GPU-Based Implementation of 128-Bit Secure Eta Pairing over a Binary Field

Utsab Bose, Anup Kumar Bhattacharya, and Abhijit Das

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur, West Bengal, India
utsab.bose@yahoo.co.in, bhattacharya.anup@gmail.com,
abhij@cse.iitkgp.ernet.in

Abstract. Eta pairing on a supersingular elliptic curve over the binary field $F_{2^{1223}}$ used to offer 128-bit security, and has been studied extensively for efficient implementations. In this paper, we report our GPU-based implementations of this algorithm on an NVIDIA Tesla C2050 platform. We propose efficient parallel implementation strategies for multiplication, square, square root and inverse in the underlying field. Our implementations achieve the best performance when López-Dahab multiplication with four-bit precomputations is used in conjunction with one-level Karatsuba multiplication. We have been able to compute up to 566 eta pairings per second. To the best of our knowledge, ours is the fastest GPU-based implementation of eta pairing. It is about twice as fast as the only reported GPU implementation, and about five times as fast as the fastest reported single-core SIMD implementation. We estimate that the NVIDIA GTX 480 platform is capable of producing the fastest known software implementation of eta pairing.

Keywords: Supersingular elliptic curve, eta pairing, binary field, parallel implementation, GPU.

1 Introduction

Recently GPUs have emerged as a modern parallel computing platform for general-purpose programming. Many cryptographic algorithms have been implemented efficiently using GPU-based parallelization. Pairing (assumed symmetric) is a bilinear mapping of two elements in a group to an element in another group. Elliptic curves are widely used to realize various forms of pairing, like Weil pairing, Tate pairing, and eta pairing. Eta pairing, being one of the most efficient pairing algorithms, has extensive applications in identity-based and attribute-based encryption, multi-party communication, identity-based and short signatures, and autonomous authentication [6]. Investigating the extent of parallelizing eta pairing on GPU platforms is an important area of current research. Although several implementations of eta pairing have already been published in the literature [3,5,11], most of them are CPU-based, and

A. Youssef, A. Nitaj, A.E. Hassanien (Eds.): AFRICACRYPT 2013, LNCS 7918, pp. 26–42, 2013.

aim at improving the performance of single eta-pairing computations. However, many applications (like authentication in vehicular ad hoc networks) requir e computing large numbers of eta pairings in short intervals.

In this paper, we report efficient implementations of eta pairing on a supersingular elliptic curve defined over a field of characteristic two. This is a standard curve studied in the literature. At the time this work was done, this curve was believed to provide 128-bit security in cryptographic applications. Recent developments [13] tend to indicate that this security guarantee may be somewhat less.[1] To the best of our knowledge, there are no previous GPU-based implementations for this curve. The only GPU-based implementation of eta pairing reported earlier [15] is on a supersingular elliptic curve defined over a field of characteristic three, which also used to provide 128-bit security (but no longer now; see [19] and also [13]). Our implementation is about twice as efficient as this only known GPU implementation. We attempt to parallelize each pairing computation alongside multiple pairing computations, so as to exploit the GPU hardware as effectively as possible. In other words, we use both intra- and inter-pairing parallelization.

We start with parallel implementations of the binary-field arithmetic. We use López-Dahab multiplication [16] with four-bit windows in tandem with one-level Karatsuba multiplication [14] to obtain the best performance. We report the variation in performance of multiplication with the number of threads. We also report our GPU implementations of the square, square-root, inverse and reduction operations in the field. Finally, we use these parallel field-arithmetic routines for implementing eta pairing. Our best eta-pairing implementation is capable of producing up to 566 eta pairings per second on an NVIDIA TESLA C2050 platform. This indicates an average time of 1.76 ms per eta-pairing computation, which is comparable with the fastest known (1.51 ms) software implementation [3] (which is a multi-core SIMD-based implementation). The fastest reported single-core SIMD implementation [3] of eta pairing on this curve takes about 8 ms for each eta pairing. The only reported GPU-based implementation [15] of eta pairing is capable of producing only 254 eta pairings per second. We estimate, based upon previously reported results, that our implementation when ported to an NVIDIA GTX 480 platform is expected to give a 30% improvement in throughput, thereby producing 736 eta pairings per second, that is, 1.36 ms for computing one eta pairing.

It is worthwhile to note here that our work deals with only software implementations. Hardware implementations, significantly faster than ours, are available in the literature. For example, some recent FPGA implementations are described in [1,9]. The paper [1] also reports ASIC implementations. Both these papers use the same curve as we study here. Other types of pairing functions (like Weil, Tate, ate, and R-ate pairings) are also widely studied from implementation perspectives. The hardware and software implementations reported in [2,7,11] (to name a recent few) use other types of pairing.

[1] We are not changing the title of this paper for historical reasons, and also because of that the analysis presented in [13] is currently only heuristic.

The rest of the paper is organized as follows. In Section 2, a short introduction to GPUs and GPU programming is provided. Section 3 sketches the eta-pairing algorithm which we implement. Our implementation strategies for the binary-field arithmetic are detailed in Section 4. This is followed in Section 5 by remarks about our implementation of eta pairing. In Section 6, our experimental results are supplied and compared with other reported results. Section 7 concludes this paper after highlighting some future scopes of work.

2 NVIDIA Graphics Processing Units

In order to program in GPU, it is important to know the architectural details and how tasks are divided among threads at software level. Here, we briefly describe the Fermi architecture of CUDA and the programming model. Some comments on the GPU memory model are also in order.

