texts (e.g. [25, §3.6.3] or [14, §31.9]) therefore cover it in some depth. We briefly describe the algorithm below; however, the interested reader is encouraged to consult one of the aforementioned texts for a more thorough description of the algorithm and analysis of its runtime.

Pollard's rho method. Pollard's rho algorithm is essentially just a clever way to exploit the well-known birthday paradox. In a nutshell, the birthday paradox tells us that, on average, one needs only select about $\sqrt{\pi n/2}$ random elements from a set of n alternatives (with replacement) before encountering a collision (wherein a previously selected element is selected again). This fact allows one to compute the discrete logarithm x of $h \in \mathbb{G}$ with respect to g by repeatedly selecting random exponents $a_i, b_i \in_R [0, n-1]$ to obtain random group elements $g^{a_i} h^{b_i} \in \mathbb{G}$. After sufficiently many such random selections, a collision will occur; in particular, the process eventually yields two triples $(a_{i_1}, b_{i_1}, g^{a_{i_1}} h^{b_{i_1}})$ and $(a_{i_2}, b_{i_2}, g^{a_{i_2}} h^{b_{i_2}})$ such that $g^{a_{i_1}} h^{b_{i_1}} = g^{a_{i_2}} h^{b_{i_2}}$ and $b_{i_1} \not\equiv b_{i_2} \bmod n$, whence it follows that $a_{i_1} + b_{i_1} x \equiv a_{i_2} + b_{i_2} x \bmod n$, and therefore $x = (a_2 - a_1)(b_1 - b_2)^{-1} \bmod n$. The birthday paradox tells us that an expected number of $\sqrt{\pi n/2}$ random selections suffice to find such a collision; thus, the natural algorithmic instantiation of this process solves the discrete logarithm problem with an expected runtime in $\Theta(\sqrt{n})$, and uses $\Theta(\sqrt{n})$ storage.

Pollard's big idea was to modify the above observation to find the collisions without having to store $\Theta(\sqrt{n})$ such triples. To do this, he proposed using a function $f : \mathbb{G} \to \mathbb{G}$, called an *iteration function*, that is chosen so that 1) it is efficient to compute, 2) it behaves heuristically like a randomly selected mapping from $\mathbb{G}$ to itself, and 3) it maps a group element $g^{a_1} h^{b_1}$ to $g^{a_2} h^{b_2}$ in such a way that a_2 and b_2 are easy to compute from a_1 and b_1. The actual instantiation for f that Pollard proposed is

$$f(x) = \begin{cases} h\,x & \text{if } 1 \leq x < \frac{N}{3}, \\ x^2 & \text{if } \frac{N}{3} \leq x < \frac{2N}{3}, \text{ and} \\ g\,x & \text{if } \frac{2N}{3} \leq x < N, \end{cases} \tag{1}$$

which is essentially the same function that we use in our implementation.[5] The algorithm then proceeds by starting with a random group element $g^{a_0} h^{b_0}$ and iteratively applying f to select the subsequent "random" group elements. It is easy to see that when a collision eventually occurs (after an expected $\sqrt{\pi n/2}$ iterations, assuming perfectly random behaviour of f), the subsequent iterations of the process form a cycle, which can be detected using a cycle finding algorithm such as that of Floyd [16] or Brent [10]. Moreover, the group element $g^{a_i} h^{b_i} = f^{(i)}(g^{a_0} h^{b_0})$, where $f^{(i)}(\cdot)$ denotes that f is iteratively applied to the operand i times, is easily expressed in terms of g and h, so once a cycle (hence, collision) is found, one can use it to compute the discrete logarithm as above.

The parallel rho method. Regrettably, Pollard's rho method as presented above does not parallelize well. The reason for this is that iterative application of f is an inherently serial process that each thread of execution must perform independently. It turns out that invoking the procedure Ψ times in parallel yields sublinear expected speedups (in particular, the expected speedup is proportional to $\sqrt{\Psi}$ rather than Ψ). Van Oorschot and Wiener [40] proposed an ingenious way to bypass this limitation using the notion of *distinguished points*. (A distinguished point is simply some group element that has an easily testable property, such as a certain

[5] Teske [39] subsequently proposed a better choice for f, which can reportedly reduce number of iterations by $\approx 20\%$ on average. It is possible that switching to Teske's iteration function would lead to speedups in our implementation, although we suspect that the overhead associated with evaluating the more complicated function on a GPU would increase the cost of an iteration so much as to negate any purported performance increases. Nonetheless, it would be interesting and worthwhile to experiment with Teske's alternative iteration function (or some middle ground between Teske's function and Pollard's function) as a direction for future work.

number of trailing zeros in its binary representation.) Under their regime, one thread acts as a server and Ψ threads act as clients; each client thread starts the iteration process at a different random group element with known representation in terms of g and h and iterates until it hits a distinguished point. When a client thread encounters its first distinguished point $g^{a_i}h^{b_i} = f^{(i)}(g^{a_0}h^{b_0})$, it sends the triple $(a_i, b_i, g^{a_i}h^{b_i})$ to the server thread and starts the iteration process anew with a fresh random group element. Upon receiving a triple $(a_{i_1}, b_{i_1}, g^{a_{i_1}}h^{b_{i_1}})$ from some client such that $g^{a_{i_1}}h^{b_{i_1}} = g^{a_{i_2}}h^{b_{i_2}}$ and $b_{i_1} \not\equiv b_{i_2} \bmod n$ for some previously received triple $(a_{i_2}, b_{i_2}, g^{a_{i_2}}h^{b_{i_2}})$, the server computes the discrete logarithm just as it did before. By the same observation used above, if two threads ever encounter a collision (be it at a distinguished point or not), then all subsequent iterations of those two threads necessarily follow identical trails; thus, the next distinguished point that either thread encounters is also necessarily a collision. In particular, each of the Ψ threads is searching for collisions with any group element encountered by any other thread, and the expected speedup becomes linear in Ψ.

