Efficient Implementation of the η_T Pairing on GPU

Yosuke Katoh[1], Yun-Ju Huang[2], Chen-Mou Cheng[3], and Tsuyoshi Takagi[4]

[1] Graduate School of Mathematics, Kyushu University,
744 Motooka, Nishi-ku, Fukuoka, 819-0395, Japan.
[2] Institute of Information Science, Academia Sinica,
128, Section 2, Academia Road, Taipei, 115, Taiwan.
[3] Department of Electrical Engineering, National Taiwan University,
1, Section 4, Roosevelt Road, Taipei, 106, Taiwan.
[4] Institute of Mathematics for Industry, Kyushu University,
744 Motooka, Nishi-ku, Fukuoka, 819-0395, Japan.

Abstract. Recently, efficient implementation of cryptographic algorithms on graphics processing units (GPUs) has attracted a lot of attention in the cryptologic research community. In this paper, we deal with efficient implementation of the η_T pairing on supersingular curves over finite fields of characteristics 3. We report the performance results of implementations on NVIDIA GTX 285, GTX 480, Tesla C1060, and Tesla C2050 graphics cards. We have implemented η_T pairing in three different ways, namely, one pairing by one thread (Implementation I), one pairing by multiple threads (Implementation II), and multiple pairings by multiple threads in a bitsliced fashion (Implementation III). The timing for Implementation III on a single GTX 285 is 1.47, 8.15, and 140.7 milliseconds for η_T pairing over $\mathbb{F}_{3^{97}}$, $\mathbb{F}_{3^{193}}$, and $\mathbb{F}_{3^{509}}$, respectively. On a single GTX 480, the throughput performance of Implementation III is 33710, 4970, and 332 η_T pairings per second over $\mathbb{F}_{3^{97}}$, $\mathbb{F}_{3^{193}}$, and $\mathbb{F}_{3^{509}}$, respectively. To the best of our knowledge, this is the first implementation of η_T pairing on GPU. Furthermore, it is currently the software implementation that achieves the highest single-chip throughput for η_T pairing.

Key words: η_T pairing, graphics processing unit

1 Introduction

Use of graphics processing unit (GPU) to accelerate general computational applications is usually called "general-purpose computation on graphics processing units" (GPGPU). GPGPU generally follows the paradigm of SPMD (Single Program Multiple Data), in which many processors simultaneously execute the same commands, much like they would do in SIMD (Single Instruction Multiple Data) but with a more relaxed synchronization constraint. Unlike multi-core x86 microprocessors, GPUs usually have simpler instruction sets and smaller caches, allowing most transistors to be used in arithmetic circuitries.

In recent years, GPGPU has attracted a lot of attention in computational research communities. For example, cryptographic algorithms are studied and implemented on GPU, such as AES [9], RSA [10], and elliptic-curve point scalar arithmetic [17]. Furthermore, cryptanalysis computation can also be carried out with GPU [5, 6]. In particular, Bernstein *et al.* has demonstrated that GPGPU can be more efficient and cost-effective than traditional CPU, e.g., a single NVIDIA GTX 295 graphics card can complete 41.88 million modular multiplications every second for 280-bit moduli, compared with 14.85 million achieved by an Intel Core 2 Quad Q9550 CPU [5]. It is also shown that GPGPU can be more efficient than several other platforms including AMD Phemon III and Sony PlayStation3 [4]. Furthermore, it is shown that GPUs can perform not only modular multiplications but also binary-field multiplications in, e.g., $\mathbb{F}_{2^{131}}$ [2].

Recently, pairing-based cryptography has attracted a lot of attention in cryptography due to its novel cryptographic applications such as identity-based encryption. In this paper, we deal with efficient implementation of the η_T pairing on supersingular curves over finite fields of characteristics 3. The timing of several previously reported implementations of the Tate pairing is quite efficient in both software and hardware platforms [7, 13]. However, to the best of our knowledge, there are no reports on implementations of the η_T pairing.

We first report our experience implementing η_T pairing on NVIDIA GTX 285, GTX 480, Tesla C1060, and Tesla C2050 graphics cards. For multi-threaded GPU implementation, efficient exploitation of the inherent parallelism in the algorithms is of vital importance. We will show three different η_T pairing implementation strategies in the following sections. In Implementation I, we use one single thread per pairing computation, while in Implementation II, we use multiple threads per pairing computation. In Implementation III, we compute several parings using multiple threads in a bitsliced fashion at the same time. For Implementation III, it takes 1.47, 8.15, and 140.7 milliseconds to compute one η_T pairing over $\mathbb{F}_{3^{97}}$, $\mathbb{F}_{3^{193}}$, and $\mathbb{F}_{3^{509}}$, respectively. On a single GTX 480, the throughput performance of Implementation III is 33710, 4970, and 332 η_T pairings per second over $\mathbb{F}_{3^{97}}$, $\mathbb{F}_{3^{193}}$, and $\mathbb{F}_{3^{509}}$, respectively. To the best of our knowledge, this is the first GPU implementation result for η_T pairing.