2.1 GPU Architecture

The next-generation CUDA architecture, code-named *Fermi* [10], is an advanced and yet commonly available GPU computing architecture. With over three billion transistors and featuring up to 512 CUDA cores, it is one of the fastest GPU platforms provided by CUDA. Our implementations are made on one such Fermi-based GPU called TESLA C2050.

In TESLA C2050, there are 14 streaming multiprocessors (SMs) with a total of 448 CUDA cores. Each SM contains 32 CUDA cores along with 4 special function units and 16 load/store units. The detailed specification can be found in [17]. The 32 CUDA cores are arranged in two columns of 16 cores each. A program is actually executed in groups of 32 threads called *warps*, and a Fermi multiprocessor allocates a group of 16 cores (half warp) to execute one instruction from each of the two warps in two clock cycles. This allocation is done by two warp schedulers which can schedule instructions for each half warp in parallel. Each SM has a high-speed shared memory to be used by all threads in it.

2.2 GPU Programing Model

In order to get a good performance by parallel execution in GPUs, it is important to know how the threads can be organized at software level, and how these threads map to the hardware. A CUDA program is written for each thread. There is a programmer- or compiler-level organization of threads which directly project on to the hardware organization of threads. At user level, threads are grouped into blocks, and each block consists of several threads. A *thread block* (also called a *work group*) is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. All the threads of a block must reside in the same SM. As the number of threads in a SM is limited because of limited number of registers and shared memory, there is a bound on the maximum number of threads that a single work group can have.

But we can have as many work groups as we want. In Fermi, the maximum work-group size is 1024. However, the threads within a work group can be arranged in one, two or three dimensions, and the work groups themselves can again be organized into one of the three dimensions. Each thread within a work group has a local ID, and each work group has a block ID. The block ID and the local ID together define the unique global ID of a thread.

In Fermi, there can be 48 active warps (1536 threads) and a maximum of eight active work groups per SM, that can run concurrently. So it is preferred to have a work group size of 192 in order to perfectly utilize all the active threads, provided that there is scope for hiding memory latency. The number of resident work groups in a SM is also bounded by the amount of shared memory consumed by each work group and by the number of registers consumed by each thread within each group.

2.3 GPU Memory Architecture

Each GPU is supplied with 3GB of device memory (also known as the global memory), which can be accessed by all the threads from all the multiprocessors of the GPU. One of the major disadvantages of this memory is its low bandwidth. When a thread in a warp (a group of 32 threads) issues a device memory operation, that instruction may eat up even hundreds of clock cycles. This performance bottleneck can be overcome to some extent by memory coalescing, where multiple memory requests from several threads in a warp are coalesced into a single request, making all the threads request from the same memory segment. There is a small software-managed data cache (also known as the shared memory) associated with each multiprocessor. This memory is shared by all the threads executing on a multiprocessor. This low-latency high-bandwidth indexable memory running essentially at the register speed is configurable between 16KB and 48KB in Fermi architectures. In TESLA C2050, we have 48KB shared memory. We additionally have 16KB of hardware cache meant for high-latency global memory data. The hardware cache is managed by the hardware. Software programs do not have any control over the data residing in the hardware cache.

3 Eta Pairing in a Field of Characteristic Two

Here, we present the algorithm for eta (η_T) pairing [4] over the supersingular curve $y^2+y = x^3+x$ (embedding degree four) defined over the binary field $F_{2^{1223}}$ represented as an extension of F_2 by the irreducible polynomial $x^{1223} + x^{255} + 1$. As the embedding degree of the supersingular curve is four, we need to work in the field $F_{(2^{1223})^4}$. This field is represented as a tower of two quadratic extensions over $F_{2^{1223}}$, where the basis for the extension is given by $(1, u, v, uv)$ with $g(u) = u^2 + u + 1$ being the irreducible polynomial for the first extension, and with $h(v) = v^2 + v + u$ defining the second extension. The distortion map is given by $\phi(x, y) = (x + u^2, y + xu + v)$.

All the binary-field operations (addition, multiplication, square, square-root, reduction and inverse) are of basic importance in the eta-pairing algorithm, and

are discussed in detail in Section 4. Now, we show the eta-pairing algorithm which takes two points P and Q on the supersingular curve $y^2 + y = x^3 + x$ as input with P having a prime order r, and which computes an element of μ_r as output, where μ_r is the group (contained in $F^*_{(2^{1223})^4}$) of the r-th roots of unity.

Algorithm 1. Eta-pairing algorithm for a field of characteristic two

Input: $P = (x_1, y_1), Q = (x_2, y_2) \in E(F_{2^{1223}})[r]$
Output: $\eta_T(P, Q) \in \mu_r$

```
1  begin
2      T ← x1 + 1
3      f ← T · (x1 + x2 + 1) + y1 + y2 + (T + x2)u + v
4      for i =1 to 612 do
5          T ← x1
6          x1 ← √x1, y1 ← √y1
7          g ← T · (x1 + x2) + y1 + y2 + x1 + 1 + (T + x2)u + v
8          f ← f · g
9          x2 ← x2^2, y2 ← y2^2
10     end
11     return f^((q^2−1)(q−2√q+1)), where q = 2^1223
12 end
```

In Algorithm 1, $f \leftarrow f \cdot g$ is a multiplication in $F_{(2^{1223})^4}$ requiring eight multiplications in $F_{2^{1223}}$. This number can be reduced to six with some added cost of linear operations, as explained in [11]. Thus, the entire for loop (called the *Miller loop*) executes 1224 square-roots, 1224 squares, and 4284 multiplications. Multiplication being the most frequently used operation, its efficiency has a direct consequence on the efficiency of Algorithm 1.

