International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 6, Number 1, pages 42–63, January 2016

Bulk execution of Euclidean algorithms on the CUDA-enabled GPU

Toru Fujita, Koji Nakano, and Yasuaki Ito
Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashihiroshima 739-8527, Japan

Received: August 1, 2015
Revised: October 26, 2015
Accepted: December 1, 2015
Communicated by Akihiro Fujiwara

Abstract

The bulk execution of a sequential algorithm is to execute it for many different inputs in turn or at the same time. A sequential algorithm is oblivious if the address accessed at each time unit is independent of the input. It is known that the bulk execution of an oblivious sequential algorithm can be implemented to run on a GPU very efficiently. The main purpose of our work is to implement the bulk execution of a Euclidean algorithm computing the GCD (Greatest Common Divisor) of two large numbers in a GPU. We first present a new efficient Euclidean algorithm that we call the Approximate Euclidean algorithm. The idea of the Approximate Euclidean algorithm is to compute an approximation of quotient by just one 64-bit division and to use it for reducing the number of iterations of the Euclidean algorithm. Unfortunately, the Approximate Euclidean algorithm is not oblivious. To show that the bulk execution of the Approximate Euclidean algorithm can be implemented efficiently in the GPU, we introduce a semi-oblivious sequential algorithms, which is almost oblivious. We show that the Approximate Euclidean algorithm can be implemented as a semi-oblivious algorithm. The experimental results show that our parallel implementation of the Approximate Euclidean algorithm for 1024-bit integers running on GeForce GTX Titan X GPU is 90 times faster than the Intel Xeon CPU implementation.

Keywords: Euclidean algorithm, GPGPU, CUDA, bulk execution

1 Introduction

The GPU (Graphics Processing Unit) is a specialized hardware designed to accelerate computation for building and manipulating images [5, 22, 24]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [5, 6, 7, 17, 25]. NVIDIA provides a parallel computing platform and application programming interface model called *CUDA* (Compute Unified Device Architecture) [14, 15], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [10], since they have thousands of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [15]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-96K bytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-12 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *bank conflicts* of the shared memory access and *coalescing* of the global memory access [6, 7, 10, 11, 14]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the shared memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the throughput between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory. Also, the latency of the global memory access is several hundred clock cycles, while that of the shared memory access is quite small [15]. Hence, we should minimize the memory access to the global memory to maximize the performance. In fact, there are several types of memory except the shared memory and the global memory in CUDA such as the texture memory, the surface memory, and the constant memory [15]. However, these memories are placed to off-chip memory where the global memory is stored. They are used for different purposes in accordance to the use. For example, the texture memory is a read-only memory for GPU programs and its special application program interfaces are supported. However, they are essentially the same memory.

It is well known that the Euclidean algorithm [8] can compute the GCD of two numbers very efficiently. The original Euclidean algorithm repeats modulo computation of two numbers until one of them reaches zero and the other one stores the GCD. However, modulo computation of large numbers takes a lot of time. Hence, the Binary Euclidean algorithm [20], which does not use modulo computation, is often used to compute the GCD. Basically, the Binary Euclidean algorithm repeats subtraction of two numbers and arithmetic shifts until one of them reaches zero. The Binary Euclidean algorithm needs more iterations than the original Euclidean algorithm, but the computation of each iteration of the Binary Euclidean algorithm takes less time than that of the original Euclidean algorithm. Totally, the Binary Euclidean algorithm runs faster than the original Euclidean algorithm, and it is commonly used to compute the GCD.

The main contribution of this paper is to present a new Euclidean algorithm for computing the GCD that can be implemented in CUDA-enabled GPUs. The idea of our new Euclidean algorithm that we call *the Approximate Euclidean algorithm* is to compute a good approximation of quotient by simple 64-bit division and to use it for reducing the number of iterations of the Euclidean algorithm. It runs much faster than the original Euclidean algorithm and the Binary Euclidean algorithm. We also present an implementation of the Approximate Euclidean algorithm optimized for CUDA-enabled GPUs.

In our previous papers [21, 23], we introduced *obliviousness* of a sequential algorithm and showed that the bulk execution of an oblivious sequential algorithm can be implemented very efficiently in CUDA-enabled GPUs. A sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input. More formally, there exists a function a such that the algorithm accesses address $a(t)$ or does not access any address at every time t (≥ 0). For example, let b be an array of size n and we want to determine if there exists i such that $b[i] \neq 0$. In other words, we compute the logical OR of all $b[i]$'s. This can be done by reading the value of $b[i]$ from $i = 0$ to $n - 1$ one by one. Once it finds i such that $b[i] \neq 0$, it terminates and $b[i + 1]$, $b[i + 2]$, $\ldots$ are not accessed. If it accesses $b[n - 1]$ and finds $b[n - 1] = 0$, then all $b[i]$'s are zero. Clearly, this algorithm is oblivious because at each i-th iteration, it accesses $b[i]$ or does not access the memory. *The bulk execution* of a sequential algorithm is to execute it for many different inputs in turn or at the same time. Suppose that each input of the bulk execution is assigned to a CUDA thread and each CUDA thread executes the sequential algorithm for an assigned input. Since each thread accesses the same address at each time unit, memory access to the global memory is coalesced if the input and the work space is arranged in the CUDA global memory in column-wise. Hence, this implementation of the bulk execution of a sequential algorithm runs very fast.

We also introduce a sequential algorithm with synchronization, which can execute sync instruction. The execution of it can be partitioned into sub-executions by sync instructions such that a sub-execution is an execution between two consecutive sync instructions. We say that a sequential algorithm with synchronization is oblivious if all sub-executions are oblivious. We will show that when a sequential algorithm with synchronization is oblivious, the bulk execution of it can be implemented in the GPU very efficiently. The concept of the obliviousness of a sequential algorithm with synchronization is necessary to show that our Approximate Euclidean algorithm can be implemented to run in the GPU efficiently.

However, unfortunately, our Approximate Euclidean algorithm is not oblivious. Hence, we further introduce *semi-obliviousness*, and show that our Approximate Euclidean algorithm is *semi-oblivious* in the sense that an address accessed at each of almost all time units. In other words, semi-oblivious algorithm may access different addresses in few time units. If each of the CUDA threads executes a semi-oblivious sequential algorithm for the bulk execution, then they may perform non-coalesced access to the global memory. However, if the ratio of non-coalesced access is small enough, the bulk execution of a semi-oblivious sequential algorithm still runs efficiently on the GPU. We will show that the Approximate Euclidean algorithm can be implemented as a semi-oblivious sequential algorithm with sync, and it runs on CUDA-enabled GPUs efficiently. The implementation results on GeForce GTX Titan X show that the GCD of two randomly generated 1024-bit numbers can be computed in 0.482 microseconds per GCD computation. Also, it is 90 times faster than the Approximate Euclidean algorithm on a single processor.

One of the applications of GCD computation is to break RSA keys [18]. Suppose that we have a lot of RSA moduli collected from the Web. If a pair of two RSA moduli in them shares a prime number, each of them can be decomposed into two prime numbers easily by computing the GCD. If an RSA moduli can be decomposed into two prime numbers, the corresponding RSA decryption key can be obtained. Such pairs of RSA keys are called weak RSA keys. By computing the GCD of all pairs of RSA moduli corrected from the Web, we can find weak RSA keys if exist. Actually, several public keys collected from the Web includes weak RSA keys [9]. Several previously published papers have presented GPU implementations of the Binary Euclidean algorithm for breaking weak RSA keys in CUDA-enabled GPU. Fujimoto [2] has implemented the Binary Euclidean algorithm using CUDA and evaluated the performance on GeForce GTX285 GPU. The experimental results show that the GCDs for 131072 pairs of 1024-bit numbers can be computed in 1.431932 seconds. Hence, his implementation runs 10.9 microseconds per one 1024-bit GCD computation. Scharfglass *et al.* [19] have presented a GPU implementation of the Binary Euclidean algorithm. It performs the GCD computation of all 199990000 pairs of 20000 RSA moduli with 1024 bits in 2005.09 seconds using GeForce GTX 480 GPU. Thus, their implementation performs each 1024-bit GCD computation in 10.02 microseconds. Quite recently, White [26] has showed that the same computation can be performed in 63.0 seconds on Tesla K20Xm. It follows that it computes each 1024-bit GCD in 3.15 microseconds. In our conference version of this paper [3], we have shown that the GCD computation for breaking weak RSA keys runs 0.346 microseconds per GCD computation. Note that, this implementation of the conference version is faster than this journal version paper, because the GCD computation for breaking RSA keys can terminate earlier. For example, a 1024-bit RSA modulo is the product of two 512-bit prime numbers. Hence, we can terminate a Euclidean algorithm as soon as we know that the GCD has less than 512 bits.

On the other hand, it has been presented [4] that a sequential algorithm can find a weak RSA keys much faster than the pairwise GCD computation for all pairs of two RSA moduli. The idea is to compute, for each RSA modulo, the GCD with the product of all other RSA moduli. If an RSA modulo shares a prime number with one of all other RSA moduli, then the GCD is the prime number. The computing time can be reduced by creating a remainder tree of all RSA moduli by repeating complicated but efficient modulo computation [1]. This sequential algorithm runs faster than a parallel implementation of pairwise GCD computation using the GPU. Hence it makes no sense to use the pairwise GCD computation for breaking weak RSA keys. However, our GPU implementation of pairwise GCD computation is still significant in the area of GPU computation. The efficient sequential algorithm uses very complicated remainder tree technique and fast modulo computation, and it just finds a pair of RSA moduli sharing a prime number in a large set of RSA moduli. It works only for the case that most of pairs of RSA moduli are coprime, and very few pairs share a prime number. Hence, the sequential algorithm using the remainder tree technique cannot be used to compute the GCD for all pairs. Since we want to compute the GCD of all pairs, the efficient sequential algorithm for breaking RSA moduli using a remainder tree cannot be used for this purpose. Our GPU implementation is the best for this task.

