International Journal of Foundations of Computer Science
© World Scientific Publishing Company

Fast Simulation of Conway's Game of Life using Bitwise Parallel Bulk Computation on a GPU

Toru Fujita, Koji Nakano, and Yasuaki Ito

Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashihirosima 739-8527, Japan

Received (Day Month Year)
Accepted (Day Month Year)
Communicated by (xxxxxxxxxx)

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, each of 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 if it is alive and two neighbors are alive. The main contribution of this paper is to develop several acceleration techniques for simulating the Game of Life using a GPU as follows: (1) the states of 32/64 cells in 32/64-bit words (integers) and the next states are computed by the Bitwise Parallel Bulk Computation (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) multi-step simulation is performed to reduce the overhead of data transfer and invoking CUDA kernel. The experimental results show that, the performance of our GPU implementation using GeForce GTX TITAN X is 1350×10^9 updates per second for 16K-step simulation of 512K× 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. Thus, these techniques work very efficiently on a GPU.

Keywords: Cellular automaton, parallel algorithms, CUDA, GPGPU

1. Introduction

Conway's Game of Life was created by John Horton Conway, a mathematician at Gonville and Caius College of the University of Cambridge [1, 6]. *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 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.

The GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [7, 23, 27]. Latest GPUs

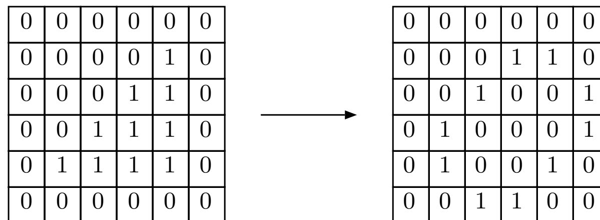

Fig. 1. One-step simulation of the Game of Life

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 [7, 8, 9, 24, 25]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [15, 18], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [11], since they have thousands of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [18]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-12 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *bank conflicts* of the shared memory access and *coalescing* of the global memory access [8, 9, 11, 13, 14, 15]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the shared memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the throughput between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory. Also, the latency of the global memory access is several hundred clock cycles, while that of the shared memory access is around 10 clock cycles [21]. Hence, we should minimize the memory access to the global memory to maximize the performance. Further, CUDA-enabled GPUs with Kepler [16] and Maxwell [20] architectures support *warp shuffle* instructions that directly exchanges data stored in registers of threads in the same warp [18]. It is faster than inter-thread communication implemented by reading/writing the shared memory. Thus, we should use warp shuffle instructions whenever possible. Actually, appropriate use of warp

shuffle instructions can accelerate the computation [3, 28].

In our previous paper [5], we have introduced *the Bitwise Parallel Bulk Computation (BPBC) technique* to accelerate the computation. The BPBC technique supports ultimate fine grained bit parallelism and thus can achieve very high acceleration over the straightforward sequential computation. The BPBC technique simulates a combinational logic circuit for a lot of instances at the same time using the bitwise logical operations. More formally, let f be a function computed by a combinational logic circuit and $X_0, X_1, \ldots, X_{M-1}$ be the M inputs. By the BPBC technique $f(X_0)$, $f(X_1)$, ..., $f(X_{M-1})$ can be computed very efficiently. The idea of the BPBC technique is

- to store a bit of each input instance in a particular bit of words of data, say 32-bit integers, and
- to simulate the combinational logic circuit for 32 input vectors at the same time by bitwise logic operations supported by computing devices such as CPUs and GPUs.

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 file 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 of cells. More specifically, we focus on accelerating simulation that computes the values of all cells in the universe after T steps for a given T.

The main contribution of this paper is to develop several acceleration techniques for simulating the Game of Life using a GPU as follows.

(1) the states of 32/64 cells in 32/64-bit words (integers) and the next states are computed by the Bitwise Parallel Bulk Computation (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) multi-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 [4] 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 [26]. Also, the simulation of the Game of Life can be done by *stencil codes*,

a class of kernels updating elements in an array according to some fixed 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 [12].

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 [2, 22], 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 states of cells [4]. 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 [4] achieved 24.7×10^9 updates per second on GeForce GTX 480 GPU. We will show that our GPU implementation achieves 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 1401MHz and 1000MHz, 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× 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. To evaluate the efficiency of the proposed method, we compare the proposed GPU implementation with a straightforward GPU implementation method without acceleration technique proposed in this paper. The straightforward method can compute 14.8×10^9 updates per second for a $16K \times 16K$ ($2^{14} \times 2^{14}$) array with the same GPU. Thus, we achieved a speed-up factor of 134 using the proposed method.