4 Arithmetic of the Binary Field

An element of $F_{2^{1223}}$ is represented by 1223 bits packed in an array of twenty 64-bit words. All binary-field operations discussed below operate on these arrays.

4.1 Addition

Addition in $F_{2^{1223}}$ is word-level bit-wise XOR of the operands, and can be handled by 20 threads in parallel. In binary fields, subtraction is same as addition.

4.2 Multiplication

The multiplication operation is associated with some precomputation, where the results of multiplying the multiplicand with all four-bit patterns are stored in a two-dimensional array, called the precomputation matrix P. The quadratic

multiplication loop involves processing four bits together from the multiplier. See [16] for the details. After the multiplication, the 40-word intermediate product is reduced back to an element of $F_{2^{1223}}$ using the irreducible polynomial $x^{1223} + x^{255} + 1$. To sum up, the multiplication consists of three stages: precomputation, computation of the 40-word intermediate product, and polynomial reduction. These three stages are individually carried out in parallel, as explained below.

Precomputation. The precomputation matrix P has 320 entries (words). We can use 320 threads, where each thread computes one $P(i, j)$. It is also possible to involve only 160 or 80 threads with each thread computing two or four matrix entries. The exact number of threads to be used in this precomputation stage is adjusted to tally with the number of threads used in the second stage (generation of the intermediate product).

Let us use 320 threads in this stage. Each thread uses its thread ID to determine which entry in P it should compute. Let $\Theta_{i,j}$ denote the thread responsible for computing $P(i, j)$. Let us also denote the i-th word of the multiplicand A by $A_i = (a_{64i+63}a_{64i+62} \ldots a_{64i+1}a_{64i})$, where each a_k is a bit, and the most significant bit in the word A_i is written first. Likewise, w_j is represented by the bit pattern $(b_3b_2b_1b_0)$. The thread $\Theta_{i,j}$ performs the following computations:

Initialize $P(i, j)$ to $(000 \ldots 0)$.
If $b_0 = 1$, XOR $P(i, j)$ with $(a_{64i+63}a_{64i+62} \ldots a_{64i+1}a_{64i})$.
If $b_1 = 1$, XOR $P(i, j)$ with $(a_{64i+62}a_{64i+61} \ldots a_{64i}a_{64i-1})$.
If $b_2 = 1$, XOR $P(i, j)$ with $(a_{64i+61}a_{64i+60} \ldots a_{64i-1}a_{64i-2})$.
If $b_3 = 1$, XOR $P(i, j)$ with $(a_{64i+60}a_{64i+59} \ldots a_{64i-2}a_{64i-3})$.

In addition to the word A_i, the thread $\Theta_{i,j}$ needs to read the three most significant bits $a_{64i-1}a_{64i-2}a_{64i-3}$ from A_{i-1}. This is likely to incur conflicts during memory access from L1 cache, since the thread $\Theta_{i-1,j}$ also accesses the word A_{i-1}. In order to avoid this, three most significant bits of the words of A are precomputed in an array M of size 20. As a result, $\Theta_{i,j}$ reads only from A_i and M_i, whereas $\Theta_{i-1,j}$ reads from the different locations A_{i-1} and M_{i-1}. Since M depends only on A (not on w_j), only 20 threads can prepare the array M, and the resulting overhead is negligible compared to the performance degradation that was associated with cache conflicts.

Intermediate Product Computation. This stage proceeds like school-book multiplication. Instead of doing the multiplication bit by bit, we do it by chunks of four bits. Each word of the multiplier B contains sixteen such four-bit chunks. Figure 1 shows the distribution of the work among several threads. The threads use a temporary matrix R for storing their individual contributions. The use of R is necessitated by that the different threads can write in mutually exclusive cells of R. In practice, R is implemented as a one-dimensional array. However, Figure 1 shows it as a two-dimensional array for conceptual clarity. After all the entries in R are computed by all the threads, the 40-word intermediate product is obtained by adding the elements of R column-wise.

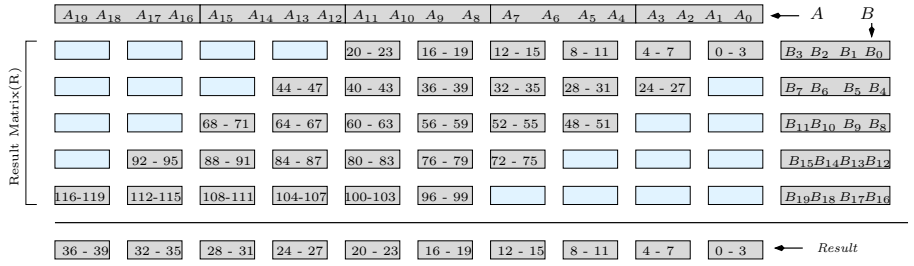

Fig. 1. Result Matrix Computation with 120 Threads

In Figure 1, we show how the result matrix R is computed by 120 threads. Here, R is of size 200 words, and is maintained as a 40×5 matrix. The figure, however, does not show these individual words (for lack of space). Instead, each box of R represents a group of four consecutive words. The range of four consecutive integers written within a box of R represents the IDs of the four threads that compute the four words in that box. In order to prevent synchronization overheads arising out of the race condition, different threads compute pair-wise different cells in R.