This paper is organized as follows. We first review several Euclidean algorithms in Section 2. We then go on to present our Approximate Euclidean algorithm that computes an approximation of quotient of two numbers in Section 3. In Section 4, we show that our GPU implementation for the bulk execution of the Approximate Euclidean algorithm can be accelerated further using PTX instructions, which are assembly language instruction for CUDA-enabled GPUs. Section 5 reviews the definition of obliviousness of a sequential algorithm and the bulk execution. It also shows that the bulk execution of a oblivious sequential algorithm can be implemented efficiently on the Unified Memory Machine (UMM), which is a theoretical model for GPU computing. In Section 6, we introduce a sequential algorithm with synchronization and show that its bulk execution can be implemented efficiently on the UMM if it is oblivious. Section 7 introduces a semi-oblivious sequential algorithm and shows that the bulk execution can be implemented efficiently. Finally, we show experimental results of the performance of the Euclidean algorithms both on the CPU and on the GPU in Section 8. It shows that the Approximate Euclidean algorithm run faster than the others. Section 9 concludes our work.

2 Euclidean algorithms for computing the GCD

The main purpose of this section is to review a classical Euclidean algorithm for computing the GCD of two numbers X and Y.

Let $\texttt{GCD}(X, Y)$ denote the GCD of X and Y. Euclidean algorithms for computing the GCD are based on the following fact:

Lemma 1. *For any integer $\alpha \geq 0$ such that $X > Y \cdot \alpha$, we have $\texttt{GCD}(X, Y) = \texttt{GCD}(X - Y \cdot \alpha, Y)$.*

Proof. Suppose that $\texttt{GCD}(X, Y) = 1$, that is, X and Y are coprime. We will prove that $\texttt{GCD}(X - Y \cdot \alpha, Y) = 1$ always holds by contradiction. If Y and $X - Y \cdot \alpha$ are not coprime, there exists a common divisor $h \geq 2$ such that $Y = Y' \cdot h$ and $X - Y \cdot \alpha = X' \cdot h$. We have $X = (Y' \cdot \alpha + X') \cdot h$, and thus X and Y have common divisor h, a contradiction. Suppose that $\texttt{GCD}(X, Y) = g$ (≥ 2). Clearly, there exists two coprime numbers x and y such that $X = x \cdot g$ and $Y = y \cdot g$. Since $X - Y \cdot \alpha = (x - y \cdot \alpha)g$, it is sufficient to show that y and $x - y \cdot \alpha$ are coprime. This can be proved in the same way by contradiction. □

Let $\texttt{rshift}(X)$ be the function such that it returns an odd number X' such that $X = X' \cdot 2^i$ for some integer $i \geq 0$. In other words, it returns the number obtained by removing consecutive 0 bits from the least significant bit of X. The reader should have no difficulty to confirm that the following lemma is correct.

Lemma 2. *For any even numbers X and Y, $\texttt{GCD}(X, Y) = 2 \cdot \texttt{GCD}(\frac{X}{2}, \frac{Y}{2})$ always holds. Also, for any odd number X and even number Y, $\texttt{GCD}(X, Y) = \texttt{GCD}(X, \frac{Y}{2}) = \texttt{GCD}(X, \texttt{rshift}(Y))$ always holds.*

For simplicity, we assume that both inputs X and Y are odd and $X \geq Y$ holds when we compute $\texttt{GCD}(X, Y)$. From Lemma 2, it should have no difficulty to modify all GCD algorithms shown in this paper to handle even input numbers. For later reference, let s denote the number of inputs bits of X and Y.

Let $\texttt{swap}(X, Y)$ denote a function to exchange the values of X and Y. We can write a standard Euclidean algorithm for computing the GCD of X and Y as follows:

```
[Original Euclidean algorithm]
gcd($X, Y$){
    do {
        $X \leftarrow X \bmod Y$; // $X < Y$ always holds
        swap($X, Y$); // $X > Y$ always holds
    } while($Y \neq 0$)
    return($X$);
}
```

Let $\alpha = \lfloor \frac{X}{Y} \rfloor$. From $X \bmod Y = X - Y \cdot \lfloor \frac{X}{Y} \rfloor$ and Lemma 1, this algorithm returns the GCD correctly. We will show that the Original Euclidean algorithm runs no more than $2s$ iterations of the do-loop. If $X < 2Y$, then X will store $X - Y$, which is less than $\frac{X}{2}$. Otherwise, X will store the value less than Y, which is no more than $\frac{X}{2}$. Hence, the value of X is halved or smaller and thus the number of bits in X is decreased by one or more. Since the number of bits of one of the two numbers is decreased by one, the Original Euclidean algorithm performs no more than $2s$ iterations.

Since modulo computation is costly, the Binary Euclidean algorithm, which does not perform modulo computations, is often used to compute the GCD efficiently:

```
[Binary Euclidean algorithm]
gcd($X, Y$){
    do{
        if($X$ is even) $X \leftarrow \frac{X}{2}$;
        else if($Y$ is even) $Y \leftarrow \frac{Y}{2}$;
        else $X \leftarrow \frac{X-Y}{2}$ // $X - Y$ is always even
        if($X < Y$) swap($X, Y$);
    } while ($Y \neq 0$)
    return($X$);
}
```

Clearly, when $\frac{X-Y}{2}$ is computed, both X and Y are odd. Hence, $X - Y$ is even and it makes sense to compute $\frac{X-Y}{2}$. If X (or Y) is even, then the number of bits in X (or in Y) is decreased by one. If both X and Y are odd, the number of bits in X is decreased by one or more. Thus, the number of iterations of the do-loop of the Binary Euclidean algorithm is also no more than $2s$.

Note that the Binary Euclidean algorithm removes 0 in the least significant bit. We can reduce the number of iterations of the do-loop by removing consecutive 0 bits. Using the `rshift` function, we can accelerate the Binary Euclidean algorithm as follows:

```
[Fast Binary Euclidean algorithm]
gcd($X, Y$){
    do {
        $X \leftarrow$rshift($X - Y$);
        if($X < Y$) swap($X, Y$)
    } while ($Y \neq 0$)
    return($X$);
}
```

From Lemmas 1 and 2, $\texttt{GCD}(X, Y) = \texttt{GCD}(X - Y, Y) = \texttt{GCD}(\texttt{rshift}(X - Y), Y)$ for all odd X and Y and thus, this algorithm correctly computes the GCD. In each iteration of the Fast Binary Euclidean algorithm, X and Y are always odd and the number of bits in X or in Y can be decreased by one or more. Hence, for any input numbers, the number of iterations of the do-while loop of the Fast Binary Euclidean algorithm is no larger than that of the Binary Euclidean algorithm.

For the reader's benefits, we use both the decimal system and the binary system to represent numbers. For example, numbers 223 in the decimal system or 11011111 in the binary system is denoted by "223", "1101,1111", or "1101,1111 (223)." Table 1 shows an example of computation performed by the Binary Euclidean algorithm and the Fast Binary Euclidean algorithm for

$$X = 1111, 1110, 1101, 1100, 1011(1043915) \text{ and,}$$
$$Y = 1011, 1011, 1011, 1011, 1011(768955).$$

We can confirm that the output 0101(5) is equal to the GCD of X and Y. The Euclidean algorithm computes the GCD in 24 iterations, while the Fast Euclidean algorithm runs only 16 iterations.

Table 1: An example of computation performed by the Binary Euclidean algorithm and the Fast Binary Euclidean algorithm

		Binary Euclidean algorithm	Fast Binary Euclidean algorithm
1	X	1111,1110,1101,1100,1011	1111,1110,1101,1100,1011
	Y	1011,1011,1011,1011,1011	1011,1011,1011,1011,1011
2	X	1011,1011,1011,1011,1011	1011,1011,1011,1011,1011
	Y	0010,0001,1001,0000,1000	0100,0011,0010,0001
3	X	1011,1011,1011,1011,1011	0101,1011,1100,0100,1101
	Y	0001,0000,1100,1000,0100	0100,0011,0010,0001
4	X	1011,1011,1011,1011,1011	0001,0101,1110,0100,1011
	Y	1000,0110,0100,0010	0100,0011,0010,0001
5	X	1011,1011,1011,1011,1011	1000,1101,1001,0101
	Y	0100,0011,0010,0001	0100,0011,0010,0001
6	X	0101,1011,1100,0100,1101	0100,0011,0010,0001
	Y	0100,0011,0010,0001	0001,0010,1001,1101
7	X	0010,1011,1100,1001,0110	0001,0010,1001,1101
	Y	0100,0011,0010,0001	1100,0010,0001
8	X	0001,0101,1110,0100,1011	1100,0010,0001
	Y	0100,0011,0010,0001	0001,1001,1111
9	X	1000,1101,1001,0101	0101,0100,0001
	Y	0100,0011,0010,0001	0001,1001,1111
10	X	0100,0011,0010,0001	0001,1101,0001
	Y	0010,0101,0011,1010	0001,1001,1111
11	X	0100,0011,0010,0001	0001,1001,1111
	Y	0001,0010,1001,1101	0001,1001
12	X	0001,1000,0100,0010	1100,0011
	Y	0001,0010,1001,1101	0001,1001
13	X	0001,0010,1001,1101	0101,0101
	Y	1100,0010,0001	0001,1001
14	X	1100,0010,0001	0001,1001
	Y	0011,0011,1110	1111
15	X	1100,0010,0001	1111
	Y	0001,1001,1111	0101
16	X	0101,0100,0001	0101
	Y	0001,1001,1111	0101
17	X	0001,1101,0001	0101
	Y	0001,1001,1111	0000
18	X	0001,1001,1111	
	Y	0001,1001	
19	X	1100,0011	
	Y	0001,1001	
20	X	0101,0101	
	Y	0001,1001	
21	X	0001,1110	
	Y	0001,1001	
22	X	0001,1001	
	Y	1111	
23	X	1111	
	Y	0101	
24	X	0101	
	Y	0101	
	X	0101	
	Y	0000	

