High-Speed Elliptic Curve Cryptography on the NVIDIA GT200 Graphics Processing Unit

Shujie Cui[1,2], Johann Großschädl[2], Zhe Liu[2,⋆], and Qiuliang Xu[1]

[1] Shandong University,
School of Computer Science and Technology,
Shunhua Road 1500, Jinan 250101, Shandong, P.R. China
shujiecui1@gmail.com, xql@sdu.edu.cn
[2] University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security,
6, rue Richard Coudenhove-Kalergi, L–1359 Luxembourg
{johann.groszschaedl,zhe.liu}@uni.lu

Abstract. This paper describes a high-speed software implementation of Elliptic Curve Cryptography (ECC) for GeForce GTX graphics cards equipped with an NVIDIA GT200 Graphics Processing Unit (GPU). In order to maximize throughput, our ECC software allocates just a single thread per scalar multiplication and aims to launch as many threads in parallel as possible. We adopt elliptic curves in Montgomery as well as twisted Edwards form, both defined over a special family of finite fields known as Optimal Prime Fields (OPFs). All field-arithmetic operations use a radix-2^{24} representation for the operands (i.e. 24 operand bits are contained in a 32-bit word) to comply with the native (24×24)-bit integer multiply instruction of the GT200 platform. We implemented the OPF arithmetic without conditional statements (e.g. if-then clauses) to prevent thread divergence and unrolled the loops to minimize execution time. The scalar multiplication on the twisted Edwards curve employs a comb approach if the base point is fixed and uses extended projective coordinates so that a point addition requires only seven multiplications in the underlying OPF. Our software currently supports elliptic curves over 160-bit and 224-bit OPFs. After a detailed evaluation of numerous implementation options and configurations, we managed to launch 2880 threads on the 30 multiprocessors of the GT200 when the elliptic curve has Montgomery form and is defined over a 224-bit OPF. The resulting throughput is 115k scalar multiplications per second (for arbitrary base points) and we achieved a minimum latency of 19.2 ms. In a fixed-base setting with 256 precomputed points, the throughput increases to some 345k scalar multiplications and the latency drops to 4.52 ms.

1 Introduction

Driven by the requirements of 3D computer games, Graphics Processing Units (GPUs) have evolved into massively parallel processors consisting of hundreds

⋆ Co-first author, supported by the FNR Luxembourg (AFR grant 1359142).

X. Huang and J. Zhou (Eds.): ISPEC 2014, LNCS 8434, pp. 202–216, 2014.

of cores that are capable of running thousands of threads concurrently [14]. In contrast, recent general-purpose CPUs feature a maximum of 12 cores and can handle only few threads per core. They dedicate a large portion of their silicon area to support a hierarchical memory organization (i.e. multi-level cache) and sophisticated flow control mechanisms (e.g. branch prediction, out-of-order execution). In a modern GPU, on the other hand, the vast majority of transistors (more than 80% according to [19]) is devoted to data processing (i.e. numerical computations) rather than data caching and flow control. Over the past couple of years, the performance of CPUs doubled roughly every 18 months, whereas the computational power of GPUs increased significantly faster with an average doubling rate of just about six months ("Moore's law cubed") [13]. Today, the floating-point performance of contemporary GPUs exceeds that of CPUs of the same or similar price by more than an order of magnitude. The unprecedented computational power and relatively low cost of modern GPUs has made them an attractive platform for various "number-crunching" applications outside the graphics domain, e.g. in cryptography [4,6] and cryptanalysis [5].

The recent literature contains several case studies that demonstrate the use of a GPU as "accelerator" for cryptographic workloads; a well-known example is SSLShader [12], a GPU-based reverse proxy for SSL servers. SSL, along with its successor TLS, is the current de-facto standard protocol for enabling secure communication over an insecure network like the Internet. The most expensive part of SSL/TLS is the handshake sub-protocol, whose task is to authenticate the server to the client[1] and establish a so-called pre-master secret [12]. When an RSA-based cipher suite is used for the handshake, the server has to execute computation-intensive modular exponentiations, which causes excessive delays and hampers throughput. SSLShader tackles this problem by "off-loading" the modular exponentiations to one or more GPUs, thereby alleviating the burden of the server's CPU. Practical experiments in [12] show that GPU acceleration of the handshake increases the number of SSL transactions per second by a factor of 2.5 (1024-bit RSA) and 6.0 (2048-bit RSA) compared to a configuration where the CPU performs the exponentiations. Even though [12] only considers RSA-based cipher suites, the idea of accelerating SSL via one or more GPUs is also applicable to handshakes using Elliptic Curve Cryptography (ECC).