The rest of this paper is organized as follows. In Section 2, we will give some necessary background information about the development environment and the various NVIDIA GPUs that we use for our implementations. In Section 3, we will present η_T pairing over $\mathbb{F}_{3^m}$ and our design that maps the mathematical computation onto GPU. We will then describe our implementation in detail and present the performance results in Section 4. Finally, we will conclude this paper in Section 5.

2 NVIDIA GPU

In this section, we will give some background information about CUDA (Compute Unified Device Architecture), the integrated development environment for

NVIDIA GPU, as well as the various NVIDIA GPUs that we use for our implementations.

2.1 CUDA

CUDA allows application programmers to program NVIDIA GPU using a high-level, C-like programming language [15]. A GPU program is then compiled as follows. First, the usual C/C++ code running on CPU is separated and compiled using a standard C/C++ compiler such as `gcc`. The other part of the program, called the "kernel," which is written with CUDA extension and targeted for running on GPU, is compiled by `nvcc`, a proprietary CUDA compiler. The output of `nvcc` is in a format called PTX (Parallel Thread eXecution), an assembly-language-like intermediary language, which is translated into the real machine code by CUDA driver before loading onto GPU.

CUDA adopts an SPMD paradigm, in which multiple data are processed simultaneously. When launching a kernel, usually a large number of threads are created to process a large amount of data in parallel. In order to reduce hardware cost and allow transparent scalability, threads are organized according to a two-level hierarchy in CUDA. Under this thread organization, each kernel uses one single grid of thread blocks, each of which can consist up to several hundreds of threads. The threads from a same thread block can share data and cooperate with each other via shared memory and barrier synchronization primitives. Threads from different thread blocks can not cooperate and hence must be able to run independently.

2.2 G2xx series cards

We use NVIDIA GeForce 200 series cards as one of our target platforms. Each GPU on such a card contains 2–30 streaming multiprocessors (SMs). Each SM contains 8 ALUs (arithmetic-logic units; in CUDA's own terminology, streaming processors, or SPs) and 2 super function units (SFUs). For example, the GT200b GPU on a GTX 285 graphics card contains 30 SMs, meaning that it has 240 SPs (or "cores,"as NVIDIA puts it). Moreover, each SM has only one instruction decode and dispatch unit, so all of its 8 ALUs must execute the same instructions simultaneously. In fact, to accommodate the throughput difference between instruction decoding and execution, CUDA defines a notion of minimal scheduling unit called a "thread warp," or simply a warp. On GT200b, one thread warp consists of 32 threads. Therefore, on GT200b, each normal instruction for the 32 threads in a warp can be processed by the 8 SPs in 4 cycles.

Unlike SIMD, threads from the thread warp *can* diverge and execute different commands at a cost of performance degradation. That is, if the threads of a same warp have different commands to execute, then the execution of the warp will be serialized. However, that threads of a warp can execute different commands does not mean they should. On the contrary, all threads within the same warp better execute the same commands as much as possible to make efficient use of

the execution engine. Therefore, it is very important to use the warp unit wisely to have an efficient GPU implementation.

CUDA also uses a multi-level memory model, which will be described in detail below. We also describe the available memory for Compute Capability 1.3, which is a way that NVIDIA differentiates the capabilities of GPUs of different generations.

- Register file: 16384 32-bit registers, or a 64 KB register file on each SM. It allows the fastest read-write operation to the data stored in it. Registers are private and can only be read and written by the owning threads.
- Shared memory: 16 KB read-write memory on each SM. It is organized into 16 32-bit banks and allows fast access (same speed as register file if no bank conflict). Shared memory can only be read and written by the threads belonging to the same thread block.
- Global memory: 1–2 GB read-write memory. This is known to be 100 times slower in terms of latency than accessing shared memory. Global memory can be read and written by all the threads on the GPU.
- Constant memory: 64 KB read-only memory. It is faster than global memory because it is cached. Constant memory can only be read by all the threads on the GPU.

Finally, we note that though each SM has a larger register file than shared memory, registers are private to each thread and hence can not be shared across different threads. Furthermore, registers can only be addressed from within instructions and hence are not as flexible as shared memory in terms of addressing modes. Therefore, it is important to use the shared memory efficiently. For compute Capability 1.3, each SM can have up to 32 simultaneous warps (1024 threads), 16384 32-bit registers, and 16 KB shared memory. We are advised to run as many thread blocks as possible under the resource constraints to allow efficient latency hiding. For example, consider a thread block consisting of 4 warps. If we consider the maximal number warps per SM, then we are able to run up to 8 thread blocks per SM. However, if each thread uses 20 registers, then we can only run 6 thread blocks per SM. If one thread block uses 4 KB shared memory, then we are restricted to 4 blocks per SM. In conclusion, in order to obtain the maximal performance, it is important to balance the number of parallel threads per SM with register and shared memory usage.

2.3 G4xx series cards

In our implementation, we also use the GTX 480 graphics card, which uses the new GF100 GPU chip. GF100 belongs to the Fermi architecture family and has Compute Capability 2.x, whose most cited improvement over previous generations is the improved performance of double-precision floating-point arithmetic. However, as we do not use double-precisions, this improvement is irrelevant in the context of this paper, so we focus on the other aspects of the Fermi family in the following discussion.