Using the idea of removing consecutive 0 bits used in the Fast Binary Euclidean algorithm, we can accelerate the Original Euclidean algorithm. Let " div " denote quotient operator such that X div $Y = \lfloor \frac{X}{Y} \rfloor$, that is, the rounded-down integer of $\frac{X}{Y}$. Clearly, we have $X \bmod Y = X - Y \cdot (X \text{ div } Y)$. Thus, we can rewrite the Original Euclidean algorithm as follows:

```
[Original Euclidean algorithm using div ]
gcd($X$,$Y$){
  do {
    $Q \leftarrow X$ div $Y$;
    $X \leftarrow X - Y \cdot Q$;
    swap($X$,$Y$);
  } while($Y \neq 0$);
  return($X$);
}
```

If $X - Y \cdot Q$ is even, then we can reduce the number of bits in X by `rshift`(X). Since X and Y are odd, $X - Y \cdot Q$ is even when Q is odd. However, if Q is even then $X - Y \cdot Q$ is odd and `rshift` does not remove 0 bits. Hence, it makes sense to decrease Q by one if Q is even. Using this idea, we can further accelerate the Original Euclidean algorithm as follows:

```
[Fast Euclidean algorithm]
gcd($X$, $Y$){
  do {
    $Q \leftarrow X$ div $Y$;
    if($Q$ is even) $Q \leftarrow Q - 1$
    $X \leftarrow$ rshift($X - Y \cdot Q$);
    if($X < Y$) swap($X$,$Y$);
  } while($Y \neq 0$);
  return($X$);
}
```

From Lemmas 1 and 2, `GCD`(X, Y) = `GCD`(`rshift`($X - Y \cdot Q$), Y) and thus, this algorithm correctly computes the GCD. Note that, X may be larger than Y after executing $X \leftarrow X - Y \cdot Q$. For example, if $X = 15$ and $Y = 7$, then X div $Y = 2$. Hence, $X = 15 - 7 \cdot (2 - 1) = 8$ and $X > Y$ holds. Thus, we need to compare X and Y and exchange them if $X < Y$, to guarantee that $X \geq Y$ holds for the next iteration.

Table 2 shows an example of computation performed by the Original Euclidean algorithm and the Fast Euclidean algorithm for the same input numbers X and Y as Table 1. We can see that they perform fewer iterations than the Binary Euclidean algorithm and the Fast Binary Euclidean algorithm. Also, the Fast Euclidean algorithm performs fewer iterations than the Original Euclidean algorithm. However, for some input numbers, the Fast Euclidean algorithm performs more iterations than the Original Euclidean algorithm. For example, if $X = 39$ and $Y = 9$, then the GCD is computed as follows. The Original Euclidean algorithm runs 2 iterations: $(39, 9) \rightarrow (9, 3) \rightarrow (3, 0)$. The Fast Euclidean algorithm runs 3 iterations: $(39, 9) \rightarrow (12, 9) \rightarrow (9, 3) \rightarrow (3, 0)$. Although such examples exist, the Fast Euclidean algorithm takes fewer iterations than the Original Euclidean algorithm for most input numbers.

3 The Approximate Euclidean algorithm for computing the GCD

The main purpose of this section is to show our new Euclidean algorithm called *the Approximate Euclidean algorithm*.

The Approximate Euclidean algorithm is based on the Fast Euclidean algorithm presented in the previous section. The computation of quotient for large numbers performed by Fast Euclidean algorithm is costly. Our new idea is to find a good approximation of quotient by small computing

Table 2: An example of computation performed by the Original Euclidean algorithm and the Fast Euclidean algorithm

		Original Euclidean algorithm		Fast Euclidean algorithm	
		$X\&Y$	Q	$X\&Y$	Q
1	X	1111,1110,1101,1100,1011	1	1111,1110,1101,1100,1011	1
	Y	1011,1011,1011,1011,1011		1011,1011,1011,1011,1011	
2	X	1011,1011,1011,1011,1011	2	1011,1011,1011,1011,1011	43
	Y	0100,0011,0010,0001,0000		0100,0011,0010,0001	
3	X	0100,0011,0010,0001,0000	1	0100,0011,0010,0001	9
	Y	0011,0101,0111,1001,1011		0111,0101,0011	
4	X	0011,0101,0111,1001,1011	3	0111,0101,0011	11
	Y	1101,1010,0111,0101		1001,1011	
5	X	1101,1010,0111,0101	1	1001,1011	1
	Y	1100,1000,0011,1100		0101,0101	
6	X	1100,1000,0011,1100	10	0101,0101	1
	Y	0001,0010,0011,1001		0010,0011	
7	X	0001,0010,0011,1001	1	0010,0011	1
	Y	0001,0010,0000,0010		0001,1001	
8	X	0001,0010,0000,0010	83	0001,1001	5
	Y	0011,0111		0101	
9	X	0011,0111	1	0101	
	Y	0010,1101		0000	
10	X	0010,1101	4		
	Y	1010			
11	X	1010	2		
	Y	0101			
	X	0101			
	Y	0000			

costs. We assume that X and Y are stored in multiple d-bit words, and let $D = 2^d$. The Approximate Euclidean algorithm is described as follows:

```
[Approximate Euclidean algorithm]
gcd($X, Y$){
  do {
    $(\alpha, \beta) \leftarrow$ approx($X, Y$);
    if($\beta = 0$){
      if($\alpha$ is even) $\alpha = \alpha - 1$; //$\alpha$ is odd
      $X \leftarrow$ rshift($X - Y \cdot \alpha$); //$Y \cdot \alpha$ is odd
    } else $X \leftarrow$ rshift($X - Y \cdot \alpha \cdot D^\beta + Y$); //$\alpha \cdot D^\beta$ is even
    if($X < Y$) swap($X, Y$);
  } while ($Y \neq 0$);
  return($X$);
}
```

From Lemmas 1 and 2, $\texttt{GCD}(X, Y) = \texttt{GCD}(X - Y \cdot \alpha, Y) = \texttt{GCD}(\texttt{rshift}(X - Y \cdot \alpha \cdot D^\beta + Y))$ holds, and thus, this algorithm is correct. In this algorithm, $\texttt{approx}(X,\ Y)$ is a function to compute a pair (α, β) such that $\alpha \cdot D^\beta$ $(\leq Q)$ is a good approximation of $Q = X$ div Y, and the computing cost of $\texttt{approx}(X, Y)$ is much smaller than that of X div Y. Also, to guarantee that X is even, $X - Y \cdot (\alpha \cdot D^\beta - 1)$ is computed if $\alpha \cdot D^\beta$ is even. Note that if $\alpha \cdot D^\beta$ is always 1, that is, $(\alpha, \beta) = (1, 0)$ then the Approximate Euclidean algorithm is the same as the Fast Binary Euclidean algorithm. Since the value of $\alpha \cdot D^\beta$ can be more than 1, the number of iterations in the Approximate Euclidean algorithm may be smaller than Binary Euclidean algorithms.

We first show the idea of implementation of approx(X, Y). Suppose that X and Y are represented by l_X d-bit words $x_1 x_2 \cdots x_{l_X}$ and l_Y d-bit words $y_1 y_2 \cdots y_{l_Y}$. In other words,

$$X = x_1 D^{l_X - 1} + x_2 D^{l_X - 2} + \cdots + x_{l_X} D^0$$

and

$$Y = y_1 D^{l_Y - 1} + y_2 D^{l_Y - 2} + \cdots + y_{l_Y} D^0$$

hold. It should be clear that $l_X \geq l_Y$ always holds from $X \geq Y$. Let $\langle x_1 x_2 \rangle (= x_1 \cdot D + x_2)$ and $\langle y_1 y_2 \rangle (= y_1 \cdot D + y_2)$ be integers represented most significant two d-bit words of X and Y. Basically, approx(X, Y) returns a pair $(\langle x_1 x_2 \rangle \text{ div } (\langle y_1 y_2 \rangle + 1), l_X - l_Y)$. Hence, $\alpha \cdot D^\beta = (\langle x_1 x_2 \rangle \text{ div } (\langle y_1 y_2 \rangle + 1)) \cdot D^{l_X - l_Y}$ is used as an approximation of $Q = X \text{ div } Y$. Also, it is guaranteed that $\alpha \cdot D^\beta \leq Q$. Thus, $X - Y \cdot \alpha \cdot D^\beta$ is always non-negative.

We show an example using 4-bit words, that is, $d = 4$. Let $X = 1101, 1001, 0000, 0011(55555)$, and $Y = 0100, 1101, 0010(1234)$. If this is the case, $l_X = 4$, $l_Y = 3$, $\langle x_1 x_2 \rangle = 1101, 1001(217)$ and $\langle y_1 y_2 \rangle = 0100, 1101(77)$. Hence we have, $\langle x_1 x_2 \rangle \text{ div } (\langle y_1 y_2 \rangle + 1) = 217 \text{ div } (77 + 1) = 2$ and $l_X - l_Y = 1$. Thus, approx(X, Y) returns $(\alpha, \beta) = (2, 1)$ and we have $\alpha \cdot D^\beta = 2 \cdot 16^1 = 32$, which approximates X div $Y = 45$. Using this idea, the following function approx computes a pair (α, β):

```
approx($X, Y$){
  if($l_X \leq 2$)
    return ($X$ div $Y, 0$); // Case 1
  if($l_Y = 1$) {
    if($x_1 \geq y_1$)
      return ($x_1$ div $y_1, l_X - 1$); // Case 2-A
    else
      return ($\langle x_1 x_2 \rangle$ div $y_1, l_X - 2$); // Case 2-B
  }
  if($l_Y = 2$) {
    if($\langle x_1 x_2 \rangle \geq \langle y_1 y_2 \rangle$)
      return ($\langle x_1 x_2 \rangle$ div $\langle y_1 y_2 \rangle, l_X - 2$);// Case 3-A
    else
      return ($\langle x_1 x_2 \rangle$ div $(y_1 + 1), l_X - 3$); // Case 3-B
  }
  if($\langle x_1 x_2 \rangle > \langle y_1 y_2 \rangle$)
    return ($\langle x_1 x_2 \rangle$ div $(\langle y_1 y_2 \rangle + 1), l_X - l_Y$); // Case 4-A
  if($l_X > l_Y$)
    return ($\langle x_1 x_2 \rangle$ div $(y_1 + 1)), l_X - l_Y - 1$); // Case 4-B
  return $(1, 0)$; // Case 4-C
}
```

The reader should have no difficulty to confirm that operands of " div " have at most 2 words, that is, $2d$ bits. Also, the resulting value of " div " has at most d bits.

