Efficient GPU Implementations for Bulk Computations

(バルク計算のための効率的な GPU 実装手法)

by

Toru Fujita

A dissertation submitted
in partial fulfillmen of the requirements for the degree of
Doctor of Engineering
in Information Engineering

Under Supervision of
Professor Koji Nakano

Department of Information Engineering,
Graduate School of Engineering,
Hiroshima University

March, 2017

Summary

A GPU(Graphics Processing Unit) is a specialized circuit designed for manipulating images and video encoding. 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. The bulk computation is to perform the same algorithm for a lot of instances. The bulk computation has many applications and can be implemented in the GPU very efficiently. This dissertation shows efficient GPU implementations of bulk computations.

First, we implement the bulk execution of Euclidean algorithms computing the Greatest Common Divisor (GCD) of two large numbers in a GPU. We 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. The experimental results show that our parallel implementation of the approximate Euclidean algorithm for 1024-bit integer running on GeForce GTX TITAN X GPU is 90 times faster than the Intel Xeon CPU implementation.

Second, we present Bitwise Parallel Bulk Computation (BPBC) technique for accelerating the bulk computation. The idea of the BPBC technique is to simulate a combinational logic circuit using bitwise logic operations. Bitwise logic operations compute logical OR, AND, NOT, and XOR operations for the individual bits of 32-bit word. In this technique, we store 32 inputs for the combinational logic circuit into a particular bit of the words. And we apply the bitwise logic operations corresponding to the combinational logic circuit to the words. Thus, we can compute 32 circuits for 32 inputs at the same time. We apply the BPBC technique to two computations, the simulation of

Conway's Game of Life and the CKY parsing.

Conway's Game of Life is the most well-known cellular automaton. The universe of the Game of Life is a 2-dimensional array of cells, which takes two possible states, alive or dead. The state of every cell is repeatedly updated according to those of eight neighbors. A cell will be alive if exactly three neighbors are alive, or it is alive and two or three neighbors are alive. This dissertation shows several acceleration techniques for simulating the Game of Life using a GPU as follows: (1) the states of 32/64 cells are stored in 32/64-bit words (integers) and the next states are computed by the BPBC technique, (2) the states of cells stored in 2 words are updated at the same time by a thread, (3) warp shuffle instruction is used to directly transfer the current states stored in registers, and (4) multiple-step simulation is performed to reduce the overhead of data transfer and invoking CUDA kernel. The experimental results show that our GPU implementation using GeForce GTX TITAN X performs 1350×10^9 updates per second for 16K-step simulation of 512K $\times$ 512K cells stored in the SSD. Since Intel Core i7 CPU using the same technique performs 13.4×10^9 updates per second, our GPU implementation for the Game of Life achieves a speedup factor of 100.

The CKY parsing determines whether a context-free grammar derives an input string. The idea of the CKY parsing is to compute a 2-dimentional table called CKY table by the dynamic programming technique. The CKY parsing can be done by iterative simulation of the combinational logic circuit. In this dissertation, we show that the CKY parsing can be implemented in the GPU efficiently using the BPBC technique. The experimental results using Intel Core i7 CPU and GeForce GTX TITAN X GPU show that the GPU implementation for the CKY parsing runs more than 400 times faster than the CPU implementation.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

The GPU (Graphics Processing Unit) is designed for accelerating to generate and manipulate images [23, 31, 36]. Latest GPUs are used for general purpose computing such as cryptography, natural language processing and so on. NVIDIA provides CUDA (Compute Unifie Device Architecture) [12, 14], the computing engine for NVIDIA GPUs. CUDA provides a parallel computing platform and programming model. The GPU consists of multiple Streaming Multiprocessors (SMs) and the global memory. Each SM has hundreds of processing cores and the shared memory. Thus, there are a lot of processing cores and multiple types of memories in the GPU, GPU can accelerate the computation traditionally handled by the CPU.

The computation of bulk execution is to perform the same algorithm for a lot of instances in turn on the sequential machine such as CPUs or in parallel on the parallel machine such as GPUs. Bulk computation has many applications such as the FFT algorithm.

1.2 Contributions

In this dissertation, we present the following three GPU implementations of bulk computations.

1.2.1 GCD computation for large numbers

It is well known that the Euclidean algorithm [28] 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 [42], 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.

In this dissertation, we 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.

Previous works [43, 44] 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 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 find i such that $b[i] \neq 0$, it terminates and $b[i + 1], b[i + 2]$,... are not accessed. If it accesses $b[n-1]$ and find $b[n-1] = 0$, 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.

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 is independent of the input. 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 [39]. 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 modulo 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 collected from the Web, we can fin weak RSA keys if exist. Actually, several public keys collected from the Web includes weak RSA keys [30]. Several previously published papers have presented GPU implementations of the binary Euclidean algorithm for breaking weak RSA keys in CUDA-enabled GPU. Fujimoto [19] has implemented the binary Euclidean algorithm using CUDA and evaluated the performance on GeForce GTX 285 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. [41] 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 [47] has showed that the same computation can be performed in 631.417 seconds on Tesla K20Xm. It follows that it computes each 1024-bit GCD in 3.15 microseconds.

On the other hand, it has been presented [22] that a sequential algorithm can fin 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 [4]. 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 significan in the area of GPU computation. The efficient sequential algorithm uses very complicated remainder tree technique and fast modulo computation, and it just find 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.

1.2.2 The simulation of Conway's Game of Life

Conway's Game of Life was created by John Horton Conway, a mathematician at Gonville and Caius College of the University of Cambridge [1, 20]. The universe of the Game of Life is an 2-dimensional array of cells, each of which takes one of two states, 1 (alive) and 0 (dead). The state of every cell is updated by the current states of the eight neighbors as follows: The next state of a cell is alive if and only if it has three alive neighbors, or if it is alive and has two alive neighbors. Figure 1.1 shows an example of one-step simulation of the Game of Life. For simplicity, we assume that the universe of the Game of Life is square and wrapped around to handle the boundary cells.

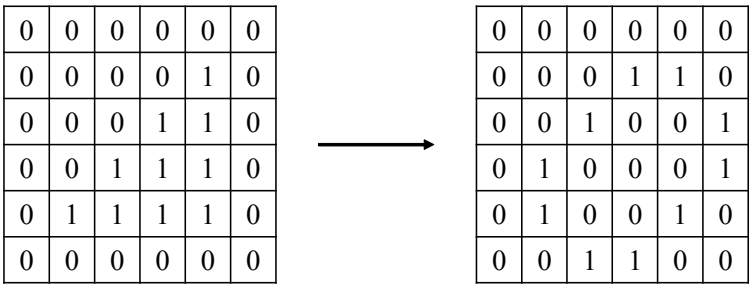


Figure 1.1: One-step simulation of the Game of Life

We are interested in how we can accelerate the simulation of the Game of Life using CUDA-enabled GPUs. Sometimes, simulation of the Game of Life means that the states of all cells of every step is output to a fil or a computer display. However, in such simulation, the overhead for output of cells is much larger than that for computation of cells. Since we are interested in computation of states of cells, we ignore the overhead for output cells. More specificall , we focus on accelerating simulation that computes the values of all cells in the universe after T steps for a given T.

We develop several acceleration techniques for simulating the Game of Life using a GPU as follows.

1. the states of 32/64 cells are stored in 32/64-bit words (integers) and the next states are computed by the BPBC technique,
2. the states of cells stored in 2 words are updated at the same time by a thread,
3. warp shuffle instruction is used to directly transfer the current states stored in registers, and
4. multiple-step simulation is performed to reduce the overhead of data transfer and invoking CUDA kernel.

It is easy to write a program for simulating the Game of Life if the state of a cell is stored in a word of data such as an 8-bit character or a 32-bit integer. However, for accelerating the simulation, it makes sense to use bit-per-cell arrangement [18] in which the state of a cell is stored as a bit of a word. For example, a 32-bit integer is used to store the states of consecutive 32 cells. A very sophisticated way to compute the next states of cells stored in a word by bitwise operations has been presented [45]. Also, the simulation of the Game of Life can be done by stencil cores, a class of kernels updating elements in an array according to some fi ed pattern, called stencil. Hence, it is easy to implement the simulation using a framework of stencil computation. For example, it can be implemented on GPUs with few codes using stencil operations of MATLAB [33].

As far as we know, there is no published technical paper aiming to accelerate the simulation. Very few papers presented GPU implementations of the simulation [3, 38], but their implementations are straightforward and did not aim to accelerate the simulation. On the other hand, there are a lot of web sites that present GPU implementations

of the Game of Life. For example, bitwise logical operations for the bit-per-cell arrangement are used to compute the next stats of cells [18]. Our implementations also use the bit-per-cell arrangement. Moreover, we developed a multiple-step simulation technique, which reduces memory access to the global memory. Also, we store the states of cells in registers of threads, and data transfer between registers is performed by a warp shuffle instruction. Using these techniques, we have obtained extremely fast GPU implementation for simulating the Game of Life using GPUs. For simulating the Game of Life with more than 1,000,000,000 cells, the best GPU implementation in [18] achieved 24.7×10^9 updates per second on GeForce GTX 480 GPU. We will show that our GPU implementation achieved 1990×10^9 updates per second on GeForce GTX TITAN X GPU. Hence, our implementation more than 80 times faster than the previously published implementation. GeForce GTX 480 and GTX TITAN X have 480 and 3072 processor cores running 1401 MHz and 1000 MHz respectively. Thus, our implementation is much more efficient even if the difference of computing power of different GPUs is taking into account. Further, we have implemented fast simulation of a very large universe stored in the SSD (Solid State Drive). The performance of our simulation on GeForce GTX TITAN X is 1350×10^9 updates per second for 16K-step simulation of 512K $\times$ 512K cells stored in the SSD. Since Intel Core i7 CPU performs 13.4×10^9 updates per second, our GPU implementation for the Game of Life with a very large universe achieves a speedup factor of 100.

1.2.3 The CKY parsing

Let $G = (N, \Sigma, P, S)$ denote a context-free grammar [32] such that N is a set of non-terminal symbols, Σ is a set of terminal symbols, P is a finit production rules, and

S ($\in N$) is the start symbol. Let $f(G, x)$ be a function such that G is a context-free grammar, $x = x_0x_1...x_{n-1}$ is a string of length n, and $f(G, x)$ returns a Boolean value. Function $f(G, x)$ returns TRUE if and only if G derives x. It is well-known that the CKY (Cocke-Kasami-Younger) parsing [2] computes $f(G, x)$ in $O(n^3)$ time, where n is the length of x. The idea of the CKY parsing is to compute a 2-dimensional table $T[i, j]$ called CKY table by the dynamic programming technique. Each element of $T[i, j]$ stores a subset of non-terminal symbols that can derive substring $x_ix_{i+1}...x_j$ by repeatedly applying production rules in P. Usually, each element of $T[i, j]$ is implemented as an array of size $|N|$ to maintain the subset of N. More specificall , $T[i, j][k] = 1$ if the k-th non-terminal symbol in N can derive substring $x_ix_{i+1}...x_j$ by applying production rules. In most implementations of the CKY parsing, the value of each $T[i, j][k]$ is stored in a word, such as a 32-bit integer. Since each $T[i, j][k]$ stores 1-bit Boolean value, it is inefficient to use an array of words to store $T[i, j]$. Our idea to apply the BPBC technique to the CKY parsing is to compute 32 CKY tables for 32 input strings at the same time. Suppose that 32 input strings are given and we want to perform the CKY parsing for each of them. Let $T_0, T_1, ..., T_{31}$ denote 32 CKY tables to be computed. We store 32 values $T_0[i, j][k], T_1[i, j][k], ..., T_{31}[i, j][k]$ in a 32-bit integer for each i, j, and k. The CKY parsing can be done by iterative simulation of combinational logic circuit [5, 6], the BPBC technique can be applied to it.

The parsing of context-free languages has many applications in various areas including natural language processing [9, 46], compiler construction [2], informatics [40], among others. Several studies have been devoted for accelerating the paring of context-free languages [8, 21, 29, 46]. It has been shown that parsing of a string of length n can be done in $O((\log n)^2)$ time using n^6 processors on the PRAM [21]. Also, using the

mesh-connected processor arrays, the parsing can be done in $O(n^2)$ time using n processors as well as in $O(n)$ time using n^2 processors [29]. Later in [8], an algorithm that runs on a systolic array with n^2 finite-stat processors with one-way communication running in linear time has been presented. In [24], it was shown that parsing can be accomplished on a one-way linear array of n^2 finite-stat processors in linear time. Since these parallel algorithms need at least n processors, they are unrealistic for large n. Ciressan et al. [10, 11] and Bordim et al. [5, 6] have presented hardwares for the CKY parsing for context-free grammars and have tested them using FPGAs. In [6], it has been shown that the CKY parsing with 64 non-terminal symbols and 8192 production rules can be done in 162 microseconds for an input string of length 32 using an APEX20K family FPGA. Our GPU implementation can do the same task in only 3.68 microseconds per one input string. Hence our implementation is more than 40 times faster than the FPGA implementation. Quite recently, GPU implementations of the CKY parsing have been presented [27, 49]. However, these implementations use the straightforward bottom-up process, which performs only one CKY parsing.

1.3 Dissertation Organization

This dissertation is organized as follows. In Chapter 2, we describe the GPU architecture and CUDA programming model. We show the bulk execution of sequential algorithms and the performance analysis in Chapter 3. In Chapter 4, we show the GPU implementation of Euclidean algorithms for large numbers. In Chapter 5, we describe Bitwise Parallel Bulk Computation (BPBC) technique for accelerating the bulk computation. We show the GPU implementation of the simulation of Conway's Game of Life using the BPBC technique in Chapter 6. In Chapter 7, we show the BPBC implementation of

the CKY parsing on the GPU. Finally, we conclude this dissertation in Chapter 8.

Chapter 2

GPUs and CUDA

This chapter describes the GPU architecture and the CUDA programming model. Figure 2.1 illustrates an architecture of CUDA-enabled GPUs. A GPU is a single-chip processor equipped with multiple Streaming Multiprocessors (SMs), each of which has processor cores, the shared memory and the register file The GPU processor is connected to an off-chip memory. For example, GeForce GTX TITAN X has 24 SMs with 128 processor cores, a 96K bytes shared memory, and a register fil with 64K 32-bit registers each. The off-chip memory can be accessed by all processor cores in all SMs. Also, registers in a register fil are assigned processor core. The off-chip memory is quite large, say 12G bytes, but the memory access latency is quite large, say several hundred clock cycles. The memory access latency of the shared memory is around 10 cycles [37] and that of registers in the register fil is smaller. Hence, to accelerate the computation, we should minimize the global memory access. We should also use registers whenever possible.

CUDA is a general purpose parallel computing platform and programming model introduced by NVIDIA in 2006. When we develop programs running on GPUs, we can

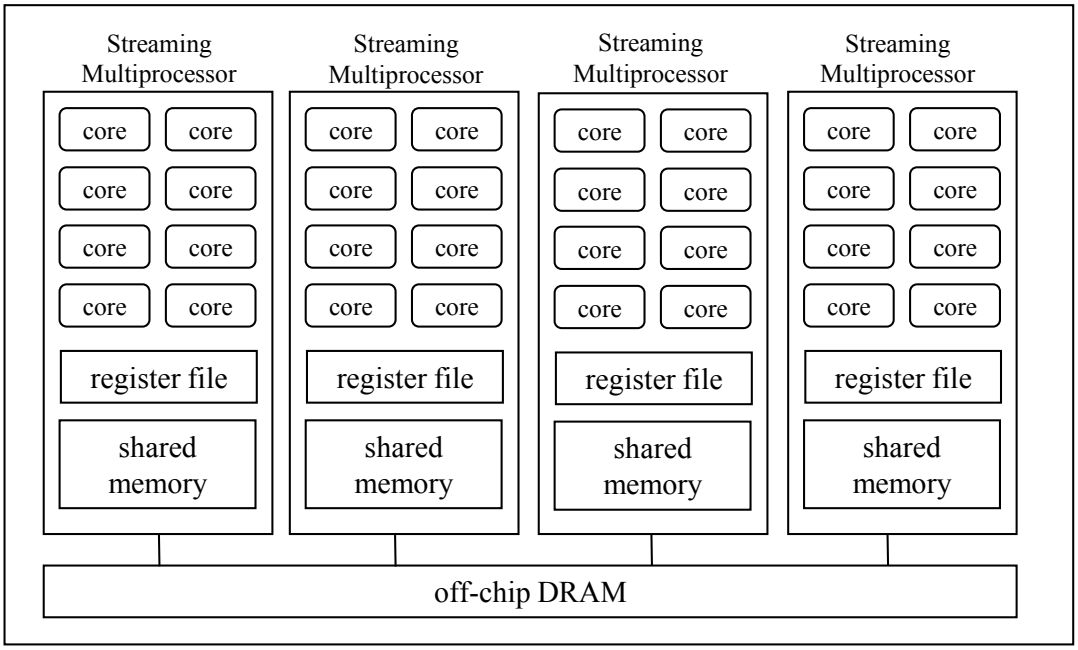


Figure 2.1: GPU architecture

use CUDA programming model illustrated in Figure 2.2 to support scalability. We assume that CUDA Compute Capability 5.2, which is available for GeForce GTX TITAN X [13]. Usually, a CUDA program executed on the host computer invokes CUDA kernels one or more times. A CUDA kernel executes one or more CUDA blocks running on SMs of the GPU. CUDA blocks in a CUDA kernel are identical in the sense that they have the same number of threads executing the same program. Each CUDA block can have up to 1024 threads, and is dispatched to one of the SM of the GPU. Since the number of CUDA blocks can be more than the number of SMs in a single GPU, they are dispatched to SMs in turn. Also, it is possible that two or more CUDA blocks are executed in a single SM at the same time. Each SM can handle up to 32 CUDA blocks with total of 2048 threads at the same time. Since each SM has 128 processor cores, at most 128 threads among them can be active and work in parallel. In other words, each

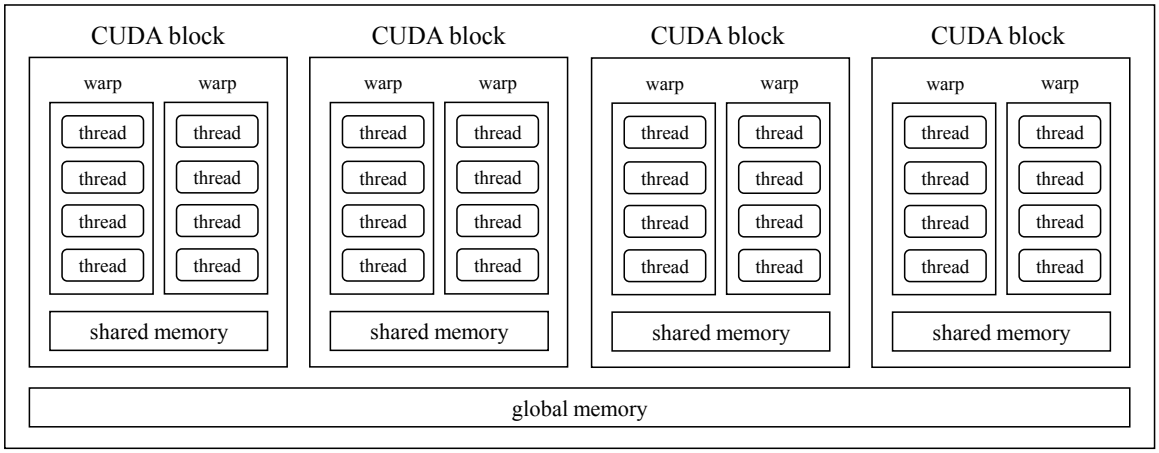


Figure 2.2: CUDA programming model

SM can have up to 2048 resident threads and 128 of them can be active on processor cores. A CUDA block can use the shared memory, which can be accessed by all threads in it. The shared memory of a CUDA block is implemented in the shared memory of the SM. Hence, its capacity is up to 96K bytes for CUDA Compute Capability 5.2 [14], and two or more CUDA blocks can be arranged in the SM at the same time only if the total shared memory capacity is no more 96K bytes. All threads in all CUDA blocks can access the global memory, which is arranged in the off-chip DRAM of the GPU. Note that after all threads in a CUDA block terminate, data stored in the shared memory are lost, because the shared memory in an SM may be used for another CUDA block. If data stored in the shared memory must be referred later, it must be copied to the global memory on developer's own responsibility.