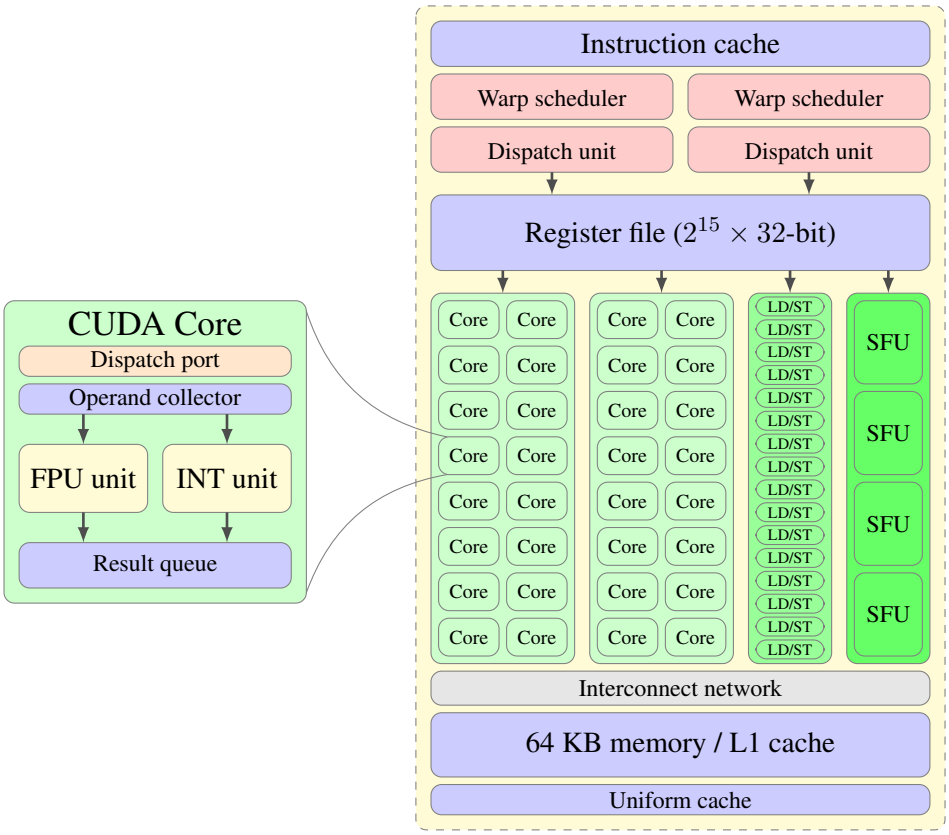
Pollard's $p - 1$ method. Pollard's $p - 1$ factoring algorithm is a special-purpose factoring algorithm that uses Fermat's Little Theorem to find certain, special factors of an integer. The key observation behind the technique is that working in the multiplicative group of units modulo n is equivalent to working in the multiplicative groups of units modulo each of n's prime-power factors. Moreover, Fermat's Little Theorem says that if $\gcd(g, p) = 1$ for a prime p, then $g^{k(p-1)} \equiv 1 \bmod p$ for all k, hence $p \mid \gcd(g^{k(p-1)} - 1, n)$. This suggests the following algorithm for finding prime factors p of n, subject to the condition that all of the prime factors of $p - 1$ are bounded above by some *smoothness* bound B (in such a case, $p - 1$ is called B-smooth): Fix a positive integer B and compute $x = \prod q^{\lfloor \log_q B \rfloor}$, where the product is taken over all primes q less than B, then compute and return $d = \gcd(g^x - 1, n)$. If $1 < d < n$, then d is a product of all prime factors p of n for which $p - 1$ is B-smooth. A simple modification involves progressively increasing B in the computation and computing the gcd at each step to recover the individual prime factors (rather than their product). Note that this procedure requires between $B/\ln 2$ and $1.5B/\ln 2$ modular multiplications, depending on whether modular exponentiations are performed with naive square-and-multiply or a somewhat more efficient algorithm.

3 GPU programming with 'C for CUDA' and PTX

To meet the demands of increasing screen resolutions, frame rates, and scene complexity seen in today's video games, modern graphics cards have evolved into extremely powerful computing platforms that leverage a large degree of parallelism compared to regular CPUs. This has led to much interest in harnessing the power of GPUs as massively parallel co-processors working alongside regular CPUs in applications outside of graphics processing. To facilitate such uses, Nvidia has developed the *Compute Unified Device Architecture* (CUDA) parallel computing platform and programming model and the *Parallel Thread eXecution* (PTX) instruction set architecture, which together form the basis for their GeForce (for consumer PCs), Quadro (for professional workstations), and Tesla (for high-performance general-purpose computing) lines of GPU devices [29].

Nvidia GPUs. The architecture of Nvidia GPU devices exhibit fundamental differences from most CPU-based systems, and effectively utilizing their computational power necessitates an understanding of these differences. §1.1.1 of Nvidia's CUDA C Best Practices Guide [31] describes the most important differences, which mostly pertain to how GPU devices handle threading and memory access; for completeness, we summarize the key architectural features of CUDA GPU devices.

A typical Nvidia GPU contains several *streaming multiprocessors* (SMPs), each of which consists of several *streaming processors* (SPs) and *special function units* (SFUs), an instruction decoder, and some



(a) An exploded view of a single CUDA core.

(b) The internal structure of one SMP in the Fermi architecture.

Figure 1: The internal structure of a streaming multiprocessor (SMP) in the Fermi architecture and an exploded view of a single streaming processor (SP), or "CUDA core". This diagram is adapted from Nvidia's documentation [30].

shared memory. The SPs are known colloquially as *CUDA cores*. The Tesla M2050 cards that we use in our experiments are based on the Nvidia Fermi architecture, which has 32 CUDA cores and 4 SFUs per SMP (the M2050 itself is comprised of 14 SMPs). Each SMP in the GPU is capable of executing a single instruction at a time, which means that the 32 CUDA cores in that SMP must each execute the same instruction simultaneously, albeit on different data (this is called *single instruction/multiple data* or SIMD). Nvidia calls a bundle of 32 threads executing in parallel on an SMP a *warp*, which constitutes the smallest executable unit of parallelism on a CUDA device. All Nvidia GPUs can support at least 24 active warps (768 active threads) per SMP — and some higher-end GPUs can support 32 active warps per SMP — where each warp has its own set of registers; once the GPU allocates registers to a warp, those registers stay allocated to that warp until it finishes execution. This makes threads on the GPU extremely lightweight compared to their counterparts on a regular CPU. An application can queue up thousands of threads and when the GPU must wait on one warp of threads, it simply begins executing work (at the next clock pulse) on another warp of threads, with no intervention from the host device and no swapping of register state. However, getting good performance out of threads in CUDA still requires the software developer to keep several caveats in mind. For example, if there is a conditional branch and some threads in a warp take this branch while others do not (called *warp divergence*), then the other threads will just idle until the branch is complete and they

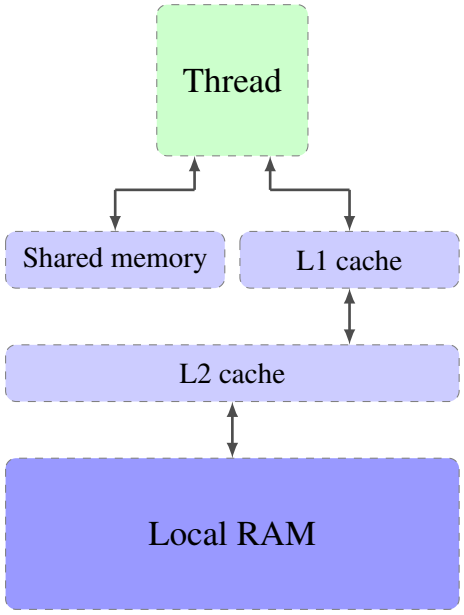


Figure 2: The memory hierarchy in CUDA GPU devices based on the Nvidia Fermi architecture. This diagram is adapted from Nvidia's documentation [30].

all converge back together on a common instruction. The situation is even worse when two or more threads from a warp each take a a different conditional branch: only one branch is executed at a time, and the overall execution time becomes the *sum* of the execution times of each branch taken (rather than the maximum execution time across all branches, as one might intuitively expect).