Let us see how approx(X, Y) computes (α, β). It has four cases determined by the values of l_X and l_Y as illustrated in Figure 1. We will show that function approx outputs a good approximation $\alpha \cdot D^\beta$ of X div Y for each cases

Case 1: X has 1 or 2 words.

Clearly, Y also has 1 or 2 words from $X \geq Y$. Hence, approx outputs (X div $Y, 0$) and we have $\alpha \cdot D^\beta = X$ div Y.

Example: If $X = 1101, 1111(223)$ and $Y = 0010, 1101(45)$ then approx outputs $(223 \text{ div } 45, 0) = (4, 0)$.

Case 2: X has more than 2 words and Y has 1 word. Case 2 has two sub-cases as follows:

l_Y \ l_X	1	2	3	4	5	6
1	1	1	2	2	2	2
2		1	3	3	3	3
3			4	4	4	4
4				4	4	4
5					4	4
6						4

Figure 1: Cases for the values of l_X and l_Y

Case 2-A: If $x_1 \geq y_1$ then `approx` outputs $(x_1 \text{ div } y_1, l_X - 1)$.
Example: If $X = 1001, 0010, 1001(2345)$ and $Y = y_1 = 0100(4)$ then $x_1 = 1001(9)$ and $x_1 \geq y_1$ hold. If this is the case, `approx` outputs $(9 \text{ div } 4, 3 - 1) = (2, 2)$. We can confirm that $\alpha \cdot D^\beta = 2 \cdot 16^2 = 512$ approximates $X \text{ div } Y = 2345 \text{ div } 4 = 586$.

Case 2-B: If $x_1 < y_1$ then `approx` outputs $(\langle x_1, x_2\rangle \text{ div } y_1, l_X - 2)$.
Example: If $X = 0100, 1101, 0010(1234)$ and $Y = 1100(12)$ then $x_1 = 0100(4)$ and $\langle x_1, x_2\rangle = 0100, 1101(77)$ hold. Hence, $x_1 < y_1$ is satisfied and `approx` outputs $(77 \text{ div } 12, 3-2) = (6, 1)$. We can confirm that $\alpha \cdot D^\beta = 6 \cdot 16^1 = 96$ approximates $X \text{ div } Y = 1234 \text{ div } 12 = 102$.

Case 3: X has more than 2 words and Y has 2 words. Case 3 has two sub-cases as follows:

Case 3-A: If $\langle x_1x_2\rangle \geq \langle y_1y_2\rangle$ then `approx` outputs $(\langle x_1x_2\rangle \text{ div } \langle y_1y_2\rangle, l_X - l_Y)$.
Example: If $X = 1001, 0010, 1001(2345)$ and $Y = 0011, 1011(59)$ then $\langle x_1x_2\rangle = 1001, 0010(146)$. Hence $\langle x_1x_2\rangle \geq \langle y_1y_2\rangle$ is satisfied and `approx` outputs $(146 \text{ div } 59, 3 - 2) = (2, 1)$. We can confirm that $\alpha \cdot D^\beta = 2 \cdot 16^1 = 32$ approximates $X \text{ div } Y = 2345 \text{ div } 59 = 39$.

Case 3-B: If $\langle x_1x_2\rangle < \langle y_1y_2\rangle$ then `approx` outputs $(\langle x_1x_2\rangle \text{ div } (y_1 + 1), l_X - 3)$.
Example: If $X = 1001, 0010, 1001(2345)$ and $Y = 1110, 0111(231)$ then $\langle x_1x_2\rangle = 1001, 0010(146)$ and $y_1 = 1110(14)$. Since $\langle x_1x_2\rangle < \langle y_1y_2\rangle$ satisfied, `approx` outputs $(146 \text{ div } (14+1), 3-3) = (9, 0)$. We can confirm that $\alpha \cdot D^\beta = 9 \cdot 16^0 = 9$ approximates $X \text{ div } Y = 2345 \text{ div } 231 = 10$.

Case 4: Both X and Y have more than 2 words. Case 4 has three sub-cases as follows:

Case 4-A: If $\langle x_1x_2\rangle > \langle y_1y_2\rangle$ then `approx` outputs $(\langle x_1x_2\rangle \text{ div } (\langle y_1y_2\rangle + 1), l_X - l_Y)$. Note that, from $\langle x_1x_2\rangle > \langle y_1y_2\rangle$, we always have $\langle y_1y_2\rangle \leq D^2 - 1$. Hence $\langle y_1y_2\rangle$ has at most $2d$ bits.
Example: If $X = 1101, 0100, 0011, 0001(54321)$ and $Y = 0100, 1101, 0010(1234)$ then $\langle x_1x_2\rangle = 1101, 0100(212)$ and $\langle y_1y_2\rangle = 0100, 1101(77)$. Since $\langle x_1x_2\rangle > \langle y_1y_2\rangle$ is satisfied, `approx` outputs $(212 \text{ div } (77+1), 4-3) = (2, 1)$. We can confirm that $\alpha \cdot D^\beta = 2 \cdot 16^1 = 32$ approximates $X \text{ div } Y = 54321 \text{ div } 1234 = 44$.

Case 4-B: If $\langle x_1x_2\rangle \leq \langle y_1y_2\rangle$ and $l_X > l_Y$ then `approx` outputs $(\langle x_1x_2\rangle \text{ div } (y_1 + 1), l_X - l_Y - 1)$.
Example: If $X = 1101, 0100, 0011, 0001(54321)$ and $Y = 1111, 1010, 0000(4000)$ then $\langle x_1x_2\rangle = 1101, 0100(212)$ and $\langle y_1y_2\rangle = 1111, 1010(250)$ hold. Hence, $\langle x_1x_2\rangle \leq \langle y_1y_2\rangle$ holds. Since $y_1 = 1111(15)$, `approx` outputs $(212 \text{ div } (15 + 1), 4 - 3 - 1) = (13, 0)$. We can confirm that $\alpha \cdot D^\beta = 13 \cdot 16^0 = 13$ approximates $X \text{ div } Y = 54321 \text{ div } 4000 = 13$.

Case 4-C: If this is the case, $\langle x_1x_2\rangle \leq \langle y_1y_2\rangle$ and $l_X \leq l_Y$ hold. Recall that $X \geq Y$ and thus $\langle x_1x_2\rangle = \langle y_1y_2\rangle$ and $l_X = l_Y$ must be satisfied. Hence the values of X and Y are almost the same and it makes sense to return $(1, 0)$ and $\alpha \cdot D^\beta = 1 \cdot 16^0 = 1$ if this is the case.

Table 3: An example of computation performed by the Approximate Euclidean algorithm

		X & Y	CASE	(α, β)
1	X	$\underline{1111{,}1110}$,1101,1100,1011	4-A	$(1, 0)$
	Y	$\underline{1011{,}1011}$,1011,1011,1011		
	X	$\underline{1011{,}1011}$,1011,1011,1011	4-A	$(2, 1)$
	Y	$\underline{0100{,}0011}$,0010,0001		
3	X	$\underline{1110{,}0110}$,1010,1111	4-A	$(3, 0)$
	Y	$\underline{0100{,}0011}$,0010,0001		
4	X	$\underline{0100{,}0011}$,0010,0001	4-B	$(7, 0)$
	Y	$\underline{0111}$,0101,0011		
5	X	$\underline{0111{,}0101}$,0011	4-A	$(1, 0)$
	Y	$\underline{0011{,}1111}$,0111		
6	X	$\underline{0011{,}1111}$,0111	3-B	$(3, 0)$
	Y	$\underline{1101}$,0111		
7	X	1101,0111	1	$(1, 0)$
	Y	1011,1001		
8	X	1011,1001	1	$(11, 0)$
	Y	1111		
9	X	1111	1	$(3, 0)$
	Y	0101		
	X	0101		
	Y	0000		

Table 3 shows an example of computation performed by the Approximate Euclidean algorithm for 4-bit words, that is, $d = 4$ and $D = 16$. It computes the GCD for the same inputs used in Tables 1 and 2 in 9 steps. The values used to compute α in `approx` are underlined. We can confirm that the Approximate Euclidean algorithm outputs 0101(5), the GCD of X and Y correctly.

Recall that the Fast Euclidean algorithm computes the exact value of quotient $Q = X \text{ div } Y$. On the other hand, the Approximate Euclidean algorithm uses an approximation $\alpha \cdot D^{\beta}$ of quotient Q. Hence, the Approximate Euclidean algorithm may take more iterations than the Fast Euclidean algorithm. Actually, from Tables 2 and 3, we can see that the Fast Euclidean algorithm and the Approximate Euclidean algorithm perform 8 and 9 iterations, respectively, for the same input numbers.