In this paper, we present an efficient implementation of ECC (or, more precisely, of scalar multiplication in an elliptic curve group) for NVIDIA graphics cards featuring a Tesla GPU [14]. Our implementation is specifically optimized for high throughput, which means we aimed at maximizing the number of scalar multiplications the GPU can execute per second. The basic idea we pursue is to employ just one single thread for each scalar multiplication, but launch as many threads in parallel as possible. This contrasts with the bulk of previous work, which followed a relatively "fine-grained" approach to parallel processing by invoking several threads to cooperatively compute one scalar multiplication [2]. Avenues for exploiting thread-level parallelism to speed up ECC on GPUs

[1] Client authentication is optional in SSL. Web applications usually authenticate the client (i.e. user) through a higher-level protocol, e.g. by entering a password.

exist in both the field arithmetic (i.e. modular multiplication and squaring, see e.g. [1,4]) and the group arithmetic (i.e. point addition and point doubling, see e.g. [6,11]). A major challenge of such a "many-threads-per-task" strategy is to partition the task (scalar multiplication in our case) into independent subtasks that can be executed in parallel with little communication and synchronization overhead. The goal is to find a partitioning that keeps all threads busy all the time so that no resources are wasted by idling threads, which is difficult due to the iterative (i.e. sequential) nature of scalar multiplication algorithms. On the other hand, a "one-thread-per-task" strategy avoids these issues and is easy to implement because all involved operations are executed sequentially by a single thread. Therefore, this approach has the virtue of (potentially) better resource utilization when launching a large number of threads. However, the problem is that the threads, even though they are independent of each other, share certain resources such as registers or fast memory, which are sparse. The more threads are active at a time, the fewer resources are available per task.

This paper seeks to shed new light on the question of how to "unleash" the full performance of GPUs to achieve maximum throughput for scalar multiplication. To this end, we combine the state-of-the-art in terms of implementation options for ECC with advanced techniques for parallel processing on GPUs, in particular the NVIDIA GT200 [14,19]. Our implementation currently supports elliptic curves in Montgomery [18] and twisted Edwards form [3], both defined over a special type of prime field known as Optimal Prime Field (OPF) [9]. In order to ensure a fair comparison with previous work (most notably [1,6]), we benchmarked our ECC software on a GeForce GTX285 graphics card equipped with a GT200 processor. Even though the GT200 is already five years old and has a (by today's standards) rather modest compute capability of 1.3 [20], its integer performance is still "remarkably good," as was recently noted by Bos in [6, Section 5]. This is not surprising since, in the past few years, NVIDIA has focused primarily on cranking up the performance of single-precision floating-point operations, whereas integer performance improved at a rather slow pace from one GPU generation to the next. A peculiarity of the GT200 GPU are its integer multipliers, which "natively" support only (24×24)-bit multiplications and MAC operations, even though the integer units, including registers, have a 32-bit datapath. (32×32)-bit multiplications can be executed, but they need to be composed of several `mul24` instructions and are, therefore, slow.

2 Preliminaries

In this section, we first discuss some basic properties and features of NVIDIA's GT200 platform (Subsection 2.1) and then recap the used elliptic curve models as well as the underlying prime field (Subsection 2.2).

2.1 Graphics Processing Units (GPUs)

A large number of multi-core GPU platforms exist today, e.g. the Tesla, Fermi and Kepler families from NVIDIA, or the Radeon series from AMD. We use an

NVIDIA GeForce GTX285 card for our implementation due to its attractive price-performance ratio and easy programmability. The main component of an NVIDIA GPU is a scalable array of multi-threaded Streaming Multiprocessors (SMs), in which the actual computations are carried out. A GT200 is composed of exactly 30 SMs, each coming with its own control units, registers, execution pipelines and caches. The main components of an SM are Streaming Processors (SPs), which are essentially just ALUs, referred to as "cores" in NVIDIA jargon [20]. Each SM contains eight cores and two Special Function Units (SFUs). The SMs are designed to create, manage, schedule, and execute a large number of threads concurrently following the SIMT (Single-Instruction, Multiple-Thread) principle. A batch of 32 threads executed physically in parallel is called a *warp*. At each cycle, the SM thread scheduler chooses a warp to execute. We should note that a warp executes one common instruction at a time. If there is a data-dependent conditional branch, divergent paths will be executed serially. So, in order to obtain full performance, all the threads of a warp should have the same execution path, i.e. conditional statements should be avoided.

The so-called Compute Unified Device Architecture (CUDA) is a parallel programming model introduced by NVIDIA to simplify software development for GPUs, including software for general-purpose processing on GPUs (GPGPU). It provides both a low-level and high-level API and also defines the memory hierarchy. The parallel portion of an application is executed on GPUs as kernels (a *kernel* is a grid of thread blocks). A *block* is a group of threads, whereby all threads in one block can cooperate with each other. A thread is the smallest unit of parallelism and only threads with the same instructions can be executed synchronously. Our implementation launches thousands of threads to compute thousands of scalar multiplications in parallel on the GT200.