Figure 1 illustrates the structure of SMPs in the Nvidia Fermi architecture; 1(a) gives an exploded view of a single CUDA core within the SMP, while 1(b) shows the internal structure of the entire SMP. Each CUDA core resembles a regular CPU core but is much simpler, reflecting its heritage as a pixel shader. It has a pipelined floating-point unit (FPU), a pipelined integer unit (INT), some logic for dispatching instructions and operands to these units, and a queue for holding results, but it lacks its own general-purpose register file, L1 cache, function units for each data type, and load/store units for retrieving and saving data.

The other important difference between CUDA GPU devices and CPU-based systems is the memory hierarchy. Memory on the GPU is segmented, both physically and virtually, into several different types, each of which has its own special purpose and performance characteristics. For one thing, the GPU has a very large frame buffer, which Nvidia calls *local RAM*, on which application developers can store their data; the local RAM is further subdivided into read-write *global memory*, read-only *constant memory*, and read-only *texture memory*. There is also a small *shared memory* and some L1 cache that is local to each CUDA core, and an L2 cache that all SMPs on the GPU device share. Figure 2 illustrates the memory hierarchy in CUDA GPU devices based on the Nvidia Fermi architecture. Carefully managing these different types of memory is important, as the latencies experienced when a thread reads from or writes to memory depends on that memory's proximity to the CUDA core running the thread. In the extreme case of fetching data from the host device's memory, these data must travel along the PCIe bus to get to the GPU device, thus incurring extremely high latencies and throughputs that are an order of magnitude or more slower than fetching data from memory on the device. According to Nvidia's documentation [30], access to local RAM requires 200 – 300 clock cycles, while access to on-chip memory (registers, shared memory, L1 cache) requires only one clock cycle. Thus, when reading blocks from (or writing blocks to) local RAM or memory on the host device, it is imperative to use a *coalesced* access pattern, wherein the blocks occupy consecutive memory addresses;

this allows CUDA to batch many small transfers into a single larger transfer. To facilitate effective use of shared memory, cache, and local RAM, CUDA-enabled programming languages such as 'C for CUDA' (see below) include new variable-type qualifiers (__device__, __constant__, and __shared__), allowing programmers to specify where to store the data referenced by a variable.

C for CUDA. Software developers can use CUDA-enabled variants of several industry-standard programming languages to access the virtual instruction set and memory of the parallel computing elements in CUDA GPUs. The most common CUDA-enabled language, and the one used in this work, is a variant of C with some additional Nvidia extensions called 'C for CUDA'. CUDA applications are partitioned into completely encapsulated GPU *kernels* with C statements interleaved; the kernels are executed on the GPU and the C statements on the host CPU. Function-type qualifiers (analogous to the aforementioned variable-type qualifiers) specify where a function should run: the __host__ function-type qualifier specifies that the host device both invokes and runs the function; the __global__ function-type qualifier specifies that function is a kernel, meaning that the host device invokes the function, but it runs on the GPU device; and the __device__ function-type qualifier specifies that code on the GPU device invokes the function and the function runs on the GPU. CUDA imposes a two-tier hierarchical structure on its threads, which the application developer specifies using a new $\lll \cdots, \cdots \ggg$ syntax; groups of threads form *thread blocks*, and groups of thread blocks form a *thread grid*. For example, an invocation of the form $\texttt{kernel}\lll \texttt{1}, \texttt{N} \ggg (\cdots)$; executes kernel N times in parallel by N different threads, where each of these threads has a unique *thread ID* that is accessible within the kernel through the built-in threadIdx variable. Threads within a block always run on a single CUDA core, which ensures that synchronization and cooperation between threads within a block is inexpensive, whereas different thread blocks may run on different MPs and therefore run independently. This design simplifies scaling, since it enables GPUs with more SMPs to process more blocks in parallel without requiring changes to the program or kernel configuration. Nvidia's nvcc compiler translates C for CUDA device source code into device-independent PTX code.

The PTX ISA. PTX is a device-independent pseudo-assembly language for CUDA GPU devices. It provides a means for software developers to make fine-grained optimizations to their code before the ptxas compiler converts it into the final device-specific binary file, which is later loaded and executed on the GPU. PTX exposes several useful instructions that the nvcc compiler fails to utilize; most relevant to our implementation of modular multiplication are the instructions for add-with-carry-in and optional carry-out (ADDC{.cc}), subtract-with-borrow-in and optional borrow-out (SUBC{.cc}), and the fused integer multiply-and-add instruction (MAD{.hi, .lo, .wide}). The latter instruction enables our implementation to multiply two 32-bit unsigned integers, and then add a third 64-bit integer, placing the full 64-bit result in a 64-bit register. The PTX ISA also gives developers some control over the allocation and use of registers, which is helpful in minimizing unnecessary copying of register values when the input to one instruction is the output of some earlier instruction.

4 Arbitrary-precision modular multiplication and parallel rho on GPUs

Montgomery multiplication. Our implementation of arbitrary-precision modular multiplication uses the well-known Montgomery multiplication and reduction techniques [26]. We briefly recall how ordinary Montgomery multiplication and reduction work, before discussing the coarsely-integrated operand scanning (CIOS) algorithm for modular Montgomery multiplication, which is the variant that we found to give the best performance in our CUDA implementation.