Threads in a CUDA block are partitioned into groups of 32 threads each called warps. It is guaranteed that 32 threads in the same warp execute the same instruction at the same time. Hence, if a CUDA block has at most 32 threads, they are executed synchronously. However, threads in different warps may not be executed at the same time.

All threads in a CUDA block can call `__syncthreads()` for barrier synchronization if necessary. However, the cost of `__syncthreads()` is not negligibly small. Hence, it makes sense to use a CUDA block with 32 threads for avoiding barrier synchronization using `__syncthreads()`, if we need to synchronize all threads in a CUDA block frequently. Also, to synchronize all threads in all CUDA blocks, we need to use separate CUDA kernel calls, because SMs in the GPU execute CUDA blocks in turn. Since the synchronization of all CUDA blocks are very costly, we should minimize the number of such synchronization operations.

Efficient usage of the global memory and the shared memory is a key for CUDA developers to accelerate applications using GPUs. To maximize the throughput between the GPU and the off-chip memory, the consecutive addresses of the global memory must be accessed at the same time. For example, in Figure 2.3 (1), threads in a CUDA block access nonconsecutive addresses of the global memory. Since the global memory access for nonconsecutive addresses is serialized, the memory access efficiency is decreased. On the other hand, in Figure 2.3 (2), they access consecutive addresses of the global memory, and these addresses can be accessed at once. Hence, threads in a CUDA block should perform coalesced access when they access the global memory [12, 25].

Since the shared memory consists of 32 memory banks, memory access by 32 threads in a warp must be destined for distinct memory banks. In other words, bank conflict [12, 26, 37] by a warp should be avoided to maximize the shared memory access performance. For example, in Figure 2.4, we assume that the shared memory consists of 4 memory banks and a warp has 4 threads. In Figure 2.4 (1), thread 0 and thread 1 in a warp access the same memory bank. This memory access has a bank conflic and the access is serialized. On the other hand, in Figure 2.4 (2), all 4 threads in

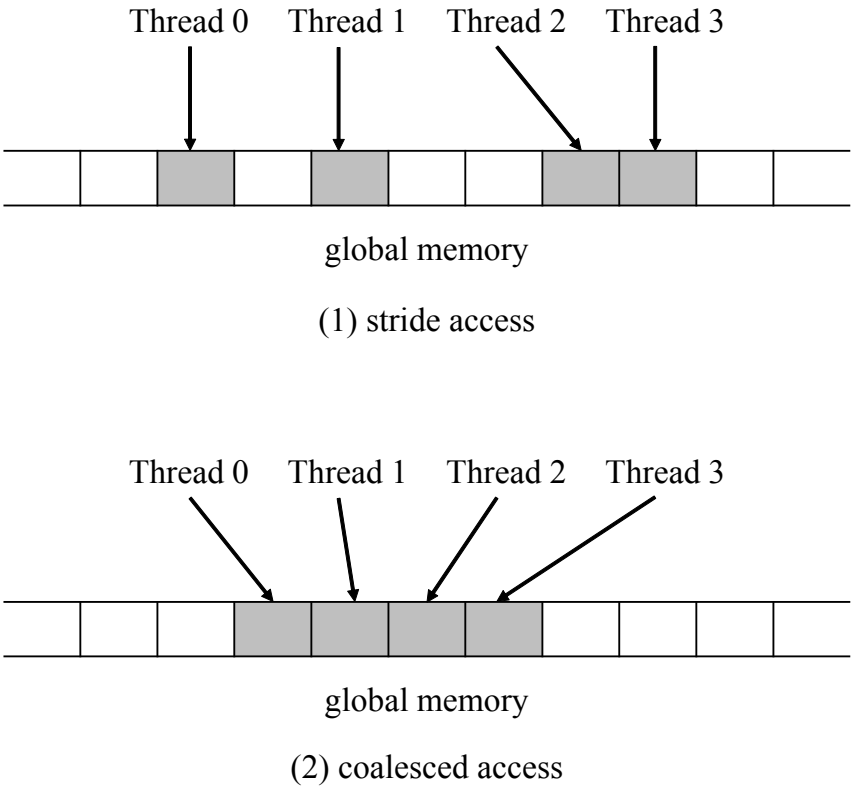
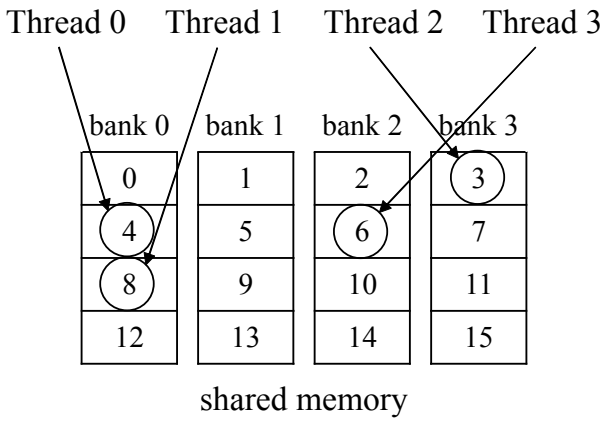


Figure 2.3: Stride access and coalesced access to the global memory

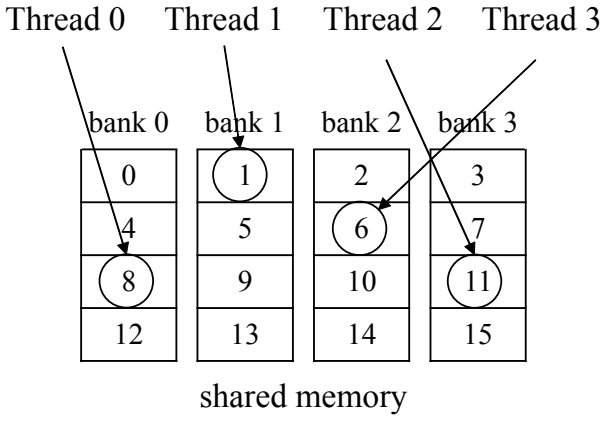
a warp access distinct memory banks. This access has no bank conflict and it can be performed at once.

The communication between threads can be done through the global memory or the shared memory. Note that the communication between threads in different CUDA blocks in the same CUDA kernel call is not possible, because CUDA blocks may be dispatched to SMs in an arbitrary order. What threads in a CUDA kernel can do is to send data to threads in the following CUDA kernel by reading/writing the global memory.

CUDA Compute Capability 3.0 and later support warp shuffle instructions that permit exchanging of data stored in registers in threads in a warp. The data exchange occurs



(1) conflict access



(2) conflict-free access

Figure 2.4: Conflic access and conflict-fre access to the shared memory

at the same time for all active threads in warp. For example, if `__shfl(`a, i`)` is executed by a CUDA block with a warp of 32 threads, the value of register a of thread i is returned. Since the data size for warp shuffle instructions must be 32 bits, two separate invocations are necessary to exchange 64-bit data. Warp shuffle instructions are more efficient than a conventional data exchanging method using write/read operations to the shared memory. Thus, we should use warp shuffle instructions whenever possible. Actually, appropriate use of warp shuffle instructions can accelerate the computation [7, 48].

Listing 2.1 and 2.2 show a standard C code and a parallel C code of SAXPY (Serial-precision Alpha X Plus Y) [17]. SAXPY is to compute $Y[i] = \alpha \cdot X[i] + Y[i]$ for each i ($0 \leq i \leq n$), where X and Y are one-dimensional arrays of length n, and α is a real value. In Listing 2.1, the computation of $Y[i]$ for each i is performed in turn. On the other hand, in Listing 2.2, the computation of them is performed in parallel. In Listing 2.2, `blockIdx.x` and `threadIdx.x` represent the block ID and the thread ID, respectively. `blockDim.x` represents the number of threads in a block. By computing `i = blockIdx.x*blockDim.x + threadIdx.x`, each thread obtains a unique ID over the GPU. The kernel function call `saxpy_serial<<<4096,256>>>` launches 4096 blocks, each of which has 256 threads. Hence, $4096 \cdot 256 = 1048576$ threads are launched in total, and all threads perform $Y[i] = \alpha \cdot X[i] + Y[i]$ for assigned i. Thus, we can perform the computation in parallel on the GPU.

Listing 2.1: Standard C Code of SAXPY

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

//Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

Listing 2.2: Parallel C Code of SAXPY

```
__global__
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

//Perform SAXPY on 1M elements
saxpy_serial<<<4096,256>>>(n, 2.0, x, y);
```

Chapter 3

Bulk execution of sequential algorithms and performance analysis

In this chapter, 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 [44] for the details. We further defin *semi-obliviousness* of sequential algorithms. Intuitively, a semi-oblivious sequential algorithm is not oblivious, but it is almost oblivious.

3.1 The obliviousness of sequential algorithms and the bulk execution

A sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input [44]. More specificall , there exists a function $a : \{0, 1, ..., t-1\} \rightarrow \mathcal{N}$, 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 ($0 \leq i \leq t-1$). In other words, at each time i ($0 \leq i \leq t-1$), 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 bulk execution of a sequential algorithm on a GPU, we firs defin the UMM (the Unifie Memory Machine) [34, 35] which captures the essence of the global memory access of CUDA-enabled GPUs. We then go on to show that the bulk execution of oblivious algorithms can be implemented very efficiently on the UMM.

Let us defin the UMM with width w and latency l. The memory of the UMM is partitioned into address groups $A[0], A[1], ...$, such that each $A[j]$ ($j \geq 0$) involves $j \cdot w, j \cdot w + 1, ..., (j+1) \cdot w - 1$. The reader should refer Figure 3.1 that illustrates the memory for $w = 4$. Also, the memory access is performed through l-stage pipeline registers as illustrated in the figure 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 specificall , p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), ..., W(\frac{p}{w} - 1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w + 1), ..., T((i+1) \cdot w - 1)\}$. Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory in the same time. More specificall , $W(0), W(1), ..., W(\frac{p}{w} - 1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not dispatched for memory access. When $W(i)$ is dispatched, w threads in $W(i)$ send memory access requests, one request per thread, to the memory banks.

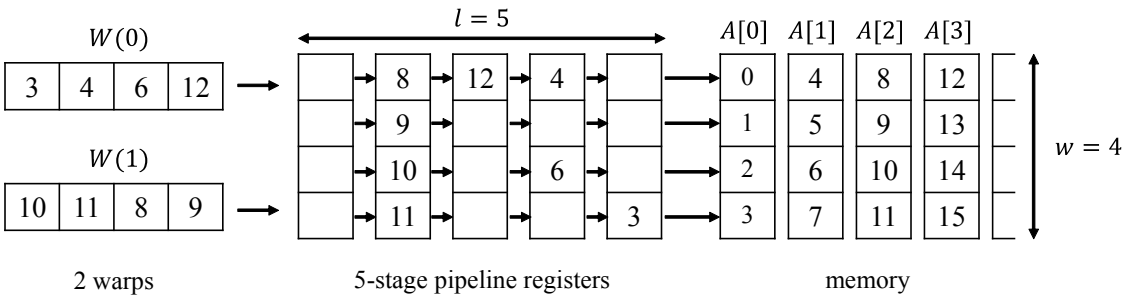


Figure 3.1: The UMM with width $w = 4$ and latency $l = 5$

For the memory access, each warp sends memory access requests to the memory 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 memory access requests by a warp are destined for k address groups, they occupy k stages. We assume that the memory access is completed as soon as the request is dequeued from the pipeline. Thus, all memory access requests by $W(0)$ and $W(1)$ in the figur are completed in 3(address group) + 1(address group) + 5(latency)−1 = 8 time units. We also assume that a thread 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

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

Figure 3.2: Column-wise arrangement for $p = 8$ arrays of size $n = 4$ each

column-wise arrangement to allocate p arrays as illustrated in Figure 3.2. More specifi cally, 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 $b_0[i], b_1[i], ..., b_{p-1}[i]$ at the same time. Clearly, they are arranged in addresses $i \cdot p + 0, i \cdot p + 1, ..., i \cdot p + (p - 1)$ 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 oblivious sequential algorithm, p threads perform 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 terminate in $(\frac{p}{w} + l - 1) \cdot t = O(\frac{pt}{w} + lt)$ time units. Thus we have,

Lemma 3.1.1 *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 [44], they have proved that Lemma 3.1.1 is time-optimal.

3.2 Semi-obliviousness of a sequential algorithm and the bulk execution

In this section, we defin 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 specificall , 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 oblivious 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 3.1.1, 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 3.2.1 *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 [44], they 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 [14], 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.

3.3 Oblivious sequential algorithms with synchronization

The main purpose of this section is to defin *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 can say that a sequential algorithm with synchronization is oblivious if every sub-execution is oblivious. Since the definitio 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* define 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 ← 0 to n − 1 do
  b[i] ← 0;
  for j ← 0 to n − 1 do
    if a[i][j] ≠ 0 then
      b[i] ← 1;
      exit for-loop;
```

The algorithm firs 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 ← 0 to n − 1 do
  b[i] ← 0;
  for j ← 0 to n − 1 do
    if a[i][j] ≠ 0 then
      b[i] ← 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, $a[i][0], a[i][1], ...$ 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 form $[0, 2^d - 1]$ 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 firs 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 integer 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 $a[0][2]$ 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 3.3.1 *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.*

Chapter 4

Bulk computation of Euclidean algorithms on the GPU

In this chapter, we describe several previous Euclidean algorithms and approximate Euclidean algorithm for computing the GCD. Also, we show the GPU implementation of bulk computation of approximate Euclidean algorithm and the experimental results.

4.1 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 GCD(X, Y) denote the GCD of X and Y. Euclidean algorithms for computing the GCD are based on the following fact:

Lemma 4.1.1 *For any integer $\alpha \geq 0$ such that $X > Y \cdot \alpha$, we have GCD(X, Y)=GCD($X - Y \cdot \alpha$, Y).*

Proof Suppose that GCD(X, Y) = 1, that is, X and Y are coprime. We will prove that

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 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 $X - Y \cdot \alpha = (x - y \cdot \alpha) \cdot g$, 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 `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 significan bit of X. The reader should have no difficulty to confir that the following lemma is correct.

Lemma 4.1.2 *For any even numbers X and Y, $GCD(X, Y) = 2 \cdot GCD(\frac{X}{2}, \frac{Y}{2})$ always holds. Also, for any odd number X and even number Y, $GCD(X, Y) = GCD(X, \frac{Y}{2}) = GCD(X,$* `rshift`*$(Y))$ always holds.*

For simplicity, we assume that both inputs X and Y are odd and $X \geq Y$ holds when we compute $GCD(X, Y)$. From Lemma 4.1.2, it should have no difficulty to modify all GCD algorithms shown in this dissertation to handle even input numbers. For later reference, let s denote the number of inputs bits of X and Y.

Let `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 ← X mod Y; // X < Y always holds
    swap(X, Y); // X > Y always holds
  } while(Y ≠ 0)
  return(X);
}
```

Let $\alpha = \lfloor \frac{X}{Y} \rfloor$. From $X \bmod Y = X - Y \cdot \lfloor \frac{X}{Y} \rfloor$ and Lemma 4.1.1, this algorithm returns the GCD correctly. We will show that the original Euclidean algorithm runs no more than $2s$ iterations of the do-while 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 ← X/2;
    else if(Y is even) Y ← Y/2;
    else X ← (X−Y)/2; // X − Y is always even
    if(X < Y) swap(X, Y);
  } while(Y ≠ 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-while loop of the binary Euclidean algorithm is also no more than $2s$.

Note that the binary Euclidean algorithm removes 0 in the least significan bit. We can reduce the number of iterations of the do-while 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 4.1.1 and 4.1.2, $GCD(X, Y) = GCD(X - Y, Y) = GCD(\texttt{rshift}(X - Y), Y)$ 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, a number 223 in the decimal system or 11011111 in the binary system is denoted by "223", "1101,1111", or "1101,1111(223)". Table 4.1 shows an example of computation performed by the binary Euclidean algorithm and the fast binary Euclidean algorithm for

$X = 1111, 1110, 1101, 1100, 1011(1043915)$ and,

$Y = 1011, 1011, 1011, 1011, 1011(768955)$.

Table 4.1: 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	0100, 0011, 0010, 0001	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	

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

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 ← X div Y;
    X ← X − Y · Q;
    swap(X, Y);
  } while(Y ≠ 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 ← X div Y;
    if(Q is even) Q ← Q − 1;
    X ← rshift(X − Y · Q);
    if(X < Y) swap(X, Y);
  } while(Y ≠ 0)
  return(X);
```

}

From Lemmas 4.1.1 and 4.1.2, $GCD(X, Y) = GCD(\texttt{rshift}(X - Y \cdot Q), Y)$ 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 4.2 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 4.1. 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.

Table 4.2: 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	5
	Y	0010, 1101		0000	
10	X	0010, 1101	4		
	Y	1010			
11	X	1010	2		
	Y	0101			
	X	0101			
	Y	0000			

4.2 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 the fast Euclidean algorithm is costly. Our new idea is to fin a good approximation

of quotient by small computing 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{
    (α, β) ← approx(X, Y);
    if(β = 0){
      if(α is even) α ← α − 1; // α is odd
      X ← rshift(X − Y · α); // Y · α is odd
    } else X ← rshift(X − Y · α · D^β + Y); // α · D^β is even
    if(X < Y) swap(X, Y);
  } while(Y ≠ 0)
  return(X);
}
```

From Lemma 4.1.1 and 4.1.2, $GCD(X, Y) = GCD(X - Y \cdot \alpha, Y) = GCD(\texttt{rshift}(X - Y \cdot \alpha \cdot D^{\beta} + Y), Y)$ holds, and thus, this algorithm is correct. In this algorithm,`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 `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 the binary Euclidean algorithms.

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

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

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

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 significan 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 $X \text{ div } Y = 45$. Using this idea, the following function `approx` computes a pair (α, β):

```
approx(X, Y){
  if(l_X ≤ 2)
    return (X div Y, 0); // Case 1
  if(l_Y = 1){
    if (x_1 ≥ y_1)
      return (x_1 div y_1, l_X − 1); // Case 2-A
    else
      return (⟨x_1x_2⟩ div y_1, l_X − 2); // Case 2-B
  }
  if(l_Y = 2){
    if (⟨x_1x_2⟩ ≥ ⟨y_1y_2⟩)
      return (⟨x_1x_2⟩ div ⟨y_1y_2⟩, l_X − 2); // Case 3-A
    else
      return (⟨x_1x_2⟩ div (y_1 + 1), l_X − 3); // Case 3-B
  }
  if(⟨x_1x_2⟩ > ⟨y_1y_2⟩)
    return (⟨x_1x_2⟩ div (⟨y_1y_2⟩ + 1), l_X − l_Y); // Case 4-A
  if(l_X > l_Y)
    return (⟨x_1x_2⟩ div (y_1 + 1), l_X − l_Y − 1); // Case 4-B
  return (1,0); // Case 4-C
}
```

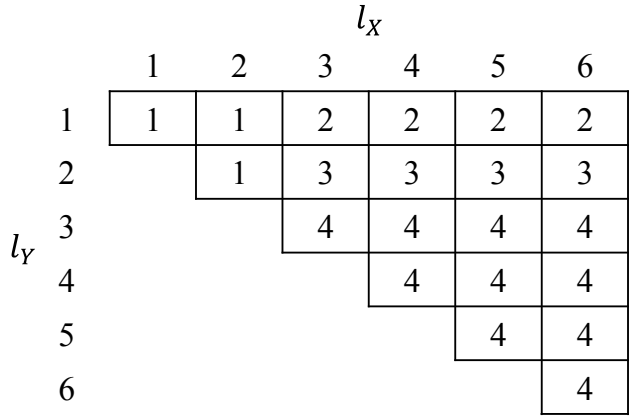


Figure 4.1: Cases for the values of l_X and l_Y

The reader should have no difficulty to confir 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 4.1. 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:

- **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 confir 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 satisfie and `approx` outputs $(77 \text{ div } 12, 3 - 2) = (6, 1)$. We can confir 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 word.