CUDA provides a hierarchical memory model, including registers, shared memory, global memory, and constant memory [20]. Registers are on-chip memories, which are private to individual threads. Variables that reside in registers can be accessed at the highest speed in a highly parallel manner. On a GT200, each SM has 16384 registers of a width of 32 bits. However, registers can not be addressed. Shared memory is also located on chip and can, therefore, be accessed at a high speed. Shared memory is allocated to a thread block. All the threads in one block can cooperate by sharing their input data and intermediate results through shared memory. In the GT200 series, each SM has 16 kB shared memory. Global memory and constant memory are off-chip memories. Global memory is the only one that can be accessed by the host processor, so it is normally used to exchange data with host memory. Constant memory can only be read and is optimized for one-dimensional locality of accesses. One can achieve optimal performance by carefully considering the advantages of the different variants of memory. As registers and shared memory are the fastest memory spaces, we mainly use them in our implementation. In order to get the best performance, it is vital to balance the number of parallel threads per block with the utilization of the limited registers and shared memory. Furthermore, one has to be careful to prevent bank conflicts [20] when accessing shared memory.

2.2 Elliptic Curve Cryptography (ECC)

Twisted Edwards Curve. Twisted Edwards curves were presented by Bernstein et al [3] and are widely considered to be one of the most efficient models for implementers. Let K be a field with $\mathrm{char}(K) \neq 2$. A twisted Edwards curve over K can be defined as

$$E_{T,a,d} : ax^2 + y^2 = 1 + dx^2y^2 \tag{1}$$

where a and d are distinct non-zero elements of K, i.e. $ad(a-d) \neq 0$.

Our implementation adopts the idea of extended coordinates from [11] to perform a point addition and point doubling. A point in extended projective coordinates can be represented as $(X : Y : T : Z)$ whereby the corresponding extended affine coordinates have the form $(X/Z, Y/Z, T/Z)$ with $Z \neq 0$. The auxiliary coordinate T has the property $T = XY/Z$. Fixing the parameter a to -1 allows for a further reduction of the cost of point operations as described in [11]. We follow the approach from [7] and use a quintuple with two variables E and H instead of T to represent a point, whereby $E \cdot H = T$. In this case, a point doubling can be performed with three multiplications and four squarings (i.e. 3M + 4S), while the point addition costs seven multiplications (7M).

To reach high throughput, our implementation adopts a comb method [10] for scalar multiplication, which can only be used in scenarios where the base point is fixed. Given the amount of constant memory the GTX285 provides, we chose a window width of $w = 8$ for the comb method. Consequently, 256 points (one of which is the neutral element) have to be pre-computed off-line and then transferred to constant memory before the actual execution of the scalar multiplication. To prevent thread divergence and protect our implementation against timing-based side-channel attacks, we simply exploit the completeness of the Edwards addition law (i.e. we add the neutral element when an 8-bit digit of the scalar is zero) to achieve a branchless execution path.

Montgomery Curve. Peter Montgomery introduced in 1997 a special family of elliptic curves with outstanding implementation properties [18]. A Montgomery curve E_M with coefficients A and B over $\mathbb{F}_p$ is defined as

$$E_{M,A,B} : By^2 = x^3 + Ax^2 + x \tag{2}$$

Montgomery curves allow a special ladder technique to perform a scalar multiplication, which is generally referred to as "Montgomery ladder". Instead of using conventional (x, y) coordinates, the scalar multiplication on a Montgomery-form curve can be computed using only the x coordinate of the base point. Due to this feature, all point additions and doublings can be executed in an efficient way since they never involve a y coordinate. Therefore, the point addition has an operation count of of only 3M + 2S, where M represents a field multiplication and S a squaring operation. Doubling a point costs 2M + 2S + 1C, where C stands for a multiplication of a field element by the constant $(A + 2)/4$. In our implementation, the parameter A is chosen such that this constant is small.

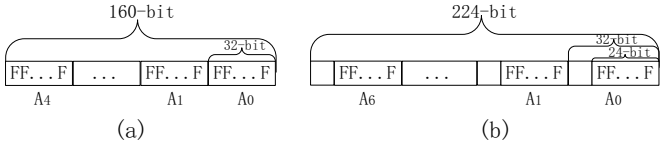

Fig. 1. Radix-2^{24} representation of a 160-bit integer using 32-bit words

Optimal Prime Fields (OPFs). We use a special class of finite field, known as *Optimal Prime Field* (OPF) [15]. OPFs are defined via a prime of the form $p = u \cdot 2^k + v$, whereby u and v are small in relation to 2^k. It is obvious that there exist many such primes for a given bitlength. In our implementation, v is always 1 and u is a 16-bit integer. A concrete example for a 160-bit prime is $p = 65356 \cdot 2^{144} + 1 =$ 0xff4c0000000000000000000000000000000000001. Primes of such form have a low Hamming weight, i.e. they contain many zero words [15]. Generic modular reduction algorithms, e.g. Montgomery reduction, can be optimized for these primes as only the non-zero words must be processed.

3 Implementation

This section describes our implementation in detail. First, we demonstrate the advantage of using a radix-2^{24} representation for the field elements in Section 3.1, and then describe the field arithmetic operations in Section 3.2 and finally the group arithmetic along with the scalar multiplication in Section 3.3.

3.1 Integer Representation

One of the fundamental questions when implementing multi-precision arithmetic for a given architecture is how to represent the operands so as to take best advantage of its computational resources. In general, multiplication and carry propagation are of primary concern.