This paper is organized as follows. In Section 2, we first briefly explain the GPU architecture and CUDA programming model necessary to understand GPU implementations of the Game of Life for the reader's benefit. Section 3 defines the Game of Life formally and shows straightforward implementations. In Section 4, we introduce the BPBC technique and show how we can apply it to simulation of the Game of Life. Section 5 shows that we can reduce the number of bitwise operations by updating states in two words at the same time. Section 6 presents an idea of multiple-step simulation, which copies the states of cells to the shared memory and repeats the simulation several times. Using this simulation technique, we can reduce

the number of CUDA kernel calls and the total amount of global memory access of the GPU. In Section 7, we show that simulation can be performed by a warp shuffle instruction of the GPU. Section 8 shows how we simulate the Game of Life for a very large universe stored in the SSD. Finally, Section 9 shows experimental results. Section 10 concludes our work.

2. GPU architecture and CUDA programming model

This section briefly describes the GPU architecture and the CUDA programming model necessary to understand GPU implementations of the Game of Life. Please see [18] for the details.

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*, *a shared memory* and *a register file*. The GPU processor is connected to *an off-chip memory*. For example, GeForce GTX TITAN X has 24 SMs [a] with 128 processor cores, a 96Kbyte shared memory, and a register file with 64K 32bit registers each. The off-chip memory can be accessed by all processor cores in all SMs, while the shared memory can be accessed only by processor cores in the same SM. Also, registers in a register file are assigned to a processor core, and they can be accessed only by the 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 [21] and that of registers in the register file is smaller. Hence, to accelerate the computation, we should minimize the global memory access. We should also use registers whenever possible.

When we develop programs running on GPUs, we can use CUDA programming model illustrated in Figure 2 (2) to support scalability. We assume CUDA Compute Capability 5.2, which is available for GeForce GTX TITAN X [17]. 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 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*,

[a]Since the architecture of GeForce GTX TITAN X is called Maxwell, its SM is particularly termed Maxwell Streaming Multiprocessor (SMM) [19].

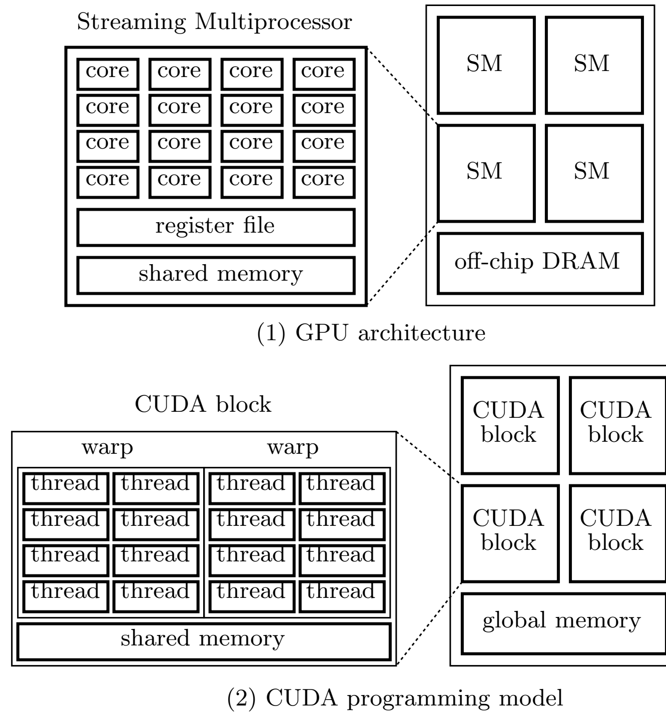

Fig. 2. GPU architecture and CUDA programming model

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 [18], 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 than 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 is 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 the 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 executes 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. Hence, threads in a CUDA block should perform *coalesced access* when they access the global memory [8, 15]. 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 conflicts* [10, 15, 21] by a warp should be avoided to maximize the shared memory access performance.

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 supports *warp shuffle* instructions that permit exchanging of data stored in registers in threads in a warp. The data exchange occurs at the same time for all active threads in a 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.

3. 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 $u_0, u_1, \ldots$ 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) defined as follows:

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

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

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

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

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

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 $u_0(i,j)$ $(0 \leq i,j \leq \sqrt{n}-1)$ 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 3, 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.