Let N be a fixed, odd k-bit integer, let $R = 2^k$ (so $2^{k-1} < N < 2^k$ and $R > N$ with $\gcd(N, R) = 1$) and let x and y be two integers in the range $[0, N-1]$. Montgomery multiplication allows for the computation

of $xy \bmod N$ without explicitly carrying out the costly classical modular reduction step. To do this, the multiplicands x and y must first be "Montgomerized" to N-*residues*: $\tilde{x} = xR \bmod N$ and $\tilde{y} = yR \bmod N$. The Montgomery multiplication algorithm computes the N-residue $\tilde{z}$ of $z = xy$ from $\tilde{x}$ and $\tilde{y}$, which turns out to be much faster than computing $xy \bmod N$ directly from x and y (because modular reduction and division by $R = 2^k$ in binary reduces to truncation and rightward bit-shifts). Define R' ($= R^{-1} \bmod N$) and N' to be integers that satisfy Bézout's identity [20, §1.2], $R \cdot R' + N \cdot N' = 1$; these values are easily computed with the extended Euclidean algorithm [25, §2.4]. The Montgomery product of $\tilde{x}$ and $\tilde{y}$ is

$$\begin{aligned}\tilde{z} &= \tilde{x}\tilde{y}R' \bmod N \\ &= (xR)(yR)R^{-1} \bmod N \\ &= (xy)R \bmod N.\end{aligned}$$

The cost savings come from the observation that one can evaluate the above expression using the following procedure. Compute $t = \tilde{x}\tilde{y}$, then $u = (t + (tN' \bmod R)N)/R$; if $u > N$ then return $u - N$, else return u. The desired product z is then obtained by computing $z = \tilde{z}R' \bmod N$. Note that the Montgomery method incurs some overhead in computing R' and N', and that conversion to and from N-residues each require a reduction modulo N. However, if an algorithm computes many modular multiplications with respect to the same modulus to produce only a small set of outputs (such as modular exponentiation, or — in our case — the iterative collision search in parallel rho), the more efficient Montgomery multiplication step results in significant cost savings.

Coarsely-integrated operand scanning. Several alternative algorithms exist for computing the Montgomery multiplication step. In our implementation, we use the *coarsely-integrated operand scanning* (CIOS) method due to Koç et al. [21]. The algorithm is *integrated* because it alternates between multiplication and reduction steps in the computation (that is, it integrates the two procedures into one). The *coarsely-* prefix refers to the frequency with which the algorithm alternates between the two steps; CIOS alternates after processing an array of words, which is in contrast to a *finely-integrated* method, which alternates after processing a single word. Finally, *operand scanning* refers to the fact that the outer loop in the algorithm is over the words of the operands (an alternative approach is *product scanning*, wherein the outer loop is over the words of the product itself). The reader should consult Koç et al.'s paper [21, §5] for full details of the CIOS algorithm.

Interestingly, Bernstein et al. report that "schoolbook" (Montgomery) multiplication gave the best performance in their CUDA implementation of 192-bit modular multiplication [5]; however, our experiments indicate that the CIOS algorithm gives superior performance. This is likely due to its smaller auxiliary storage requirements (the integrated nature of the CIOS method means that it requires just $s+2$ words of auxiliary storage for an s-word modulus, contrasted with $2s + 2$ words for the schoolbook method; this enables more threads to run in parallel on the GPU without exhausting the register pool). Neves and Araujo [28] suggest that the *finely-integrated product scanning* (FIPS) Montgomery multiplication method (also from Koç et al. [21]) yields better performance on GPUs than CIOS does, since each word of the final product can be calculated individually in parallel (whereas the long carry chains in CIOS can make instruction-level parallelism difficult). However, as Bernstein et al. have pointed out [5], using a single thread to compute an entire s-word multiplication leads to improved compute-to-memory-access ratios and eliminates synchronization overhead, thus resulting in better overall performance.

Implementing CIOS with CUDA and PTX. Our implementation of CIOS Montgomery multiplication follows the algorithm given by Koç et al. in §5 of their paper almost exactly, including the suggested improvement for integrating the shifting into the reduction. Our initial implementation looked much like the pseudocode in that paper; however, its performance was underwhelming, and a mysterious bug caused it to

```
// x <- x - y; x and y are both WORDS words long
__device__ void _sub(uint32_t *x, const uint32_t *y)
{
    asm("sub.cc.u32 %0, %1, %2;"
         :  "=r"(x[0]) :  "r"(x[0]), "r"(y[0]));
    for (int i = 1; i < WORDS; i++)
    {
        asm("subc.cc.u32 %0, %1, %2;"
             :  "=r"(x[i]) :  "r"(x[i]), "r"(y[i]));
    }
    asm("subc.u32 %0, %1, %2;"
         :  "=r"(x[WORDS]) : "r"(x[WORDS]), "r"(y[WORDS]));
}
```

Figure 3: PTX-based implementation of arbitrary-precision subtraction.

```
// return x * y + c
static inline __device__ uint64_t mad_u32(
      const uint32_t x, const uint32_t y, const uint64_t c)
{
    uint64_t out;
    asm("mad.wide.u32 %0, %1, %2, %3;"
             :  "=l"(out) :  "r"(x), "r"(y), "l"(c));
    return out;
}
```

Figure 4: PTX-based implementation of fused multiply-and-add.

produce random results in some invocations. (Rather frustratingly, the exact same code *always* succeeded when we ran it in the now deprecated CUDA device emulator.) Eventually, we traced the root of the problem to our use of `memset` to zero the auxiliary array; a race condition was occurring wherein subsequent lines of code sometimes used that memory before the GPU had finished zeroing it. To avoid this, we factored the first iteration out of the outer CIOS loop and modified it to work regardless of the state of the auxiliary array. This modification had the fringe benefit of making the code slightly faster (by entirely avoiding the call to `memset` and several unnecessary additions of 0). Likewise, we factored the last iteration of the outer CIOS loop to place the result directly into the return value, thus avoiding an unnecessary copy at the end of the algorithm.

By far the greatest performance gains occurred when we replaced arithmetic that used standard C-like syntax with inline PTX assembly. For example, rewriting our arbitrary-precision subtraction code to use the subtract-with-borrow-in and optional borrow-out PTX instruction shaved several nanoseconds off the average execution time of each modular multiplication. Figure 3 shows the optimized arbitrary-precision subtraction function; note that in our final implementation, the `for` loop is unrolled to save a few more clock cycles. Similarly, we used PTX's fused multiply-and-add instruction to get significant speedups in the inner CIOS loops (see Figure 4).

Manual loop unrolling proved to be another crucial optimization; at first glance, this optimization appears to be somewhat incompatible with implementing *arbitrary*-precision arithmetic. We solved this by writing a simple Perl script that generates completely unrolled PTX assembly for a given modulus size, which we then link into the binary at compile time. The output of the Perl script also operates entirely on registers instead

of arrays, which is much faster in CUDA, and not possible using an ordinary `for` loop.

Parallel rho with GPUs. We leverage our arbitrary-precision modular multiplication code to implement the parallel rho method. We run the "server thread" on the CPU and each of the "client threads" on one of two Nvidia Tesla M2050 GPU cards (cf. §2). Using Nvidia's CUDA Occupancy Calculator[6] and some experimentation, we found that launching each kernel with 16 warps $= 512$ threads per thread block and 50 total thread blocks per card (which is 25,600 threads per thread grid) provides reasonably good occupancy when our implementation is run to solve discrete logarithms with a 768-bit modulus on our Tesla M2050 cards.