Multiplication plays an important role in ECC implementations, especially when projective coordinates are used. The GT200 series is based on the Tesla architecture, which means the native integer multiply instruction calculates a (24×24)-bit product. A 32-bit integer multiplication is actually performed via a combination of several 24-bit multiplications, shifts and additions. According to the CUDA C programming guide [20], eight 24-bit integer multiplications can be executed per clock cycle on each SM, which is more efficient than the integer multiplication using a straightforward 32-bit representation. Thus, we adopt a 24-bit representation for the field elements in our work.

Multi-precision operands are typically represented by arrays of w-bit words whereby w is determined by the word-size of the target processor. When using a straightforward 32-bit-per-word representation, a 160-bit operand X can be stored in an array of five 32-bit words. On the other hand, a radix-2^{24} representation (i.e. 24 bits per word) requires seven 32-bit words as shown in

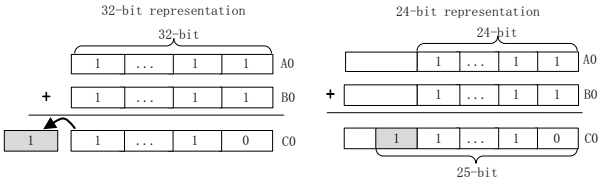

Fig. 2. Comparison of multi-precision addition (radix-2^{32} vs radix-2^{24})

Figure 1. The most significant byte is 0 in every word and the most significant word of a 160-bit integer contains only 16 bits. Even though this representation takes two additional 32-bit words (namely seven in total instead of five), it yields significantly better field-multiplication performance on a GT200.

2^{24} Addition vs 2^{32} Addition. As shown in the left side of Figure 2, when using the radix-2^{32} representation, the sum of two words may overflow, and the resulting carry bit has to be added to the next-higher word. This can be performed efficiently in PTX assembly language using the `add.cc` instruction [21]. On the other hand, the radix-2^{24} representation provides enough space in the unused most significant byte to hold the carry. However, there are extra instructions necessary to extract the carry bit and then add it to the next-higher pair of words. Thus, the radix-2^{24} representation makes multi-precision addition slightly slower, but this is more than compensated by a significant gain in multiplication performance as will be described below.

2^{24} Multiplication vs 2^{32} Multiplication. We use the product-scanning method [10], which can be optimized to take advantage our 24-bit integer representation. Figure 3 shows an example of its implementation. As mentioned before, CUDA provides the `[u]mul24.lo/hi` instructions, whereby the former multiplies the 24 Least Significant Bits (LSBs) of the operands and returns the 32 LSBs of the 48-bit product. On the other hand, `[u]mul24.hi` also multiplies the 24 LSBs of the operands, but returns the 32 Most Significant Bits (MSBs) of the product [21, p. 60]. Therefore, the 48-bit product is written to two 32-bit registers. In the inner loop of the product-scanning method, the partial products of the same column are added together. Due to the 24-bit representation, we have 8 unused bits, which allows the carries to be added as part of the operands (we only need to extract the carries at the end of the inner loop). Hence, only two 32-bit additions are needed in each iteration of the inner loop.

The inner loop is much slower when using a 32-bit representation, which has two main reasons. First, a 32-bit integer multiplication takes much longer than the native 24-bit multiply instruction. Second, the processing of the carry bits requires additional effort because both a 32-bit addition (to accumulate the lower part) and a 64-bit addition (to accumulate the higher part) has to be executed per loop iteration. A further disadvantage is the need for extra registers. Figure 4 compares the execution time of the multiplication using a radix-2^{24} and

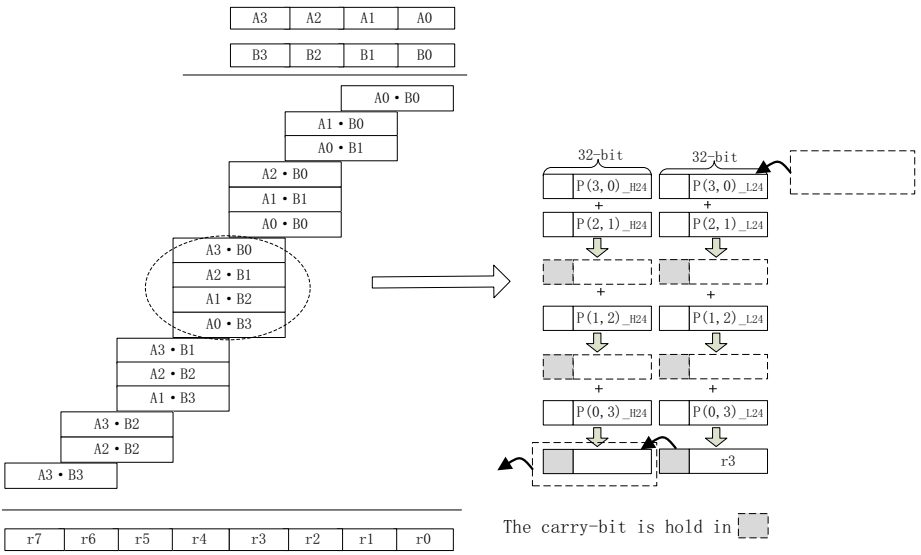

Fig. 3. Product-scanning method for multi-precision multiplication

a radix-2^{32} representation. The figure also includes a radix-2^{29} representation, which uses 32-bit multiply instructions to get the partial products, but handles the carries in the same way as the radix-2^{24} representation. Our results show that the 24-bit representation outperforms the other two approaches by far.