Case 3 has two sub-cases as follows:

- **Case 3-A:** If $\langle x_1 x_2 \rangle \geq \langle y_1 y_2 \rangle$ then `approx` outputs $(\langle x_1 x_2 \rangle \text{ div } \langle y_1 y_2 \rangle, l_X - l_Y)$.

  Example: If $X = 1001, 0010, 1001(2345)$ and $Y = 0011, 1011(59)$ then $\langle x_1 x_2 \rangle = 1001, 0010(146)$. Hence $\langle x_1 x_2 \rangle \geq \langle y_1 y_2 \rangle$ is satisfie and `approx` outputs $(146 \text{ div } 59, 3 - 2) = (2, 1)$. We can confir that $\alpha \cdot D^{\beta} = 2 \cdot 16^1 = 32$ approximates $X \text{ div } Y = 2345 \text{ div } 59 = 39$.

- **Case 3-B:** If $\langle x_1 x_2 \rangle < \langle y_1 y_2 \rangle$ then `approx` outputs $(\langle x_1 x_2 \rangle \text{ div } (y_1 + 1), l_X - 3)$.

  Example: If $X = 1001, 0010, 1001(2345)$ and $Y = 1110, 0111(231)$ then $\langle x_1 x_2 \rangle = 1001, 0010(146)$ and $y_1 = 1110(14)$. Since $\langle x_1 x_2 \rangle < \langle y_1 y_2 \rangle$ is satisfied `approx` outputs $(146 \text{ div } (14 + 1), 3 - 3) = (9, 0)$. We can confir 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 $\langle x_1 x_2 \rangle > \langle y_1 y_2 \rangle$ then `approx` outputs $(\langle x_1 x_2 \rangle \text{ div } (\langle y_1 y_2 \rangle + 1), l_X - l_Y)$. Note that, from $\langle x_1 x_2 \rangle > \langle y_1 y_2 \rangle$, we always have $\langle y_1 y_2 \rangle + 1 \leq D^2 - 1$. Hence $\langle y_1 y_2 \rangle + 1$ has at most $2d$ bits.

  Example: If $X = 1101, 0100, 0011, 0001(54321)$ and $Y = 0100, 1101, 0010(1234)$ then $\langle x_1 x_2 \rangle = 1101, 0100(212)$ and $\langle y_1 y_2 \rangle = 0100, 1101(77)$. Since $\langle x_1 x_2 \rangle > \langle y_1 y_2 \rangle$ is satisfied `approx` outputs $(212 \text{ div } (77 + 1), 4 - 3) = (2, 1)$. We can confir that $\alpha \cdot D^\beta = 2 \cdot 16^1 = 32$ approximates X div $Y = 54321$ div $1234 = 44$.

- **Case 4-B:** If $\langle x_1 x_2 \rangle \leq \langle y_1 y_2 \rangle$ and $l_X > l_Y$ then `approx` outputs $(\langle x_1 x_2 \rangle \text{ div } (y_1 + 1), l_X - l_Y - 1)$.

  Example: If $X = 1101, 0100, 0011, 0001(54321)$ and $Y = 1111, 1010, 0000(4000)$ then $\langle x_1 x_2 \rangle = 1101, 0100(212)$ and $\langle y_1 y_2 \rangle = 1111, 1010(250)$ hold. Hence, $\langle x_1 x_2 \rangle \leq \langle y_1 y_2 \rangle$ holds. Since $y_1 = 1111(15)$, `approx` outputs $(212 \text{ div } (15 + 1), 4 - 3 - 1) = (13, 0)$. We can confir that $\alpha \cdot D^\beta = 13 \cdot 16^0 = 13$ approximates X div $Y = 54321$ div $4000 = 13$.

- **Case 4-C:** If this is the case, $\langle x_1 x_2 \rangle \leq \langle y_1 y_2 \rangle$ and $l_X \leq l_Y$ hold. Recall that $X \geq Y$ and thus $\langle x_1 x_2 \rangle = \langle y_1 y_2 \rangle$ 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 4.3 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 4.1 and 4.2 in 9 steps. The values used to compute α in `approx` are underlined. We can confir that the approximate Euclidean algorithm outputs 0101(5), the GCD of X and Y correctly.

Table 4.3: An example of computation performed by the approximate Euclidean algorithm

		X & Y	CASE	(α, β)
1	X	1111 1110 1101 1100 1011	4-A	$(1, 0)$
	Y	1011 1011 1011 1011 1011		
2	X	1011 1011 1011 1011 1011	4-A	$(2, 1)$
	Y	0100 0011 0010 0001		
3	X	1110 0110 1010 1111	4-A	$(3, 0)$
	Y	0100 0011 0010 0001		
4	X	0100 0011 0010 0001	4-B	$(7, 0)$
	Y	0111 0101 0011		
5	X	0111 0101 0011	4-A	$(1, 0)$
	Y	0011 1111 0111		
6	X	0011 1111 0111	3-B	$(3, 0)$
	Y	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		

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 Table 4.2 and 4.3, we can see that the fast Euclidean algorithm and the approximate Euclidean algorithm performs 8 and 9 iterations, respectively, for the same input numbers.

Table 4.4 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 2 Gbytes totally and evaluated the number of iterations. For example, when $s = 1024$, we have generated 8 Mega pairs of 1024-bit odd integers in $[2^{1023}, 2^{1024})$.

Table 4.4: The average number of iterations performed by the Euclidean algorithm, (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

	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

Each iteration of (A) and (B) is costly, because they compute quotient/modulo of two s-bit numbers. On the other hand, (C) and (D) involves no division/multiplication operations. Further, each iteration of (E) involves one division of two 64-bit numbers, and $\frac{s}{32}$ repetitions of 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 iteration of (C), (D), and (E).

From Table 4.4, we can see that

1. the number of iterations is proportional to the number of input bits,
2. the number of iterations of (E) is about 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 that 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 4.4, we have recorded the value of β for each call of function `approx`. From the record, we have obtained Table 4.5, which shows the probability that $\beta > 0$. We can see that the probability is very small and it is very rare that $X \leftarrow \texttt{rshift}(X - Y \cdot \alpha \cdot D^{\beta} + Y)$ is executed.

Table 4.5: 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}

4.3 Further acceleration using PTX instructions

PTX is a low-level parallel thread execution virtual machine and instruction set architecture of GPU [15]. 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 $\texttt{rshift}(X - Y \cdot \alpha)$ and $\texttt{rshift}(X - Y \cdot \alpha \cdot D^{\beta} + Y)$.

Let $lo(X)$ and $hi(X)$ denote the least significan 64-bit and the most significan 64-bit unsigned integers of 128-bit unsigned integer X. More specificall , $lo(X) = x_{63}x_{62}\cdots x_0$ 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 PTX instructions to handle carry propagation:

setp.lo.s32.u64 *c*, *a*, *b*: $c \leftarrow (a < b)? -1 : 0$ 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 // $c \leftarrow (a < b)? - 1 : 0$

slct.u64.s32 $d, 0, 1, c$ // $d \leftarrow (c \geq 0)?0 : 1$

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 integer variables that constitute a 256-bit unsigned integer and α be a 64-bit unsigned integer variable. We can compute $Z = Y \cdot \alpha$, where $Z = z_0z_1z_2z_3z_4$ be f ve 64-bit unsigned integer variables, by the following PTX program.

(1)	mul.lo.u64	z_4, y_4, α	// $z_4 \leftarrow lo(y_4 \cdot \alpha)$
(2)	mul.hi.u64	c, y_4, α	// $c \leftarrow hi(y_4 \cdot \alpha)$
(3)	mad.lo.u64	z_3, y_3, α, c	// $z_3 \leftarrow lo(y_3 \cdot \alpha + c)$
(4)	setp.lo.s32.u64	t, z_3, c	// $t \leftarrow (z_3 < c)? - 1 : 0$
(4)	slct.u64.s32	$c, 0, 1, t$	// $c \leftarrow (t \geq 0)?0 : 1$
(5)	mad.hi.u64	c, y_3, α, c	// $c \leftarrow hi(y_3 \cdot \alpha + c \cdot 2^{64})$
(6)	mad.lo.u64	z_2, y_2, α, c	// $z_2 \leftarrow lo(y_2 \cdot \alpha + c)$
(7)	setp.lo.s32.u64	t, z_2, c	// $t \leftarrow (z_2 < c)? - 1 : 0$
(7)	slct.u64.s32	$c, 0, 1, t$	// $c \leftarrow (t \geq 0)?0 : 1$
(8)	mad.hi.u64	c, y_2, α, c	// $c \leftarrow hi(y_2 \cdot \alpha + c \cdot 2^{64})$
(9)	mad.lo.u64	z_1, y_1, α, c	// $z_1 \leftarrow lo(y_1 \cdot \alpha + c)$
(10)	setp.lo.s32.u64	t, z_1, c	// $t \leftarrow (z_1 < c)? - 1 : 0$
(10)	slct.u64.s32	$c, 0, 1, t$	// $c \leftarrow (t \geq 0)?0 : 1$
(11)	mad.hi.u64	z_0, y_1, α, c	// $z_0 \leftarrow hi(y_1 \cdot \alpha + c \cdot 2^{64})$

In this PTX program, c is a 64-bit unsigned integer variable to store the carry and t is a 32-bit signed integer variable. Figure 4.2 illustrates the $Z = Y \cdot \alpha$ by the PTX program. This PTX program computes the values of Z from the least significan 64-bit digit one by one. Although the computation of $X \leftarrow \texttt{rshift}(X - Y \cdot \alpha)$ and $X \leftarrow \texttt{rshift}(X - Y \cdot \alpha \cdot D^{\beta} + Y)$ are much more complicated, they can be done by a similar way using these PTX instructions.

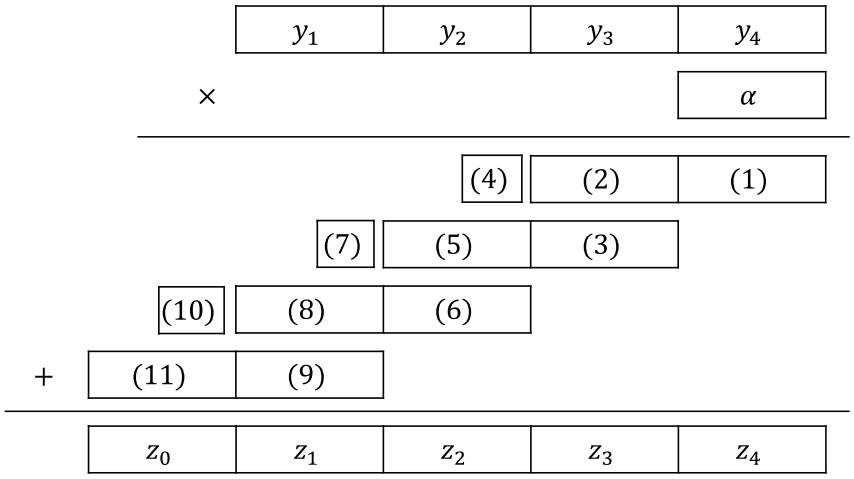


Figure 4.2: Illustrating the computation of $Z = Y \cdot \alpha$ using PTX instructions

4.4 Semi-oblivious implementation of the approximate Euclidean algorithm

This section shows that the approximate Euclidean algorithm can be implemented as a semi-oblivious sequential algorithm introduced in Section 3.2.

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 specificall , 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 4.3 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.

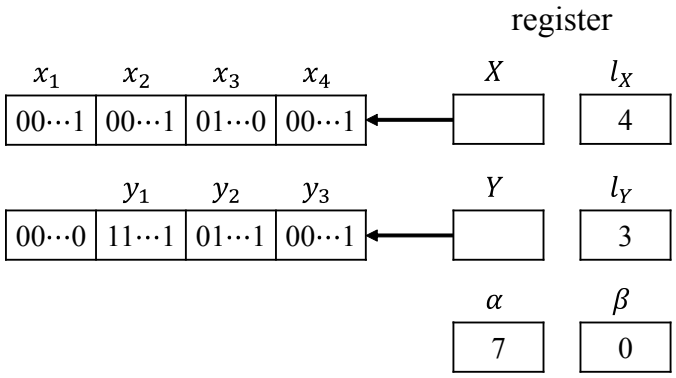


Figure 4.3: Implementation of X and Y

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 significan 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 4.3 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

$x_4 \leftarrow (z >> r)\&0xFFFFFFFF$

$z \leftarrow (z >> 32) + (x_2 << 32) - y_2 \cdot \alpha$

$x_3 \leftarrow (z >> r)\&0xFFFFFFFF$

$z \leftarrow (z >> 32) + (x_1 << 32) - y_1 \cdot \alpha$

$x_2 \leftarrow (z >> r)\&0xFFFFFFFF$

$x_1 \leftarrow z >> (r + 32)$

Clealy, 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 significan word. More specificall , 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 3.3.1, 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.

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 $X \leftarrow \texttt{rshift}(X - Y \cdot \alpha)$ can be done in $3\frac{s}{d}$ memory access operations, reading from Y, reading from X, and writing in X. Similarly, $X \leftarrow \texttt{rshift}(X - Y \cdot \alpha \cdot D^{\beta} + Y)$ 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 efficientl . If we use 32-bit unsigned integers on CUDA-enabled GPUs, then $w = d = 32$. Thus, this condition is satisfie if $s \geq 1024$.

4.5 Experimental results

This section shows the running time of the Euclidean algorithms. We have used Xeon X7460 (2.66 GHz) 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 4.6 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 2 Gbytes 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`$(X - Y \cdot \alpha \cdot D^{\beta} + Y)$.

Table 4.6: 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
$\frac{CPU}{GPU}$	(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

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 firs 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 an 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 4.5. Hence all threads execute 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 4.6. Namely, all instructions and addresses of memory access executed at the same time within a warp are identical. Table 4.7 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 4.7: 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

Chapter 5

Bitwise Parallel Bulk Computation technique

The main purpose of this chapter is to show the idea of Bitwise Parallel Bulk Computation (BPBC) technique. This idea works well not only for a multi-core machine but also for a single CPU.

Let $f : \{0,1\}^m \to \{0,1\}^n$ be a function with m input bits and n output bits. Since f is a function, there exists a combinational logic circuit that computes f. Let $X_0, X_1, ..., X_{d-1}$ be d inputs of m bits each. Suppose that we want to compute $f(X_0), f(X_1), ..., f(X_{d-1})$. We can evaluate these values one by one using a single CPU. Also, we can use d processor cores and compute $f(X_i)$ for each X_i $(0 \leq i \leq d-1)$ using one processor each. The Bitwise Parallel Bulk Computation (BPBC) technique can perform this computation much faster than these straightforward sequential and parallel algorithms simulating the combinational logic circuit independently for all inputs.

Let $x_{i,0}x_{i,1}\cdots x_{i,m-1}$ denote m bits of each X_i $(0 \leq i \leq d-1)$. Further, let $x_{0,j}x_{1,j}\cdots x_{d-1,j}$ be $\mathcal{X}_j$ $(0 \leq j \leq m-1)$. We assume that CPU can handle d-bit word and each $\mathcal{X}_j$ is stored

in a d-bit word. By bitwise logic operations for $\mathcal{X}_j$, we can simulate a combinational logic circuit for computing f, and can obtain the values of $f(X_0), f(X_1), ..., f(X_{d-1})$ at the same time. For example, let $f(a, b, c) = (y, z)$ such that $y = (a \wedge b) \vee (b \wedge c) \vee (c \wedge a)$ and $z = a \oplus b \oplus c$. In other words, f is a function simulating a full adder. Also, let $a_i b_i c_i$ denote three bits of each X_i $(0 \leq i \leq d-1)$. We assume that we have three d-bit words $A = a_0 a_1 ... a_{d-1}$, $B = b_0 b_1 ... b_{d-1}$, and $C = c_0 c_1 ... c_{d-1}$. We want to compute $Y = y_0 y_1 ... y_{d-1}$ and $Z = z_0 z_1 ... z_{d-1}$ such that $(y_i, z_i) = f(a_i, b_i, c_i)$ for all i $(0 \leq i \leq d-1)$. Two words Y and Z can be computed simply by bitwise XOR ($\oplus$), bitwise AND ($\wedge$), and bitwise OR ($\vee$) as follows:

$$\begin{aligned} Y &\leftarrow (A \wedge B) \vee (B \wedge C) \vee (C \wedge A), \\ Z &\leftarrow A \oplus B \oplus C. \end{aligned}$$

Hence, we can compute Y and Z in 7 bitwise binary operations. For later reference, we show Y and Z can be computed in 5 bitwise binary operations using a temporal word T as follows:

$$\begin{aligned} T &\leftarrow A \oplus B, \\ Z &\leftarrow T \oplus C, \\ Y &\leftarrow (A \wedge B) \vee (T \wedge C). \end{aligned}$$

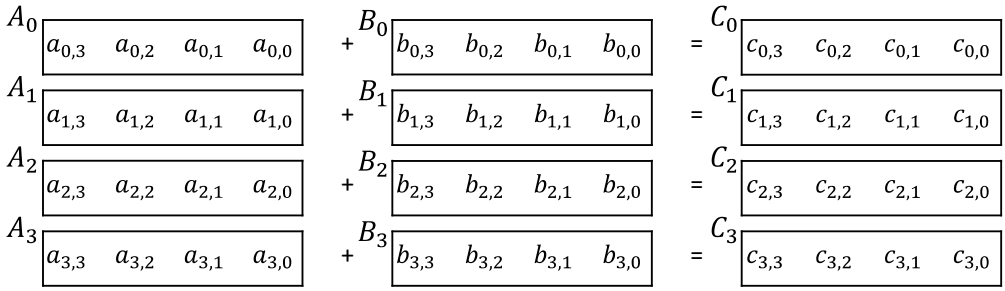
5.1 Bitwise summing technique

As an example of application of the BPBC technique, we show the pairwise sums of integers can be accelerated if the value of integers are small. Suppose that 32 pairs $(A_0, B_0), (A_1, B_1), ..., (A_{31}, B_{31})$ of 32-bit integers are given, and we want to compute the

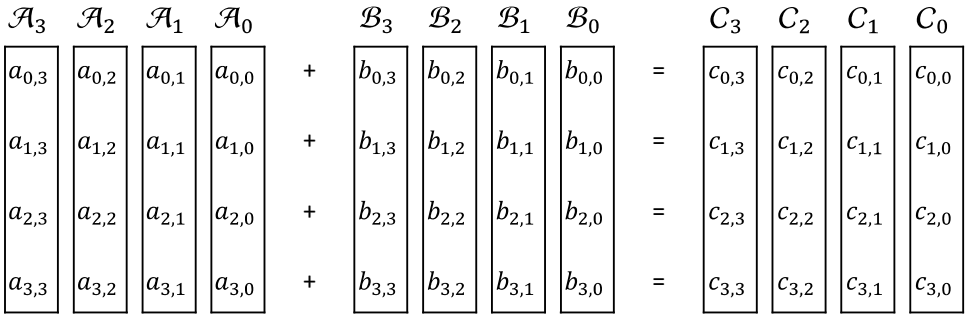
pairwise sums $A_0+B_0, A_1+B_1, ..., A_{31}+B_{31}$. Clearly, 32 addition operations can complete the computation of pairwise sums. The BPBC technique for this task simulates thirty two 32-bit ripple-carry adders by 32-bit bitwise operations, such as bitwise OR, AND, and XOR. Figure 5.1 illustrates the naive pairwise addition and the BPBC pairwise addition for four pairs of 4-bit numbers. The naive pairwise addition performs addition operation four times. On the other hand, the BPBC pairwise addition simulates a 4-bit ripple-carry adder for $(\mathcal{A}_3, \mathcal{A}_2, \mathcal{A}_1, \mathcal{A}_0)$ and $(\mathcal{B}_3, \mathcal{B}_2, \mathcal{B}_1, \mathcal{B}_0)$ to obtain (C_3, C_2, C_1, C_0). In most practical application programs, A's and B's do not have full 32 bits. In some applications, all integers stored in 32-bit integer may have only 10-20 bits. The running time of a naive pairwise addition cannot be reduced even if these numbers are very small. On the other hand, the computation of pairwise sums by the BPBC can be accelerated if values of A's and B's are small. For example, if these numbers are less than 2^{16}, then we can compute the pairwise sums by simulating 16-bit ripple-carry adders.