In the new Fermi architecture, NVIDIA has renamed an SP to a CUDA core. The number of CUDA cores per GPU is increased from 240 to 512, and each SM now has 32 CUDA cores instead of 8 SPs. As a result, the total number of SMs per GPU is decreased from 30 to 16. Furthermore, in order to have better manufacturing yield, NVIDIA disables one SM per GTX 480, leaving only 15 active SMs (or 480 CUDA cores). For the high-end server market, they disable two SMs per Telsa C2050 card, leaving only 14 active SMs (or 448 CUDA cores).

Each SM on GF100 has two instruction units (or warp scheduler) instead of one in the previous generation. The number of SFUs is also doubled from two to four. It now takes two cycles to execute one instruction for one warp. Moreover, each SM now has 16–48 KB of L1 cache, and all SMs share 768 KB of coherent L2 cache. The latency of accessing L1 and L2 caches is dozens and hundreds of clock cycles, respectively. The shared memory is increased from 16 to 48 KB, and the number of shared memory banks is also increased from 16 to 32. However, shared memory and L1 cache must add up to 64 KB in total, as they are implemented using the same physical memory. In other words, if we use 48 KB of shared memory, then we can only have up to 16 KB of L1 cache, and vice versa. As the primary cache of the global memory, L1 cache is automatically handled by GPU. Therefore, unlike in the previous architecture, even if there is no special memory management on the shared memory, the performance will also be somewhat improved. Of course, in order to get better performance, we need to manage the fast on-die memory by ourselves. There are 32768 32-bit registers in one SM, totaling about 128 KB. However, the overall fast memory on GPU is only slightly increased from 1920 to 2048 KB.

It is of crucial importance to consider the trade-off between number of threads, available registers, and amount of shared memory per thread in the Fermi architecture, just as it was for GPUs of previous generations. Within the limit of memory resource consumption, we would like to run as many thread blocks as possible to maximize the GPU utilization. With the G2xx series cards, we are able to use a large number of threads to hide the memory latency and use a small amount of shared memory efficiently to speed up the calculation. In contrast, the Fermi architecture has a larger amount of shared and register memory per SM, which has even a smaller latency. Therefore, it would result in further performance improvement if we use the on-die memory efficiently. When we fix the number of threads, we can use more registers on Fermi than G2xx series GPU in order to maximally utilize all compute resources.

Telsa is NVIDIA's dedicated GPGPU product. In this paper, we test our implementation on Tesla C1060 and C2050. Basically, Tesla C1060 and C2050 corresponds to the architecture of G2xx and G4xx series cards, respectively. We summarize the characteristics of the graphics cards we have used to test our implementation in Table 1.

Table 1. Specifications of several NVIDIA graphics cards

Specifications	GTX 285	GTX 480	Tesla C1060	Tesla C2050
CUDA Cores	240	480	240	448
Arithmetic Clock (GHz)	1.476	1.401	1.296	1.15
Single-precision GFLOPS	1063	1345	933	1030
Standard Memory Configuration (GB)	1	1.536	4	3
Memory Bandwidth (GB/sec)	159	177.4	102	144

3 Parallel η_T pairing computation

In this section, we will describe in detail the η_T pairing over $\mathbb{F}_{3^m}$, the optimal representation of elements, multiplication in $\mathbb{F}_{3^m}$ for GPU, as well as the parallel thread assignment.

3.1 Definition of η_T pairing

The η_T pairing with characteristic 3 can be defined on the supersingular elliptic curve

$$E : y^2 = x^3 - x + b,\ b = \pm 1.$$

Let r be the largest prime such that $r \mid \#E(\mathbb{F}_{3^m})$, and $r \mid (3^{6m} - 1)$. Let the subgroup with order r in $E(\mathbb{F}_{3^m})$ be denoted as $E(\mathbb{F}_{3^m})[r]$. Then the η_T pairing is a bilinear mapping

$$\eta_T : E(\mathbb{F}_{3^m})[r] \times E(\mathbb{F}_{3^{6m}})/rE(\mathbb{F}_{3^{6m}}) \longrightarrow \mathbb{F}^*_{3^{6m}}/(\mathbb{F}^*_{3^{6m}})^r.$$

3.2 Representation of elements in $\mathbb{F}_{3^m}$

We have tried three approaches for parallel computation of η_T paring on GPU. One approach is to use multiple threads to calculate one single element in $\mathbb{F}_{3^m}$ in parallel, while another approach is to use multiple threads to calculate multiple elements in $\mathbb{F}_{3^m}$ in a bitsliced fashion. In this paper, we call the former approach Implementation II (Parallel), and the latter, Implementation III (Bitsliced). For comparison purposes, we also include the basic approach that does not use any parallel computation, which we call Implementation I (Serial). We are going to describe the representation of elements in $\mathbb{F}_{3^m}$ for Implementation I, II, and III, as summarized in Table 2.