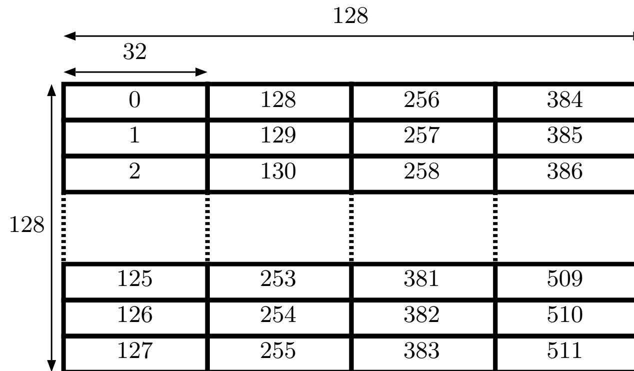

Fig. 3. Bit-per-cell arrangement of 128×128 universe of 32-bit words with column-major order arrangement

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.

4. Bitwise Parallel Bulk Computation (BPBC) technique and application to the Game of Life

The main purpose of this section is to show the idea of Bitwise Parallel Bulk Computation (BPBC). This idea works well not only for a multi-core machine but also for a single CPU. We also show how the BPBC technique can be applied to simulation of the Game of Life.

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 $X_0, X_1, \ldots, X_{d-1}$ be d inputs of m bits each. Suppose that we want to compute $f(X_0), f(X_1), \ldots, f(X_{d-1})$. 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.

As an example of application of the BPBC technique, we show *the bitwise summing technique*, which compute the bitwise sums. 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 $f(X_0), f(X_1), \ldots, f(X_{d-1})$ 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 $a_ib_ic_i$ denotes three bits of each X_i $(0 \leq i \leq d-1)$. We assume that we have three d-bit words $A = a_0a_1\cdots a_{d-1}$, $B = b_0b_1\cdots b_{d-1}$, and $C = c_0c_1\cdots c_{d-1}$. We want to compute $Y = y_0y_1\cdots y_{d-1}$ and $Z = z_0z_1\cdots z_{d-1}$ 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:

$$Y \leftarrow (A \wedge B) \vee (B \wedge C) \vee (C \wedge A),$$
$$Z \leftarrow A \oplus B \oplus C.$$

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

$$T \leftarrow A \oplus B,$$
$$Z \leftarrow T \oplus C,$$
$$Y \leftarrow (A \wedge B) \vee (T \wedge C).$$

We use this technique to compute the number of alive cells using fewer bitwise operations.

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 (1) and (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 [26].

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 4 shows the computation to obtain the next states of 4 cells in I. We need $2 \cdot 4 + 6 = 14$ neighboring cells are necessary for this computation. We first store neighboring cells in eight 4-bit words $A, B, \ldots, H$ as illustrated in Figure 4. After that, we compute the bitwise sums as shown in Figure 4 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 $A, B, \ldots, H$ is 2 and 3, respectively. Clearly, using I_2, I_3, and the current value of I, we can compute the next state 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 $([A\text{-}H]_3, [A\text{-}H]_2, [A\text{-}H]_1, [A\text{-}H]_0)$ denote the bitwise sums of each bit of $A, B, \ldots, H$. Also, let $[A\text{-}H]_{23} = [A\text{-}H]_2 \vee [A\text{-}H]_3$. Clearly, $I_2 = 1$ if $([A\text{-}H]_{23}, [A\text{-}H]_1, [A\text{-}H]_0) = (0,1,0)$ and $I_3 = 1$ if $([A\text{-}H]_{23}, [A\text{-}H]_1, [A\text{-}H]_0) = (0,1,1)$. Hence, we can compute I_2 and I_3 from $([A\text{-}H]_{23}, [A\text{-}H]_1, [A\text{-}H]_0)$.

We will show Algorithm SINGLE-WORD that computes the next states of I using this idea. We first 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 $([A\text{-}D]_2, [A\text{-}D]_1, [A\text{-}D]_0)$. This can be done by computing the sums from the least significant bit. Similarly, we obtain the sum $([E\text{-}H]_2, [E\text{-}H]_1, [E\text{-}H]_0)$. Finally, we compute the sum of $([A\text{-}D]_2, [A\text{-}D]_1, [A\text{-}D]_0)$ and $([EH]_2, [EH]_1, [EH]_0)$ and obtain three bits $([AH]_{23}, [AH]_1, [AH]_0)$. 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 state