Suppose that we have two sequences of d numbers with d bits each. Let $A = A_0, A_1, ..., A_{d-1}$ and $B = B_0, B_1, ..., B_{d-1}$ denote these two sequences, and $a_{i,j}$ and $b_{i,j}$ be the j-th bits of A_i and B_i, respectively. Our goal is to compute sequence $C_0, C_1, ..., C_{d-1}$ of d numbers with d bits each, such that $C_i = (A_i + B_i) \bmod 2^d$ for each i $(0 \leq i \leq d-1)$. Clearly, we can obtain C by computing the pairwise sums of two sequences A and B in $O(d)$ time by an obvious sequential algorithm as illustrated in Figure 5.1 (1). Let us apply the BPBC technique for this problem. Let $\mathcal{A}_j$ and $\mathcal{B}_j$ be words of d bits each such that $\mathcal{A}_j = a_{0,j}a_{1,j}\cdots a_{d-1,j}$ and $\mathcal{B}_j = b_{0,j}b_{1,j}\cdots b_{d-1,j}$. The goal is to compute the sum $C_j = c_{0,j}c_{1,j}\cdots c_{d-1,j}$ such that $c_{i,j}$ is the j-th bit of C_i as illustrated in Figure 5.1 (2). We can obtain C by simulating a d-bit ripple-carry adder as follows:

$$C_0 \leftarrow \mathcal{A}_0 \oplus \mathcal{B}_0$$



(1) the naive pairwise addition



(2) the BPBC pairwise addition

Figure 5.1: The naive pairwise addition and the BPBC pairwise addition for four pairs of 4-bit numbers

$$
\begin{aligned}
\mathcal{T} &\leftarrow \mathcal{A}_0 \wedge \mathcal{B}_0 \\
C_1 &\leftarrow \mathcal{A}_1 \oplus \mathcal{B}_1 \oplus \mathcal{T} \\
\mathcal{T} &\leftarrow (\mathcal{A}_1 \wedge \mathcal{B}_1) \vee (\mathcal{A}_1 \wedge \mathcal{T}) \vee (\mathcal{B}_1 \wedge \mathcal{T}) \\
C_2 &\leftarrow \mathcal{A}_2 \oplus \mathcal{B}_2 \oplus \mathcal{T} \\
\mathcal{T} &\leftarrow (\mathcal{A}_2 \wedge \mathcal{B}_2) \vee (\mathcal{A}_2 \wedge \mathcal{T}) \vee (\mathcal{B}_2 \wedge \mathcal{T}) \\
&\vdots \\
C_{d-1} &\leftarrow \mathcal{A}_{d-1} \oplus \mathcal{B}_{d-1} \oplus \mathcal{T}
\end{aligned}
$$

In this algorithm, $\mathcal{T}$ is a d-bit temporal variable used to store carry bits. Clearly, this

algorithm runs $O(d)$ time. Hence, the computing time is the same as the obvious sequential algorithm. However, if the numbers stored in A and B are small, we can omit the computation. More specificall , suppose that all numbers in A and B are less than K. Since the pairwise sums in C are less than $2K$, all C_i's ($\log K + 1 \leq i \leq d - 1$) are zero. Thus, we can omit the computation for these C_i's and the computing time can be decreased to $O(\log K)$.

Let us evaluate the computation performed by the BPBC technique. Again, let $f : \{0,1\}^m \rightarrow \{0,1\}^n$ be a function with m input bits and n output bits. We assume that f can be computed by a combinational logic circuit with s gates. We have M inputs $X_0, X_1, ..., X_{M-1}$ with m bits each and compute $f(X_i)$ for all i ($0 \leq i \leq M - 1$). Since $f(X_i)$ can be computed in $O(s)$ time by simulating the combinational logic circuit, we can evaluate all $f(X_i)$ for all i in $O(sM)$ time.

To apply our bitwise computation technique, we partition the input into $\frac{M}{d}$ groups of d inputs each. Let $x_{i,j}$ denote the j-th bit of X_i as before. Since the firs group has d inputs $X_0, X_1, ..., X_{d-1}$, we store each j-th bit ($0 \leq j \leq d - 1$) in a d-bit integer $\mathcal{X}_j$ such that $\mathcal{X}_j = x_{0,j}x_{1,j}\cdots x_{d-1,j}$. Clearly, the values of $f(X_i)$ for all i ($0 \leq i \leq d - 1$) can be computed in $O(s)$ time by simulating the combinational logic circuit by bitwise logic operations. The values of $f(X_i)$'s remaining groups can be computed in the same way. Thus, we have,

Theorem 5.1.1 *The bulk computation of simulating a combinational logic circuit with s gates for M inputs can be done in $O(\frac{sM}{d})$ time using a single d-bit processor.*

5.2 The BPBC on the UMM and performance analysis

The main purpose of this section is to show that the BPBC can be implemented in the GPU very efficiently. For this purpose, we use the Unifie Memory Machine (UMM) introduced in Section 3.1. We evaluate the performance of algorithms using the BPBC technique on the UMM to show that they are efficient from the theoretical point of view. Since the BPBC technique implemented in the GPU does not use the shared memory, the UMM can be used for the theoretical analysis of the performance.

Let us implement the BPBC technique in the UMM and evaluate the performance. As before, we have M inputs $X_0, X_1, ..., X_{M-1}$ with m bits each and compute $f(X_i)$ for all i ($0 \leq i \leq M-1$), where f is a function computed by a combinational logic circuit with s gates. For this purpose, we partition the inputs into $\frac{M}{d}$ groups of d inputs each and each thread i ($0 \leq i \leq \frac{M}{d}-1$) on the UMM computes the value of i-th group by simulating the combinational logic circuit. For example, thread 0 computes the values of $f(X_0), f(X_1), ..., f(X_{d-1})$. Each thread also uses s words to store the output of s gates, and thus, it totally uses $m+s$ words. We arrange $m+s$ words used by $\frac{M}{d}$ threads in a 2-dimensional array of $(m+s) \times \frac{M}{d}$ d-bit words as illustrated in Figure 5.2. Each thread accesses to $m+s$ words in a column and simulates the combinational logic circuit for d inputs. We call this arrangement the column-wise arrangement. Since words in the same row are allocated in consecutive addresses, memory access to words in the same row by w threads in a warp occupies pipeline registers in one stage. Thus, memory access requests by $\frac{M}{d}$ threads to the same row occupies $\frac{M}{wd}$ stages and the memory access is completed in at most $O(\frac{M}{wd}+l)$ time units. Since every thread issues $O(s)$ memory access requests to simulate the combinational logic circuit, we have,

Theorem 5.2.1 *The BPBC represented by a combinational logic circuit with s gates for*

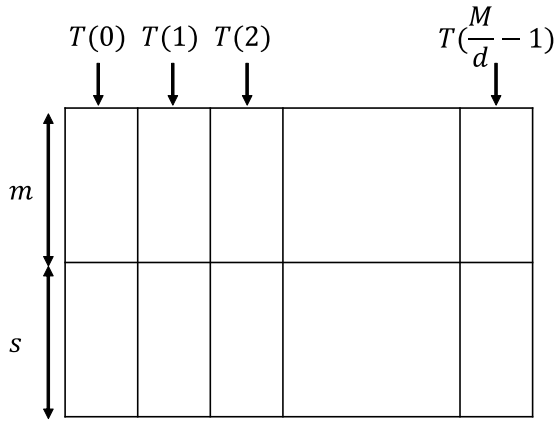


Figure 5.2: The column-wise arrangement of $m + s$ words on the memory

M inputs can be done in $O(\frac{sM}{wd} + sl)$ time units using $\frac{M}{d}$ threads on the UMM with d-bit words, width w and latency l.

Clearly, at most wd bits in the memory of the UMM can be accessed in a time unit. To simulate a combinational logic circuit with s gates for M inputs, at least sM memory requests are necessary. Thus, this task takes at least $\Omega(\frac{sM}{wd})$ time on the UMM and it is time optimal if $sl \leq \frac{sM}{wd}$, that is, if $M \geq wdl$ holds.

Next, let us evaluate the time for pairwise sum of M pairs of two numbers using Theorem 5.2.1. We assume that all numbers are less than K. As we have discussed, the combinational logic circuit to compute the pairwise sums has $O(\log K)$ gates. Thus, from Theorem 5.2.1, we have,

Corollary 5.2.2 *The pairwise sums of M pairs of two numbers less than K can be computed in $O(\frac{M \log K}{wd} + l \log K)$ time units using $\frac{M}{d}$ threads on the UMM with d-bit words, width w and latency l.*

Similarly, this computation is optimal if $M \geq wdl$.

5.3 Experimental results of pairwise summing

The main purpose of this section is to show experimental results using Intel Core i7-4790 (3.6GHz) CPU and GeForce GTX TITAN X (1GHz) GPU. GeForce GTX TITAN X has 24 streaming multiprocessors with 128 cores each. Hence, it has totally $24 \times 128 = 3072$ processor cores. Since we use the BPBC technique, we have not used the shared memory of streaming multiprocessors on the GPU. We have evaluated the running time for pairwise summing for $2^{20} = 1048576$ pairs of 32-bit unsigned integers stored in the global memory of the GPU. We assume that all numbers are less than 2^k and evaluated the performance for every k ($1 \leq k \leq 32$). The implementation of a sequential algorithm for naive pairwise summing is obvious. It simply computes the sum of two integers one by one. Bitwise pairwise summing is performed by simulating k-bit ripple-carry adder circuit for two k-bit integers. In GPU implementations, we invoke 1024 CUDA blocks with 32 threads each and each thread computes the sums of 32 pairs by the BPBC.

Table 5.1 shows the running time and the speed-up factors. The naive sequential algorithm runs about 0.676 ms, while GPU performs the same computation in 0.0566 ms. Thus, the GPU implementation is 11.9 times faster than the CPU for the naive algorithm. The running time of bitwise CPU/GPU implementation is almost proportional to the value of k. Also, the bitwise implementations are faster if $k \leq 19$ on the CPU and $k \leq 29$ on the GPU. We can say that, the BPBC is a potent method if the upper bound of input numbers is limited, and the GPU acceleration works very effectively. For example, if input numbers are guaranteed to be in the range of 16-bit short integer, the pairwise sums for 1048576 pairs can be computed only in 0.0223 ms by the BPBC technique. On the other hand, naive pairwise summing takes 0.0565 ms. The sequential algorithm for the bitwise summing technique runs 0.562 ms. Hence, the GPU bitwise

pairwise summing is the best method among others.

Table 5.1: The running time (in milliseconds) of the pairwise sums for 1048576 pairs and the speed-up factors

k	naive		bitwise		speed-up factors			
	CPU	GPU	CPU	GPU	GPU over CPU		bitwise over naive	
					naive	bitwise	CPU	GPU
1	0.682	0.0564	0.0475	0.00119	12.1	39.8	14.4	47.2
2	0.676	0.0566	0.109	0.00230	11.9	47.6	6.18	24.6
3	0.676	0.0564	0.149	0.00337	12.0	44.1	4.55	16.7
4	0.677	0.0566	0.196	0.00445	12.0	44.1	3.45	12.7
5	0.676	0.0566	0.221	0.00487	11.9	45.4	3.06	11.6
6	0.676	0.0569	0.250	0.00601	11.9	41.6	2.70	9.46
7	0.676	0.0565	0.278	0.00703	12.0	39.5	2.44	8.04
8	0.676	0.0564	0.310	0.00814	12.0	38.1	2.18	6.93
9	0.676	0.0566	0.340	0.00863	11.9	39.4	1.99	6.56
10	0.674	0.0563	0.376	0.0119	12.0	31.7	1.79	4.75
11	0.676	0.0565	0.403	0.0134	12.0	30.2	1.68	4.23
12	0.675	0.0565	0.434	0.0171	12.0	25.4	1.56	3.31
13	0.675	0.0565	0.464	0.0176	12.0	26.4	1.45	3.22
14	0.676	0.0566	0.496	0.0199	11.9	24.9	1.36	2.85
15	0.676	0.0565	0.529	0.0209	12.0	25.3	1.28	2.70
16	0.680	0.0565	0.562	0.0223	12.0	25.2	1.21	2.54
17	0.676	0.0566	0.592	0.0224	12.0	26.4	1.14	2.52
18	0.676	0.0570	0.623	0.0241	11.9	25.8	1.08	2.36
19	0.676	0.0567	0.657	0.0271	11.9	24.2	1.03	2.09
20	0.676	0.0567	0.691	0.0289	11.9	23.9	0.977	1.96
21	0.676	0.0569	0.726	0.0327	11.9	22.2	0.930	1.74
22	0.676	0.0565	0.761	0.0359	12.0	21.2	0.889	1.57
23	0.676	0.0567	0.796	0.0387	11.9	20.6	0.849	1.47
24	0.677	0.0570	0.830	0.0410	11.9	20.3	0.815	1.39
25	0.676	0.0565	0.865	0.0440	12.0	19.6	0.782	1.28
26	0.676	0.0566	0.902	0.0476	12.0	19.0	0.750	1.19
27	0.675	0.0564	0.936	0.0494	12.0	19.0	0.721	1.14
28	0.676	0.0568	0.973	0.0519	11.9	18.7	0.695	1.09
29	0.676	0.0567	1.01	0.0544	11.9	18.5	0.671	1.04
30	0.676	0.0566	1.04	0.0570	11.9	18.3	0.649	0.993
31	0.676	0.0569	1.08	0.0598	11.9	18.0	0.627	0.951
32	0.675	0.0568	1.41	0.0616	11.9	22.9	0.480	0.923

Chapter 6

Efficient GPU implementations of Conway's Game of Life

In this chapter, we briefl describe Conway's Game of Life and the BPBC implementation. Also, we show several techniques for accelerating the computation and the method for simulating a very large universe.

6.1 Conway's Game of Life and a conventional implementation

The universe of Conway's Game of Life is a 2-dimensional array of cells, each of which takes one of two states, 1 (alive) or 0 (dead). For simplicity, we assume that the size of the array is $\sqrt{n} \times \sqrt{n}$. Let $u_0, u_1, ...$ denote the states of the universe such that universe u_0 stores the initial states, and each u_t ($t \geq 1$) is the states after t-step transition. Let $u_t(i, j)$ denote the state of a cell at position (i, j) ($0 \leq i, j \leq \sqrt{n} - 1$). For simplicity, we assume that the universe is wrapped around to handle the state of cells outside of the array. For

example, the value of $u_t(i,-1)$ is that of $u_t(i,\sqrt{n}-1)$. Let $s_t(i,j)$ be the number of alive cells in eight neighbors of cell (i,j) define as follows:

$$\begin{aligned} s_t(i,j) = {} & u_t(i-1,j-1)+u_t(i-1,j)+u_t(i-1,j+1)+u_t(i,j-1) \\ & +u_t(i,j+1)+u_t(i+1,j-1)+u_t(i+1,j)+u_t(i+1,j+1). \end{aligned} \tag{6.1}$$

The state $u_t(i,j)$ $(0 \le i,j \le \sqrt{n}-1)$ is determined by the following formula:

$$\begin{aligned} u_t(i,j) &= 1\text{(alive) if } s_{t-1}(i,j)=3 \text{ or} \\ &\quad (u_{t-1}(i,j)=1 \text{ and } (s_{t-1}(i,j)=2 \text{ or } s_{t-1}(i,j)=3)), \\ &= 0\text{(dead) otherwise.} \end{aligned}$$

Hence, we can compute the value of $u_t(i,j)$ by the following Boolean formula:

$$u_t(i,j) = (s_{t-1}(i,j)=3) \vee (u_{t-1}(i,j)=1 \wedge s_{t-1}(i,j)=2) \tag{6.2}$$

We have two arrangements, the word-per-cell and the bit-per-cell arrangements for simulating the Game of Life not only on the GPU but also on the CPU. The word-per-cell arrangement is a conventional arrangement in which the state of each cell is stored in a word of the memory, such as a 32-bit integer or an 8-bit character. For example, we can store the states $u_0(i,j)$ $(0 \le i,j \le \sqrt{n}-1)$ of cells in a $\sqrt{n} \times \sqrt{n}$ 2-dimensional array of 8-bit characters. For more storage-efficient implementation of 2-dimensional array of cells, we can use the bit-per-cell arrangement, which arranges each cell to a bit of a word. For example, we use a 32-bit unsigned integer to store the states of consecutive 32 cells. In general, d consecutive cells in the same row are stored in a d-bit word and thus n cells are stored in a $\sqrt{n} \times \frac{\sqrt{n}}{d}$ array of d-bit words. As illustrated in Figure 6.1, consecutive 32 cells in the same row is arranged in a 32-bit word. Since a square block of 32×32 cells are arranged in consecutive address, we use column-major order addressing as shown in the figure

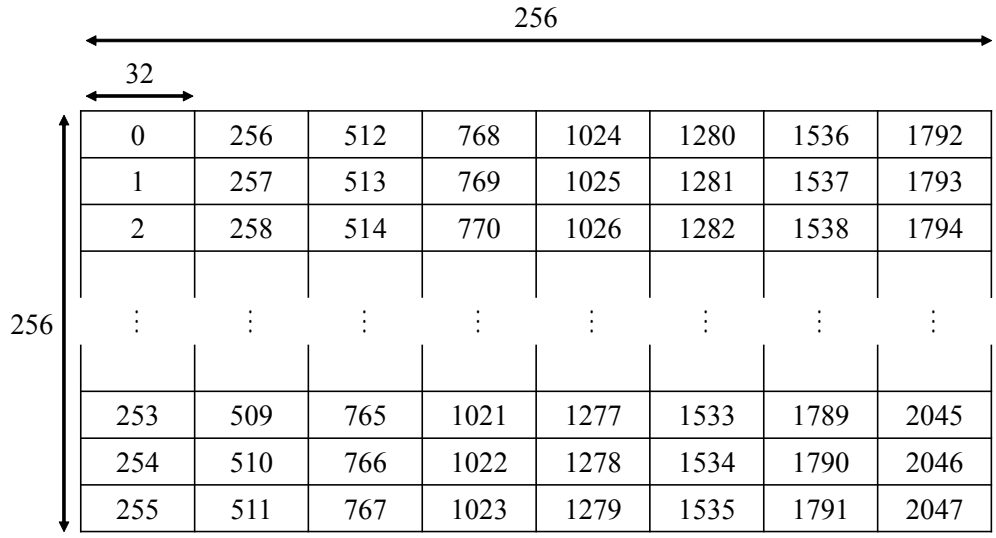


Figure 6.1: Bit-per-cell arrangement of 256×256 universe of 32-bit words with column-major order arrangement

Let us see a straightforward GPU implementation for the Game of Life using the word-per-cell arrangement. We assume that the initial states of cells are stored in the global memory of the GPU. We use a CUDA kernel with n threads to compute the next states $u_1(i, j)$. For example, a CUDA kernel invokes $\frac{n}{32}$ CUDA blocks with 32 threads each. Each thread is assigned to a cell, and computes the next state $u_1(i, j)$ and write it in the global memory. Note that it is not possible to compute $u_2(i, j)$ by the same CUDA kernel, because threads in different CUDA blocks cannot communicate with each other. Thus, after a thread computes and writes $u_1(i, j)$, it must terminate. A CUDA kernel terminates when all threads complete the computation of next states of cells. After that, the same CUDA kernel to compute $u_2(i, j)$ is invoked. In other words, one CUDA kernel call is necessary to simulate one-step transition and thus, T CUDA kernel calls are performed for T-step simulation.