Besides addition and multiplication, the proposed radix-2^{24} representation is also beneficial for modular reduction. The reason is twofold:

- **No Reduction Operation:** The idea of incomplete modular reduction was described in detail by Yanik et al [23]. This technique allows the result of an operation to be greater than the prime p, but it must have the same bit length (denoted as s). Normally, if $p < 2^s < 2p - 1$, we require the result of a field arithmetic operation to be in the range $[0, 2^s - 1]$, but it does not necessarily need to be smaller than p. Consequently, the reduction operation can be avoided when this condition is met, which means incompletely reduced results can save execution time. However, if the result does not fit into s bits, we need to reduce it until it is in the range $[0, 2^s - 1]$. Our implementation does not need to perform the reduction operation for every field operation since the excess bits can be held in the unused bits without additional memory or register usage.
- **No Conditional Branches:** Addition and Montgomery reduction may require a final subtraction of p, which can cause thread divergence and leak side-channel information if implemented in a naive way. As we pointed out before, the radix-2^{24} representation does not have this problem.

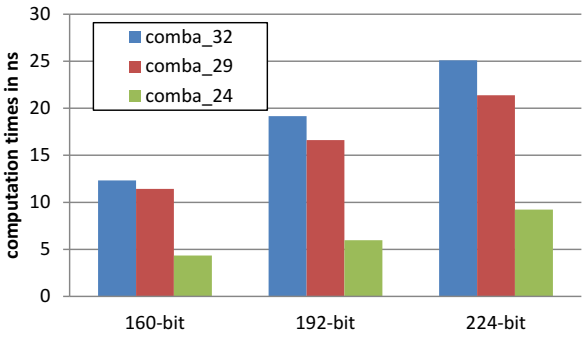

Fig. 4. Comparison of the execution time of multi-precision multiplication for 160, 192 and 224-bit operands (radix-2^{24} vs radix-2^{29} vs radix-2^{32} representation)

3.2 Field Operations

"Lazy" Modular Addition and Subtraction. The modular addition and subtraction are basic operations in ECC. We implemented them efficiently using our special integer representation. For modular addition, we replace the field addition $a + b \bmod p$ by an ordinary integer addition $a + b$ without reduction operation. Since the unused bits in the most significant word can hold excess bits (i.e. carries), the conditional subtraction can be eliminated. In a modular subtraction, it is not possible to get a negative result due to the final reduction operation (i.e. addition of p). However, this is not true for an ordinary subtraction. In our work, we compute $kp + a - b$ instead of $a - b$ to avoid to get negative results, which is more efficient than doing a reduction since we just need to add two 24-bit words before subtracting. The problem is to decide how many p have to be added, which of course depends on the operands a and b. If $a > b$ is always true then we could just compute $a - b$. Unfortunately, there is no guarantee that this is the case. In our implementation, the field-arithmetic operations are invoked by the point addition and point doubling. In these two operations, the inputs of a modular subtraction are generally the outputs of addition, modular multiplication or squaring. The outputs of modular multiplication and squaring are always in $[0, 2p)$. On the other hand, the output of an addition is in the range of $[0, 4p)$. Therefore, we can avoid a negative result when $k = 4$, which means we can simply replace $a - b \bmod p$ by $4p + a - b$.

Efficient Field Multiplication and Squaring. Modular multiplication and modular squaring are the two most performance-critical arithmetic operations in ECC. In our implementation, they are realized through Montgomery's modular reduction technique introduced in [17]. We use a special variant of the so-called Montgomery multiplication, the so-called Finely Integrated Product Scanning (FIPS) method. The OPF primes we use have a very low Hamming weight so that only the most significant and the least significant 24-bit word needs to be considered in the reduction. We implemented both modular multiplication and

squaring on basis of Liu et al's OPF-FIPS algorithm introduced in [15], which simply ignores all the zero-words and, in this way, achieves very high performance. We refer to [15] for a detailed description of the implementation. Some further optimizations are possible when using a radix-2^{24} representation and following the approach of incomplete modular reduction. In this way, the final subtraction in both the Montgomery multiplication and squaring does not need to be carried out. However, the result of a modular multiplication/squaring is now in the range of $[0, 2p - 1]$. Note that, since the FIPS method is based on the product-scanning approach, we can process carries efficiently as described before. All loops are fully unrolled for performance reasons. The operands are loaded into registers and then we perform the computation in a word by word fashion. Finally, the result is written back to memory.