The first row of R corresponds to the multiplication of A by the least significant four words of B (that is, B_0, B_1, B_2, B_3). This partial product occupies 24 words which are computed by the threads with IDs 0–23, and stored in the columns 0–23 (column numbering begins from right). The second row of R is meant for storing the product of A with the next more significant four words of B (that is, B_4, B_5, B_6, B_7). This 24-word partial product is computed by a new set of 24 threads (with IDs 24–47), and stored in the second row of R with a shift of four words (one box). Likewise, the third, fourth and fifth rows of R are computed by 72 other threads. Notice that each row of R contains sixteen unused words, and need to be initialized to zero. After the computation of R, 40 threads add elements of R column-wise to obtain the intermediate product of A and B. Algorithm 2 elaborates this intermediate product generation stage. The incorporation of the precomputation table P is explicitly mentioned there. We use the symbols $\oplus$ (XOR), AND, OR, LEFTSHIFT and RIGHTSHIFT to stand for standard bit-wise operations.

In Algorithm 2, the values of $c - 4r - ID$ and $c - 4r - ID - 1$ at Lines 12 and 13 may become negative. In order to avoid the conditional check for negative values (which degrades performance due to warp divergence [8]), we maintain the precomputation matrix P as a 28×16 matrix instead of a 20×16 matrix. The first four and the last four columns of each row are initialized with zero entries. The actual values are stored in the middle 20 columns in each row.

[1] By a barrier, we mean that any thread must start executing instructions following MEM FENCE, only after all the currently running threads in a work group have completed execution up to the barrier. This ensures synchronization among the running threads.

Algorithm 2. Code for the i-th thread during intermediate product computation

Input: $A, B \in F_{2^{1223}}$ with precomputations on A stored in P
Output: The intermediate product $C = A \times B$ (a 40-word polynomial)

begin
  $r \leftarrow i/24$
  $c \leftarrow i \bmod 24$
  $r \leftarrow r + 4c$
  $t_1 \leftarrow 0$, $t_2 \leftarrow 0$
  $r \leftarrow r + 4$
  for $ID = 0$ ***to*** 4 **do**
    $w_2 \leftarrow B_{4r+ID}$
    for $j = 0$ ***to*** 16 **do**
      $bit \leftarrow \text{RIGHTSHIFT}(w_2, 4j)$ AND `0x0F`
      $w_1 \leftarrow P(bit, c - 4r - ID)$
      $w_0 \leftarrow P(bit, c - 4r - ID - 1)$
      $t_1 \leftarrow \text{LEFTSHIFT}(w_1, 4j) \oplus \text{RIGHTSHIFT}(w_0, 64 - 4j)$
      $t_2 \leftarrow t_2 \oplus t_1$
    end
  end
  $r \leftarrow r - 4$
  $R_{r,c} \leftarrow t_2$
  barrier(MEM FENCE)^2
  if $i < 40$ **then**
    $Result_i \leftarrow R_{0,i} \oplus R_{1,i} \oplus R_{2,i} \oplus R_{3,i} \oplus R_{4,i}$
  end
  barrier(MEM FENCE)
end

In the above implementation, each thread handles four words of the multiplier B. This can, however, be improved. If each thread handles only three consecutive words of B, we need $\lceil 20/3 \rceil = 7$ rows in R and $20 + 3 = 23$ used columns in each row. This calls for $7 \times 23 = 161$ threads. Since the 40-th word of the result is necessarily zero (the product of two polynomials of degrees < 1223 is at most $2444 \leq 2496 = 39 \times 64$), we can ignore the 161-st thread, that is, we can manage with 160 threads only.

Each thread can similarly be allowed to handle 1, 2, 5, 7, 10, or 20 words of B. The number of threads required in these cases are respectively 420, 220, 100, 60, 80 and 40. Increasing the number of threads to a high value may increase the extent of parallelism, but at the same time it will restrict the number of work groups that can be simultaneously active for concurrent execution in each multiprocessor. On the other hand, decreasing the number of threads to a small value increases the extent of serial execution within each thread. Thus, we need to reach a tradeoff between the number of threads and the amount of computation by each thread. Among all the above choices, we obtained the best performance with 160 threads, each handling three words of B.

One-level Karatsuba Multiplication. Using Karatsuba multiplication in conjunction with López-Dahab multiplication can speed up eta-pairing computations further, because Karatsuba multiplication reduces the number of F_2 multiplications at the cost of some linear operations. If we split each element A of $F_{2^{1223}}$ in two parts A_{hi} and A_{lo} of ten words each, López-Dahab multiplication of A and B requires four multiplications of ten-word operands ($A_{hi}B_{hi}$, $A_{lo}B_{lo}$, $A_{lo}B_{hi}$, and $A_{hi}B_{lo}$), as shown below (where $n = 640$):

$$(A_{hi}x^n + A_{lo})(B_{hi}x^n + B_{lo}) = A_{hi}B_{hi}x^{2n} + (A_{hi}B_{lo} + A_{lo}B_{hi})x^n + A_{lo}B_{lo}.$$

Karatsuba multiplication computes only the three products $A_{hi}B_{hi}$, $A_{lo}B_{lo}$, and $(A_{hi} + A_{lo})(B_{hi} + B_{lo})$, and obtains

$$A_{hi}B_{lo} + A_{lo}B_{hi} = (A_{hi} + A_{lo})(B_{hi} + B_{lo}) + A_{hi}B_{hi} + A_{lo}B_{lo}$$

using two addition operations. Each of the three ten-word multiplications is done by López-Dahab strategy. Each thread handles two words of the multiplier, and the temporary result matrix R contains five rows and twenty columns. Twelve threads write in each row, so the total number of threads needed for each ten-word multiplication is 60. All the three ten-word multiplications can run concurrently, thereby using a total of 180 threads.