6.2 BPBC implementation for the Game of Life

The main purpose of this section is to show how the BPBC technique can be applied to simulation of the Game of Life.

To simulate the Game of Life stored in the bit-per-cell arrangements, we can retrieve the state of an individual cell by bitwise AND operation, compute the sum of neighbors by formulas 6.1 and 6.2, and write the next state by bitwise OR operation. However, this straightforward implementation of the bit-per-cell arrangement is not efficient. We should use the bitwise summing technique, which computes the bitwise sum of words by the BPBC technique. The original idea using the bitwise summing technique has been shown in [45].

To compute the next states of d cells stored in a d-bit word, the states of $2d + 6$ neighboring cells are necessary. For example, Figure 6.2 shows the computation to obtain the next states of 4 cells in I. We need $2 \cdot 4 + 6 = 14$ neighboring cells for this computation. We firs store neighboring cells in eight 4-bit words $A, B, ..., H$ as illustrated in Figure 6.2. After that, we compute the bitwise sums as shown in Figure 6.2 and obtain two words I_2 and I_3, where each bit of I_2 and I_3 is 1 if and only if the number of 1's in the corresponding position of eight words $A, B, ..., H$ is 2 and 3, respectively. Clearly, using I_2, I_3 and the current value of I, we can compute the next states of all cells in I by evaluating $(I \wedge I_2) \vee I_3$. Next, we will show how I_2 and I_3 are computed. Let $([A-H]_3, [A-H]_2, [A-H]_1, [A-H]_0$ denote the bitwise sums of each bit of $A, B, ..., H$. Also, let $[A-H]_{23} = [A-H]_2 \vee [A-H]_3$. Clearly, $I_2 = 1$ if $([A-H]_{23}, [A-H]_1, [A-H]_0) = (0, 1, 0)$ and $I_3 = 1$ if $([A-H]_{23}, [A-H]_1, [A-H]_0) = (0, 1, 1)$. Hence, we can compute I_2 and I_3 from $([A-H]_{23}, [A-H]_1, [A-H]_0)$.

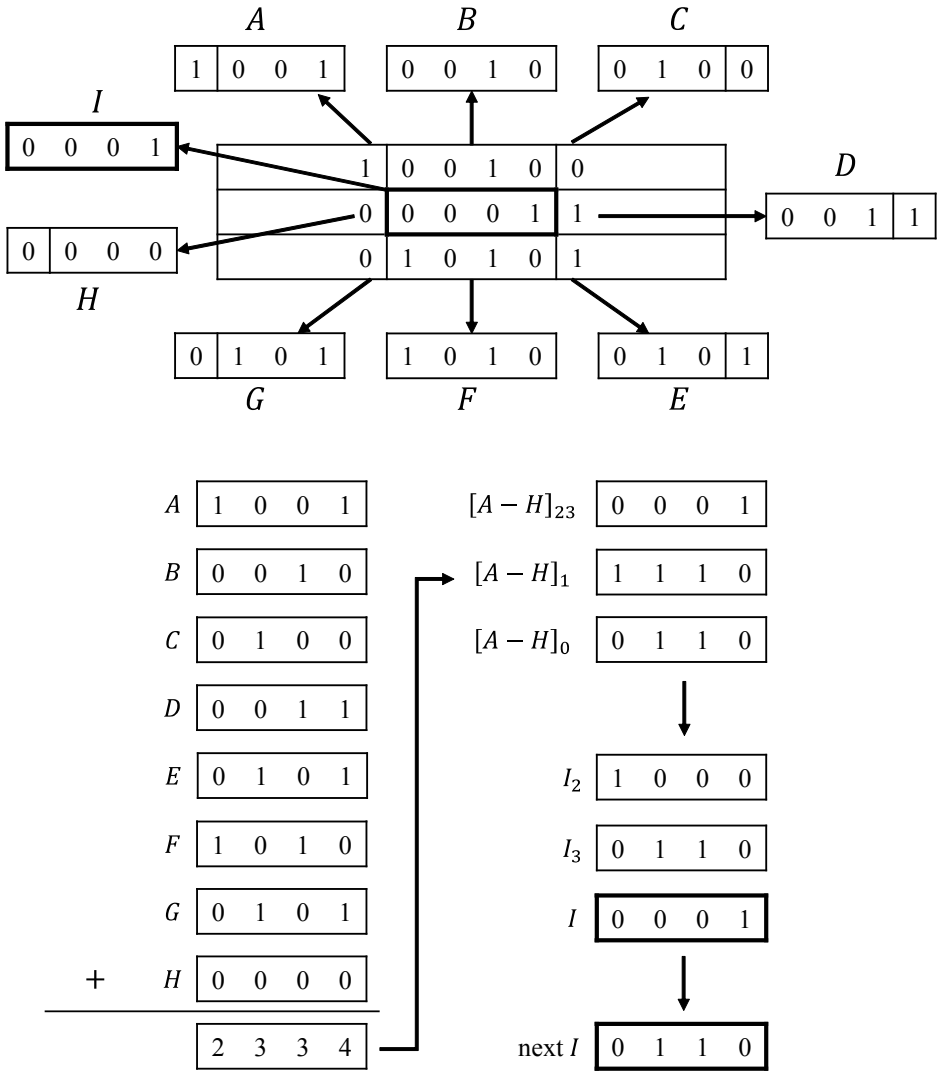


Figure 6.2: The computation of the next states of 4 cells in a 4-bit word by Algorithm SINGLE-WORD

We will show Algorithm SINGLE-WORD that computes the next states of I using this idea. We firs compute the bitwise sums of each of four pairs of two words. For example, by computing $([AB]_1, [AB]_0) \leftarrow (A \wedge B, A \oplus B)$, we obtain two bits $([AB]_1, [AB]_0)$ which represent the sum of A and B. Similarly, we can obtain $([CD]_1, [CD]_0)$, $([EF]_1, [EF]_0)$, and $([GH]_1, [GH]_0)$. After that, we compute the sum of pairs $([AB]_1, [AB]_0)$ and $([CD]_1, [CD]_0)$,

and obtain three bits $([A-D]_2, [A-D]_1, [A-D]_0)$. This can be done by computing the sums from the least significan bit. Similarly, we obtain the sum $([E-H]_2, [E-H]_1, [E-H]_0)$. Finally, we compute the sum of $([A-D]_2, [A-D]_1, [A-D]_0)$ and $([E-H]_2, [E-H]_1, [E-H]_0)$ and obtain three bits $([A-H]_{23}, [A-H]_1, [A-H]_0)$. From these three bits, the values of I_2 and I_3 can be obtained and then, the next states of I can be computed. The details of an algorithm, Algorithm SINGLE-WORD that computes I_2, I_3, and the next states of I are as follows:

[Algorithm SINGLE-WORD]

1. $([AB]_1, [AB]_0) \leftarrow (A \wedge B, A \oplus B)$
2. $([CD]_1, [CD]_0) \leftarrow (C \wedge D, C \oplus D)$
3. $([EF]_1, [EF]_0) \leftarrow (E \wedge F, E \oplus F)$
4. $([GH]_1, [GH]_0) \leftarrow (G \wedge H, G \oplus H)$

// $([A-D]_2, [A-D]_1, [A-D]_0) \leftarrow ([AB]_1, [AB]_0) + ([CD]_1, [CD]_0)$

5. $[A-D]_0 \leftarrow [AB]_0 \oplus [CD]_0$
6. $[A-D]_1 \leftarrow [AB]_1 \oplus [CD]_1 \oplus ([AB]_0 \wedge [CD]_0)$
7. $[A-D]_2 \leftarrow [AB]_1 \wedge [CD]_1$

// $([E-H]_2, [E-H]_1, [E-H]_0) \leftarrow ([EF]_1, [EF]_0) + ([GH]_1, [GH]_0)$

8. $[E-H]_0 \leftarrow [EF]_0 \oplus [GH]_0$
9. $[E-H]_1 \leftarrow [EF]_1 \oplus [GH]_1 \oplus ([EF]_0 \wedge [GH]_0)$
10. $[E-H]_2 \leftarrow [EF]_1 \wedge [GH]_1$

// $([A-H]_{23}, [A-H]_1, [A-H]_0) \leftarrow ([A-D]_2, [A-D]_1, [A-D]_0) + ([E-H]_2, [E-H]_1, [E-H]_0)$

11. $[A-H]_0 \leftarrow [A-D]_0 \oplus [E-H]_0$
12. $X \leftarrow [A-D]_0 \wedge [E-H]_0$
13. $Y \leftarrow [A-D]_1 \oplus [E-H]_1$
14. $[A-H]_1 \leftarrow X \oplus Y$
15. $[A-H]_{23} \leftarrow [A-D]_2 \vee [E-H]_2 \vee ([A-D]_1 \wedge [E-H]_1) \vee (X \wedge Y)$

// $(I, I_2, I_3) \leftarrow (I, [A-H]_{23}, [A-H]_1, [A-H]_0)$

16. $Z \leftarrow \overline{[A-H]_{23}} \wedge [A-H]_1$
17. $I_2 \leftarrow \overline{[A-H]_0} \wedge Z$
18. $I_3 \leftarrow [A-H]_0 \wedge Z$
19. $I \leftarrow (I \wedge I_2) \vee I_3$

Note that, when we compute $([A-D]_2, [A-D]_1, [A-D]_0) \leftarrow ([AB]_1, [AB]_0) + ([CD]_1, [CD]_0)$, the values of $([AB]_1, [AB]_0)$ and $([CD]_1, [CD]_0)$ can not be $(1, 1)$. Hence, $[A-D]_2$ can be computed by formula $[AB]_1 \wedge [CD]_1$.

Let us evaluate the total number of binary operations and unary operations performed in this algorithm for bit-per-cell arrangement. For computing $([AB]_1, [AB]_0) \leftarrow (A \wedge B, A \oplus B)$, two binary operations are performed. Thus, the sums of four pairs can be computed by 8 binary operations. Five binary operations are performed for computing the sum of two bits, $([A-D]_2, [A-D]_1, [A-D]_0) \leftarrow ([AB]_1, [AB]_0) + ([CD]_1, [CD]_0)$. This computation is executed twice, and thus, 10 binary operations are performed. For computing $([A-H]_{23}, [A-H]_1, [A-H]_0)$, 9 binary operations are performed. Finally, (I, I_2, I_3) is computed in 5 binary operations and 2 unary operations. Thus, the total number of operations is $4 \times 2 + 2 \times 5 + 9 + 5 + 2 = 34$. Hence, we have,

Lemma 6.2.1 *The next states of cells stored in a word by the bit-per-cell arrangement can be computed in 34 operations.*

Let us implement bitwise summing technique in the GPU. Since CUDA supports 32-bit and 64-bit bitwise operations, it makes sense to use a 32-bit or 64-bit integer to store 32 or 64 cells. Suppose that we use 64-bit integers to store cells. Each thread is assigned a word storing 64 cells, and it is responsible for computing the next states of these cells. We can invoke a CUDA kernel with $\frac{n}{64 \cdot 32}$ CUDA blocks with 32 threads

each for n cells. Each word with 64 cells and 8 neighboring words are read by a thread assigned to it. The thread computes 8 words $A, B, ..., H$ from these words, and computes the next states of I by 34 operations. After that, it writes the resulting next states of I in the global memory and terminates. After all threads terminate, the CUDA kernel terminates. In this way, one-step simulation is performed by a single CUDA kernel call. The same CUDA kernel call is repeatedly performed T times to compute the T-step simulation.

6.3 Bitwise summing technique for two words

We can reduce the number of operations if next states of cells in two words are computed at the same time. If we just executed Algorithm SINGLE-WORD twice, we need 68 operations. We will show that it can be reduced to 59 operations by sharing the computation for two words. For this purpose, we partition the cells as illustrated in Figure 6.3. We compute the next states of cells in two words K and L in the figur at the same time. For updating K, the sum of words A, B, C, D, E, I, J, L is computed. Also, the sum of D, E, F, G, H, I, J, K is computed for word L. More specificall , we compute $([A-EIJL]_{23}, [A-EIJL]_1, [A-EIJL]_0)$ and $([D-K]_{23}, [D-K]_1, [D-K]_0)$. Clearly, four words D, E, I, J are included in both sets of words. Hence, by computing the sum of these words first we can reduce the total number of operations. Once we have $(K, [A-EIJL]_{23}, [A-EIJL]_1, [A-EIJL]_0)$, we can compute (K, K_2, K_3) where K stores the next states of K, and each bit of K_2 and K_3 is 1 if and only if the number of 1's in the corresponding position of eight words A, B, C, D, E, I, J, L is 2 and 3, respectively. Similarly, we can obtain (L, L_2, L_3) using $(L, [D-K]_{23}, [D-K]_1, [D-K]_0)$.

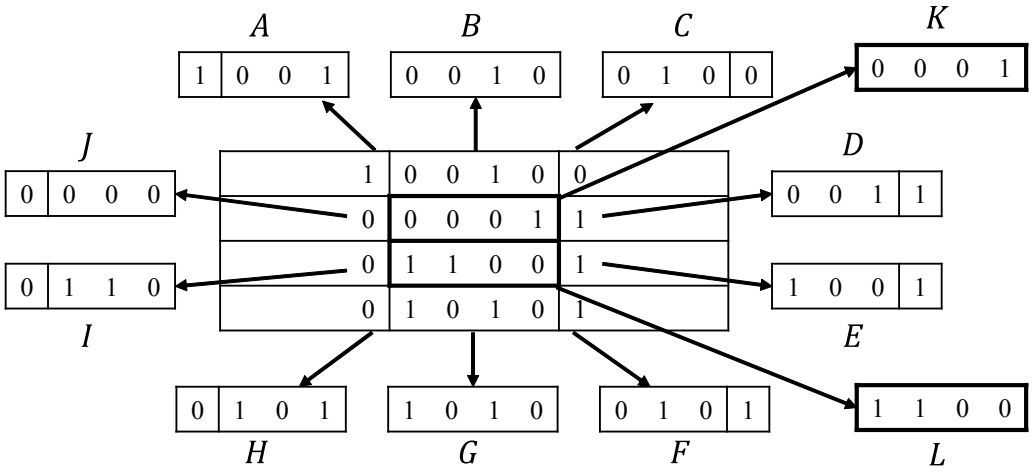


Figure 6.3: Illustrating 12 words for computing next states of cells in two words by Algorithm DOUBLE-WORD

Using this idea, next states of cells in two words can be computed by Algorithm DOUBLE-WORD as follows:

[Algorithm DOUBLE-WORD]

1. $([DE]_1, [DE]_0) \leftarrow (D \wedge E, D \oplus E)$
2. $([IJ]_1, [IJ]_0) \leftarrow (I \wedge J, I \oplus J)$
3. $([AB]_1, [AB]_0) \leftarrow (A \wedge B, A \oplus B)$
4. $([CL]_1, [CL]_0) \leftarrow (C \wedge L, C \oplus L)$
5. $([FG]_1, [FG]_0) \leftarrow (F \wedge G, F \oplus G)$
6. $([HK]_1, [HK]_0) \leftarrow (H \wedge K, H \oplus K)$
7. $([DEIJ]_2, [DEIJ]_1, [DEIJ]_0) \leftarrow ([DE]_1, [DE]_0) + ([IJ]_1, [IJ]_0)$
8. $([ABCL]_2, [ABCL]_1, [ABCL]_0) \leftarrow ([AB]_1, [AB]_0) + ([CL]_1, [CL]_0)$
9. $([FGHK]_2, [FGHK]_1, [FGHK]_0) \leftarrow ([FG]_1, [FG]_0) + ([HK]_1, [HK]_0)$
10. $([A - EIJL]_{23}, [A - EIJL]_1, [A - EIJL]_0) \leftarrow ([ABCL]_2, [ABCL]_1, [ABCL]_0) + ([DEIJ]_2, [DEIJ]_1, [DEIJ]_0)$
11. $([D{-}K]_{23}, [D{-}K]_1, [D{-}K]_0) \leftarrow ([FGHK]_2, [FGHK]_1, [FGHK]_0) + ([DEIJ]_2, [DEIJ]_1, [DEIJ]_0)$

12. $(K, K_2, K_3) \leftarrow (K, [A - EIJL]_{23}, [A - EIJL]_1, [A - EIJL]_0)$
13. $(L, L_2, L_3) \leftarrow (L, [D - K]_{23}, [D - K]_1, [D - K]_0)$

Let us evaluate the total number of operations. Each of Lines 1-6 can be done in two binary operations. Lines 7-9 can be done in 5 binary operations each. Lines 10 and 11 can be performed in 9 binary operations each. Finally, lines 12 and 13 takes 5 binary operations and 2 unary operations. Thus, the total number of operations is $6 \times 2 + 3 \times 5 + 2 \times 9 + 2 \times 7 = 59$, and we have,

Lemma 6.3.1 *The next states of cells stored in two words by the bit-per-cell arrangement can be computed in 59 operations*

Similarly to the GPU implementation using the algorithm SINGLE-WORD, we can implement the algorithm for Lemma 6.3.1 in CUDA programming model. For example, a CUDA kernel with $\frac{n}{(64 \cdot 32 \cdot 2)}$ CUDA blocks with 32 threads each is repeatedly invoked. Each thread is responsible for computing the next states of two words. Since the memory access to the global memory can be shared for updating two words, we can further accelerate the computation.

6.4 Multiple-step simulation using the shared memory

We can accelerate the computation if multiple steps simulation is performed on the shared memory. More specificall , a CUDA block is assigned to multiple words, say, 32 words. It copies words storing the cell states to the shared memory and simulates multiple steps on the shared memory. The resulting states are copied to the global memory.

If multiple-step simulation is performed in a block of the universe, cells in the boundary of the block may not have correct states. More specificall , suppose that we have a block of $d \times d$ cells in a large 2-dimensional array of cells. Since we do not have the states of cells outside of the block, we simply assume that those cells always take state 0.

We can say that the boundary cells are *dirty* after one-step simulation in the sense that their states may not be correct, because at least one of neighboring cells of each boundary cell is not taken into account. Also, cells inside the boundary are *clean* in the sense that their states are guaranteed to be correct. After another step simulation, neighboring cells of the dirty cells, that is, the boundary cells of clean cells become dirty. In general, cells in the distance t from the boundary become dirty after t-step simulation and $m \times m$ cells are clean, where $m = d - 2t$, as illustrated in Figure 6.4.