Because there is no lightweight way to interrupt a kernel once it is invoked, we instead have each thread perform some fixed number of iterations before it returns; in particular, each of the 25,600 threads performs 1000 iterations. For the iteration function, we use a "Montgomerized" version of Pollard's original iteration function (Equation (1)), but we note that our use of the iteration function differs somewhat from its regular usage in van Oorschot and Wiener's parallel rho method. When a client thread encounters a distinguished point, it simply outputs the triple $(a_i, b_i, g^{a_i}h^{b_i})$ to the server thread on the host and then continues iterating from that element rather than starting over from a new random group element. We do this to avoid having to initialize a new random element from the server between invocations, and to avoid having all of the threads in the same warp stall for the remainder of the current kernel invocation.[7] If the server does not receive any collisions during a kernel invocation, it simply relaunches the kernel without having to reinitialize or modify any memory on the GPU, and the threads continue iterating from where they left off. This keeps overhead low, but it also means that some client threads could get caught in cycles that do not contain any distinguished points (using a cycle-finding algorithm to detect this would be detrimental to overall performance). However, the expected cycle size is about $\sqrt{n}$, where n is the order of the group $\mathbb{G}$; thus, if the frequency F of distinguished points is such that $F \gg \frac{1}{\sqrt{n}}$, then the probability that this happens is low enough that we can safely ignore it. We define our distinguished points to be elements of $x \in \mathbb{G}$ such that the binary representation of the N-residue of x (i.e., of the Montgomery representation of x) has at least ten trailing zeros, so that about one in every 1024 iterations yields a distinguished point. Since each thread performs 1000 iterations per invocation, the server thread receives about $2^{14.6}$ distinguished points — or one per client thread — each time it invokes the kernel. In the case that $n \leq 2^{20}$ (so that $\frac{1}{1024} \gg \frac{1}{\sqrt{n}}$ does not hold), we change the definition of distinguished points to make them more numerous. In each kernel invocation, the client threads perform a combined total of about $2^{24.6}$ multiplications; thus, for n up to about 2^{50} a single kernel invocation usually suffices to solve the discrete logarithm.

5 Performance evaluation

For our performance benchmarks, we used a server running an Intel Xeon E5620 quad core processor (2.4 GHz) and 2×4 GB of DDR3-1333 RAM, which is equipped with $2 \times$ Tesla M2050 GPU cards. Table 1 below summarizes the per-card performance of our arbitrary-precision modular multiplication implementation; that is, it displays the number of modular multiplications with a k-bit modulus that our implementation can compute on a single Tesla M2050 card for various choices of k, as well as the (amortized) time required

[6] `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

[7] Of course, this is not the only strategy for efficient handling of distinguished points in a GPU environment. For instance, one of the anonymous SHARCS 2012 reviewers points out that "[there] are good reasons not to continue walking from a distinguished point: One can skip all bookkeeping for counting the a and b and instead store only the starting value together with the distinguished point found. If two points collide the server can redo the computations, this time keeping track of the coefficients. For that to work each walk should be reasonably short. See Bernstein et al. [4] and Bernstein, Lange and Schwabe [6], for details on how to handle this in a SIMD environment".

to do each modular multiplication. We point out that as k increases, the execution time increases not only because it naturally takes longer to multiply larger numbers, but also because larger moduli use more registers, and so each core can compute fewer multiplications in parallel.

Table 1: Number of k-bit modular multiplications per second and (amortized) time required per k-bit modular multiplication on each Tesla M2050 GPU card, for various k.

Bit length of modulus	**Modmults per thread**	**Number of threads**	**Number of trials**	**Time per trial ± std dev**	**Amortized time per modmult**	**Modmults per second**
192	100,000	256,000	100	30.538 s ± 4 ms	1.19 ns	≈ 840,336,000
256	100,000	256,000	100	50.916 s ± 5 ms	1.98 ns	≈ 505,050,000
512	100,000	256,000	100	186.969 s ± 4 ms	7.30 ns	≈ 136,986,000
768	100,000	256,000	100	492.6 s ± 200 ms	19.24 ns	≈ 51,975,000
1024	100,000	256,000	100	2304.5 s ± 300 ms	90.02 ns	≈ 11,108,000

Table 2 shows the average time required to compute a discrete logarithm with a 1536-bit RSA modulus $N = pq$ such that $p - 1$ and $q - 1$ are both 768-bit B-smooth integers, for various choices of B. Our implementation uses the approach of Pohlig and Hellman [32] to solve the discrete logarithm independently modulo p and modulo q using the parallel rho method, and then combines the results via the Chinese Remainder Theorem [25, §2.4.3] to get the final discrete logarithm modulo N. In particular, each of the discrete logarithm computations in Table 2 consists of $2 \cdot \lceil 768/\lg B \rceil$ smaller discrete logarithm computations, each at a cost proportional to $\sqrt{B}$. We therefore expect the total cost of the larger discrete logarithm computation to be proportional to $\sqrt{B}/\lg B = B^{0.5-(\lg\lg B/\lg B)}$. When $B \approx 2^{53}$, as in Table 2, we have that $\lg\lg B/\lg B \approx (\lg 53)/53 \approx 0.108$, so that the total running time should be near $c \cdot B^{0.39}$ for some constant of proportionality c.

Figure 5 plots data from Table 2. The exponent on the trend line is slightly less than the expected value of 0.39 because of overhead that is more significant at lower smoothness bounds. One source of overhead is the Chinese remaindering step that we perform after computing the discrete logarithm modulo p and modulo q. Another source of overhead comes from processing "remainder" submoduli of $p - 1$ and $q - 1$: to generate N we choose all but one prime factor of $p - 1$ and $q - 1$ to be k_B bits long, and the last prime factor is about $768 \bmod k_B$ bits. Since our implementation is not optimized for these smaller submoduli, they tend to introduce some additional, nearly constant overhead to the computation time.

Table 2: Time to compute discrete logarithms modulo a product $N = pq$ of two 768-bit primes, such that $\varphi(N)$ is B-smooth, for various choices of B. The discrete logarithm is solved independently modulo p and q using Pohlig-Hellman [32] and parallel rho, and the results are combined via the Chinese Remainder Theorem [25, §2.4.3]. The runtime reported in the final column is the time required to compute all three steps.