3.3 Group Operations and Scalar Multiplication

Point Addition and Doubling. The most efficient way to represent a point $P = (x, y)$ on a twisted Edwards curve is to use extended projective coordinates of the form $(X : Y : T : Z)$ as proposed by Hisil et al in [11]. However, in order to further optimize the point arithmetic, our implementation omits the multiplication that produces the auxiliary coordinate T and outputs the two factors E, H it is composed of instead (see [7] for details). In this way, one can obtain more efficient point addition formulae, especially when the curve parameter $a = -1$. By applying these optimizations, the cost of addition and doubling amounts to 7M and $3\mathrm{M} + 4\mathrm{S}$, respectively.

We implemented the point addition/doubling on the Montgomery curve in a straightforward way using exactly the formulae given in [18].

Scalar Multiplication. We benchmark our GPU implementation with two different scalar multiplication techniques. In the case of an arbitrary point (i.e. a base point that that neither constant or known in advance), we use the standard Montgomery ladder on the Montgomery curve. In this way, we have to always execute exactly one point addition and one point doubling for each bit of the scalar, which amounts to $5M + 4S$ per bit. On the other hand, if the base point is fixed, our implementation uses a regular version of the *comb method* with 256 pre-computed points, similar as described in [16]. The idea of the regular comb method is as follows: Since the base point P is fixed, we can do an off-line pre-computation of multiples $d \cdot P$ of P and store them in a table. Then, during the actual scalar multiplication, we process 8 bits of the scalar at a time, and add the corresponding entry from the table to the previous intermediate result. In this way, the number of point doublings is reduced by a factor of 8 compared to the straightforward double-and-add method. The number of point additions is exactly the same as the number of doublings since we exploit the completeness of the Edwards addition law and add the neutral element $\mathcal{O} = (0, 1)$ when an 8-bit block of the scalar is 0. The overall cost of our comb method with 256 pre-computed points amounts to $\frac{10}{8}n\mathrm{M} + \frac{4}{8}n\mathrm{S}$ for an n-bit scalar.

4 Experimental Results

Our experimental platform is an NVIDIA GTX285 graphics card; it contains a GT200 GPU clocked at a frequency of 1476 MHz. The GT200 is nowadays considered a low-end GPU with a compute capability of 1.3.

4.1 Throughput and Latency

The number of blocks per grid and the number of threads per block are two essential parameters of an execution configuration since they directly impact the utilization of the GPU. Furthermore, these two parameters interplay with the memory and register usage. In our evaluation, we focus on three parameters, namely the number of blocks running on each SM, the number of threads per block, and the usage of on-chip memory.

The threads are assigned to an SM in a group of blocks. A block of threads gets scheduled to one available multiprocessor. We can use more than one block to expand the throughput by taking advantages of the 30 SMs. In [1], the best performance was achieved by launching 30 blocks on the GT200. However, the GT200 allows for up to 512 threads per SM, provided that there is sufficient on-chip memory and registers available for each thread. Unfortunately, both is severely limited, which requires to carefully balance the number of threads per block with register and shared memory usage. Furthermore, the performance also varies depending on what kind of memory is used. There are three basic implementation options; we briefly describe them below taking variable-base scalar multiplication on the 160-bit Montgomery curve as example.

- Shared memory can be accessed very fast, but is small. We can achieve the lowest latency when all operands are held in shared memory. However, due to its limited capacity of 16 kB per SM, only up to 80 threads per block can be launched. We call this number of threads the *threads limit point*.
- Global memory is large. Thus, we can move some operands that are not frequently used into global memory. In this case, the threads limit point increases to 144. However, due to slower access time, the latency rises.
- To launch even more threads, we can put all operands into global memory. In this case, the threads limit point is 160 threads per block, determined by the register restriction. Unfortunately, the latency becomes very high.

The resulting throughput and latency of all three cases are illustrated in Figure 5. We can see that the blue line representing the latency is flat until the first threads limit point of 80. Thereafter (i.e. from 96 onwards), the latency rises slightly since now global memory is used to hold parts of operands. After passing the second threads limit point (i.e. 144), only global memory is used, and therefore the latency increases sharply. In our work, throughput refers to the number of point multiplications that can be executed per second, which is an important performance metric. Figure 5 shows that the green bars representing this metric keep increasing until the second thread limit point of 144, where the

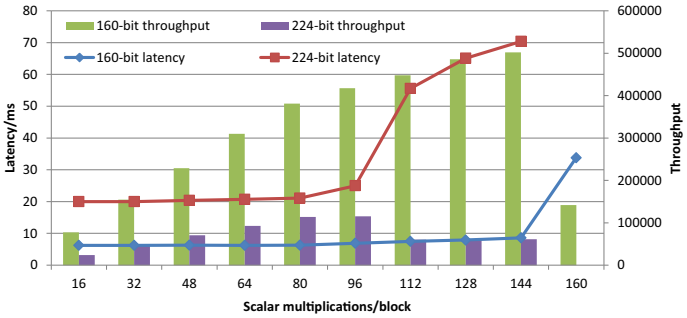

Fig. 5. Throughput and latency for a different number of variable-base scalar multiplications per block (using the Montgomery ladder on a Montgomery curve)

peak is reached. After this point, the throughput declines sharply. Hence, we achieve the highest throughput, namely 502k scalar multiplications per second, with 4320 threads (i.e. 144 threads per block). This shows that one can increase throughput by sacrificing latency; we did this by using both shared memory and global memory for storing operands. For the 224-bit Montgomery ladder, the highest throughput of 115k scalar multiplications per second is achieved when 96 threads per block are launched (i.e. 2880 threads altogether).