Table 2. Characteristics of the implementations

Implementation	I	II	III
Characteristic	Serial	Parallel	Bitsliced

Implementation I and II. Let $\mathbb{F}_3[x]$ be the set of polynomials with coefficient from the prime field $\mathbb{F}_3 = \{0, 1, 2\}$. Let $f(x)$ be an irreducible polynomial with degree m in $\mathbb{F}_3[x]$. In this case, the field $\mathbb{F}_{3^m}$ can be expressed as $\mathbb{F}_{3^m} = \mathbb{F}_3[x]/(f(x))$. Two bits are necessary to represent an element in $\mathbb{F}_3$. In this paper, we use representation given by Kawahara, which uses one "hi bit"and one "lo bit"to represent each element in $\mathbb{F}_3$ [11]. For $A(x) \in \mathbb{F}_{3^m}$ with degree $m-1$,

$$A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1 x + a_0$$

is the representation of $A(x)$. To represent $A(x)$, we need m hi bits and m lo bits, that is, with 32 bits in a word, $2 \times \lceil m/32 \rceil$ words are needed. For example, if m is 97, then an element in $\mathbb{F}_{3^{97}}$ is represented as follows.

$$\begin{aligned}
A_{hi}[3] &= (0, \ldots, 0, (a_{96})_{hi}) \\
A_{lo}[3] &= (0, \ldots, 0, (a_{96})_{lo}) \\
A_{hi}[2] &= ((a_{95})_{hi}, \ldots, (a_{65})_{hi}, (a_{64})_{hi}) \\
A_{lo}[2] &= ((a_{95})_{lo}, \ldots, (a_{65})_{lo}, (a_{64})_{lo}) \\
A_{hi}[1] &= ((a_{63})_{hi}, \ldots, (a_{33})_{hi}, (a_{32})_{hi}) \\
A_{lo}[1] &= ((a_{63})_{lo}, \ldots, (a_{33})_{lo}, (a_{32})_{lo}) \\
A_{hi}[0] &= ((a_{31})_{hi}, \ldots, (a_1)_{hi}, (a_0)_{hi}) \\
A_{lo}[0] &= ((a_{31})_{lo}, \ldots, (a_1)_{lo}, (a_0)_{lo})
\end{aligned}$$

Implementation III. Similar to Implementation II, Implementation III uses hi bits and lo bits to represent $A(x)$, too. Specifically, it uses $2 \times m$ words, with 32 bits in a word to store 32 $\mathbb{F}_{3^m}$ elements. This strategy is known as bitslicing, in which a word is treated as a vector of bits from different elements [2]. In this paper, the notation of the j-th element $A(x)_{[j]}$ in $\mathbb{F}_{3^m}$ is

$$A(x)_{[j]} = a_{(m-1)[j]}x^{m-1} + \cdots + a_{1[j]}x + a_{0[j]}.$$

For example, if m is 97, then the 32 elements in $\mathbb{F}_{3^{97}}$ are represented as follows.

$$\begin{aligned}
A_{hi}[96] &= ((a_{96\,[31]})_{hi}, \cdots, (a_{96\,[1]})_{hi}, (a_{96\,[0]})_{hi}) \\
A_{lo}[96] &= ((a_{96\,[31]})_{lo}, \cdots, (a_{96\,[1]})_{lo}, (a_{96\,[0]})_{lo}) \\
&\vdots \\
A_{hi}[0] &= ((a_{0\,[31]})_{hi}, \cdots, (a_{0\,[1]})_{hi}, (a_{0\,[0]})_{hi}) \\
A_{lo}[0] &= ((a_{0\,[31]})_{lo}, \cdots, (a_{0\,[1]})_{lo}, (a_{0\,[0]})_{lo})
\end{aligned}$$

Unlike Implementation II, Implementation III does not contain any redundant 0's in the representation. In other words, in Implementation II, it takes $32 \times 2 \times \lceil m/32 \rceil$ words to represent 32 $\mathbb{F}_{3^m}$ elements, while in Implementation III, $2 \times m$.

3.3 Multiplication in $\mathbb{F}_{3^m}$

The bulk of the η_T pairing computation is multiplication. We will show the multiplication algorithms of Implementation I, II, and III in $\mathbb{F}_{3^m}$. To obtain the optimal performance, we try our best to fit the entire input in the shared memory and perform all operations within the shared memory to avoid going to the slow global memory. However, for the Implementation III over $\mathbb{F}_{3^{509}}$ in pre-Fermi architecture, we are unable to store all the data in shared memory, and hence some temporary data have to be stored in global memory.

Implementation I. For elements $A(x), B(x)$ in F_{3^m} with irreducible polynomial $f(x)$, the multiplication $C(x) = A(x) \cdot B(x) \bmod f(x)$ over the finite field $\mathbb{F}_{3^m}$ requires a polynomial multiplication of $A(x) \cdot B(x)$ followed by a reduction by $f(x)$. The Comb method [12] is used as the polynomial multiplication in Implementation I, as shown in Algorithm 1.