Bit length of modulus	**Smoothness of group order**	**Number of trials**	**Time to compute discrete logarithm**
$1536 = 2 \times 768$	2^{48}	100	23 s ± 1 s
$1536 = 2 \times 768$	2^{50}	100	32 s ± 2 s
$1536 = 2 \times 768$	2^{51}	100	41 s ± 3 s
$1536 = 2 \times 768$	2^{52}	100	49 s ± 4 s
$1536 = 2 \times 768$	2^{53}	100	63 s ± 5 s
$1536 = 2 \times 768$	2^{54}	100	85 s ± 7 s
$1536 = 2 \times 768$	2^{55}	100	110 s ± 10 s
$1536 = 2 \times 768$	2^{56}	100	140 s ± 20 s
$1536 = 2 \times 768$	2^{58}	100	270 s ± 30 s

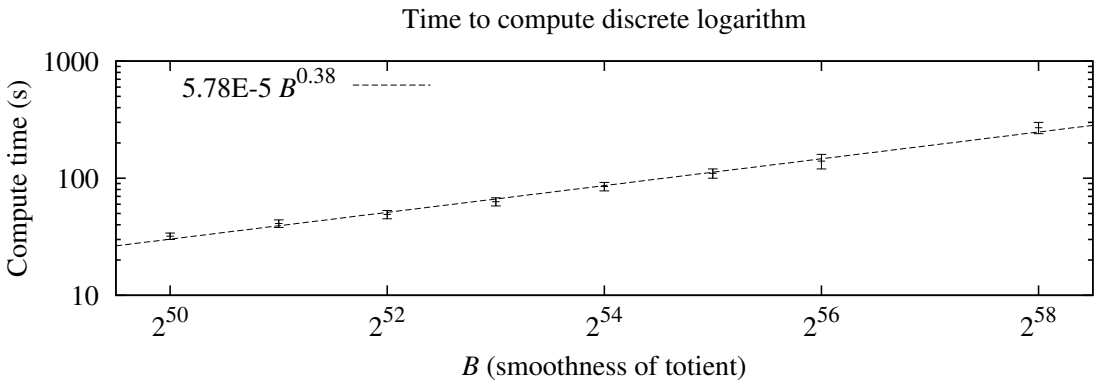


Figure 5: Plot of data from Table 2. The exponent on the trend line is slightly less than 0.39 because of overhead that is more significant at lower smoothness bounds.

By extrapolating to $B = 2^{80}$, we see that it should be feasible to solve discrete logarithms in groups whose order is 2^{80}-smooth in approximately 23 hours.

6 Analysis

6.1 Implications to existing cryptosystems

The idea of exploiting the smoothness of a group's order for attacking cryptosystems whose security relies on the hardness of factoring or computing discrete logarithms is not new. In fact, many cryptographers advocate the use of *safe primes* (primes of the form $p = 2q + 1$ for another prime q) specifically to avoid such attacks (since $p - 1 = 2q$ is not B-smooth for any $B \ll p$). However, the celebrated *elliptic curve factorization method* [23] (ECM) renders these defenses ineffective by considering random elliptic curves over $\mathbb{Z}_p$, which, by Hasse's theorem [38, §V], have orders that are (essentially randomly) distributed between $p + 1 - 2\sqrt{p}$ and $p + 1 + 2\sqrt{p}$. Thus, one can argue that, due to ECM, explicitly choosing safe primes is not helpful, since

to ensure security we must already assume that $p-1$ is non-smooth for a random prime p. Pomerance and Shparlinski [36] study the distribution of smooth integers, and derive rigorous upper bounds on the number of k-bit prime numbers p for which $p-1$ is smooth or has a large smooth factor. Their findings suggest that a *randomly selected*, cryptographically large number is not "sufficiently smooth" to make smoothness-based attacks feasible.

We point out, however, that a bad actor could choose the modulus for a discrete-logarithm-based cryptosystem with malice, inserting a trapdoor for his own use. Our experiments in §5 indicate that computing discrete logarithms in groups of B-smooth order, for B up to at least $\approx 2^{80}$, is entirely realistic with under 24 hours of computation on readily available commodity hardware. For a prime modulus N such that $N-1$ is 2^{80}-smooth, one could use ECM (or some other related technique) to factor $N-1$ and thus learn about its insecurity (although this would require substantial computational effort on the part of the would-be victim). On the other hand, detecting insecure *composite* moduli N is not so simple, since even determining the value of $\varphi(N)$ is equivalent to factoring N [25, §8.2.2]. Our analysis in the next subsection suggests that the massive parallelism afforded by GPUs only helps to widen the gap between the feasibility and detectability of such attacks. Fortunately, most cryptosystems that base their security on the difficulty of computing discrete logarithms work over prime moduli; however, below we point out a concrete — and realistic — attack on zero-knowledge range proofs, which uses a difficult-to-detect composite modulus with smooth totient.

An attack on zero-knowledge 'range proofs'. At Eurocrypt 2000, Boudot proposed a novel zero-knowledge proof that allows a prover to convince a verifier that a committed value is in a specific interval [9]. Boudot's *range proof* relies on Lagrange's four-square theorem [37], which states that an integer can be expressed as a sum of (at most) four squares if and only if it is nonnegative. Suppose that a prover wishes to convince a verifier that a commitment, say $C = g^x \bmod p$, is to a value x in the interval $[a, b]$. To do this, the prover and verifier each compute $C_a = C/g^a = g^{x-a} \bmod p$ and $C_b = g^b/C = g^{b-x} \bmod p$, and then the prover engages the verifier in a zero-knowledge proof of knowledge of two tuples of integers (c, d, e, f) and (h, i, j, k) such that $C_a = g^{c^2+d^2+e^2+f^2} \bmod p$ and $C_b = g^{h^2+i^2+j^2+k^2} \bmod p$. Of course, for soundness the proof assumes that the prover does not know the group order, since otherwise the prover could simply find, say, (c, d, e, f) such that $c^2 + d^2 + e^2 + f^2 = x - a + \varphi(p)$ and thereby fool the prover. Therefore, when p is prime (which it usually is), the verifier chooses a composite modulus N for which the prover does not know the factorization, then the prover commits to x modulo N and proves in zero knowledge that this new commitment is to the same x as the original commitment. Finally, the prover and verifier do the above range proof using the commitment modulo N. If the verifier chooses a modulus N with smooth totient (thus violating one of the assumptions needed to prove computational zero-knowledge), then when the prover commits to his secret x in the group modulo N, the verifier can compute the discrete logarithm to learn x.[8]

6.2 Trapdoor discrete logarithm groups

In earlier work [18], we discussed using a CPU-based implementation of parallel rho to construct *trapdoor discrete logarithm groups*; that is, groups in which computing discrete logarithms is easy for anyone in possession of a special trapdoor key, but cryptographically hard for everybody else. The GPU-based implementation of parallel rho that we consider in this work allows for the same construction, but with a considerably improved margin of security.