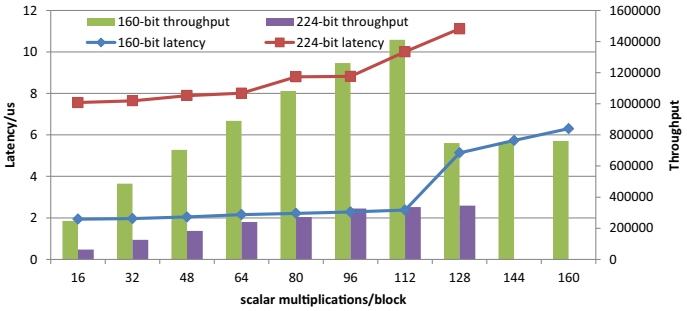

Fig. 6. Throughput and latency for a different number of fixed-base scalar multiplications per block (using a comb method with 256 points on a twisted Edwards curve)

Our implementation of the comb method stores the pre-computed points in constant memory. Figure 6 shows the latency and throughput for a twisted Edwards curves over a 160 and 224-bit OPF, respectively. Table 1 summarizes the maximum performance of the four implementations. The throughput of the comb method is about 1412k and 345k in the 160 and 224-bit case, respectively. The performance is highly dependent on choosing the proper number of scalar multiplications per block. Our results show that 112 and 128 are the best choices for 160 and 224-bit curves, respectively, if one aims for high throughput.

Table 1. Minimum latency and maximum throughput of scalar multiplication

Implementation	Latency [ms]	Throughput [op/s]
160-bit Montgomery ladder	5.9	502326
160-bit Comb method	1.84	1411756
224-bit Montgomery ladder	19.2	115200
224-bit Comb method	4.52	345417

4.2 Comparison with Related Work

In recent years, numerous ECC implementations for GPUs have been reported in the literature. In [1], Antão et al introduced a parallel algorithm for point multiplication using a Residue Number System (RNS) to expose parallelism in the multi-precision integer arithmetic. Their results on the GTX285 platform suggest a maximum throughput of 9990 scalar multiplications per second and a latency of 24.3 ms if the underlying field has a size of 224 bits. Szerwinski and Güneysu [22] presented an implementation on an NVIDIA 8800GTS based on the operand-scanning method for multi-precision multiplication. Their results indicate a throughput of 1412 scalar multiplications per second using the NIST P-224 curve. In [8], Giorgi et al did a comprehensive evaluation of both prime-field arithmetic and point arithmetic (including scalar multiplication) on the NVIDIA 9800 GX2 GPU for operands of different length. When using a 224-bit field, they achieved throughput of 1972 scalar multiplications per second.

Table 2. Comparison of GPU implementations of 224-bit scalar multiplication

Implementation	Platform	Latency [ms]	Throughput [op/s]	Processor clock [MHz]
Szerwinski [22]	8800 GTS	305	1412.6	n./a.
Giorgi [8]	9800 GX2	n./a.	1972	n./a.
Antão [1]	GTX 285	24.3	9990	1476
Bos [6]	GTX 295	10.6	79198	1242
Our work (var. point)	GTX 285	19.2	115200	1476
Our work (fixed point)	GTX 285	4.52	345417	1476

To our knowledge, Bos reported in [6] the best previous result for ECC over a 224-bit prime field on the GT200, even though he optimized latency instead of throughput. He used a Montgomery ladder on a Weierstrass curve for scalar multiplication, which is implemented with 8 threads so as to exploit parallelism in the point operations. As shown in Table 2, he reached a throughput of 79198 scalar multiplications per second, but one has to consider that the GTX295 he used for benchmarking contains two GT200 GPUs, which are clocked with a slightly lower frequency than in our GTX285. Taking these differences into account, our throughput in the variable-base setting is 2.45 times higher than that of Bos. On the other hand, the latency differs by a factor of 2.15.

5 Conclusions

In this work, we combined Optimal Prime Fields (OPFs) with twisted Edwards and Montgomery curves, and implemented both the field and curve arithmetic to match the characteristics of the NVIDIA GT200 GPU. To optimize the field arithmetic with respect to 24-bit integer multipliers of the GT200, we adopted a radix-2^{24} representation for the field elements. This representation facilitates lazy or incomplete modular addition and subtraction since the most significant word contains (at least) 8 vacant bits. We use OPFs as underlying algebraic structure, which allows for very fast modular reduction since only the non-zero words of the prime need to be processed. For point operations on the twisted Edwards curve, extended coordinates are used to represent the points, which allows the point addition to be performed with only seven multiplications in the underlying OPF, while a point doubling requires three multiplications and four squarings. We adopted the complete point addition formulae for curves with parameter $a = -1$. The scalar multiplication uses a regular variant of the comb method with 256 pre-computed points. Regarding the implementation options related to memory (resp. register) usage and number of threads, we scarified latency to get a higher throughput by moving temporary arrays from shared memory to the un-cached global memory. In this way, we managed to achieve a significantly higher throughput than the state-of-the-art.