To simulate multiple steps of all cells, the $\sqrt{n} \times \sqrt{n}$ 2-dimensional array in the global memory is partitioned into $\frac{\sqrt{n}}{m} \times \frac{\sqrt{n}}{m}$ slices of size $m \times m$ each as illustrated in Figure 6.5. Each slice is expanded by t cells for every direction, and we obtain a $d \times d$ block. A CUDA block is assigned to a block and performs t-step simulation using the shared memory. For this purpose, it copies the states of $d \times d$ cells in a block to the shared memory. Note that each row of $d \times d$ cells is stored in one or two d-bit words. Thus, we read at most $2d$ words to copy $d \times d$ cells from the global memory. In the shared memory, t-step simulation is performed. After that, the resulting states in the $m \times m$ slice are written in the global memory. Similarly, we need to perform write operations for at most $2m$ words to the global memory. Since this t-step simulation for all blocks must be completed before the next t-step simulation is performed. Hence, each t-step simulation must be performed by one CUDA kernel call and thus T-step simulation can

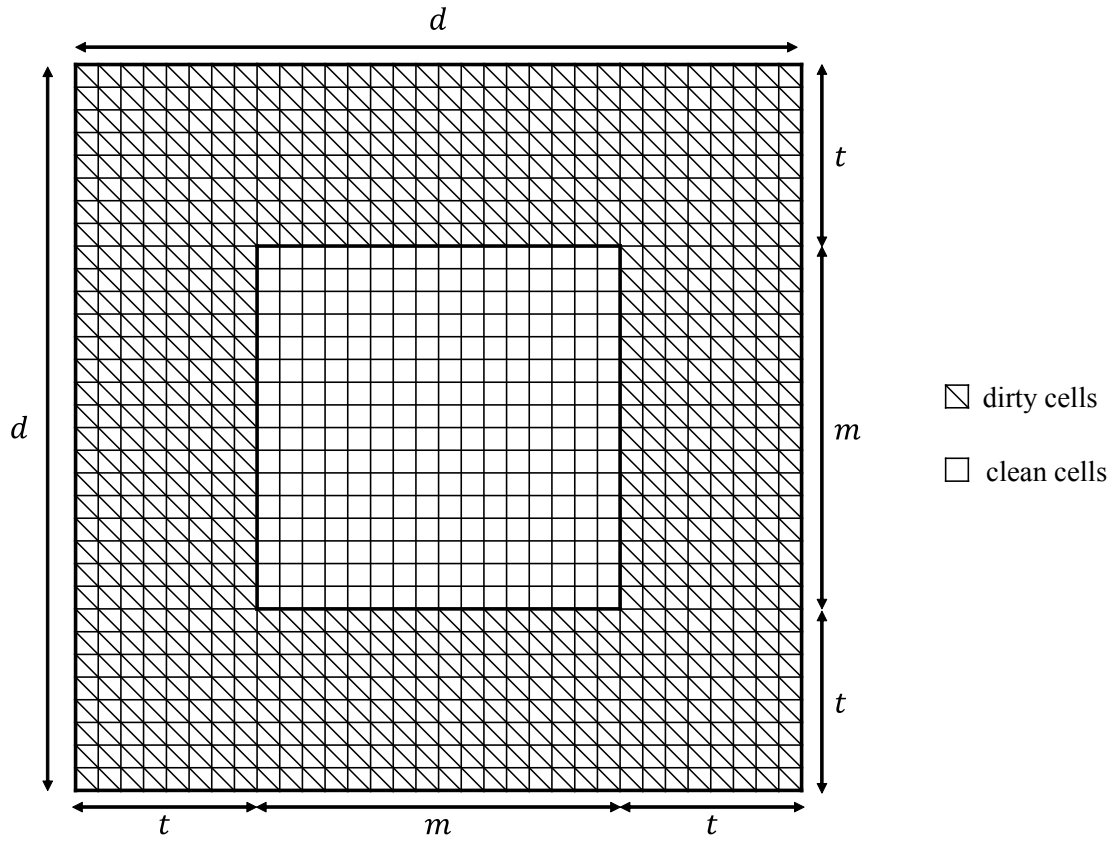


Figure 6.4: Clean and dirty cells

be done by $\frac{T}{t}$ CUDA kernel calls.

We can observe that, we should select an appropriate value of t ($1 \leq t \leq \frac{d}{2}$) for fi ed n and d that minimizes the running time. For simplicity, we assume that the cost for computing the next states of d cells stored in a word is one unit. Also, let c be the cost of miscellaneous overhead for dispatching CUDA blocks and reading/writing the states of d cells in the global memory. Under this assumption, we can write that the cost of t-step simulation of a slice of size $m \times m$ is $t + c$. Hence, the cost of T-step simulation of $\sqrt{n} \times \sqrt{n}$ cells is:

$$\frac{T}{t} \times \frac{n}{m^2} \times (t + c) = \frac{nT(t + c)}{t(d - 2t)^2}$$

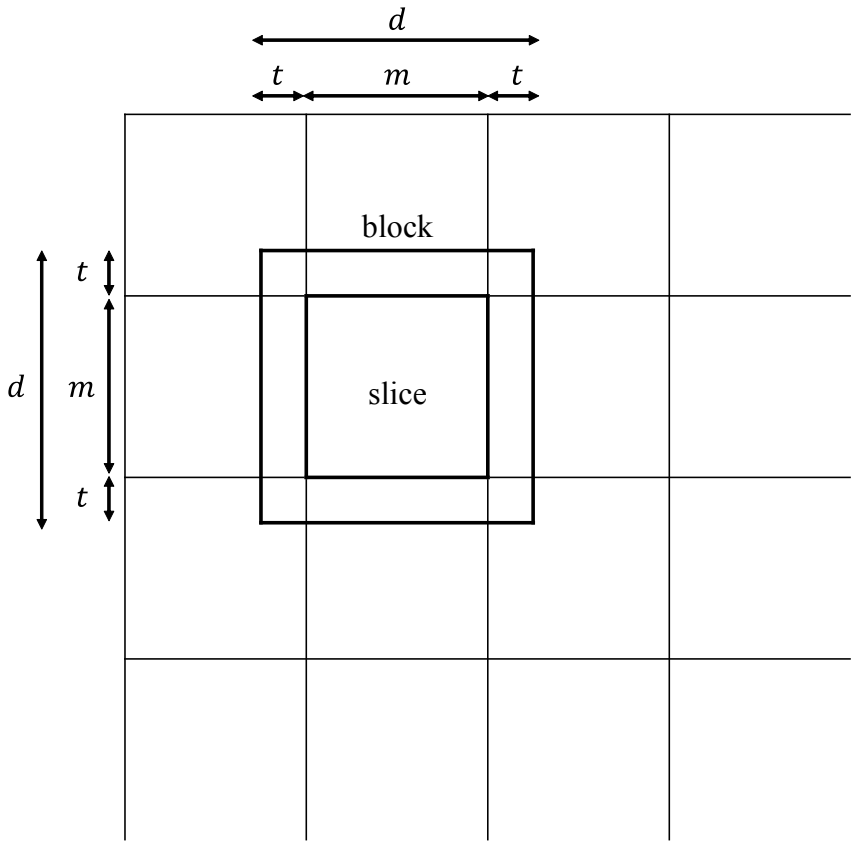


Figure 6.5: An $m \times m$ slice and a $d \times d$ block in a large 2-dimensional array

This cost is minimized when $4t^2 + 6ct - dc = 0$, that is,

$$t = \frac{\sqrt{9c^2 + 4dc} - 3c}{4}$$

Cleary, t is an increasing function of c and the value of t is in the range $[0, \frac{d}{6}]$. Intuitively, this is reasonable because the number $\frac{T}{t}$ of CUDA kernel calls should be smaller when the overhead c is larger.

If Algorithm DOUBLE-WORD is implemented using the shared memory as it is, memory access to the shared memory has bank conflicts In Algorithm DOUBLE-WORD, a block of 64×64 cells stored in 64 64-bit words are updated by 32 threads as illustrated in Figure 6.6. For example, thread 1 is responsible for updating 128 cells

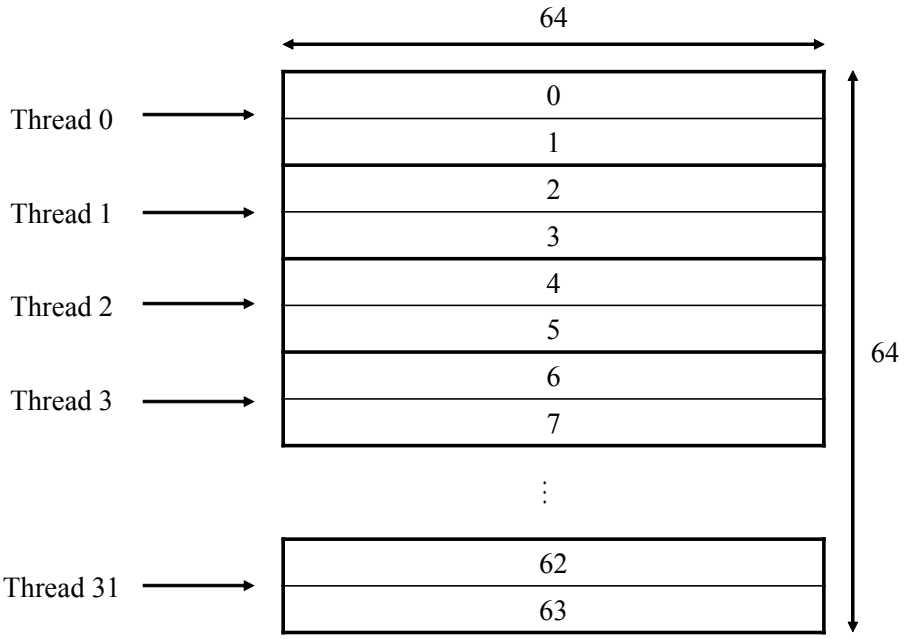


Figure 6.6: Words accessed by threads executing Algorithm DOUBLE-WORD

in rows 2 and 3. For this purpose, it accesses cells in rows $1, 2, 3$, and 4. Thus, threads $0, 1, 2, ..., 31$ may access words $k + 0, k + 2, k + 4, ..., k + 62$ at the same time for each $k = -1, 0, 1, 2$. Note that dummy rows -1 and 64 can be arranged in the shared memory to avoid out-of-bound memory access.

The shared memory of Maxwell GPU architecture has 32 memory banks with 32-bit width [16]. If we store 64-bit data in the shared memory, each of them are stored in two adjacent banks. In other words, a pair of two adjacent banks are used to store 64-bit number. Hence, we can think that the shared memory has 16 memory banks, bank $0, 1, ..., 15$ with 64-bit width. If 64 cells are arranged as it is, memory access has bank conflict as illustrated in Figure 6.7. If 32 threads access to rows $2, 4, 6, ..., 32$, then two memory access operations are performed to the same banks as illustrated in the figure To avoid such bank conflicts we use shift arrangement as illustrated in Figure 6.8, in

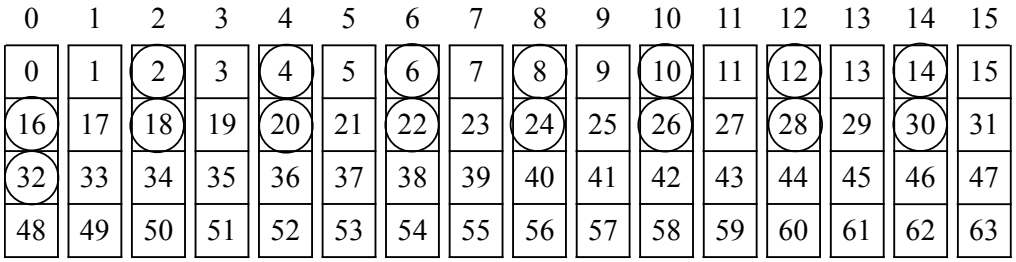


Figure 6.7: Regular arrangement

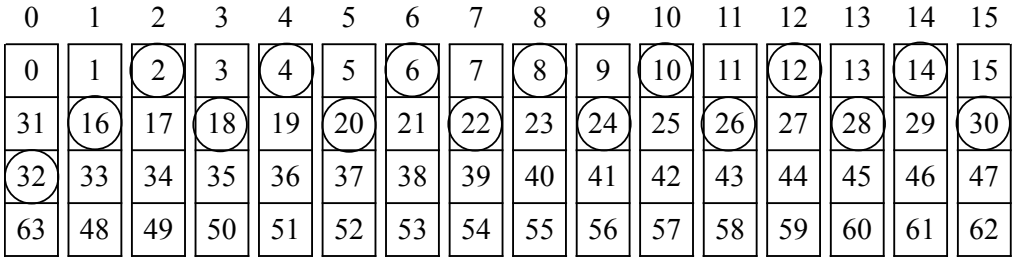


Figure 6.8: Shift arrangement

which elements are shifted by one in every two rows.

6.5 Further acceleration using warp shuffle

The memory access latency of the shared memory is not small [37]. Hence, if we can implement words of cells as registers, we can further accelerate the computation. We will show that t-step simulation can be done using registers without using the shared memory.

The algorithm is almost the same as in Section 6.4, which uses the shared memory for t-step simulation. Instead of using the shared memory, we use registers which can be accessed faster than the shared memory. However, registers are assigned to a thread, and they can be accessed only by the assigned thread. Hence, we use a warp shuffle

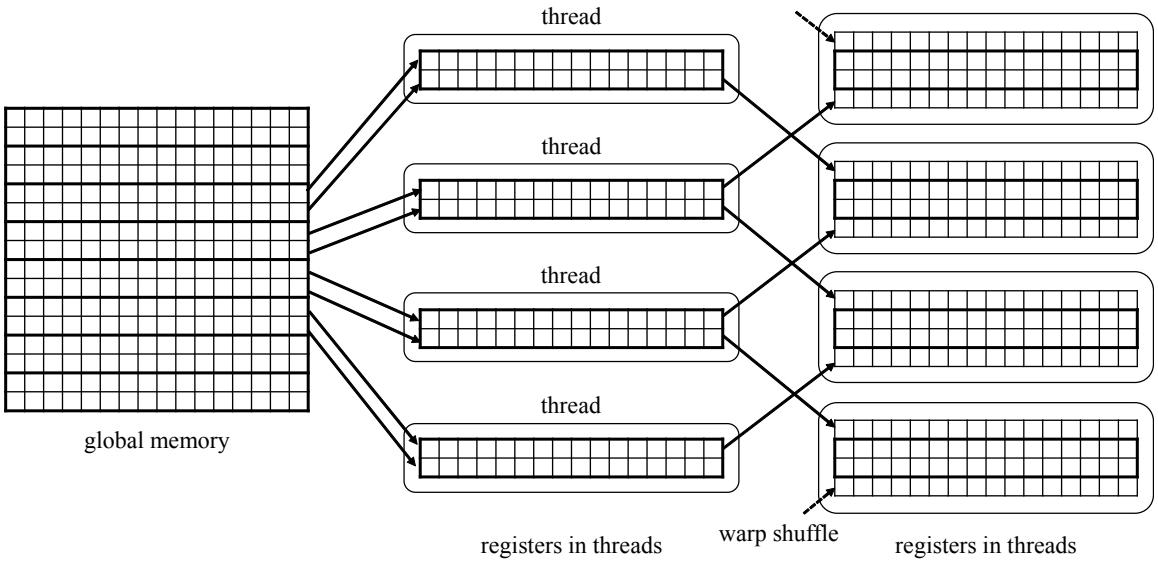


Figure 6.9: Copying words storing cells using a warp shuffle instruction

instruction, which copies registers of threads in the same warp, as illustrated in Figure 6.9. First, each thread copies two words storing cells from the global memory. For one-step simulation, each thread copies registers of two neighboring threads. After that, one-step simulation is performed for two words. This operation is repeated t times for t-step simulation. The resulting states of cells are copied from the registers to the global memory.

6.6 Simulation of a very large universe

This section shows how we perform simulation of Conway's Game of Life for a universe so large that it cannot be stored in the global memory of the GPU and in the main memory of the host PC. Consider that a very large universe of size $\sqrt{N} \times \sqrt{N}$ is stored in the SSD connected to the host PC. Our goal is to simulate the Game of Life for a large universe and store the resulting states after T-step simulation in the SSD.

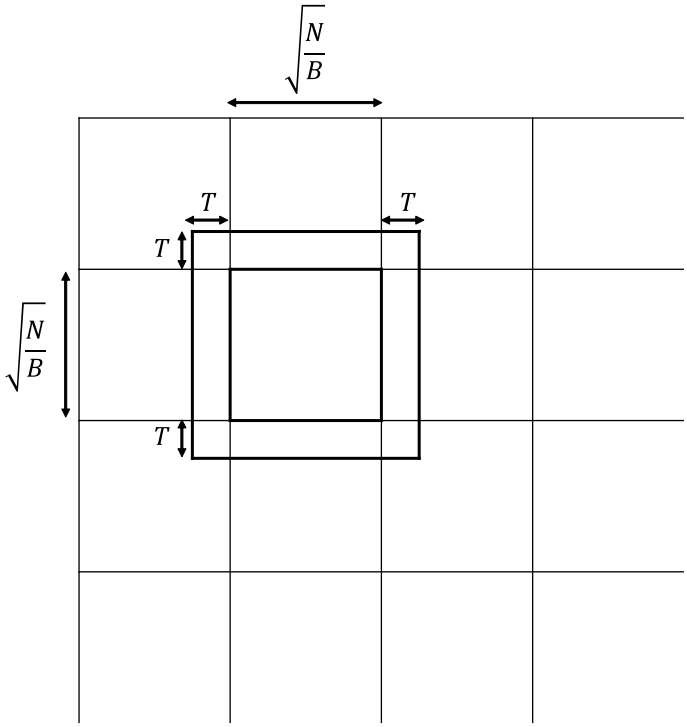


Figure 6.10: Partition of a large universe of size $\sqrt{N} \times \sqrt{N}$ into B sub-universes of size $\sqrt{\frac{N}{B}} \times \sqrt{\frac{N}{B}}$

To complete T-step simulation, we partition the universe into B sub-universes of size $\sqrt{\frac{N}{B}} \times \sqrt{\frac{N}{B}}$ each and perform t-step simulation $\frac{T}{t}$ times, as illustrated in Figure 6.10. For this purpose, similarly to multiple-step simulation shown in Section 6.4, we extend the sub-universe by T cells for each direction such that it has $(\sqrt{\frac{N}{B}} + 2T) \times (\sqrt{\frac{N}{B}} + 2T)$ cells. Using the GPU, t-step simulation of every sub-universe is performed in turn. More specificall , each extended sub-universe is copied from the SSD to the main memory of the host PC. The host PC performs T-step simulation using the GPU. Clearly, the resulting sub-universe has $\sqrt{\frac{N}{B}} \times \sqrt{\frac{N}{B}}$ clean cells. These clean cells are copied to the corresponding sub-universe in the SSD. This operation for every extended sub-universe is repeated $\frac{T}{t}$ times to complete T-step simulation.

6.7 Experimental results

The main purpose of this section is to show the performance of algorithms for Game of Life. We have used GeForce GTX TITAN X and Intel Core i7-4790 CPU (3.66GHz) for the experiment. GeForce GTX TITAN X has 24 streaming multiprocessors with 128 cores each.

We have evaluated the running time of straightforward implementations for 1K-step (2^{10}-step) simulation for a 16K $\times$ 16K ($2^{14} \times 2^{14}$) array. In the word-per-cell, we have used 8-bit unsigned characters to store the states of cells. In other words, a 2-dimensional array of 16K $\times$ 16K unsigned characters are used and evaluated formulas 6.1 and 6.2 to obtain the next states. The CPU implementation of the word-per-cell is obvious; the next state of every cell is computed one by one. To implement the word-per-cell in the GPU, each cell is assigned one thread. More specificall , a CUDA kernel computing 1-step transition invokes 2^{23} CUDA blocks with 32 threads each. The 2-dimensional array storing the states of cells are arranged in the global memory. Each thread reads the current states of cells necessary compute the next state of an assigned cell. It computes the next state of a cell by formulas 6.1 and 6.2 and writes the resulting state in the global memory. Note that, a CUDA kernel call can compute only 1-step transition and thus 1024 CUDA kernel calls are necessary to compute the states after 1024 steps. Table 6.1 shows the performance of these straightforward implementations. The performance is evaluated by the number of updates per second. For example, the CPU implementation runs 921.7 seconds for 1K-step simulation for 16K $\times$ 16K cells. Thus, the performance is $1\text{K} \cdot 16\text{K} \cdot 16\text{K}/921.7 \approx 0.298 \times 10^9$ updates per second.

Table 6.1: The performance (10^9 updates per second) of CPU implementation and GPU implementation (global memory)

	word-per-cell	bit-per-cell	
		SINGLE-WORD	DOUBLE-WORD
CPU	0.298	7.71	10.9
GPU	14.8	478	398
speed-up	49.7	62.0	36.5

From Table 6.1, we can see that the bit-per-cell arrangement is much more efficient than the word-per-cell arrangement. Since the state of one cell is stored using 8 bits in the word-per-cell, we can expect that an implementation of the bit-per-cell is 8 times faster than that of the word-per-cell. Quite surprisingly, the bit-per-cell implementation can be more than 30 times faster than the word-per-cell implementation. This is because memory access to 8-bit words is not efficient in 64-bit processor architecture. Thus, we should not use word-per-cell arrangement and must use bit-per-cell arrangement for 64-bit words. Further, we can see that Algorithm DOUBLE-WORD on the CPU is much faster than Algorithm SINGLE-WORD. On the other hand, Algorithm DOUBLE-WORD on the GPU does not achieve an improvement over Algorithm SINGLE-WORD. This is because a straightforward implementation of Algorithm DOUBLE-WORD involves stride memory access to the global memory, while that of Algorithm SINGLE-WORD does not.