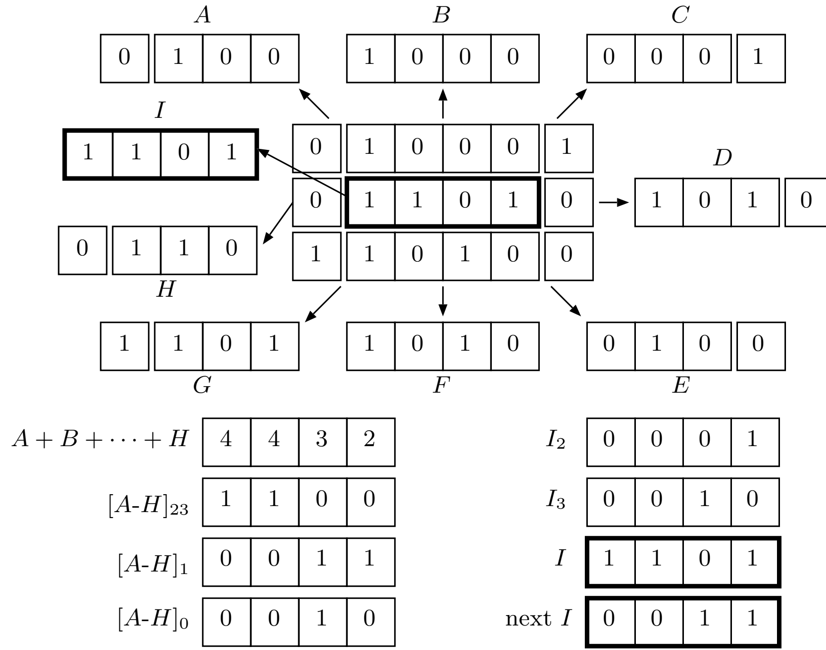

Fig. 4. The computation of the next states of 4 cells in a 4-bit word by Algorithm SINGLE-WORD

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. $([GH]_1, [GH]_0) \leftarrow (G \wedge H, G \oplus H)$

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

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

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

8. $[EH]_0 \leftarrow [EF]_0 \oplus [GH_0]$
9. $[EH]_1 \leftarrow [EF]_1 \oplus [GH]_1 \oplus ([EF]_0 \wedge [GH]_0)$
10. $[EH]_2 \leftarrow [EF]_1 \wedge [GH]_1$

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

11. $[A\text{-}H]_0 \leftarrow [A\text{-}D]_0 \oplus [E\text{-}H]_0$
12. $X \leftarrow [A\text{-}D]_0 \wedge [E\text{-}H]_0$
13. $Y \leftarrow [A\text{-}D]_1 \oplus [E\text{-}H]_1$
14. $[A\text{-}H]_1 \leftarrow X \oplus Y$

15. $[A\text{-}H]_{23} \leftarrow [A\text{-}D]_2 \vee [E\text{-}H]_2 \vee ([A\text{-}D]_1 \wedge [E\text{-}H]_1) \vee (X \wedge Y)$
// $(I, I_2, I_3) \leftarrow (I, [A\text{-}H]_{23}, [A\text{-}H]_1, [A\text{-}H]_0)$
17. $Z \leftarrow \overline{[A\text{-}H]_{23}} \wedge [A\text{-}H]_1$
18. $I_2 \leftarrow \overline{[A\text{-}H]_0} \wedge Z$
19. $I_3 \leftarrow [A\text{-}H]_0 \wedge Z$
20. $I \leftarrow (I \wedge I_2) \vee I_3$

Note that, when we compute $([A\text{-}D]_2, [A\text{-}D]_1, [A\text{-}D]_0) \leftarrow ([AB]_1, [AB]_0) + ([CD]_1, [CD]_0)$, the values of $([AB]_1, [AB]_0)$ and $([CD]_1, [CD]_0)$ can not be $(1, 1)$. Hence, $[A\text{-}D]_2$ 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, $([A\text{-}D]_2, [A\text{-}D]_1, [A\text{-}D]_0) \leftarrow ([AB]_1, [AB]_0) + ([CD]_1, [CD]_0)$. This computation is executed twice, and thus, 10 binary operations are performed. For computing $([A\text{-}H]_{23}, [A\text{-}H]_1, [A\text{-}H]_0)$, 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 $4\times2+2\times5+9+5+2 = 34$. Hence we have,

Lemma 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 $\frac{n}{64\cdot32}$ 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 $A, B, \ldots, H$ from these words, and computes the next state 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 complete the T-step simulation.

5. 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 execute 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 5. We compute the next states of cells in two words K and L in the figure 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 specifically, we compute $([A\text{-}EIJL]_{23}, [A\text{-}EIJL]_1, [A\text{-}EIJL]_0)$ and $([D\text{-}K]_{23}, [D\text{-}K]_1, [D\text{-}K]_0)$. 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 $(K, [A\text{-}EIJL]_{23}, [A\text{-}EIJL]_1, [A\text{-}EIJL]_0)$, we can compute (K, K_2, K_3) where K stores 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 $(L, [D\text{-}K]_{23}, [D\text{-}K]_1, [D\text{-}K]_0)$.