Two levels of Karatsuba multiplication require only nine five-word multiplications instead of 16. Although this seems to yield more speedup, this is not the case in practice. First, the number of linear operations (additions and shifts that cannot be done concurrently) increases. Second, precomputation overheads associated with López-Dahab multiplication also increases. Using only one level of Karatsuba multiplication turns out to be the optimal strategy.

Reduction. The defining polynomial $x^{1223} + x^{255} + 1$ is used to reduce the terms of degrees ≥ 1223 in the intermediate product. More precisely, for $n \geq 0$, the non-zero term x^{1223+n} is replaced by $x^{255+n} + x^n$. This is carried out in parallel by 20 threads, where the i-th thread reduces the $(20 + i)$-th word. All the threads first handle the adjustments of x^{255+n} concurrently. Reduction of the entire $(20 + i)$-th word affects both the $(5 + i)$-th and the $(4 + i)$-th words. In order to avoid race condition, all the threads first handle the $(5 + i)$-th words concurrently. Subsequently, after a synchronization, the $(4 + i)$-th words are handled concurrently again. The adjustments of x^n in a word level are carried out similarly. Note that for large values of n, we may have $255 + n \geq 1223$. This calls for some more synchronization of the threads.

4.3 Square

We precompute the squares of all 8-bit patterns, and store them in an array Q of 256 words [12]. The parallel implementation of squaring in $F_{2^{1223}}$ is done by a total of 40 threads. Each thread handles a half word (that is, 32 bits) of the operand A, consults the precomputation table Q four times, and stores the

partial result in an array R. More precisely, the threads $2i$ and $2i+1$ read the word A_i. The $2i$-th thread reads the least significant 32 bits of A_i, and writes the corresponding square value in R_{2i}, whereas the $(2i+1)$-st thread writes the square of the most significant 32 bits of A_i in R_{2i+1}. The threads write in pairwise distinct words of R, so they can run concurrently without synchronization.

Algorithm 3. Code for the i-th thread during squaring

Input: An element of $A \in F_{2^{1223}}$ and the precomputed table Q
Output: The intermediate 40-word square $R = A^2$

```
1  begin
2      T ← A_{i/2}
3      if i is odd then
4          T ← RIGHTSHIFT(T, 32)
5      end
6      RT ← 0
7      for j = 0 to 3 do
8          byte ← T AND 0xFF
9          T ← RIGHTSHIFT(T, 8)
10         RT ← RT ⊕ RIGHTSHIFT(Q[byte], 16j)
11     end
12     Result_i ← RT
13     barrier(MEM FENCE)
14 end
```

4.4 Square-Root

Write an element A of $F_{2^{1223}}$ as $A = A_{even}(x) + xA_{odd}(x)$, where

$$A_{even}(x) = a_{1222}x^{1222} + a_{1220}x^{1220} + \cdots + a_2x^2 + a_0,$$
$$A_{odd}(x) = a_{1221}x^{1220} + a_{1219}x^{1218} + \cdots + a_3x^2 + a_1.$$

Then,

$$\begin{aligned}\sqrt{A} &= A_{even}(\sqrt{x}) + \sqrt{x}A_{odd}(\sqrt{x}) \\ &= (a_{1222}x^{611} + a_{1220}x^{610} + \cdots + a_2x + a_0) + \\ &\quad \sqrt{x}(a_{1221}x^{610} + a_{1219}x^{609} + \cdots + a_3x + a_1).\end{aligned}$$

Moreover, since $x^{1223} + x^{255} + 1 = 0$, we have $\sqrt{x} = x^{612} + x^{128}$.

We use 40 threads for this computation. Twenty threads $\Theta_{even,i}$ compute $A_{even}(\sqrt{x})$, and the remaining twenty threads $\Theta_{odd,i}$ compute $A_{odd}(\sqrt{x})$. For $j = 0, 1, 2, \ldots, 9$, the thread $\Theta_{even,2j}$ reads only the even bits of A_{2j}, that is, $(a_{128j+62} \ldots a_{128j+2}a_{128j})$, and stores them in the least significant 32 bits of an array $T_{even}[0][j]$. On the other hand, for $j = 0, 1, 2, \ldots, 9$, the thread $\Theta_{even,2j+1}$

reads only the even bits of A_{2j+1}, that is, $(a_{128j+126} \dots a_{128j+66} a_{128j+64})$ and stores them in the most significant 32 bits of $T_{even}[1][j]$. Likewise, for $j = 0, 1, 2, \dots, 9$, $\Theta_{odd,2j}$ writes $(a_{128j+63} \dots a_{128j+3} a_{128j+1})$ in the least significant 32 bits of $T_{odd}[0][j]$, and $\Theta_{odd,2j+1}$ writes $(a_{128j+127} \dots a_{128j+67} a_{128j+65})$ in the most significant 32 bits of $T_{odd}[1][j]$. After all these threads finish, ten threads add T_{even} column-wise, and ten other threads add T_{odd} column-wise. The column-wise sum T_{odd} is shifted by 612 and 128 bits, and the shifted arrays are added to the column-wise sum of T_{even}. The details are shown as Algorithm 4, where we have used flattened representations of various two-dimensional arrays.