Construction. The idea behind the trapdoor discrete logarithm group construction is to work in the multiplicative group of units modulo a k_N-bit RSA modulus $N = pq$ such that $p \approx q$, and both $p-1$ and

[8] Of course, one can thwart this attack by having the verifier prove to the prover that the composite modulus is the product of two safe primes, for example by using the technique of Camenisch and Michels [12].

$q-1$ are products of distinct k_B-bit primes; here k_N and k_B are carefully selected parameters. The public key is N and the private (trapdoor) key is the factorization of $\varphi(N)$ into k_B-bit primes. Computing discrete logarithms with knowledge of the trapdoor key requires $\Theta\left(\frac{k_N}{k_B} \cdot 2^{k_B/2}\right)$ highly parallelizable work, whereas the most efficient way to compute discrete logarithms *without* knowledge of the trapdoor key seems to be factoring N into p and q, and then factoring $p-1$ and $q-1$ to recover the trapdoor key. In our original application [18], we actually wanted trapdoor discrete logarithm groups such that computing the discrete logarithm with the trapdoor key is *tunably costly*, but feasible; other applications might wish to set the cost as low as possible subject to the construction staying secure.

Security analysis. Using the parallel rho method, the expected number of k_N-bit modular multiplications needed to compute the discrete logarithm in such a trapdoor group is $c \cdot \left(\frac{k_N}{k_B}\right) \cdot 2^{k_B/2}$, for some constant of proportionality c. (Note that these multiplications are almost completely parallelizable.) Given a parallelism factor of Ψ cores, this takes about

$$\frac{k_N}{k_B} \cdot \frac{c \cdot 2^{k_B/2}}{\Psi \cdot \mu} \text{ seconds,} \tag{2}$$

where μ is the number of multiplications modulo an $(k_N/2)$-bit modulus that are computable per core-second. Thus, to tune the parameters such that discrete logarithm computations require a specific time Γ on average, we solve for k_B in the expression

$$\frac{2^{k_B/2}}{k_B} \approx \frac{\Gamma \cdot \mu \cdot \Psi}{k_N \cdot c}. \tag{3}$$

Through experimentation, we observe that it takes the "progressively increase B" variant of Pollard's $p-1$ method at least about $\frac{3}{5} \cdot 2^{k_B}$ modular multiplications with a k_N-bit modulus to factor N. (Note that this is fewer multiplications than the estimate given in §2, since we only need to consider those primes q such that $2^{k_B} < q < 2^{k_B+1}$, and since we can assume each prime has multiplicity one in $\varphi(N)$, given our prior knowledge about N.) We stress that — in contrast to the case of *general* exponentiation, which can potentially benefit from some parallelism — the adversary must perform these multiplications in a sequential manner [11]; even with a very large degree of parallelism, only a very small speedup is obtainable.[9] It may be possible to parallelize the "non-progressive" variant of Pollard's $p-1$; however, this variant will output the *product* of all prime factors p of N such that $p-1$ is B-smooth, which in the trapdoor discrete logarithm case is just N itself. Therefore, when $k_B \ll 85$, an adversary requires about $\frac{3}{5} \cdot \frac{2^{k_B}}{\mu}$ seconds to factor N. We ran some experiments on an Intel Q9550 quad core CPU (2.83 GHz) that indicate a value of $\mu = 385{,}000$ $^{\text{mults}}/_{\text{second}}$ on that device (which has considerably faster individual cores than our Tesla machine does). Thus, setting k_B as low as 55 yields over 1500 years of (non-parallelizable) wall-clock time to factor N using the Pollard $p-1$ method on this CPU, while requiring less than two minutes to compute trapdoor discrete logarithms with our two M2050 cards.

Maurer and Yacobi [24, §4] point out that, since the cost of factoring increases with 2^{k_B}, while the cost of computing discrete logarithms increases with $\sqrt{2^{k_B}}$, faster cores and more parallelism only help to increase security. In particular, if μ and Ψ increase by a factor f and g, respectively, then we can revise the parameters such that security increases by a factor of fg^2.

[9] Other factoring algorithms, such as ECM [23] or the quadratic sieve algorithm (QS) [35], are highly parallelizable and can factor a general modulus with *sublinear* asymptotic complexity; however, the linear cost of Pollard's $p-1$ factoring algorithm is by far the most efficient method for factoring N, given its special form and our parameter selection. In other words, while both of the aforementioned algorithms have superior asymptotic complexity to Pollard's $p-1$ factoring algorithm (depending on how one asymptotically relates k_B and k_N), the actual position on the $O(2^{k_B})$ cost curve in our case is much smaller than the corresponding position on the cost curves for these asymptotically faster algorithms. For reference, factoring a 1536-bit RSA modulus using more efficient algorithms requires about 2^{85} (parallelizable) work; thus, the cost curves intersect when $\Psi \cdot 2^{k_B} \approx 2^{85}$ and Pollard's $p-1$ algorithm ceases to be most efficient for larger values of $\Psi \cdot 2^{k_B}$ [22].

Zero-knowledge proofs of costliness. Using a straightforward generalization of the zero-knowledge proof that a number is a product of two safe primes, due to Camenisch and Michels [12], one can prove in zero-knowledge that the prime factors q of $\varphi(N)$ each satisfy $2^{k_B} < q < 2^{k_B+1}$. Given some assumptions about available computing power, Equation (2) lets us estimate how long an average trapdoor discrete logarithm computation takes. While the validity of this proof relies on assumptions about the available computational capacity, it *does* give a reliable estimate of the computational — and thus economic — cost of being able to compute discrete logarithms, which is useful in applications such as partial key escrow [3] or our own anonymous blacklisting [18].

7 Conclusion

In this paper, we discussed our experiences with using GPUs and Nvidia's CUDA framework to accelerate the computation of discrete logarithms with respect to a special class of moduli. In particular, we presented our approach to implementing fast, arbitrary-precision modular multiplication on GPUs using C for CUDA and the PTX instruction set architecture, and then described how we were able to leverage this modular multiplication to implement a parallel version of Pollard's rho algorithm. We also examined the implications to existing cryptosystems whose security is based on the presumed intractability of computing discrete logarithms, and pointed out that efficient implementations of Pollard's rho for groups with smooth order enables the construction of cryptographically secure trapdoor discrete logarithm groups. All of our source code is open source and is freely available online from the CrySP group's website at `http://crysp.uwaterloo.ca/software/`.