Table 4 shows the average number of iterations of do-while loops performed by the Euclidean algorithms, (A) the Original Euclidean algorithm, (B) the Fast Euclidean algorithm, (C) the Binary Euclidean algorithm, (D) the Fast Binary Euclidean algorithm, (E) the Approximate Euclidean algorithm when pairs of two randomly generated s-bit unsigned odd integers for $s = 1024$, 2048, 4096, 8192, and 16384, respectively. We have generated integers with 2G bytes totally and evaluated the number of iterations. For example, when $s = 1024$, we have generated 8 Mega pairs of 1024-bit odd integers in $[2^{1023}, 2^{1024})$.

Each iteration of (A) and (B) is very costly, because they computes quotient/modulo of two s-bit numbers. On the other hand, (C) and (D) involves no division/multiplication operation. Further, each iteration of (E) involves one division of two 64-bit numbers, and $\frac{s}{32}$ repetitions 32-bit multiplications. Hence, the computation of each iteration takes more time than that of (C) and (D). However, they perform the same memory access operations to X and Y. Thus, if memory access latency is large like GPUs, the computing time of each iteration of (E) is just little larger than that of (C) and (D). Hence, it makes sense to evaluate and compare the number of iterations of (C), (D), and (E).

From Table 4, we can see that

1. the number of iterations is proportional to the number of input bits,

Table 4: The average number of iterations performed by the Euclidean algorithms, (A) the Original Euclidean algorithm, (B) the Fast Euclidean algorithm, (C) the Binary Euclidean algorithm, (D) the Fast Binary Euclidean algorithm, and (E) the Approximate Euclidean algorithm for 100000 random odd numbers

	1024	2048	4096	8192	16384
(A) Original Euclidean algorithm	598.5	1196.7	2393.1	4785.8	9571.5
(B) Fast Euclidean algorithm	380.9	761.7	1523.1	3046.0	6091.6
(C) Binary Euclidean algorithm	1444.8	2890.6	5782.3	11565.6	23132.2
(D) Fast Binary Euclidean algorithm	723.4	1446.4	2892.2	5783.8	11567.1
(E) Approximate Euclidean algorithm	380.9	761.7	1523.2	3046.0	6091.7
(E)−(B)	0.0059	0.0120	0.0227	0 .0457	0.0923

Table 5: The probability that $\beta > 0$ when the Approximate Euclidean algorithm is executed

1024	2048	4096	8192	16384
4.38099×10^{-9}	5.63433×10^{-9}	6.88731×10^{-9}	5.00944×10^{-9}	5.94903×10^{-9}

2. the number of iterations of (E) is about a half of (D) and about a quarter of (C), and

3. the number of iterations of (B) is the same as that of (E).

To see the small difference of (B) and (E), the table also show the average value of (E)-(B), that is, the number of iterations of (E) minus that of (B). Quite surprisingly, their difference is only 0.001%-0.002%. Recall that (B) computes the exact quotient by division of two large numbers, while (E) computes an approximation by 64-bit division. Hence, we can say that approximated quotient is sufficient for computing the GCD.

We should also note that the value of β computed by function `approx` in the Approximate Euclidean algorithm is zero with very high probability. In the experiments to obtain Table 4, we have recorded the value of β for each call of function `approx`. From the record, we have obtained Table 5, which shows the probability that $\beta > 0$. We can see that the probability is very small and it is very rare that $X \leftarrow$`rshift`$(X - Y \cdot \alpha \cdot D^{\beta} + Y)$ is executed.

4 Further acceleration using PTX instructions

PTX is a low-level parallel thread execution virtual machine and instruction set architecture of GPU [16]. We can embed PTX instructions in CUDA C program as inline assembly codes. The 128-bit product of two 64-bit unsigned integers cannot be obtained directly by a C language program. On the other hand, PTX includes instructions for computing the 128-bit product. We note that the PTX instructions are the most fundamental instructions on CUDA and cannot be divided into smaller instructions. If some libraries are provided to compute 128-bit or more product, they consists of several PTX codes. Therefore, we have used PTX instructions including the multiplication of two 64-bit unsigned integers to accelerate the computation of `rshift`$(X-Y\cdot\alpha)$ and `rshift`$(X-Y\cdot\alpha\cdot D^{\beta}+Y)$.

Let $lo(X)$ and $hi(X)$ denote the least significant 64-bit and the most significant 64-bit unsigned integers of a 128-bit unsigned integer X. More specifically, $lo(X) = x_{63}x_{62}\cdots x_0$ and $hi(X) = x_{127}x_{126}\cdots x_{64}$, where $X = x_{127}x_{126}\cdots x_0$.

For 64-bit unsigned integer variables a, b, c, and d, we use the following PTX instructions for multiplications.

mul.lo.u64 *d, a, b*: $d \leftarrow lo(a \cdot b)$ is performed.

mul.hi.u64 *d, a, b*: $d \leftarrow hi(a \cdot b)$ is performed.

mad.lo.u64 d, a, b, c: $d \leftarrow lo(a \cdot b + c)$ is performed

mad.hi.u64 d, a, b, c: $d \leftarrow hi(a \cdot b + c \cdot 2^{64})$ is performed

For 64-bit unsigned integer variables d, a, and b, and 32-bit signed integer variable c. We use the following two PTX instructions to handle carry propagation:

setp.lo.s32.u64 c, a, b: $d \leftarrow (a < b)? -1 : 0$ is performed. In other words, if $a < b$ then $d \leftarrow -1$ is performed. Otherwise, $d \leftarrow 0$ is performed.

slct.u64.s32 d, a, b, c: $d \leftarrow (c \geq 0)?a : b$ is performed. In other words, if c is non-negative then $d \leftarrow a$ is performed. Otherwise, $d \leftarrow b$ is performed.

We use these two PTX instructions as follows:

```
setp.lo.s32.u64  c, a, b       // c ← (a < b)? − 1 : 0
slct.u64.s32     d, 0, 1, c    // d ← (c ≥ 0)?0 : 1
```

Clearly, by these two PTX instructions, $d \leftarrow (a < b)?1 : 0$ is performed.

We will show an example that uses these PTX instructions. Let $Y = y_1y_2y_3y_4$ be four 64-bit unsigned integers variables that constitute a 256-bit unsigned integer and α be a 64-bit unsigned integer variable. We can compute $Z = Y\alpha$, where $Z = z_0z_1z_2z_3z_4$ be five 64-bit unsigned integer variables, by the following PTX program.

```
(1)  mul.lo.u64       z4, y4, α       // z4 ← lo(y4 · α)
(2)  mul.hi.u64       c, y4, α        // c ← hi(y4 · α)
(3)  mad.lo.u64       z3, y3, α, c    // z3 ← lo(y3 · α + c)
(4)  setp.lo.s32.u64  t, z3, c        // t ← (z3 < c)? − 1 : 0
(4)  slct.u64.s32     c, 0, 1, t      // c ← (t ≥ 0)?0 : 1
(5)  mad.hi.u64       c, y3, α, c     // c ← hi(y3 · α + c)
(6)  mad.lo.u64       z2, y2, α, c    // z2 ← lo(y2 · α + c)
(7)  setp.lo.s32.u64  t, z2, c        // t ← (z2 < c)? − 1 : 0
(7)  slct.u64.s32     c, 0, 1, t      // c ← (t ≥ 0)?0 : 1
(8)  mad.hi.u64       c, y2, α, c     // c ← hi(y2 · α + c)
(9)  mad.lo.u64       z1, y1, α, c    // z1 ← lo(y1 · α + c)
(10) setp.lo.s32.u64  t, z1, c        // t ← (z1 < c)? − 1 : 0
(10) slct.u64.s32     c, 0, 1, t      // c ← (≥ 0)?0 : 1
(11) mad.hi.u64       z0, y1, α, c    // z0 ← hi(y1 · α + c)
```

In this PTX program, c is a 64-bit unsigned integer variables to store the carry and t is a 32-bit signed integer variables. Figure 2 illustrates the $Z = Y \cdot \alpha$ by the PTX program. This PTX program computes the values of Z from the least significant 64-bit digit one by one. Although the computation of $X \leftarrow$`rshift`$(X - Y \cdot \alpha)$ and $X \leftarrow$`rshift`$(X - Y \cdot \alpha \cdot D^{\beta} + Y)$ are much more complicated, they can be done by a similar way using these PTX instructions.

5 Oblivious sequential algorithms and bulk execution

In this section we review the obliviousness of sequential algorithms and the bulk execution of them. We then go on to show that the bulk execution of oblivious sequential algorithm can be implemented very efficiently in CUDA-enabled GPUs. Please see [23] for the details. We further define *semi-obliviousness* of sequential algorithms. Intuitively, a semi-oblivious sequential algorithm is not oblivious, but it is almost oblivious.