Acknowledgements. The research described in this paper was supported, in part, by the National Science Foundation of China (61173139), the Specialized Research Fund for the Doctoral Program of Higher Education of China (20110131110027), and the Key Program of the Natural Science Foundation of Shandong Province (ZR2011FZ005).

References

1. Antão, S., Bajard, J.-C., Sousa, L.: Elliptic curve point multiplication on GPUs. In: Proceedings of the 21st IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2010), pp. 192–199. IEEE Computer Society Press (2010)
2. Antão, S., Bajard, J.-C., Sousa, L.: RNS-based elliptic curve point multiplication for massive parallel architectures. Computer Journal 55(5), 629–647 (2012)
3. Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted Edwards curves. In: Vaudenay, S. (ed.) AFRICACRYPT 2008. LNCS, vol. 5023, pp. 389–405. Springer, Heidelberg (2008)
4. Bernstein, D.J., Chen, H.-C., Chen, M.-S., Cheng, C.-M., Hsiao, C.-H., Lange, T., Lin, Z.-C., Yang, B.-Y.: The billion-mulmod-per-second PC. In: Proceedings of the 4th Workshop on Special-Purpose Hardware for Attacking Cryptographic Systems (SHARCS 2009), Lausanne, Switzerland, pp. 131–144 (September 2009)
5. Bernstein, D.J., Chen, T.-R., Cheng, C.-M., Lange, T., Yang, B.-Y.: ECM on graphics cards. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 483–501. Springer, Heidelberg (2009)

6. Bos, J.W.: Low-latency elliptic curve scalar multiplication. International Journal of Parallel Programming 40(5), 532–550 (2012)
7. Chu, D., Großschädl, J., Liu, Z., Müller, V., Zhang, Y.: Twisted Edwards-form elliptic curve cryptography for 8-bit AVR-based sensor nodes. In: Proceedings of the 1st ACM Workshop on Asia Public-Key Cryptography (AsiaPKC 2013), pp. 39–44. ACM Press (2013)
8. Giorgi, P., Izard, T., Tisserand, A.: Comparison of modular arithmetic algorithms on GPUs. In: Parallel Computing: From Multicores and GPU's to Petascale. Advances in Parallel Computing, vol. 19, pp. 315–322. IOS Press (2010)
9. Großschädl, J.: TinySA: A security architecture for wireless sensor networks. In: Proceedings of the 2nd International Conference on Emerging Networking Experiments and Technologies (CoNEXT 2006), pp. 288–289. ACM Press (2006)
10. Hankerson, D.R., Menezes, A.J., Vanstone, S.A.: Guide to Elliptic Curve Cryptography. Springer (2004)
11. Hisil, H., Wong, K.K.-H., Carter, G., Dawson, E.: Twisted Edwards curves revisited. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 326–343. Springer, Heidelberg (2008)
12. Jang, K., Han, S., Han, S., Moon, S., Park, K.: SSLShader: Cheap SSL acceleration with commodity processors. In: Andersen, D.G., Ratnasamy, S. (eds.) Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2011). USENIX Organization (2011)
13. Khan, F.G.: General Purpose Computation on Graphics Processing Units using OpenCL. Ph.D. Thesis, Politecnico di Torino, Torino, Italy (March 2013)
14. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro 28(2), 39–55 (2008)
15. Liu, Z., Großschädl, J., Wong, D.S.: Low-weight primes for lightweight elliptic curve cryptography on 8-bit processors. In: Lin, D., Xu, S., Yung, M. (eds.) The 9th China International Conference on Information Security and Cryptology — INSCRYPT 2013. LNCS. Springer, Heidelberg (to appear)
16. Liu, Z., Wenger, E., Großschädl, J.: MoTE-ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks (February 2013) (to be published)
17. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
18. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48(177), 243–264 (1987)
19. NVIDIA Corporation. NVIDIA GeForce® GTX 200 GPU Architectural Overview. Technical brief (2008), `http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf`
20. NVIDIA Corporation. CUDA C Programming Guide. Design guide (2013), `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`
21. NVIDIA Corporation. Parallel Thread Execution ISA. Application guide (2013), `http://docs.nvidia.com/cuda/pdf/ptx_isa_3.2.pdf`
22. Szerwinski, R., Güneysu, T.: Exploiting the power of GPUs for asymmetric cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 79–99. Springer, Heidelberg (2008)
23. Yanık, T., Savaş, E., Koç, Ç.K.: Incomplete reduction in modular arithmetic. IEE Proceedings – Computers and Digital Techniques 149(2), 46–52 (2002)