Algorithm 1 Comb multiplication in $\mathbb{F}_{3^m}$ [12]

INPUT: $A(x), B(x) \in \mathbb{F}_{3^m}, W$
OUTPUT: $C(x) = A(x) \cdot B(x)$
1: $C(x) \leftarrow 0$
2: **for** $i \leftarrow 1$ to $W - 1$ **do**
3: **for** $j \leftarrow 0$ to $\lfloor m/W \rfloor$ **do**
4: $C(x) \leftarrow C(x) + A[j]_i \cdot B(x)x^{jW+i}$
5: **end for**
6: **end for**
7: **return** $C(x) \leftarrow \mathit{Reduction}(C(x))$

For the reduction, we use the ROT method (Reduction Optimal Trinomials) proposed by Nakajima *et al.* [14] However, since there is no parallel processing in Implementation I, we use only one thread here.

Implementation II. Implementation II is basically Implementation I with parallelism. In order to perform in parallel the operations $A[j]_i \cdot B(x)x^{jW+i}$ at Step 4 in Algorithm 1, we assign the word length W as one block in the Comb method and perform the addition on $\mathbb{F}_{3^m}$ $\lceil m/W \rceil$ times in each thread. Thus, each thread's main work is the expanded computation in the nested loops from Step 2 to 6. Since the number of threads needs to be the same as the word length 32, we can simply use an entire warp for this part so that the SIMD efficiency is maximized at the same time. The total number of bit operations needed is $6 \times \lceil m/32 \rceil \times m + (2 \times \lceil m/32 \rceil) \times 31 \times 32$.

The reduction part is straightforward to implement but difficult to parallelize on GPU. We just note that we do not use any windowed methods because we believe that the bank conflicts of shared memory or the latency of global memory

will offset any benefits parallelization can bring when a table is simultaneously accessed by multiple threads.

Implementation III. The method used in Implementation III to perform 32 multiplications is shown in Algorithm 2. Implementation III works with $2 \times m$ arrays for hi bits and lo bits. We use m threads to operate on these arrays in order to maximize parallelism. Thus, for the expanded operations in each loop at Step 2 and 3, each array is assigned to one thread to compute.

Algorithm 2 Multiplication of Implementation III in $\mathbb{F}_{3^m}$

INPUT: $A(x)_{[j]}, B(x)_{[j]} \in \mathbb{F}_{3^m}$, $j = \{0, \ldots, m-1\}$
OUTPUT: $C(x)_{[j]} = A(x)_{[j]} \cdot B(x)_{[j]}$
1: $C(x)_{[2 \cdot j]} \leftarrow 0$
2: **for** $i \leftarrow 0$ to $m-1$ **do**
3: **for** $k \leftarrow 0$ to $m-1$ **do**
4: $\acute{C}_{hi}[i+k] \leftarrow A_{hi}[i]$ & $B_{lo}[k] \mid A_{lo}[i]$ & $B_{hi}[k]$
5: $\acute{C}_{lo}[i+k] \leftarrow A_{hi}[i]$ & $B_{hi}[k] \mid A_{lo}[i]$ & $B_{lo}[k]$
6: $C(x)_{[i+k]} \leftarrow C(x)_{[i+k]} + \acute{C}(x)_{[i+k]}$
7: **end for**
8: **end for**
9: **return** $C(x)_{[j]} \leftarrow Reduction(C(x)_{[2 \cdot j]})$

The total number of bit operations needed is $6 \times m^2 + 6 \times m^2$. We note that although the Karatsuba algorithm is also known as a fast multiplication method, it may not be as efficient for our representation of $\mathbb{F}_{3^m}$ elements, in which addition is as expensive as multiplication. In addition, at Step 4 and 5, each array only needs to be accessed by the corresponding thread. Thus, we are able to obtain a further speed-up by storing these arrays in thread-private registers. Reduction is also done by the same threads in parallel. However, same as in Implementation II, the parallelization of reduction is not as efficient due the SPMD constraints.

4 Implementation results

In this section, we first describe three different types of implementation of η_T pairing on GTX 285 that we have considered in this paper. Next, we report the performance of the best one, Implementation III, on GTX 285, GTX 480, Tesla C1060, and Tesla C2050. Then we show the experiment result of multiplication throughput for η_T pairing.

4.1 Implementation techniques

Although CUDA's SPMD model allows conditional branches within a warp, it can cause divergent warps and, as a result, performance degradation. Therefore,

to achieve optimal performance, we would need to minimize use of conditional branches in the code. We achieve so whenever possible by choosing appropriate algorithms for the operations in $\mathbb{F}_{3^m}$ that are needed for computing η_T pairing. For example, since the cubing calculation of $A(x)$ in $\mathbb{F}_{3^m}$ is fast and parallelizable, we use Fermat's method to calculate the inverse of $A(x)$ in $\mathbb{F}_{3^m}$ by taking $A(x)$ to the power of (3^m-2). We also use the algorithm proposed by Shirase *et al.* [3] as the main loop for our η_T pairing computation and the algorithm implemented with torus stated by Shirase *et al.* [16] Some temporary data produced in the pairing computation need to be cached in registers. If we finish the entire pairing computation in one kernel launch, it would require a large number of registers for storing the temporary data. Thus, in order to balance the register usage, we split the computation into two parts, namely, the Miller loop in Algorithm 2 and the final exponentiation in Algorithm 3 [16].