Intuitively, a sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input [23] . More specifically, there exists a function $a : \{0, 1, \ldots, t-1\} \rightarrow \mathcal{N}$, where t is the running time of the algorithm and $\mathcal{N}$ is a set of all non-negative integers such that, for any input of the algorithm, it accesses address $a(i)$ or does not access the memory at each time i

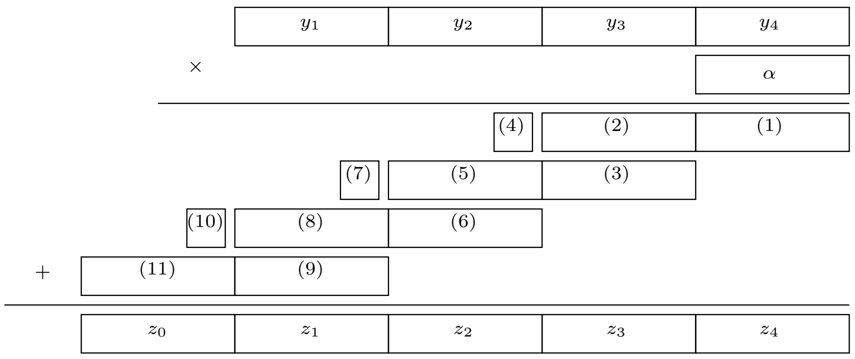

Figure 2: Illustrating the computation of $Z = Y \cdot \alpha$ using PTX instructions

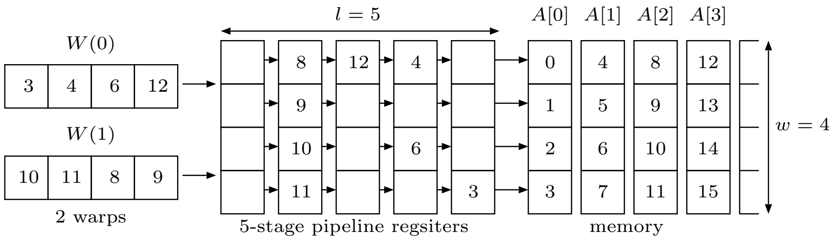

Figure 3: The UMM with width $w = 4$ and latency $l = 5$

$(0 \le i \le t-1)$. In other words, at each time i $(0 \le i \le t-1)$, it never accesses an address other than $a(i)$. Suppose that we need to execute a sequential algorithm for many different inputs on a single CPU in turn or on a parallel machine at the same time. We call such computation *bulk execution.*

For theoretical performance analysis of the Approximate Euclidean algorithm, we first define *the UMM (the Unified Memory Machine)* [12, 13] which captures the essence of the global memory access of CUDA-enabled GPUs. We then go on to show that the bulk execution oblivious algorithms can be implemented very efficiently on the UMM. Let us define the UMM with width w and latency l. The memory of the UMM is partitioned into address groups $A[0], A[1], \ldots$ such that each $A[j]$ $(j \ge 0)$ involves $j \cdot w, j \cdot w+1, \ldots, (j+1) \cdot w - 1$. The reader should refer to Figure 3 that illustrates address groups for $w = 4$. Also, the memory access is performed through l-stage pipeline registers as illustrated in Figure 3. Let p be the number of threads of the UMM and $T(0)$, $T(1)$, ..., $T(p-1)$ be the p threads. We assume that p is a multiple of w. The p threads are partitioned into $\frac{p}{w}$ groups called *warps* with w threads each. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w+1), \ldots, T((i+1) \cdot w - 1)\}$. Warps are dispatched for the memory access in turn, and w threads in a warp try to access the memory at the same time. More specifically, $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ are dispatched in a round-robin manner if at least one thread in a warp requests the memory access. If no thread in a warp needs the memory access, such warp is not dispatched for the memory access. When $W(i)$ is dispatched, w threads in $W(i)$ send the memory access requests, one request per thread, to the memory banks.

For the memory access, each warp sends the memory access requests to the memory banks through the l-stage pipeline registers. We assume that each stage can store the memory access requests destined for the same address group. For example, since the memory access requests by $W(0)$ are separated in three address groups in the figure, they occupy three stages of the pipeline registers. Also, those by $W(1)$ are in the same address group, they occupy only one stage. In general, if the memory access requests by a warp are destined for d address groups, they occupy d stages. For simplicity, we assume that the memory access is completed as soon as the request reaches the last pipeline stage. Thus, all memory access requests by $W(0)$ and $W(1)$ in the figure are completed in $3(\text{address groups}) + 1(\text{address group}) + 5(\text{latency}) - 1 = 8$ time units. We also assume that a thread

$b_0[0]$	$b_1[0]$	$b_2[0]$	$b_3[0]$	$b_4[0]$	$b_5[0]$	$b_6[0]$	$b_7[0]$
$b_0[1]$	$b_1[1]$	$b_2[1]$	$b_3[1]$	$b_4[1]$	$b_5[1]$	$b_6[1]$	$b_7[1]$
$b_0[2]$	$b_1[2]$	$b_2[2]$	$b_3[2]$	$b_4[2]$	$b_5[2]$	$b_6[2]$	$b_7[2]$
$b_0[3]$	$b_1[3]$	$b_2[3]$	$b_3[3]$	$b_4[3]$	$b_5[3]$	$b_6[3]$	$b_7[3]$

Figure 4: Column-wise arrangement for $p = 8$ arrays of size $n = 4$ each

cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread sends a memory access request, it must wait at least l time units to send a new memory access request.

We will show that the bulk execution of an oblivious sequential algorithm can be done efficiently on the UMM. Without loss of generality, we can assume that an oblivious sequential algorithm works on a 1-dimensional array b of size n. If p threads on the UMM perform the bulk execution, the global memory stores p arrays of b. We use *column-wise arrangement* to allocate p arrays as illustrated in Figure 4. More specifically, let $b_j[i]$ denote the i-th element of b for thread j. Each $b_j[i]$ is allocated in address $j \cdot p + i$. If all threads execute a same oblivious algorithm, then they access the same address at each time unit. In other words, if a sequential algorithm accesses index i at some time unit, p threads access $b_0[i], b_1[i], \ldots, b_{p-1}[i]$ at the same time. Clearly, they are arranged in addresses $i \cdot p + 0, i \cdot p + 1, \ldots, i \cdot p + (p - 1)$ in the same row of the 2-dimensional array. Hence, they are in consecutive addresses and memory access by p threads is always coalesced.

Let us evaluate the computing time for the bulk execution of an oblivious sequential algorithm on the UMM. Let t be the running time of an oblivious sequential algorithm and p be the number of inputs and the number of threads. For each memory access of the obvious sequential algorithm p threads performs coalesced memory access. Since they are in $\frac{p}{w}$ address groups, it can be completed in $\frac{p}{w} + l - 1$ time units. Since the oblivious sequential algorithm performs at most t memory access operations, p threads on the UMM terminates in $(\frac{p}{w} + l - 1) \cdot t = O(\frac{pt}{w} + lt)$ time units. Thus we have,

Lemma 3. *The bulk execution of an oblivious sequential algorithm runs $O(\frac{pt}{w} + lt)$ time units using p threads on the UMM with width w and latency l, where t is the running time of the corresponding oblivious sequential algorithm.*

In our previous paper [23], we have proved that Lemma 3 is time-optimal.

Let us define semi-obliviousness of a sequential algorithm. Suppose that a sequential algorithm runs t time units. A sequential algorithm is *semi-oblivious with parameter* γ $(0 \leq \gamma \leq 1)$ if it is oblivious in $(1 - \gamma)t$ time units. More specifically, it an address accessed at each of $(1 - \gamma)t$ time units out of t time units is independent of the input. Hence, an address accessed may be different in γt time units. Clearly, it is obvious if $\gamma = 0$.

We will prove that the bulk execution of a semi-oblivious algorithm with parameter γ can be implemented efficiently in the UMM if $\gamma \leq O(\frac{1}{w})$. Again, let t be the running time of an oblivious sequential algorithm and p be the number of inputs and the number of threads. From Lemma 3, the bulk execution of an oblivious sequential algorithm runs $O(\frac{pt}{w} + lt)$ time units on the UMM with width w and latency l. Suppose that γt memory access operations out of t operations is not oblivious. If this semi-oblivious algorithm is implemented on the UMM, each of such memory access operations occupies at most p pipeline registers. Hence, the bulk execution of a semi-oblivious sequential algorithm runs in $O(\frac{pt}{w} + lt + pt\gamma)$ time units. Thus, we have,

Lemma 4. *The bulk execution of a semi-oblivious sequential algorithm with parameter $O(\frac{1}{w})$ runs $O(\frac{pt}{w} + lt)$ time units using p threads on the UMM with width w and latency l, where t is the running time of the corresponding semi-oblivious sequential algorithm.*

Regarding performance analysis with the UMM, in our previous papers [7, 23], we showed the performance analysis on the theoretical model including the UMM. Also, its correctness has also been shown by actually verifying the performance evaluation with the GPU implementations for various problems. Therefore, the performance analysis on the UMM is reasonable for the global memory access. Moreover, since the latency of the global memory is 200 to 400 clock cycles [15], if a global memory access is not coalesced, that is, several memory access requests are issued, the number of clock cycles for the access becomes constant times more, but the performance is affected much unless the number of memory access instructions is extremely small. Therefore, the number of global memory requests is one of the important factors for the performance analysis.

6 Oblivious sequential algorithms with synchronization

The main purpose of this section is to define *a sequential algorithm with synchronization*. We also show that bulk execution of a sequential algorithm with synchronization runs in the UMM efficiently if it is oblivious.

A sequential algorithm with synchronization can execute a synchronize instruction *sync*. This instruction is something like NOP (No Operation) instruction that does nothing. However, when the bulk execution of a sequential algorithm with synchronization is performed in parallel, this instruction is used for barrier synchronization. In other words, if a thread executes sync instruction, it is stalled until all the other threads execute sync instruction.

We can think that an execution of a sequential algorithm with synchronization is separated into sub-executions by sync instruction. We say that a sequential algorithm with synchronization is oblivious if every sub-execution is oblivious. Since the definition of obliviousness for a sequential algorithm with synchronization is hard to understand, we explain it using an example.