Algorithm 4. Code for the i-th thread for square-root computation

Input: An element $A \in F_{2^{1223}}$
Output: $R = \sqrt{A}$

begin
  $d \leftarrow i/20$, $bit \leftarrow i/20$
  $ID \leftarrow i \bmod 20$
  $word \leftarrow A_i$
  for $j = 0$ **to** 31 **do**
    $W \leftarrow \text{RIGHTSHIFT}(word, bit)$ AND 1
    $T \leftarrow T \oplus \text{LEFTSHIFT}(W, bit/2)$
    $bit \leftarrow bit + 2$
  end
  if i *is odd* **then**
    $T \leftarrow \text{LEFTSHIFT}(T, 32)$
  end
  $R_{20d+10\times(ID \text{ AND } 1)+id/2} \leftarrow T$
  $EvenOdd_{20d+ID} \leftarrow R_{20d+i} \oplus R_{20d+10+i}$
  $odd1_i \leftarrow \text{ShiftBy612Bits}(EvenOdd_{20+i})$ `// multiply by` x^{612}
  $odd2_i \leftarrow \text{ShiftBy128Bits}(EvenOdd_{20+i})$ `// multiply by` x^{128}
  if $i < 20$ **then**
    $R_i \leftarrow odd1_i \oplus odd2_i \oplus EvenOdd_i$
  end
end

4.5 Inverse

Inverse is used only once during final exponentiation [18]. This is computed in parallel by the extended Euclidean gcd algorithm for polynomials [12]. In addition to finding the gcd γ of two polynomials A and B, it also finds polynomials g and h satisfying the Bézout relation $gA + hB = \gamma$. In the inverse computation, $B = f(x) = x^{1223} + x^{255} + 1$ is irreducible, and $A \neq 0$ is of degree < 1223, so $\gamma = 1$. Thus, the polynomial g computed by the algorithm is the inverse of A (modulo f). It is not necessary to compute the other polynomial h.

Algorithm 5. Code for the i-th thread for computing inverse

Input: A non-zero binary polynomial A, and the irreducible polynomial $f(x) = x^{1223} + x^{255} + 1$, each occupying 20 words
Output: $A^{-1} \bmod f$

```
1  begin
2      U_i ← f_i, V_i ← A_i, g_{1_i} ← 0, g_{2_i} ← 0
3      SV_i ← 0, SV_{i+20} ← 0, SG_i ← 0, SG_{i+20} ← 0
4      if i = 0 then
5          g_{2_i} ← 1
6      end
7      while TRUE do
8          DEG_i ← WordDegree(U_i)     // highest position of a 1-bit in U_i
9          DEG_{i+20} ← WordDegree(V_i) // highest position of a 1-bit in V_i
10         barrier(MEM FENCE)
11         if i = 0 then
12             shared terminate ← 0
13             degU ← GetDegree(DEG, 20) // Find deg(U) from DEG[0–19]
14             degV ← GetDegree(DEG, 40) // Find deg(U) from DEG[20–39]
15             shared diff ← degU − degV
16         end
17         barrier(MEM FENCE)
18         d ← diff
19         if d ≤ 1 then
20             U_i ↔ V_i, g_{1_i} ↔ g_{2_i}, d ← −d
21         end
22         barrier(MEM FENCE)
23         k ← d/64, w ← d mod 64
24         SV_{i+k} ← V_i, SG_{i+k} ← g_{2_i}
25         vw ← GetRightMostBits(V_{i−1}, w)
26         gw ← GetRightMostBits(g_{2_{i−1}}, w)
27         SV_{i+k} ← LEFTSHIFT(SV_{i+k}, w) ⊕ vw
28         SG_{i+k} ← LEFTSHIFT(SG_{i+k}, w) ⊕ gw
29         barrier(MEM FENCE)
30         U_i ← U_i ⊕ SV_i, g_{1_i} ← g_{1_i} ⊕ SG_i
31         if i ≠ 0 and U_i ≠ 0 then
32             terminate ← 1
33         end
34         barrier(MEM FENCE)
35         if (terminate = 0 and U_0 = 1) then
36             Terminate the loop
37         end
38     end
39 end
```

Two polynomials U and V are initialized to f and A. Moreover, g_1 is initialized to 0, and g_2 to 1. In each iteration, 20 threads compute the word degrees $\deg(U_i)$ and $\deg(V_i)$. Then, one thread computes $d = \deg(U) - \deg(V)$. If d is negative, the 20 threads swap U and V, and also g_1 and g_2. Finally, the threads subtract (add) $x^d V$ from U and also $x^d g_2$ from g_1. This is repeated until U reduces to 1. The detailed code is supplied as Algorithm 5.

In our implementation, this algorithm is further parallelized by using 40 threads (Algorithm 5 uses only 20 threads, for simplicity). The degree calculations of U and V can proceed in parallel. The swapping of (U, V) and (g_1, g_2) can also be parallelized. Finally, the two shifts $x^d V$ and $x^d g_2$ can proceed concurrently, and so also can the two additions $U + x^d V$ and $g_1 + x^d g_2$.

5 Parallel Implementations of Eta Pairing

Suppose that we want to compute n eta pairings in parallel. In our implementation, only two kernels are launched for this task. The first kernel runs the Miller loop for all these n eta-pairing computations. The output of the Miller loop is fed to the second kernel which computes the final exponentiation. Each kernel launches n work groups, each with 180 threads. Threads are launched as warps, so even if we use only 180 threads, the GPU actually launches six warps (that is, 192 threads) per work group.