As of the time of writing, the NVIDIA GeForce GTX 285 is considered as a mid-class graphics card. It has a GPU that runs at 1476 MHz. We are able to fit the entire computation on one single SM because our design allows all critical data structures stored entirely on the on-die memory of GT200b, as shown in Section 4.2. Then, we implement η_T pairing and multiplication in $\mathbb{F}_{3^m}$ on 30 SMs and summarize our results in Section 4.3. In the reports, we use `cudaEvent`, part of the CUDA API, for time measurement. To match the security levels of 66-, 89-, and 128-bit AES, we choose three finite fields, $\mathbb{F}_{3^{97}}$, $\mathbb{F}_{3^{193}}$, and $\mathbb{F}_{3^{509}}$ for our implementation of η_T pairing.

4.2 Performance results I

The performance results on GTX 285 over $\mathbb{F}_{3^{97}}$, $\mathbb{F}_{3^{193}}$, and $\mathbb{F}_{3^{509}}$ are shown in Table 3, Table 4, and Table 5, respectively.

In the above tables, the running time for Implementation III is normalized to per $\mathbb{F}_{3^m}$ element for comparison purposes. As we have mentioned in Section 3, the number of bit operations in multiplication in Implementation III ($12 \times m^2$) is more than that in Implementation I and II ($(192 \times m + 1984) \times \lceil m/32 \rceil$). However, from Table 3, 4, and 5, we can see that Implementation III is faster than Implementation I and II in all cases. According to Section 3.2, this is possible because the total computation time of GPU also depends on the amount of data transfer; compared to the memory footprint of Implementation I and II ($32 \times \lceil m/32 \rceil \times 2$ 32-bit arrays), that of Implementation III is more compact ($m \times 2$ arrays) and thus incurs a smaller amount of memory transfer. Moreover, unlike Implementation I and II, the polynomials multiplication in Implementation III, not including reduction, does not result in any warp divergences. That is, there are no divergent branches in the multiplication of Implementation III, which makes it much faster than its counterparts in Implementation I and II.

Based on our experiment data, Implementation III has the shortest calculation time per pairing. In Implementation III, the number of threads in one block depends on the size of the array and the degree of pairing. Therefore, to increase the flexibility, we have considered the feasibility that uses one single thread for computing 32 pairings, as well as an arbitrary number of threads for computing

Table 3. Per-SM running time (ms) for $\mathbb{F}_{3^{97}}$

Implementation	I	II	III
Addition	0.00014	0.0003	0.00002
Subtraction	0.00014	0.0003	0.00002
Multiplication	0.1748	0.0175	0.001
Cubing	0.0499	0.0096	0.00008
Inverse	22.07	3.09	0.11
η_T pairing	337.53	43.75	1.472

Table 4. Per-SM running time (ms) for $\mathbb{F}_{3^{193}}$

Implementation	I	II	III
Addition	0.00023	0.0003	0.00002
Subtraction	0.00024	0.0003	0.00002
Multiplication	0.38	0.025	0.003
Cubing	0.052	0.0116	0.0001
Inverse	83.88	7.79	0.63
η_T pairing	1323.01	109.42	8.15

Table 5. Per-SM running time (ms) for $\mathbb{F}_{3^{509}}$

Implementation	I	II	III
Addition	0.00044	0.00078	0.00003
Subtraction	0.00043	0.00080	0.00003
Multiplication	1.02	0.135	0.023
Cubing	0.194	0.029	0.0004
Inverse	421.57	52.69	10.4
η_T pairing	6100.5	732.3	140.73

32 parings in one block. Such a design turns out to be slower than implementation III. It is a trade-off between efficiency (the number of threads should be an integral multiple of 32, the number of threads in a warp) and flexibility (being able to use an arbitrary number of threads).

4.3 Performance results II

In this section, we are going to show the throughput performance of η_T pairing and multiplication in $\mathbb{F}_{3^m}$ using all the SMs provided by NVIDIA GTX 285, GTX 480, Tesla C1060, and Tesla C2050 graphics cards. We focus on Implementation III because it is the fastest among the three according to Section 4.2. As we have described in Section 2.2, the balance between number of threads, register used, and the amount of shared memory used in each block is important. When a block of threads is accessing memory or waiting for data transfer, we need other blocks to fill in the gaps in order to fully utilize the compute

resources. That is, the number of active thread blocks per SM is an important performance index. For this purpose, we choose the number of thread blocks to be an integral multiple of the number of SMs on the target GPU, which is 30 for GTX 285 and Tesla C1060, 15 for GTX 480, and 14 for Tesla C2050. We show the total number of thread blocks used in η_T pairing in Table 6 and in multiplication in Table 7. Also, the throughput of η_T pairing is shown in Table 8, while the throughput of $\mathbb{F}_{3^m}$ shown in Table 9.