Let us consider *the row-wise OR problem* defined as follows. Suppose that we have an integer matrix a of size $n \times n$. We want to compute each element $b[i]$ of an array b such that

$$\begin{aligned} b[i] &= 0 \quad \text{if } a[i][0] = a[i][1] = \cdots = a[i][n-1] = 0, \\ &= 1 \quad \text{otherwise.} \end{aligned}$$

A straightforward algorithm can compute all elements in array b as follows:

```
[Straightforward row-wise OR algorithm]
for $i \leftarrow 0$ to $n-1$ do
  $b[i] \leftarrow 0$;
  for $j \leftarrow 0$ to $n-1$ do
    if $a[i][j] \neq 0$ then
      $b[i] \leftarrow 1$;
      exit for-loop;
```

The algorithm first reads $a[0][0]$. If it is 0 then it reads $a[0][1]$. Otherwise, it reads $a[1][0]$. Hence, this straightforward algorithm is not oblivious. On the other hand, we can modify it to be oblivious using sync instruction as follows:

```
[Row-wise OR algorithm with sync]
for $i \leftarrow 0$ to $n-1$ do
  $b[i] \leftarrow 0$;
  for $j \leftarrow 0$ to $n-1$ do
    if $a[i][j] \neq 0$ then
      $b[i] \leftarrow 1$;
      exit for-loop;
    else NOP;
  sync;
```

Clearly, this algorithm performs n sync instructions. Hence, its execution is partitioned in n sub-executions, each of which computes the value of each $b[i]$. In each i-th sub-execution, $a[i][0], a[i][1], \ldots$ are read until the value is non-zero. Thus, the j-th iteration of i-th sub-execution reads $a[i][j]$ or does not perform read operation, and so this algorithm is oblivious.

Let us evaluate bulk execution of this sequential algorithm with synchronization. Suppose that p threads execute this algorithm on the UMM independently, that is, each of p threads executes this algorithm in parallel, In the worst case, all values are zero and all elements are read. Hence, each thread reads n^2 elements and the total running time of the bulk execution on the UMM is $(\frac{p}{w} + l - 1) \cdot n^2 = O(\frac{pn^2}{w} + ln^2)$. If all elements $a[i][0]$ $(0 \leq i \leq n-1)$ are non-zero, only these elements are read and each thread reads n elements. If this is the case, the total running time is $(\frac{p}{w} + l - 1) \cdot n = O(\frac{pn}{w} + ln)$.

We assume that each element of matrix is a d-bit unsigned integer and the value is selected from $[0, 2^d - 1]$ uniformly at random. We will show that the total running time is expected $O(\frac{pn}{w} + ln)$ under this assumption. Recall that a warp of w threads perform memory access at the same time. First, each of w threads in the first warp reads $a[0][0]$ of the input. Since it is zero with probability $\frac{1}{2^d}$, at least one of all w threads reads zero values with probability at most $\frac{w}{2^d}$. This probability is at most $\frac{1}{2}$ if $w \leq 2^{d+1}$. Since the number of threads in a warp of CUDA-enabled GPUs is 32, and 32-bit integers are used, it makes sense to set $w = d = 32$. Hence, from practical point of view, this condition is satisfied. If this is the case, one or more threads read $a[0][1]$ with probability at most $\frac{1}{2}$. Further, we can say that some threads read $a[0][2]$ with probability at most $\frac{1}{2^2}$. In general, $a[0][j]$ is read with probability at most $\frac{1}{2^j}$. Thus, a warp performs j reading rounds with probability at most $\frac{1}{2^j}$, and so, the expected number of reading rounds is at most

$$\sum_{j=0}^{n-1} \frac{j}{2^j} = O(1).$$

Hence, each thread performs expected $O(1)$ memory access operations for each sub-execution and expected $O(n)$ memory access operations for all n sub-executions. Since the algorithm is oblivious, the total running time is $O(\frac{pn}{w} + ln)$. Consequently, we have

Lemma 5. *The bulk execution of the row-wise OR algorithm with sync for an $n \times n$ matrix, each element of which is a d-bit unsigned integer and the value is selected from $[0, 2^d - 1]$ uniformly at random, using p threads on the UMM runs in $O(\frac{pn^2}{w} + ln^2)$ time units and in expected $O(\frac{pn}{w} + ln)$ time units.*

7 Semi-oblivious implementation of the Approximate Euclidean algorithm

This section shows that the Approximate Euclidean algorithm can be implemented as an oblivious sequential algorithm.

We assume that all numbers are stored in d-bit words. Hence, a number with s bits is stored in $\frac{s}{d}$ words. For example, a 512-bit number is stored in sixteen 32-bit words. Since the Approximate Euclidean algorithm operates large numbers stored in multiple words, naive implementations perform a lot of redundant memory access operations. We will show how we implement fundamental operations used in the Binary Euclidean algorithm, the Fast Binary Euclidean algorithm, and the Approximate Euclidean algorithm. We will show that, with high probability, $3\frac{s}{d} + O(1)$ memory access operations are performed in each iteration if X and Y with s bits are stored in d-bit words. More specifically, each iteration essentially performs three operations, reading from X, reading from Y, and writing in X, each of which involves $\frac{s}{d}$ memory access operations. Also, additional $O(1)$ reading operations are performed for X and Y.

Figure 5 illustrates how X and Y are implemented. Two s-bit numbers X and Y are stored in arrays of $\frac{s}{d}$ words. Two registers are used to store pointers that specify arrays for X and Y. Also, the values of l_X, l_Y, α, and β are stored in registers.

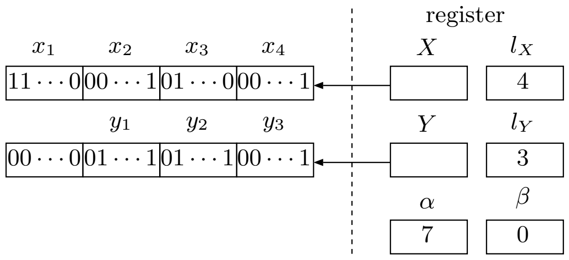

Figure 5: Implementation of X and Y

We will show that, the Approximate Euclidean algorithm can be implemented as a semi-oblivious algorithm with sync. For this purpose we show how each statement of the Approximate Euclidean algorithm can be implemented. We assume that synchronization is executed after each statement.

`approx`(X, Y): The value of `approx`(X, Y) can be determined by those of l_X, l_Y, x_1, x_2, y_1, and y_2. Hence, `approx`(X, Y) accesses at most four words x_1, x_2, y_1 and y_2 in the memory. Since addresses of these four words may change, the computation of `approx`(X, Y) is not oblivious.

$X \leftarrow$`rshift`$(X - Y \cdot \alpha)$: This operation can also be done by reading words of X and Y and writing words of X from the least significant word. For example, this operation for X with four 32-bit words x_1, x_2, x_3, x_4 and Y for three 32-bit words y_1, y_2, y_3 as illustrated in Figure 5 can be performed using a 64-bit temporary register variable z and a 16-bit temporary register variable r as follows:

```
$z \leftarrow x_4 + (x_3 << 32) - y_3 \cdot \alpha$
$r \leftarrow$ the number of consecutive 0 bits in $z$ from the LSB
$x_4 \leftarrow (z >> r)\&0xFFFFFFFF$
$z \leftarrow (z >> 32) + (x_2 << 32) - y_2 \cdot \alpha$
$x_3 \leftarrow (z >> r)\&0xFFFFFFFF$
$z \leftarrow (z >> 32) + (x_1 << 32) - y_1 \cdot \alpha$
$x_2 \leftarrow (z >> r)\&0xFFFFFFFF$
$x_1 \leftarrow z >> (r + 32)$
```

Clearly, each word in X and Y is read once, each word in X is written once. Note that this algorithm works only if $r \leq 32$. The reader should have no difficulty to modify this algorithm that works correctly even if $r > 32$. Since the memory access are performed from the LSB of X and Y, it is oblivious.

$X \leftarrow$`rshift`$(X - Y \cdot \alpha \cdot D^\beta + Y)$: This can be done in a similar way to "$X \leftarrow$`rshift`$(X - Y \cdot \alpha)$". Note that we need to perform additional reading operations from Y to compute "$+Y$."

$X < Y$: If $l_X < l_Y$ then $X < Y$ is true and if $l_X > l_Y$ then $X < Y$ is false. Thus, access to the memory is not necessary if $l_X \neq l_Y$. If $l_X = l_Y$ then we need to compare the values of X and Y from the most significant word. More specifically, x_1 and y_1 are read from the memory. If $x_1 < y_1$ then $X < Y$ is true and if $x_1 > y_1$ then $X < Y$ is false. If $x_1 = y_1$ then x_2 and y_2 are read from the memory and they are compared in the same way. If x_2 and y_2 takes 32-bit random values, then $x_2 \neq y_2$ with probability $1 - 2^{-32}$. Hence, the result of $X < Y$ can be determined without reading x_3 and y_3 with very high probability. If this is the case, only four words x_1, x_2, y_1 and y_2 in the memory are accessed. Consequently, this condition can be determined by accessing these four words with probability $1 - 2^{-32}$. Also, by a similar analysis as Lemma 5, we can prove that expected $O(1)$ memory access operations are performed. Since the position of each of these four words may change, the memory access may not be oblivious.

Table 6: The performance of the Euclidean algorithms, (C) the Binary Euclidean algorithm, (D) the Fast Binary Euclidean algorithm, (E) the Approximate Euclidean algorithm, and (F) the Approximate Euclidean algorithm with PTX: one GCD computing time in microseconds

		1024	2048	4096	8192	16384
CPU	(C) Binary Euclidean algorithm	82.0	282	1050	3990	15500
	(D) Fast Binary Euclidean algorithm	49.9	166	607	2330	8800
	(E) Approximate Euclidean algorithm	43.7	140	494	1830	7090
GPU	(C) Binary Euclidean algorithm	5.34	23.2	90.2	400	1680
	(D) Fast Binary Euclidean algorithm	1.02	4.13	15.7	64.5	257
	(E) Approximate Euclidean algorithm	0.530	2.21	8.50	34.8	138
	(F) Approximate Euclidean algorithm with PTX	0.482	1.96	7.85	30.5	120
$\frac{\text{CPU}}{\text{GPU}}$	(C) Binary Euclidean algorithm	15.3	12.2	11.6	9.98	9.23
	(D) Fast Binary Euclidean algorithm	48.8	40.3	38.6	36.1	34.2
	(E) Approximate Euclidean algorithm	82.5	63.3	58.2	52.6	51.2
	(F) Approximate Euclidean algorithm with PTX	90.6	71.6	63.0	60.1	59.1

swap(X, Y): This can be done by exchanging the pointer variables for X and Y. Hence, swap(X, Y) can be done by access to registers.