At the end of each iteration of the Miller loop, the threads of each work group are synchronized. Out of the 180 threads in a work group, 80 threads are used to compute the two squares x_2^2 and y_2^2 in parallel, and also the two square-roots $\sqrt{x_1}$ and $\sqrt{y_1}$ in parallel. For these operations, only 44.45% of the threads are utilized. For most part of the six multiplications in an iteration, all the threads are utilized (we have used Karatsuba and López-Dahab multiplications together). The linear operations (assignments and additions) in each iteration are usually done in parallel using 20 threads. In some cases, multiple linear operations can proceed in parallel, thereby utilizing more threads. Clearly, the square, square-root, and linear operations are unable to exploit available hardware resources (threads) effectively. Nevertheless, since multiplication is the most critical field operation in the eta-pairing algorithm, our implementation seems to exploit parallelism to the best extent possible.

Each multiprocessor can have up to eight active work groups capable of running concurrently. Moreover, there are 14 multiprocessors in our GPU platform. Therefore, a total of 112 work groups can be simultaneously active. As a result, at most 112 eta-pairing computations can run truly in parallel, at least in theory. Our implementations corroborate this expected behavior.

6 Experimental Results

Our implementations are done both in CUDA and in OpenCL. Here, we report our OpenCL results only, since OpenCL gives us slightly better results, potentially because of the following reasons.

- Kernel initialization time is less in OpenCL than in CUDA. This may account for better performance of OpenCL over CUDA for small data sets. For large data sets, both OpenCL and CUDA are found to perform equally well.
- OpenCL can be ported to many other devices (like Intel and AMD graphics cards) with minimal effort.
- CUDA's synchronization features are not as flexible as those of OpenCL. In OpenCL, any queued operation (like memory transfer and kernel execution) can be forced to wait on any other set of queued operations. CUDA's instruction streams at the time of implementation are comparatively more restrictive. In other words, the in-line synchronization features of OpenCL have been useful in our implementation.

We have used the `CudaEvent()` API call for measuring time in CUDA, and the `clGetEventProfilingInfo()` API call in OpenCL. Table 1 shows the number of field multiplications (in millions/sec) performed for different numbers of threads. In the table, ω represents the number of words in the multiplier, that each thread handles. This determines the number of threads to be used, as explained in Section 4.2. Only the entry with 180 threads ($\omega = 3$) uses Karatsuba multiplication in tandem with López-Dahab multiplication.

Table 1. Number of $F_{2^{1223}}$ multiplications with different thread-block sizes

ω	Number of threads	Number of multiplications (millions/sec)
20	40	1.3
10	60	1.7
7	80	1.9
5	100	2.3
4	120	2.8
3	160	3.3
2	180 *	3.5
2	220	3.1
1	420	2.3

* With Karatsuba multiplication

Table 1 shows that the performance gradually improves with the increase in the number of threads, reaches the best for 180 threads, and then decreases beyond this point. This is because there can be at most 48 concurrent warps in a multiprocessor, and the number of work groups that can reside in each multiprocessor is 8. Since each warp has 32 threads, a work-group size of $(48/8) \times 32 = 192$ allows concurrent execution of all resident threads, thereby minimizing memory latency. With more than 192 threads, the extent of concurrency is restricted, and the performance degrades.

Table 2 shows the numbers of all field operations computed per second, along with the number of threads participating in each operation. The multiplication and square operations include reduction (by the polynomial $x^{1223} + x^{255} + 1$).

Table 2. Performance of binary-field arithmetic

Field operation	Number of threads	Number of operations (millions/sec)
Addition	20	100.09
Reduction	20	62.5
Square *	40	15.8
Square root	40	6.4
Multiplication *	180	3.5
Inverse	40	0.022

* Including reduction

Table 3 presents a comparative study of the performances of some eta-pairing implementations. Currently, the fastest software implementation of 128-bit secure eta pairing over fields of small characteristics is the eight-core CPU implementation of Aranha et al. [3]. Our implementation on Tesla C2050 is slightly slower than this. Our codes are readily portable to the GTX 480 platform, but unavailability of such a machine prevents us from carrying out the actual experiment. We, however, estimate that our implementation ported to a GTX 480 platform can be the fastest software implementation of 128-bit secure eta pairing.

Table 3. Comparison with other software implementations of η_T pairing

Implementation	Field	Platform	# cores	Clock freq (GHz)	Time per eta pairing (ms)
Hankerson et al. [11]	$F_{2^{1223}}$	CPU, Intel Core2	1	2.4	16.25
	$F_{3^{509}}$	CPU, Intel Core2	1	2.4	13.75
	$F_{p_{256}}$	CPU, Intel Core2	1	2.4	6.25
Beuchat et al. [5]	$F_{3^{509}}$	CPU, Intel Core2	1	2.0	11.51
	$F_{3^{509}}$	CPU, Intel Core2	2	2.0	6.57
	$F_{3^{509}}$	CPU, Intel Core2	4	2.0	4.54
	$F_{3^{509}}$	CPU, Intel Core2	8	2.0	4.46
Aranha et al. [3]	$F_{2^{1223}}$	CPU, Intel Core2	1	2.0	8.70
	$F_{2^{1223}}$	CPU, Intel Core2	2	2.0	4.67
	$F_{2^{1223}}$	CPU, Intel Core2	4	2.0	2.54
	$F_{2^{1223}}$	CPU, Intel Core2	8	2.0	1.51
Katoh et al. [15]	$F_{3^{509}}$	GPU, Tesla C2050	448	1.1	3.93
	$F_{3^{509}}$	GPU, GTX 480	480	1.4	3.01
This Work	$F_{2^{1223}}$	GPU, Tesla C2050	448	1.1	1.76
This Work	$F_{2^{1223}}$	GPU, GTX 480	480	1.4	1.36 *

* Estimated running time

7 Conclusion

In this paper, we report our GPU-based implementations of eta pairing on a supersingular curve over the binary field $F_{2^{1223}}$. Cryptographic protocols based

on this curve offer 128-bit security, so efficient implementation of this pairing is an important issue to cryptographers. Throughout the paper, we report our optimization strategies for maximizing efficiency on a popular GPU architecture. Our implementations can be directly ported to similar GPU architectures, and have the potential of producing the fastest possible implementation of 128-bit secure eta pairing. Our implementations are most suited to applications where a large number of eta pairings need to be computed.