Table 6. Total number of blocks used for η_T pairing

Base field	GTX 285	Tesla C1060	GTX 480	Tesla C2050
$\mathbb{F}_{3^{97}}$	60	60	60	98
$\mathbb{F}_{3^{193}}$	30	30	30	56
$\mathbb{F}_{3^{509}}$	20	20	30	28

Table 7. Total number of blocks used for multiplication in $\mathbb{F}_{3^m}$

Base field	GTX 285	Tesla C1060	GTX 480	Tesla C2050
$\mathbb{F}_{3^{97}}$	120	120	120	112
$\mathbb{F}_{3^{193}}$	60	60	90	84
$\mathbb{F}_{3^{509}}$	30	30	90	84

For multiplications in $\mathbb{F}_{3^m}$, we store half of the temporary data in registers and half in shared memory. As a result, the maximal number of blocks per SM is limited by the number of available registers. For GTX 285 and Tesla C1060,

Table 8. Throughput performance of η_T pairing on NVIDIA GPUs (1/sec)

Base field	GTX 285	Tesla C1060	GTX 480	Tesla C2050
$\mathbb{F}_{3^{97}}$	23496	20619	33712	31250
$\mathbb{F}_{3^{193}}$	3257	2874	4975	4425
$\mathbb{F}_{3^{509}}$	81	53	332	254

Table 9. Throughput performance of multiplication on NVIDIA GPUs (10^6/sec)

Base field	GTX 285	Tesla C1060	GTX 480	Tesla C2050
$\mathbb{F}_{3^{97}}$	41.20	36.23	61.43	47.45
$\mathbb{F}_{3^{193}}$	11.20	9.80	17.50	13.87
$\mathbb{F}_{3^{509}}$	0.57	0.38	2.85	2.21

in the case of $\mathbb{F}_{3^{509}}$, the number of registers required exceeds that are available per SM. There are 509 threads in total, so we use 32 registers for each thread instead and spill the rest into local memory. This is faster than the alternative in which the computation is split among multiple threads because the latter would require a lot of inter-thread communication and synchronization. Moreover, for the multiplication on $\mathbb{F}_{3^{509}}$, since the temporary data can not all be stored in

shared memory, we have to store some of them in global memory instead. On the other hand, for GTX 480 and Tesla C2050 (Fermi architecture), since the number of registers and shared memory are both increased, the memory limitation is no longer a problem. As the result, performance on GTX 480 and Tesla C2050 is better than that on GTX 285 and Tesla C1060.

There have been several efficient implementations of cryptographic pairings on multi-core CPUs [1, 8, 13]. We will now compare our results with those obtained on multi-core CPUs in the literature [1, 13]. The experiment results are summarized in Table 10, in which we compare the execution time per η_T pairing on GPU and various multi-core CPU implementations of a cyptographic pairing on supersingular curves in characteristics 2 and 3. The acceleration on multi-core CPUs achieved by an n-core implementation is almost always less than the ideal $n\times$ speed-up. The timing shown in Table 10 is the normalized time per

Table 10. Performance comparison of η_T pairings on multi-core processors

	Curve	Architecture	#cores	Freq. (GHz)	Time (ms)
Beuchat *et al.* [13]	$E(\mathbb{F}_{3^{97}})$	Intel Core 2	2	2.6	0.090
This work	$E(\mathbb{F}_{3^{97}})$	NVIDIA GTX 480	480	1.4	0.029
Beuchat *et al.* [13]	$E(\mathbb{F}_{3^{193}})$	Intel Core 2	2	2.6	0.550
This work	$E(\mathbb{F}_{3^{193}})$	NVIDIA GTX 480	480	1.4	0.201
Aranha *et al.* [1]	$E(\mathbb{F}_{2^{1223}})$	Intel Xeon 45nm	8	2.0	1.51
Beuchat *et al.* [13]	$E(\mathbb{F}_{3^{509}})$	Intel Core 2	4	2.4	2.94
Beuchat *et al.* [13]	$E(\mathbb{F}_{3^{509}})$	Intel Core i7	8	2.9	1.87
This work	$E(\mathbb{F}_{3^{509}})$	NVIDIA GTX 480	480	1.4	3.01

η_T pairing, i.e., the total compute time divided by the total number of pairings. From the comparison, we can see that GTX 480 has the highest throughput performance over smaller finite fields.

Based on our experiment data on GTX 480, it would be possible to achieve 61.43 million multiplications per second in $\mathbb{F}_{3^{97}}$. For η_T pairing over $\mathbb{F}_{3^{97}}$, GPU's running time is around 29 microseconds. This is much faster than the best result on a dual-core 2.6 GHz Intel Core 2 processor [13]. However, for η_T pairing over $\mathbb{F}_{3^{509}}$, GPU's running time is around 3.01 milliseconds. This is slower than the best result on an eight-core 2.9 GHz Intel i7 processor [13] . We conclude that for larger fields, GPU implementation might be slower than CPU implementations because of the limited fast on-die memory on GPU. However, for smaller fields, GPU can easily outperform CPU because the computation uses a relatively smaller number of registers, and the large-scale parallel computing is more efficient on GPU.