For further acceleration, we have implemented multiple-step simulation with bit-per-cell arrangement using the shared memory and the registers on the GPU. Since we want to avoid barrier synchronization using `__syncthreads()`, we use CUDA blocks with a single warp of 32 threads each. Also, we implemented simulation of the Game of Life for a block with 32×32 cells and with 64×64 cells as follows:

32×32 **block:** A block of size 32×32 is implemented using 32 32-bit unsigned integers, each of which stores the states of 32 cells. A CUDA block with 32 threads is assigned 32×32 cells. Each thread computes t-step transition of 32 cells stored in a 32-bit unsigned integer by repeating Algorithm SINGLE-WORD.

64×64 **block:** A block of size 64×64 is implemented using 64 64-bit unsigned long long integers, each of which stores the states of 64 cells. Since a warp of 32 threads is used for 64 words, we execute SINGLE-WORD twice or DOUBLE-WORD once to compute 1-step transition. Each thread repeats this t times to complete t-step transition.

To fin the best value of the number t of steps computed by a single CUDA kernel call, we evaluated the running time for $t = 2, 4, 8$, and 16. Recall that the 2-dimensional array of size 16K $\times$ 16K is partitioned into $\frac{16K}{m} \times \frac{16K}{m}$ slices of size $m \times m$ each where $m = d - 2t$ and $d = 32$ for 32×32 blocks and $d = 64$ for 64×64 blocks. Hence, it makes no sense to perform 16-step simulation for 32×32 blocks, because $m = d - 2t = 0$.

Table 6.2 shows the performance (10^9 updates per second) of 1K-step simulation of the Game of Life with 16K $\times$ 16K cells. In most cases, implementations of 64×64 blocks are faster than that of 32×32 blocks, because 64-bit memory access can maximize the memory access bandwidth for the global memory and the shared memory. Also, in Algorithm DOUBLE-WORD, warp shuffle implementations are faster than the shared memory implementations. From the table, the warp shuffle implementation of 8-step simulation with 64×64 block using Algorithm DOUBLE-WORD performs 1990×10^9 updates per second, which is the maximum over all implementations that we have developed. Also, the straightforward GPU implementation using the global memory performs 478×10^9 updates per second (Table 6.1). Hence, multiple-step simulation by

the GPU can accelerate the computation by a speedup factor of 3.

Table 6.2: The performance (10^9 updates per second) of GPU implementations of multiple-step simulation

	GPU(shared memory)			GPU(register + warp shuffle)		
	32×32	64×64		32×32	64×64	
steps	SINGLE	SINGLE	DOUBLE	SINGLE	SINGLE	DOUBLE
2	451	586	768	489	672	692
4	535	808	1370	659	1330	1510
8	322	720	1560	487	1510	1990
16	-	359	873	-	790	1120

Table 6.3 shows the running time for 16K-step simulation of 512K $\times$ 512K cells. The universe of 256G cells (that is, 32G bytes) are stored in the SSD. We have partitioned the universe into 16 sub-universes of size 128K $\times$ 128K cells each, and T-step simulation for extended sub-universe is performed using the GPU. Since 16K-step simulation for the universe is performed, T-step simulation for extended sub-universe is executed $\frac{16K \cdot 16}{T} = \frac{256K}{T}$ times on the GPU. For T-step simulation of extended sub-universe stored in the global memory, we have executed warp shuffle, 8-step, 64×64 block, DOUBLE-WORD algorithm, which is the best configuratio from Table 6.2. Simulation takes more time for larger T, because extended sub-universe is larger. Also, the time for SSD read/write is inversely proportional to T. From the table, we can see that the running time is minimized when $T = 8$K. We have evaluated the running time for 16K-step simulation of 512K $\times$ 512K cells using a Core i7 CPU for T-step simulation. We found that the total running time is minimized when $T = 128$, and the performance is 13.4×10^9 updates per second. Therefore, the speedup of the GPU implementation over the CPU implementation is about $1350/13.4 \approx 100$.

Table 6.3: The running time (in seconds and 10^9 updates per second) for 16K-step simulation of 512K $\times$ 512K cells using the GPU

T	SSD read/write	simulation	total (sec)	total (10^9 updates)
1K	2470/1390	2360	6230	722
2K	1210/662	2330	4210	1070
4K	591/323	2510	3430	1310
8K	310/170	2860	3340	1350
16K	159/86.0	3550	3800	1190

Chapter 7

BPBC implementation of the CKY parsing on the GPU

In this chapter, we show that the CKY parsing for context-free grammars can be implemented in the GPU efficiently using the BPBC technique.

7.1 The CKY parsing

The main purpose of this section is to briefl describe the CKY parsing and evaluate the performance.

Let $G = (N, \Sigma, P, S)$ denote a context-free grammar such that N is a set of non-terminal symbols, Σ is a set of terminal symbols, P is a finit production rules from N to $(N \cup \Sigma)^*$, and S $(\in N)$ is the start symbol. A context-free grammar is said to be in Chomsky Normal Form (CNF), if every production rule in P is in either form $A \rightarrow BC$ (binary rule) or $A \rightarrow a$ (unary rule), where A, B, and C are non-terminal symbols and a is a terminal symbol. Note that any context-free grammar can be converted into an

equivalent CNF context-free grammar. For later reference, let p_2 and p_1 denote the numbers of binary and unary production rules, respectively.

We are interested in the parsing problem for a context-free grammar in CNF. More specificall , for a given CNF context-free grammar G and a string x over Σ, the parsing problem is a problem to determine if the start symbol S derives x by applying production rules in P. For example, let $G_{\text{example}} = (N, \Sigma, P, S)$ be a context-free grammar such that $N = \{S, A, B\}, \Sigma = \{a, b\}$, and $P = \{S \to AB, S \to BA, S \to SS, A \to AB, B \to BA, A \to a, B \to b\}$. The context-free grammar G derives *abaab*, because S derives it as follows:

$$S \Rightarrow AB \Rightarrow ABA \Rightarrow ABAA \Rightarrow ABAAB \Rightarrow \cdots \Rightarrow abaab.$$

We are going to explain the CKY parsing scheme that determines whether G derives x for a CNF context-free grammar G and a string x. Let $x = x_1 x_2 \cdots x_n$ be a string of length n, where each x_i $(1 \le i \le n)$ is in Σ. Let $T[i, j]$ $(1 \le i \le j \le n)$ denote a subset of N such that every A in $T[i, j]$ derives a substring $x_i x_{i+1} \cdots x_j$. The idea of the CKY parsing is to compute every $T[i, j]$ using the following relations:

$$\begin{aligned} T[i,i] &= \{A \mid (A \to x_i) \in P\} \\ T[i,j] &= \bigcup_{k=i}^{j-1} \{A \mid (A \to BC) \in P, B \in T[i,k], \text{ and } C \in T[k+1,j]\} \end{aligned}$$

A two-dimensional array T is called the CKY table. A grammar G generates a string x if and only if S is in $T[1, n]$. Let $\otimes_G$ denote a binary operator $2^N \times 2^N \to 2^N$ such that $U \otimes_G V = \{A \mid (A \to BC) \in P, B \in U, \text{ and } C \in V\}$. The details of the CKY parsing are spelled out as follows:

[CKY parsing]

1. $T[i, i] \leftarrow \{A \mid (A \leftarrow x_i) \in P\}$ for every i $(1 \le i \le n)$

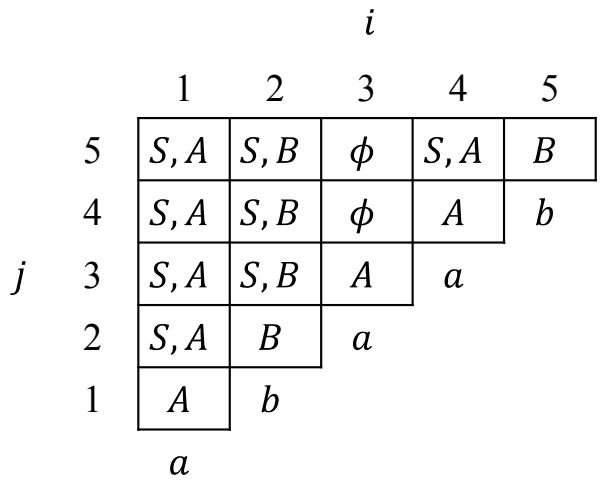


Figure 7.1: The CKY table for G_{example} and *abaab*

2. $T[i, j] \leftarrow \phi$ for every i and j $(1 \leq i < j \leq n)$
3. for $j \leftarrow 2$ to n do
4. for $i \leftarrow j - 1$ downto 1 do
5. for $k \leftarrow i$ to $j - 1$ do
6. $T[i, j] \leftarrow T[i, j] \bigcup (T[i, k] \otimes_G T[k + 1, j])$

The firs two lines initialize the CKY table, and the next four lines compute the CKY table. Figure 7.1 illustrates the CKY table for G_{example} and the string *abaab*. Since $S \in T[1, 5]$, one can see that G_{example} derives *abaab*.

Clearly, the last four lines are dominant in the CKY parsing. Let t be the computing time necessary to perform an iteration of the line 6. Then the running time is

$$\sum_{j=2}^{n} \sum_{i=1}^{j-1} \sum_{k=i}^{j-1} t = t \sum_{j=2}^{n} \sum_{i=1}^{j-1} (j - i) = \frac{1}{6} t(n^3 - n).$$

Let us evaluate the computing time t necessary to perform line 6, i.e., necessary to evaluate the binary operator $\otimes_G$. A straightforward sequential algorithm checks whether $B \in U$ and $C \in V$ for every production rule $A \rightarrow BC$ in P. Clearly, using a reasonable

data structure, this can be done in $O(1)$ time. Hence, $U \otimes_G V$ can be evaluated in $O(p_2)$ time. Thus, using the above approach, the CKY parsing can be done in $O(n^3 p_2)$ time.

Lemma 7.1.1 *The CKY parsing for an input string of length n takes $O(n^3 p_2)$ time, where p_2 is the number of binary rules.*

7.2 Bitwise Parallel Bulk Computation for CKY parsing

This section is devoted to show how we apply the BPBC technique to the CKY parsing.

Suppose that a CNF context-free grammar $G = (N, \Sigma, P, S)$ is given. Let $N = \{N_1, N_2, ..., N_b\}$ be a set of non-terminal symbols, where b is the number of non-terminal symbols. Recall that the CKY parsing repeatedly computes $U \otimes_G V = \{A \mid (A \rightarrow BC) \in P, B \in U, \text{ and } C \in V\}$. We will show that computation of $U \otimes_G V$ can be represented by a combinational logic circuit. Let U and V ($\in 2^N$) be represented by b-bit binary vectors $u_1 u_2 \cdots u_b$ and $v_1 v_2 \cdots v_b$, respectively, such that $u_i = 1$ iff $N_i \in U$. Also, let $U \otimes_G V = w_1 w_2 \cdots w_b$. For a particular w_k ($1 \leq k \leq b$), we are going to show how w_k is computed. Let $N_k \rightarrow N_{i_1} N_{j_1}, N_k \rightarrow N_{i_2} N_{j_2}, ...,$ and, $N_k \rightarrow N_{i_s} N_{j_s}$ be the binary production rules in P whose non-terminal symbol in the left-hand side is N_k. Clearly, we can compute w_k by the following formula:

$$w_k \leftarrow (u_{i_1} \wedge v_{j_1}) \vee (u_{i_2} \wedge v_{j_2}) \vee \cdots \vee (u_{i_s} \wedge v_{j_s}).$$

This formula corresponds to a combinational logic circuit with s AND gates and $s - 1$ OR gates and the value of w_k can be computed by simulating the circuit. Figure 7.2 illustrates a circuit for G_{example} in Section 7.1. Clearly, the combinational logic circuit for $\otimes_G$ has p_2 AND gates and less than p_2 OR gates.

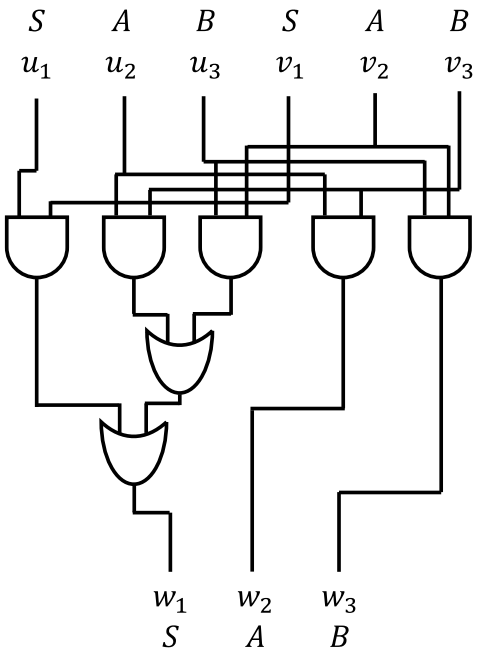


Figure 7.2: The circuit for computing $\otimes_{G_{\text{example}}}$

Since the computation of $\otimes_G$ can be done by simulating a combinational logic circuit, we can use the BPBC technique for the CKY parsing, which repeatedly computes $\otimes_G$. We assume that M input strings $X_0, X_1, ..., X_{M-1}$ of length n each are given. Our goal is to determine if $G = (N, \Sigma, P, S)$ can generate X_i for all i $(0 \leq i \leq M-1)$ by the CKY parsing. Similarly to the bitwise summing in Chapter 5, we partition the input strings into $\frac{M}{d}$ groups of d strings each, because we use a d-bit CPU. Let $x_{i,j}$ denote the j-th character of X_i. We show how we determine if G can generate X_i for the firs group. We use $|N|$ d-bit integers to represent subsets of non-terminal symbols N for d input strings of the firs group. Each bit of d-bit integers corresponds to one of the d input strings. Using these integers, we can compute $\otimes_G$ by the bitwise operations very efficiently. Figure 7.3 illustrates the computation of $\otimes_G$ for an example of a context-free grammar shown in Section 7.1. It uses three 4-bit integers to represent subsets of non-

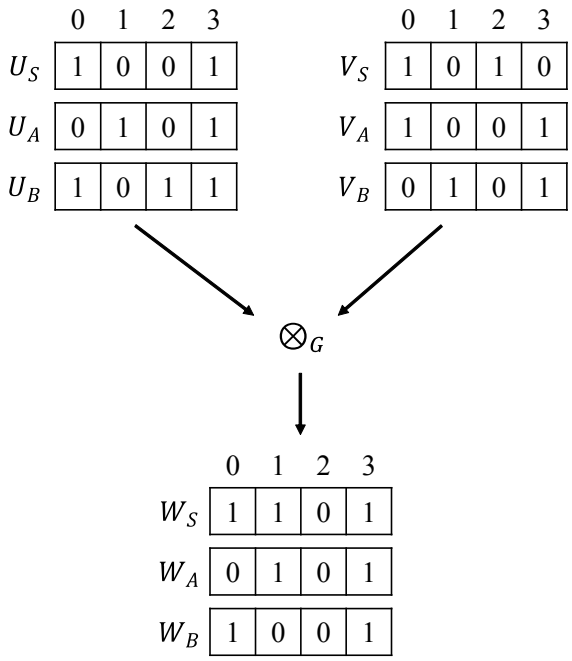


Figure 7.3: The computation of $\otimes_G$ for four instances

terminal symbols. Each of the 4 bits correspond to the following computation in terms of $\otimes_G$:

0: $\{S, B\} \otimes_G \{S, A\} \rightarrow \{S, B\}$

1: $\{A\} \otimes_G \{B\} \rightarrow \{S, A\}$

2: $\{B\} \otimes_G \{S\} \rightarrow \{\}$

3: $\{S, A, B\} \otimes_G \{A, B\} \rightarrow \{S, A, B\}$

Since the computation of $\otimes_G$ can be represented by a combinational logic circuit, we can compute $\otimes_G$ for d pairs of inputs by bitwise logic operations. For example, the computation illustrated in Figure 7.3 can be done by bitwise logic operations as follows:

$W_S \leftarrow (U_A \wedge V_B) \vee (U_B \wedge V_A) \vee (U_S \wedge V_S)$

$W_A \leftarrow U_A \wedge V_B$

$W_B \leftarrow U_B \wedge V_A$

Using this idea, we can perform the CKY parsing shown in Section 7.1. We perform the CKY parsing for d input strings at the same time. The computation of $\otimes_G$ performed in line 6 of the CKY parsing can be done by $O(p_2)$ bitwise logic operations. Hence, the CKY parsing for d input strings can be done in $O(n^3 p_2)$ time. Since we have $\frac{M}{d}$ groups, we have

Theorem 7.2.1 *The CKY parsing of M input strings of length n each can be done in $O(\frac{Mn^3 p_2}{d})$ time by the BPBC technique. From Lemma 7.1.1, the CKY parsing for M input strings of length n can be done in $O(Mn^3 p_2)$. Thus, the BPBC technique can accelerate it by a speed-up factor of d.*

7.3 The performance analysis of the CKY parsing using the BPBC technique on the UMM

The main purpose of this section is to evaluate the performance of the CKY parsing using the BPBC technique on the UMM. Let M be the number of input strings of length n. We use $\frac{M}{d}$ threads on the UMM to perform the CKY parsing. Recall that the CKY parsing for a context-free grammar G repeatedly computes the function $\otimes_G$, which can be represented by a combinational logic circuit with $O(p_2)$ gates. Let b be the number of non-terminal symbols. Each thread computes d CKY tables at the same time. Since each CKY table has $O(n^2)$ entries, each thread uses $O(bn^2)$ words to store all entries of d CKY tables. We arrange these $O(bn^2)$ words in a column of 2-dimensional array similarly to the column-wise arrangement shown in Figure 5.2. Using the column-wise arrangement, memory access is coalesced, and memory access requests by w threads in a warp occupy only one pipeline stage. Each thread performs $O(n^3 p_2)$ memory access

operations and each memory access by $\frac{M}{d}$ threads can be done in $O(\frac{M}{wd} + l)$ time units. Thus, we have,

Corollary 7.3.1 *The CKY parsing for M input strings of length n can be done in* $O(\frac{n^3 p_2 M}{wd} + n^3 p_2 l)$ *time units on the UMM with d-bit words, width w and latency l.*

7.4 GPU implementation

The main purpose of this section is to show the GPU implementation for the CKY parsing by the BPBC technique. We use the global memory of the GPU to store the CKY tables. The CKY parsing computes elements of the CKY table in the order illustrated in Figure 7.4. The elements are computed from bottom row. In each row, they are computed from right to left. Hence, we use the local memory of CUDA to cache the value of a row that is being computed. Note that the local memory may be allocated in registers in the streaming multiprocessor if small, and in the off-chip DRAM if large. Even if it is allocated in the off-chip DRAM, it may be accessed faster than the global memory, which is also arranged in the off-chip DRAM. This is because an element of the local memory is accessed by a particular thread, and cache mechanism may work efficiently for the local memory. Hence, it makes sense to use the local memory to cache a row. Also, since the capacity of the local memory is limited, it is not possible to store all elements of the CKY table in it.

Recall that the CKY parsing by the BPBC on the UMM uses $\frac{M}{d}$ threads for M input strings. Since we use 32-bit unsigned integers, $d = 32$ and $\frac{M}{32}$ threads are invoked. We arrange 32 threads for each CUDA block, a CUDA kernel for the CKY parsing invokes $\frac{M}{1024}$ CUDA blocks with 32 threads each.