We can see that $X \leftarrow$rshift$(X - Y \cdot \alpha)$ can be done in $3\frac{s}{d}$ memory access operations, reading from Y, reading from X, and writing in X. Similarly, $X \leftarrow$rshift$(X - Y \cdot \alpha \cdot D^{\beta} + Y)$ can be done in $4\frac{s}{d}$ memory access operations, because additional $\frac{s}{d}$ reading operations are necessary to compute "$+Y$." The other operations performs at most $O(1)$ non-oblivious memory access with very high probability. Hence, the Approximate Euclidean algorithm is semi-oblivious with parameter $\frac{d}{s}$ with high probability. Thus, if $\frac{d}{s} \leq \frac{1}{w}$, that is, $s \geq wd$, then the Approximate Euclidean algorithm runs on the UMM efficiently. If we use 32-bit unsigned integers on CUDA-enabled GPUs, then $w = d = 32$. Thus, this condition is satisfied if $s \geq 1024$.

8 Experimental results

This section shows the running time of the Euclidean algorithms. We have used Xeon X7460 (2.66GHz) CPU for executing the sequential Euclidean algorithms and GeForce GTX Titan X GPU for evaluating the CUDA implementations. In our CUDA implementations, we have used CUDA blocks with 64 threads in which each thread computes GCDs of a pair of two large numbers. We have used local memory arranged in the global memory to store X and Y.

Table 7 shows the time for computing one GCD in microseconds when pairs of two randomly generated s-bit unsigned odd integers for $s = 1024, 2048, 4096, 8192$, and 16384. We have generated integers with 2G bytes totally for each s, and evaluated the running time on the GPU. For example, when $s = 1024$, we have generated 8 Mega pairs of 1024-bit integers. Basically, we use 32-bit unsigned integers to store large unsigned integers. When we compute the GCD using PTX instructions, we use 64-bit unsigned integers to compute rshift$(X - Y \cdot \alpha)$ and rshift$(X - Y \cdot \alpha \cdot D^{\beta} + Y)$.

From the table, we can see that the Approximate Euclidean algorithm is faster than the others. Since the Euclidean algorithms are semi-oblivious, the speedup ratio CPU/GPU is enough large. However, the execution time ratio CPU/GPU of the Binary Euclidean algorithm is rather smaller than the others. This is due to the branch divergence of a CUDA C program for the Binary Euclidean algorithm. Since CUDA architecture is based on SIMT (Single Instruction Multiple Threads), all threads in a warp must execute the same instruction in each clock cycle. Hence, if CUDA C program has a branch using an if-else statement, then the instructions for the true case are executed first and then those for the false case are executed. Note that, if all threads execute the instructions for the same case, those for the other case are not executed. The Binary Euclidean algorithm has a if-else if-else statement to select one of the three cases: (X, Y) is (even, odd), (odd, even), and (odd, odd). Since the instructions for these three cases are executed sequentially, and the branch divergence degenerates the performance of the Binary Euclidean algorithm. On the other hand, we can ignore the branch divergence of the Approximate Euclidean algorithm. The Approximate Euclidean algorithm has if-else statement to select two cases: $\beta = 0$ or $\beta > 0$, where β is the value computed by function `approx`. However, $\beta > 0$ with probability less than 10^{-8} from Table 5. Hence all threads executes instructions for the case of $\beta = 0$ with very high probability, and those for $\beta > 0$ are not executed. Further, the 64-bit division operation for function `approx` and 32-bit multiplications for `rshift`$(X - Y \cdot \alpha)$ takes a lot of time on the CPU. On the other hand, on the GPU, time for these operations are hidden by large memory access latency. Hence, the GPU implementation for the Approximate Euclidean algorithm achieves much higher speedup ratio over the CPU. Also, we can see that the GCD computation can be about 10% faster if we use 64-bit PTX instructions.

We have also evaluated the difference of the performance between oblivious and semi-oblivious executions for the GCD computation. To evaluate the oblivious execution, we have executed the GPU implementation such that every thread performs the identical GCD computation using common two numbers instead of the input used in Table 6. Namely, all instructions and addresses of memory access executed at the same time within a warp are identical. Table 7 shows the running time of the Approximate Euclidean algorithm with PTX for oblivious and semi-oblivious executions. The oblivious execution runs at most 31% faster than the semi-oblivious one.

Table 7: The performance of the Approximate Euclidean algorithm with PTX for semi-oblivious and oblivious executions on the GPU: one GCD computing time in microseconds

	1024	2048	4096	8192	16384
semi-oblivious	0.482	1.96	7.85	30.5	120
oblivious	0.369	1.67	6.58	26.2	103
$\frac{\text{semi-oblivious}}{\text{oblivious}}$	1.31	1.17	1.19	1.16	1.17

9 Conclusion

We have presented a new Euclidean algorithm for computing the GCD of all pairs of encryption moduli. The idea of our new Euclidean algorithm that we call the Approximate Euclidean algorithm is to compute an approximation of quotient by just one 64-bit division and to use it for reducing the number of iterations of the Euclidean algorithm. We also present an implementation of the Approximate Euclidean algorithm optimized for CUDA-enabled GPUs. The experimental results show that our implementation for 1024-bit GCD on GeForce GTX Titan X runs about 90 times faster than the Intel Xeon CPU implementation.

References

[1] Daniel J. Bernstein. Fast multiplication and its applications. *Algorithmic Number Theory*, 44:325–384, 2008.

[2] Noriyuki Fujimoto. High throughput multiple-precision GCD on the CUDA architecture. In *Proc. of International Symposium on Signal Processing and Information Technology*, pages 507–512, Dec. 2009.

[3] Toru Fujita, Koji Nakano, and Yasuaki Ito. Bulk GCD computation using a GPU to break weak RSA keys. In *Proc. of International Parallel and Distributed Processing Symposium Workshops*, pages 385–394, May 2015.

[4] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proc. of the 21st USENIX Security Symposium*, page 35, Aug. 2012.

[5] Wen-mei W. Hwu. *GPU Computing Gems Emerald Edition.* Morgan Kaufmann, 2011.

[6] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito. Offline permutation algorithms on the discrete memory machine with performance evaluation on the GPU. *IEICE Transactions on Information and Systems*, Vol. E96-D(12):2617–2625, Dec. 2013.

[7] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito. An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation. In *Proc. of International Conference on Parallel Processing (ICPP)*, pages 1–10, Oct. 2013.

[8] Donald Ervin Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, 1997.

[9] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064, 2012.

[10] Duhu Man, Kenji Uda, Hironobu Ueyama, Yasuaki Ito, and Koji Nakano. Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs. *International Journal of Networking and Computing*, 1(2):260–276, July 2011.

[11] Koji Nakano. Optimal parallel algorithms for computing the sum, the prefix-sums, and the summed area table on the memory machine models. *IEICE Trans. on Information and Systems*, E96-D(12):2626–2634, 2013.

[12] Koji Nakano. Sequential memory access on the unified memory machine with application to the dynamic programming. In *Proc. of International Symposium on Computing and Networking*, pages 85–94, Dec. 2013.

[13] Koji Nakano. Simple memory machine models for GPUs. *International Journal of Parallel, Emergent and Distributed Systems*, 29(1):17–37, 2014.

[14] NVIDIA Corporation. NVIDIA CUDA C best practice guide version 3.1, 2010.

[15] NVIDIA Corporation. NVIDIA CUDA C programming guide version 7.0, Mar 2015.

[16] NVIDIA Corporation. Parallel thread execution ISA ver4.2, Mar 2015.

[17] Kohei Ogawa, Yasuaki Ito, and Koji Nakano. Efficient Canny edge detection using a GPU. In *Proc. of International Conference on Networking and Computing*, pages 279–280. IEEE CS Press, Nov. 2010.

[18] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120 – 126, 1978.

[19] Kerry Scharfglass, Darrin Weng, Joseph White, and Christopher Lupo. Breaking weak 1024-bit RSA keys with CUDA. In *Proc. of Internatinal Conference of Breaking weak 1024-bit RSA keys with CUDA*, pages 207 – 212, Dec. 2012.

[20] Josef Stein. Computational problems associated with racah algebra. *Journal of Computational Physics*, 1(3), Feb. 1967.

[21] Daisuke Takafuji, Koji Nakano, and Yasuaki Ito. A CUDA C program generator for bulk execution of a sequential algorithm. In *Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, pages 178–191, Aug. 2014.

[22] Yuji Takeuchi, Daisuke Takafuji, Yasuaki Ito, and Koji Nakano. Ascii art generation using the local exhaustive search on the GPU. In *Proc. of International Symposium on Computing and Networking*, pages 194–200, Dec. 2013.

[23] Kazuya Tani, Daisuke Takafuji, Koji Nakano, and Yasuaki Ito. Bulk execution of oblivious algorithms on the unified memory machine, with GPU implementation. In *Proc. of International Parallel and Distributed Processing Symposium Workshops*, pages 586–595, May 2014.

[24] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. Fast and accurate template matching using pixel rearrangement on the GPU. In *Proc. of International Conference on Networking and Computing*, pages 153–159. IEEE CS Press, Dec. 2011.

[25] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. An efficient GPU implementation of ant colony optimization for the traveling salesman problem. In *Proc. of International Conference on Networking and Computing*, pages 94–102. IEEE CS Press, Dec. 2012.

[26] Joseph R. White. *PARIS: A PARALLEL RSA-PRIME INSPECTION TOOL.* PhD thesis, California Polytechnic State University - San Luis Obispo, June 2013.