5 Conclusion

In this paper, we report our experience implementing η_T pairing using three different strategies on GPU. Implementation I uses a single thread to compute

one pairing, Implementation II uses multiple threads to compute one pairing in parallel, and Implementation III uses multiple threads to compute multiple pairings in a bitsliced fashion. The best result among the three implementations on a single GTX 285 is obtained with Implementation III, which takes 1.47, 8.15, and 140.7 milliseconds to compute one pairing over $\mathbb{F}_{3^{97}}$, $\mathbb{F}_{3^{193}}$, and $\mathbb{F}_{3^{509}}$, respectively. On a single GTX 480, the throughput performance of Implementation III is 33710, 4970, and 332 η_T pairings per second over $\mathbb{F}_{3^{97}}$, $\mathbb{F}_{3^{193}}$, and $\mathbb{F}_{3^{509}}$, respectively. This is the first implementation result of the η_T pairing on GPU. To the best of our knowledge, this is also the fastest single-chip software implementation over smaller finite fields. Though the result over larger finite fields such as $\mathbb{F}_{3^{509}}$ is not as ideal as it is over smaller finite fields, the implementation should be sufficient for some real-world applications.

We have implemented the η_T pairing over finite fields $\mathbb{F}_{3^m}$ of characteristic 3. Another efficient class of cryptographic pairings can be constructed on the ordinary curves over finite fields $\mathbb{F}_p$ of large characteristic p. The implementation of pairings over large characteristics and its comparison with our implementation are future works.

Acknowledgements. This work was supported in part by National Science Council, National Taiwan University, and Intel Corporation under Grants NSC99-2911-I-002-001, 99R70600, and 10R80800 when the first author was visiting National Taiwan University.

References

1. D. F. Aranha, J. López, and D. Hankerson, "High-speed parallel software implementation of the η_T pairing", CT-RSA 2010, LNCS 5985 pp.89–105, 2010.
2. D. V. Bailey, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, H.-C. Chen, C.-M. Cheng, G. van Damme, G. de Meulenaer, L. J. Dominguez Perez, J. Fan, T. Guńeysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, R. Niederhagen, C. Paar and F. R, "Breaking ECC2K-130", Cryptology ePrint, http://eprint.iacr.org/2009/541, 2009.
3. M. Shirase, Y. Kawahara, T. Takagi, and E. Okamoto, "Universal η_T Pairing Algorithm over Arbitrary Extension Degree", WISA 2007, LNCS 4867, pp.1–15, 2007.
4. D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. Yang, "The Billion-Mulmod-Per-Second PC", SHARCS 2009, pp. 131–144, 2009.
5. D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang, "ECM on Graphics Cards", Eurocrypt 2009, LNCS 5479, pp. 483–501, 2009.
6. C. Bouillaguet, H.-C. Chen, C.-M. Cheng, T. Chou, R. Niederhagen, A. Shamir, and B.-Y. Yang, "Fast Exhaustive Search for Polynomial Systems in $\mathbb{F}_2$", CHES 2010, LNCS 6225, pp. 230–218, 2010.
7. N. Estibals, "Compact hardware for computing the Tate pairing over 128-bitsecurity supersingular curves", Pairing-Based Cryptography 2010, LNCS 6487, pp. 397–416, 2010.
8. P. Grabher, J. Großschädl, and D. Page, "On software parallel implementation of cryptographic pairings", SAC 2008, LNCS 5381, pp.34–49, 2008.

9. O. Harrison and J. Waldron, "Practical Symmetric Key Cryptography on Modern Graphics Hardware", USENIX 2008, pp.195–210, 2008.
10. O. Harrison and J. Waldron, "Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware", AFRICACRYPT 2009, LNCS 5580, pp. 350–367, 2009.
11. Y. Kawahara, K. Aoki, and T. Takagi, "Faster Implementation of η_T Pairing over GF(3^m) Using Minimum Number of Logical Instructions for GF(3) Addition", Pairing 2008, LNCS 5209, pp.289–296, 2008.
12. D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer, 2004.
13. J.-L. Beuchat, E. López-Trejo, L. Martínez-Ramos, S. Mitsunari, and F. Rodríguez-Henríquez, "Multi-core Implementation of the Tate Pairing over Supersingular Elliptic Curves", CANS 2009, LNCS 5888, pp. 413–432, 2009.
14. T. Nakajima, T. Izu, and T. Takagi, "Reduction Optimal Trinomials for Efficient Software Implementation of the η_T Pairing", IWSEC 2007, LNCS 4752, pp.44–57, 2007.
15. NVIDIA, CUDA Zone, http://www.nvidia.com/object/cuda_home_new.html.
16. M. Shirase, T. Takagi , and E. Okamoto, "Some Efficient Algorithms for the Final Exponentiation of η_T Pairing", IPSEC 2007, LNCS 4464, pp. 254–268, 2007.
17. R. Szerwinski and T. Guneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography", CHES 2008, LNCS 5154, pp. 79–99, 2008.