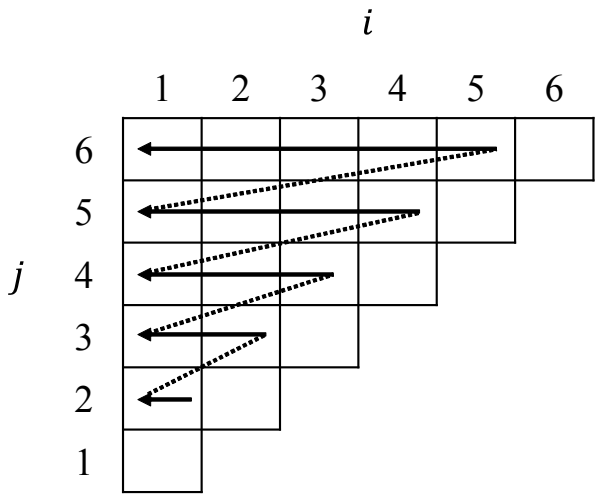


Figure 7.4: The computation order of the CKY table

7.5 Experimental results

The main purpose of this section is to show experimental results using Intel Core i7-4790 (3.6GHz) CPU and GeForce GTX TITAN X (1GHz) GPU. GeForce GTX TITAN X has 24 streaming multiprocessors with 128 cores each. Hence, it has totally $24 \times 128 = 3072$ processor cores. Since we use the BPBC technique, we have not used the shared memory of streaming multiprocessors on the GPU. Although Intel Core i7-4790 has 4 processor cores, we have used only one processor core to evaluate sequential algorithms. We may accelerate the computation by a speedup factor of up to 4 if we implement a parallel algorithm that uses all 4 processor cores. Since our goal is not to compare the computing powers of Intel Core i7-4790 and GeForce GTX TITAN X, we have not implemented a 4-parallel algorithm on Intel Core i7-4790.

Table 7.1 shows the running time of the bitwise-parallel CKY parsing for M input strings of length 32. The context-free grammar has 32/64 non-terminal symbols and 8192 production rules. Each thread uses a 32-bit unsigned register to store the current

status of 32 non-terminal symbols. Note that, the total number of possible binary production rules is $32^3 = 32768$. We have selected 8192 production rules from these rules at random. We have evaluated the running time for $M = 1024$ to 2097152 ($= 2^{21}$). Clearly, the running time of the CKY parsing by the CPU is proportional to M, because it just repeats the CKY parsing for M input strings. On the other hand, the running time by the GPU is not. Recall that, from Corollary 7.3.1, the CKY parsing on the UMM takes $O(\frac{n^3 p_2 M}{wd} + n^3 p_2 l)$ time units. Roughly speaking, from Table 7.1, we can think that $O(n^3 p_2 l) \approx 0.4$ for 32 non-terminal symbols. Also, $O(\frac{n^3 p_2 M}{wd}) \approx 3.85 - 0.4 = 3.45$ when $M = 2097152$. Hence, $O(\frac{n^3 p_2}{wd}) \approx \frac{3.45}{2097152} \approx 1.65 \cdot 10^{-6}$. Thus, we can say that, the latency overhead is 0.4 seconds and the throughput is 1.65 μs per CKY parsing. Further, M must be so large that $M \geq 262144$ to hide the latency overhead.

Table 7.1: The running time (in seconds) of the bitwise-parallel CKY parsing for 32/64 non-terminal symbols and 8192 production rules and the speed-up ratio

	32 non-terminal symbols			64 non-terminal symbols		
M	CPU	GPU	spd-up	CPU	GPU	spd-up
1024	0.679	0.412	1.65	0.637	0.432	1.47
2048	1.30	0.402	3.25	1.25	0.435	2.87
4096	2.60	0.405	6.40	2.43	0.437	5.56
8192	5.18	0.408	12.7	4.85	0.441	11.0
16384	10.4	0.387	26.8	9.71	0.423	22.9
32768	20.7	0.383	54	19.4	0.425	45.6
65536	41.4	0.399	104	38.7	0.459	84.4
131072	82.8	0.434	191	77.4	0.769	101
262144	166	0.754	220	155	0.989	157
524288	331	0.983	337	310	1.94	159
1048576	663	1.93	344	615	3.87	159
2097152	1330	3.85	344	-	-	-

Table 7.2 shows the running time per string of the CKY parsing for strings of length 32 using the BPBC technique. The experiment is performed for 32, 64, 128, 256, and 512 non-terminal symbols and $32, 64, ..., 131072$ binary production rules. Note that the

number p_2 of binary production rules must be $|N| \leq p_2 \leq |N|^3$, where $|N|$ is the number of non-terminal symbols. If $|N| > p_2$, there exists a non-terminal symbol that are not in the left-hand side of a binary production rule. Since a binary production rule in form $A \rightarrow BC$ includes 3 non-terminal symbols, it is not possible to have more than $|N|^3$ distinct binary rules. Thus, these tables does not include the experiment for values p_2 out of this range. From the capacity of the global memory of GeForce GTX TITAN X, we can implement 2097152, 1048576, 524288, 262144, and 131072 CKY tables, respectively, which occupies 8 Gbytes in the global memory of the GPU. Clearly, the running time of the CPU implementation is almost proportional to p_2, because the number of bitwise operations performed is $O(p_2)$. Also, for the same number of binary production rules, the CPU implementation for more non-terminal symbols takes more time. For example, the CKY parsing takes 1310 μs for 32 non-terminal symbols and 16384 binary rules, while it runs 1740 μs if the context-free grammar has 64 non-terminal symbols. This is because the locality of memory access. Roughly speaking, each non-terminal symbol is accessed three times for each binary rule. Hence, we can think that about $\frac{16384 \cdot 3}{32} = 1536$ memory access operations are performed for each of 32 non-terminal symbols. On the other hand, if the context-free grammar has 64 non-terminal symbols, each non-terminal symbols are accessed expected $\frac{16384 \cdot 3}{64} = 768$ times. If the context-free grammar has fewer non-terminal symbols, then each of them are accessed more frequently and the memory cache mechanism work more efficiently.

Similarly, the GPU implementation also takes more time if the context-free grammar has more non-terminal symbols. In addition to the locality of memory access, fewer active threads increase the running time. For example, if the context-free grammar has 32 non-terminal symbols, the global memory of the GPU can store 2097152 CKY tables,

Table 7.2: The running time (per string in microseconds) of the bitwise-parallel CKY parsing for strings of length 32

	32 non-terminal symbols 2097152 strings			64 non-terminal symbols 1048576 strings			128 non-terminal symbols 524288 strings		
p_2	CPU	GPU	spd-up	CPU	GPU	spd-up	CPU	GPU	spd-up
32	4.62	1.24	3.73	-	-	-	-	-	-
64	7.75	1.22	6.38	8.24	2.25	3.66	-	-	-
128	12.4	1.13	11.0	13.5	2.16	6.25	16.6	3.34	4.98
256	23.1	0.956	24.2	23.8	1.58	15.0	27.1	2.77	9.78
512	41.0	0.827	49.5	42.0	1.54	27.3	48.1	3.48	13.8
1024	78.1	0.767	102	76.1	1.48	51.6	83.6	3.99	21.0
2048	156	1.35	115	151	1.80	83.7	156	4.65	33.6
4096	310	5.56	55.7	296	3.36	88.3	295	5.02	58.7
8192	626	1.84	341	588	3.68	160	574	6.71	85.5
16384	1310	3.01	436	1740	6.06	288	1910	10.7	178
32768	2730	6.00	454	3510	10.8	325	3850	18.6	207
65536	-	-	-	7010	20.4	344	7690	34.1	226
131072	-	-	-	14100	39.5	357	16200	65.1	249

	256 non-terminal symbols 262144 strings			512 non-terminal symbols 131072 strings		
p_2	CPU	GPU	spd-up	CPU	GPU	spd-up
256	33.5	5.45	6.15	-	-	-
512	55.2	7.64	7.23	65.6	12.8	5.11
1024	95.0	9.80	9.70	107	16.1	6.63
2048	169	13.1	12.8	184	21.2	8.67
4096	306	18.9	16.2	329	32.7	10.1
8192	588	33.8	17.4	605	54.2	11.2
16384	2010	62.4	32.2	2070	97.6	21.2
32768	4070	125	32.6	4180	188	22.2
65536	8140	248	32.8	8320	370	22.5
131072	16900	501	33.8	16900	736	23.0

which are computed by $\frac{2097152}{32} = 65536$ threads. On the other hand, if the context-free grammar has 64 non-terminal symbols, the global memory of the GPU can store 1048576 CKY tables, which are computed by $\frac{1048576}{32} = 32768$ threads. In general, to maximize the memory access bandwidth, more threads must be invoked at the same time. Hence, the running time per input string is rather increased because fewer CKY tables are computed using fewer threads.

From Table 7.2, the GPU implementation is more than 400 times faster than the CPU implementation when the context-free grammar has 32 non-terminal symbols and

16384/32768 binary production rules. However, if it has many non-terminal symbols and the global memory of the GPU can store fewer CKY tables, we cannot attain high speed-up factor.

Chapter 8

Conclusions

In this dissertation, we have presented efficient GPU implementations for bulk computations, which performs the same algorithm for a lot of instances.

In Chapter 4, we have presented a new Euclidean algorithm for computing the GCD of all pairs of large numbers. 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.

In Chapter 5, we have presented Bitwise Parallel Bulk Computation (BPBC) technique for accelerating the bulk computation. The idea of the BPBC technique is to simulate a combinational logic circuit using bitwise logic operations. Bitwise logic operations compute logical OR, AND, NOT, and XOR for the individual bits of 32-bit word. In this method, we store 32 inputs for the combinational logic circuit into a par-

ticular bit of the words. And we apply the bitwise logic operations corresponding to the combinational logic circuit to the words. By this, we can compute 32 circuits for 32 inputs at the same time.

In Chapter 6, we have presented several techniques for accelerating the simulation of Conway's Game of Life. In particular, we have presented techniques of (1) the states of 32/64 cells are stored in 32/64-bit word (integers) and the next states are computed by the BPBC technique, (2) the states of cells stored in 2 words are updated at the same time by a thread, (3) warp shuffle instruction is used to transfer the current states, and (4) multiple-step simulation is performed to reduce the overhead of data transfer and invoking CUDA kenel. The experimental results show that, the performance of our GPU implementation is 1350×10^9 updates per second for 16K-step simulation of 512K $\times$ 512K cells stored in the SSD. Since Intel Core i7 CPU performs 13.4×10^9 updates per second, our GPU implementation for the Game of Life achieves a speedup factor of 100.

In Chapter 7, we apply the BPBC technique to the CKY parsing. Since the CKY parsing can be computed by a combinational logic circuit, we can perform it for multiple input strings at the same time by simulating the combinational logic circuit using bitwise operations. We also implemented it on the GPU and showed that the GPU implementation can be more than 400 times faster than the CPU implementation.

References

[1] A. Adamatzky. *Game of Life Celluar Automata*. Springer, 2015.

[2] A. V. Aho and J. D. Ullman. *The Theory of Parsing Translation and Compiling*. Prentice Hall, 1972.

[3] M. Bailey and S. Cunningham. A hands-on environment for teaching gpu programming. pages 254–258. ACM, 2007.

[4] D. J. Bernstein. Fast multiplication and its applications. *Algorithmic Number Theory*, (44):325–384, 2008.

[5] J. L. Bordim, O. H. Ibarra, Y. Ito, and K. Nakano. Instance-specifi solutions to accelerate the cky parsing for large context-free grammars. *International Journal on Foundations of Computer Science*, 15(2):403–416, April 2014.

[6] J. L. Bordim, Y. Ito, and K. Nakano. Accelerating the cky parsing using fpgas. *IEICE Transactions on Information ans Systems*, E86-D(5):811–818, 2003.

[7] N. Brunie, S. Collange, and G. Diamos. Simultaneous branch and warp interweaving for sustained gpu performance. In *International Symposium on Computer Architecture*, pages 49–60. ACM, 2012.

[8] J. Chang, O. Ibarra, and M. Palis. Parallel parsing on a one-way array of infinite-stat machines. *IEEE Transactions on Computers*, C-36(1):64–75, 1987.

[9] E. Charniak. *Satistical Language Learning*. MIT Press, 1993.

[10] C. Ciressan, E. Sanchez, M. Rajman, and J. C. Chappelier. An fpga-based coprocessor for the parsing of context-free grammars. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[11] C. Ciressan, E. Sanchez, M. Rajman, and J. C. Chappelier. An fpga-based syntactic parser for real-life almost unrestricted context-free grammars. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 590–594, 2000.

[12] N. Corporation. *NVIDIA CUDA C best practice guide version 3.1*, 2010.

[13] N. Corporation. *GeForce GTX TITAN X*. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x, 2015.

[14] N. Corporation. *NVIDIA CUDA C programming guide version 7.0*, March 2015.

[15] N. Corporation. *Parallel thread execution ISA ver4.2*, March 2015.

[16] N. Corporation. *Tuning CUDA applications for Maxwell*, 2015.

[17] N. Corporation. *CUDA parallel computing platform*. http://www.nvidia.co.jp/object/cuda-parallel-computing-platform-jp.html, 2017.

[18] M. Fisher. *Conway's Game of Life on GPU using CUDA*. http://www.marekfise .com/Projects/Conways-Game-of-Life-on-GPU-using-CUDA, 2013.

[19] N. Fujimoto. High throughput multiple-precision gcd on the cuda architecture. In *International Symposium on Signal Processing and Information Technology*, pages 507–512, December 2009.

[20] M. Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientifi Amerian*, (223):120–133, 1970.

[21] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[22] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *the 21st USENIX Security Symposium*, page 35, August 2012.

[23] W. W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[24] O. H. Ibarra, T. Jiang, and H. Wang. Parallel parsing on a one-way linear array of finite state machines. *Theoretical Computer Science*, 85(1):53–74, 1991.

[25] A. Kasagi, K. Nakano, and Y. Ito. Offline Permutation algorithms on the discrete memory machine, with performance evaluation on the GPU. *IEICE Transactions on Information and systems*, E96-D:2617–2625, 2013.

[26] A. Kasagi, K. Nakano, and Y. Ito. Offline Permutation on the cuda-enabled gpu. *IEICE Transactions on Information and systems*, E97-D:3052–3062, 2014.

[27] K. H. Kim, S. M. Choi, H. Lee, K. L. Man, and Y. S. Han. Parallel cyk membership test on gpus. In *International Conference on Network and Parallel Computing (LNCS 8707)*, pages 157–168, September 2014.

[28] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1997.

[29] S. R. Kosaraju. Speed of recognition of context-free languages by array automata. *SIAM J. on Computers*, 4:331–340, 1975.

[30] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, whit is right. Cryptology ePrint Archive, 2012. 2012/064.

[31] D. Man, K. Uda, Y. Ito, and K. Nakano. Accelerating computation of euclidean distance map using the GPU with efficient memory access. *International Journal of Parallel, Emergent and Distributed Systems*, 28(5):383–406, 2013.

[32] J. C. Martin. *Introduction to languages and the theory of computation (2nd Edition)*. Mac-Graw Hill, 1996.

[33] MathWorks. *Stencil operations on a GPU*. https://www.mathworks.com/examples/parallel-computing/mw/distcomp-ex11684379-stencil-operations-on-a-gpu, 2015.

[34] K. Nakano. Sequential memory access on the unifie memory machine with application to the dynamic programming. In *International Symposium on Computing and Networking*, pages 85–94, December 2013.

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

[36] K. Ogawa, Y. Ito, and K. Nakano. Efficient canny edge detection using a GPU. *Proc. of International Conference on Networking and Computing*, pages 279–280, November 2010.

[37] S. Okamoto, Y. Ito, K. Nakano, and J. L. Bordim. Thorough evaluation of GPU shared memory load and store instructions. In *International Symposium on Computing and Networking*, pages 614–616. IEEE CS Press, December 2015.

[38] K. S. Perumalla and B. G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on gpus. In *Spring simutation Multiconference*, pages 116–123. Society for Computer Simulation International, 2008.

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

[40] Y. Sakakibara, M. Brown, R. Hughey, I. S. Mian, K. Sjolander, R. C. Underwood, and D. Haussler. Stochastic context-free grammars for trna modeling. *Nucleic Acids Research*, 22:5112–5120, 1994.

[41] K. Scharfglass, D. Weng, J. White, and C. Lupo. Breaking weak 1024-bit rsa keys with cuda. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 207–212, December 2012.

[42] J. Stein. Computational problems associated with racah algebra. *Journal of Computational Physics*, (1(3)), Febrary 1967.

[43] D. Takafuji, K. Nakano, and Y. Ito. A cuda c program generator for bulk execution of a sequential algorithm. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 178–191, August 2014.

[44] K. Tani, D. Takafuji, K. Nakano, and Y. Ito. Bulk execution of oblivious algorithms on the unifie memory machine, with gpu implementation. In *International Parallel and Distributed Processing Symposium Workshops*, pages 586–595, May 2014.

[45] N. Tsuda. *Acceleration of Game of Life by the bit operation (bit-board)*. http://vivi.dyndns.org/tech/games/LifeGame.html, 2012.

[46] M. P. van Lohuizen. Survey on parallel context-free parsing techniques. Technical Report IMPACT-NLI-1997-1, Delft University of Technology, 1997.

[47] J. R. White. *PARIS: A parallel RSA-prime inspection tool*. PhD thesis, California Polytechnic State University - San Luis Obispo, June 2013.

[48] Y. Yang, Z. Guan, H. Sum, and Z. Chen. Accelerating rsa with fine-graine parallelism using gpu. In *Information Security Practice and Experience (LNCS)*, volume 9065, pages 454–468. Springer, 2015.

[49] Y. Yi, C. Y. Lai, S. Petrov, and K. Keutzer. Efficient parallel cky parsing on gpus. In *International Conference on Parsing Technologies*, pages 175–185, 2011.

Acknowledgment

First and foremost, I would like to express my most sincere gratitude to my supervisor, Professor Koji Nakano for his continuous encouragement, advice and support. His knowledge and research experience has been a great help in my study. Without his encouragement and suggestion, this dissertation would not have been completed. As a supervisor, he taught me skills and knowledges that will provide the value in my career.

I would also express my sincere appreciation to Associate Professor Yasuaki Ito for his continuous guidance in whole period of my study.

I would express my appreciation to Professor Satoshi Fujita for reviewing my dissertation.

I would express my heartfelt appreciation to Assistant Professor Daisuke Takafuji for his continuous support in my study.

I thank all members of computer system laboratory. They were very kind and keen to help. I am deeply grateful to all the faculty members of the Department of Information Engineering of Hiroshima University.

Last, I wish to express my thanks to my family who always encouraged me.

List of publications

Journals

[J-1] Toru Fujita, Koji Nakano, and Yasuaki Ito, Bulk execution of Euclidean algorithms on the CUDA-enabled GPU, International Journal of Networking and Computing, Vol. 6 No. 1, pp. 42–63, January 2016.

[J-2] Toru Fujita, Koji Nakano, and Yasuaki Ito, Fast Simulation of Conway's Game of Life using Bitwise Parallel Bulk Computation on a GPU, International Journal of Foundations of Computer Science, to appear.

International Conferences

[C-1] Toru Fujita, Koji Nakano, and Yasuaki Ito, Bulk GCD Computation Using a GPU to Break Weak RSA Keys, Proc. of International Parallel and Distributed Processing Symposium Workshops, pp. 385–394, May 2015.

[C-2] Toru Fujita, Daigo Nishikori, Koji Nakano, and Yasuaki Ito, Efficient GPU implementations for the Conway's Game of Life, Proc. of International Symposium on Computing and Networking (CANDAR), pp. 11–20, December 2015.

[C-3] Toru Fujita, Koji Nakano, and Yasuaki Ito, Bitwise Parallel Bulk Computation on the GPU, with Application to the CKY Parsing for Context-free Grammars, Proc.of International Parallel and Distributed Processing Symposium Workshops, pp. 589–598, May 2016.