We end this paper after highlighting some possible extensions of our work.

- Our implementations can be applied *mutatis mutandis* to compute eta pairing on another popular supersingular curve defined over the field $F_{3^{509}}$.
- Other types of pairing (like Weil and Tate) and other types of curves (like ordinary) may also be studied for GPU-based implementations.
- Large prime fields involve working with large integers. Since integer arithmetic demands frequent carry manipulation, an efficient GPU-based implementation of prime-field arithmetic is a very challenging exercise, differing substantially in complexity from implementations of polynomial-based arithmetic of extension fields like $F_{2^{1223}}$ and $F_{3^{509}}$.

Acknowledgement. The authors wish to gratefully acknowledge many useful improvements suggested by the anonymous referees.

References

1. Adikari, J., Hasan, M.A., Negre, C.: Towards faster and greener cryptoprocessor for eta pairing on supersingular elliptic curve over $\mathbb{F}_{2^{1223}}$. In: Knudsen, L.R., Wu, H. (eds.) SAC 2012. LNCS, vol. 7707, pp. 166–183. Springer, Heidelberg (2013)
2. Aranha, D.F., Beuchat, J.-L., Detrey, J., Estibals, N.: Optimal eta pairing on supersingular genus-2 binary hyperelliptic curves. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 98–115. Springer, Heidelberg (2012)
3. Aranha, D.F., López, J., Hankerson, D.: High-speed parallel software implementation of the η_T pairing. In: Pieprzyk, J. (ed.) CT-RSA 2010. LNCS, vol. 5985, pp. 89–105. Springer, Heidelberg (2010)
4. Barreto, P.S.L.M., Galbraith, S., hÉigeartaigh, C.O., Scott, M.: Efficient pairing computation on supersingular Abelian varieties. Designs, Codes and Cryptography, 239–271 (2004)
5. Beuchat, J.-L., López-Trejo, E., Martínez-Ramos, L., Mitsunari, S., Rodríguez-Henríquez, F.: Multi-core implementation of the tate pairing over supersingular elliptic curves. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 2009. LNCS, vol. 5888, pp. 413–432. Springer, Heidelberg (2009)
6. Blake, I., Seroussi, G., Smart, N.: Elliptic curves in cryptography. London Mathematical Society, vol. 265. Cambridge University Press (1999)
7. Fan, J., Vercauteren, F., Verbauwhede, I.: Efficient hardware implementation of $\mathbb{F}_p$-arithmetic for pairing-friendly curves. IEEE Transactions on Computers 61(5), 676–685 (2012)
8. Fung, W.W.L., Sham, I., Yuan, G., Aamodt, T.M.: Dynamic warp formation and scheduling for efficient GPU control flow. Micro 40, 407–420 (2007)

9. Ghosh, S., Roychowdhury, D., Das, A.: High speed cryptoprocessor for η_T pairing on 128-bit secure supersingular elliptic curves over characteristic two fields. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 442–458. Springer, Heidelberg (2011)
10. Glaskowsky, P.: NVIDIA's Fermi: The first complete GPU computing architecture. White paper, NVIDIA Corporation (2009)
11. Hankerson, D., Menezes, A., Scott, M.: Software implementation of pairings. In: Cryptology and Information Security Series, vol. 2, pp. 188–206. IOS Press (2009)
12. Hankerson, D., Menezes, A., Vanstone, S.: Guide to elliptic curve cryptography. Springer (2004)
13. Joux, A.: A new index calculus algorithm with complexity $L(1/4 + o(1))$ in very small characteristic. Cryptology ePrint Archive, Report 2013/095 (2013), http://eprint.iacr.org/2013/095
14. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. Doklady Akad. Nauk. SSSR 145, 293–294 (1962)
15. Katoh, Y., Huang, Y.-J., Cheng, C.-M., Takagi, T.: Efficient implementation of the η_T pairing on GPU. Cryptology ePrint Archive, Report 2011/540 (2011), http://eprint.iacr.org/2011/540
16. López, J., Dahab, R.: High-speed software multiplication in f2m. In: Roy, B., Okamoto, E. (eds.) INDOCRYPT 2000. LNCS, vol. 1977, pp. 203–212. Springer, Heidelberg (2000)
17. NVIDIA Corporation, CUDA: Compute unified device architecture programming guide. Technical Report, NVIDIA (2007)
18. Scott, M., Benger, N., Charlemagne, M., Dominguez Perez, L.J., Kachisa, E.J.: On the final exponentiation for calculating pairings on ordinary elliptic curves. In: Shacham, H., Waters, B. (eds.) Pairing 2009. LNCS, vol. 5671, pp. 78–88. Springer, Heidelberg (2009)
19. Shinohara, N., Shimoyama, T., Hayashi, T., Takagi, T.: Key length estimation of pairing-based cryptosystems using η_T pairing. Cryptology ePrint Archive, Report 2012/042 (2012), http://eprint.iacr.org/2012/042