Acknowledgements. We thank the anonymous reviewers for their thoughtful and constructive comments. Funding for this research was provided in part by NSERC, Mprime NCE (formerly MITACS) and the Ontario Research Fund. The first author is supported by an NSERC Vanier Canada Graduate Scholarship and a Cheriton Graduate Scholarship.

References

[1] Carlos Aguilar Melchor, Benoit Gaborit, Philippe Gaborit, Vincent Jolivet, and Pierre Rousseau. High-speed Private Information Retrieval Computation on GPU. In *Proceedings of SECURWARE 2008*, pages 263–272, Cap Esterel, France, August 2008.

[2] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. Available online: `http://eprint.iacr.org/2009/541.pdf`.

[3] Mihir Bellare and Shafi Goldwasser. Verifiable Partial Key Escrow. In *Proceedings of CCS 1997*, pages 78–91, Zurich, Switzerland, April 1997.

[4] Daniel J. Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. ECC2K-130 on NVIDIA GPUs. In *Proceedings of INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 328–346, Hyderabad, India, December 2010.

[5] Daniel J. Bernstein, Hsueh-Chung Chen, Ming-Shing Chen, Chen-Mou Cheng, Chun-Hung Hsiao, Tanja Lange, Zong-Cing Lin, and Bo-Yin Yang. The Billion-Mulmod-Per-Second PC. In *Workshop record of SHARCS'09: Special-purpose Hardware for Attacking Cryptographic Systems*, Lausanne, Switzerland, September 2009.

[6] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. On the Correct Use of the Negation Map in the Pollard rho Method. In *Proceedings of PKC 2011*, volume 6571 of *LNCS*, pages 128–146, March 2011.

[7] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Solving a 112-bit Prime Elliptic Curve Discrete Logarithm Problem on Game Consoles using Sloppy Reduction. *International Journal of Applied Cryptography*, 2011.

[8] Joppe W. Bos, Marcelo E. Kaihara, and Peter L. Montgomery. Pollard Rho on the PlayStation 3. In *Workshop record of SHARCS'09: Special-purpose Hardware for Attacking Cryptographic Systems*, Lausanne, Switzerland, September 2009.

[9] Fabrice Boudot. Efficient Proofs that a Committed Number Lies in an Interval. In *Proceedings of EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 431–444, Bruges, Belgium, May 2000.

[10] Richard P. Brent. An Improved Monte Carlo Factorization Algorithm. *BIT*, 20:176–184, 1980.

[11] Richard P. Brent. Parallel Algorithms for Integer Factorisation. *Number Theory and Cryptography*, pages 26–37, 1990.

[12] Jan Camenisch and Markus Michels. Proving in Zero-Knowledge that a Number Is the Product of Two Safe Primes. In *Proceedings of EUROCRYPT 1999*, volume 1592 of *LNCS*, pages 107–122, Prague, Czech Republic, May 1999.

[13] Certicom Corporation. Certicom ECC Challenge. Available online: `http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf`.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[15] Sebastian Fleissner. GPU-Accelerated Montgomery Exponentiation. In *Proceedings of ICCS 2007*, Beijing, China, May 2007.

[16] Robert W. Floyd. Nondeterministic Algorithms. *Journal of the ACM*, 14(4):635–644, October 1967.

[17] Owen Harrison and John Waldron. Public Key Cryptography on Modern GPU Hardware. In *Booklet of accepted posters for Eurocrypt 2009*, Cologne, Germany, April 2009.

[18] Ryan Henry, Kevin Henry, and Ian Goldberg. Making a Nymbler Nymble Using VERBS. In *Proceedings of PETS 2010*, volume 6205 of *LNCS*, pages 111–129, Berlin, Germany, July 2010.

[19] Jens Hermans, Frederik Vercauteren, and Bart Preneel. Speed Records for NTRU. In *Proceedings of CT-RSA 2010*, volume 5985 of *LNCS*, pages 73–88, San Francisco, California, March 2010.

[20] Gareth A. Jones and Josephine M. Jones. *Elementary Number Theory*. Springer-Verlag, 1998.

[21] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[22] Arjen K. Lenstra. Key Lengths. In Hossein Bidgoli, editor, *Handbook of Information Security*, volume II, pages 617–635. Wiley, December 2005.

[23] Hendrik W. Lenstra Jr. Factoring Integers With Elliptic Curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[24] Ueli M. Maurer and Yacov Yacobi. A Non-interactive Public-Key Distribution System. *Designs, Codes and Cryptography*, 9(3):305–316, 1996.

[25] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[26] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[27] Andrew Moss, Dan Page, and Nigel P. Smart. Toward Acceleration of RSA Using 3D Graphics Hardware. In *Proceedings of the IMA International Conference on Cryptography and Coding 2007*, Cirencester, UK, December 2007.

[28] Samuel Neves and Filipe Araujo. On the Performance of GPU Public-Key Cryptography. In *Proceedings of ASAP 2011*, pages 133–140, Santa Monica, California, September 2011.

[29] NVIDIA Corporation. Why Choose Tesla. Available online: `http://www.nvidia.com/object/why-choose-tesla.html`. Retrieved 18-Jan-2012.

[30] NVIDIA Corporation. NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi, February 2010. v1.1. Available online: `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf`.

[31] NVIDIA Corporation. CUDA C Best Practices Guide, May 2011. DG-05603-001_v4.0. Available online: `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf`.

[32] Stephen Pohlig and Martin Hellman. An Improved Algorithm for Computing Logarithms Over GF(p) and its Cryptographic Significance. *IEEE Transactions on Information Theory*, IT-24(1):106–110, January 1978.

[33] John M. Pollard. Theorems of Factorization and Primality Testing. *Proceedings of the Cambridge Philosophical Society*, 76(3):521–528, 1974.

[34] John M. Pollard. Monte Carlo Methods for Index Computations (mod p). *Mathematics of Computation*, 32(143):918–924, July 1978.

[35] Carl Pomerance. Analysis and Comparison of Some Integer Factoring Algorithms. In *Computational Methods in Number Theory, Part I*, pages 89–139, Amsterdam, 1982.

[36] Carl Pomerance and Igor Shparlinski. Smooth Orders and Cryptographic Applications. In *Proceedings of ANTS 2002*, volume 2369 of *LNCS*, pages 338–348, Sydney, Australia, July 2002.

[37] Michael O. Rabin and Jeffrey O. Shallit. Randomized Algorithms in Number Theory. *Communications on Pure and Applied Mathematics*, 39(S1):S239–S256, 1986.

[38] Joseph A. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer-Verlag, 1994.

[39] Edlyn Teske. Speeding Up Pollard's Rho Method for Computing Discrete Logarithms. In *Proceedings of ANTS 1998*, volume 1423 of *LNCS*, pages 541–554, Portland, Oregon, June 1998.

[40] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, January 1999.