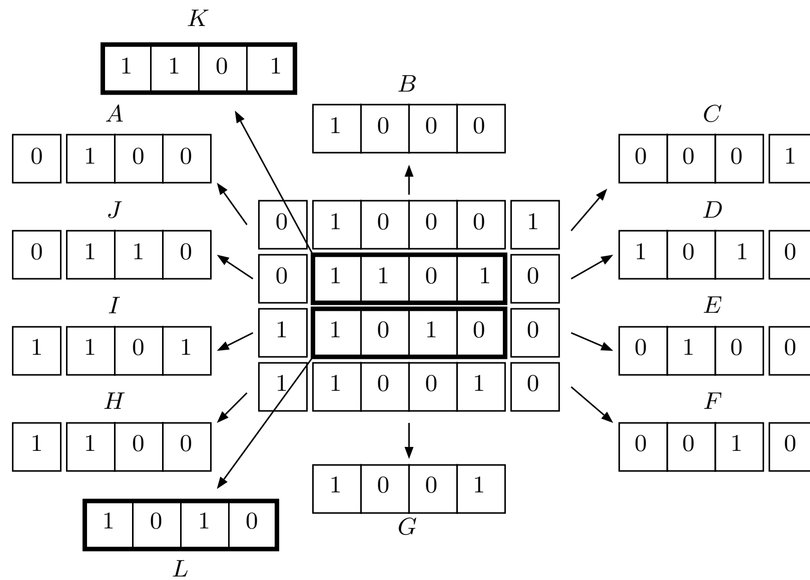

Fig. 5. Illustrating 12 words for computing next states of cells in two words by Algorithm DOUBLE-WORD

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. $([A\text{-}EIJL]_{23}, [A\text{-}EIJL]_1, [A\text{-}EIJL]_0)$
    $\leftarrow ([ABCL]_2, [ABCL]_1, [ABCL]_0) + ([DEIJ]_2, [DEIJ]_1, [DEIJ]_0)$
11. $([D\text{-}K]_{23}, [D\text{-}K]_1, [D\text{-}K]_0)$
    $\leftarrow ([FGHK]_2, [FGHK]_1, [FGHK]_0) + ([DEIJ]_2, [DEIJ]_1, [DEIJ]_0)$
12. $(K, K_2, K_3) \leftarrow (K, [A\text{-}EIJL]_{23}, [A\text{-}EIJL]_1, [A\text{-}EIJL]_0)$
13. $(L, L_2, L_3) \leftarrow (L, [D\text{-}K]_{23}, [D\text{-}K]_1, [D\text{-}K]_0)$

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 two 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 2. *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 global memory, we can implement the algorithm for Lemma 2 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. Multiple-step simulation using the shared memory

We can accelerate the computation if multiple steps simulation is performed on the shared memory. More specifically, 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 specifically, 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 Figure 6.

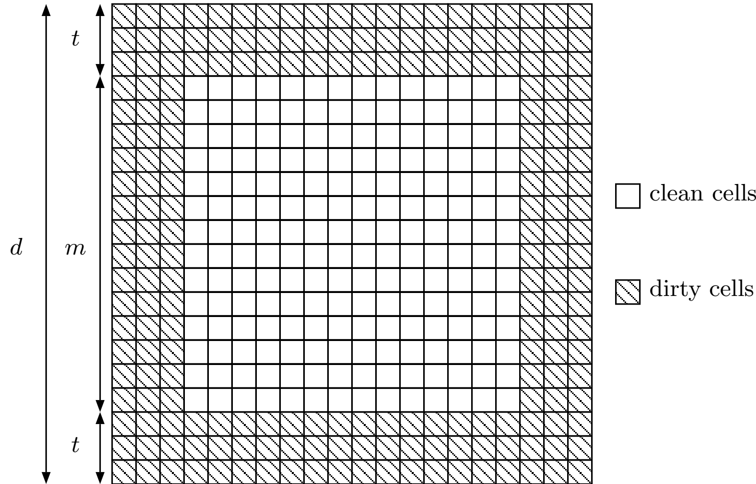

Fig. 6. Clean and dirty cells

To simulate multiple steps of all cells, the $\sqrt{n} \times \sqrt{n}$ 2-dimensional array in the global memory is partitioned into $\frac{\sqrt{n}}{m} \times \frac{\sqrt{n}}{m}$ *patches* of size $m \times m$ each as illustrated in Figure 7. Each patch 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 $m \times m$ patch 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 be done by $\frac{T}{t}$ CUDA kernel calls.

We can observe that, we should select an appropriate value of t ($1 \leq t < \frac{d}{2}$) for fixed n and d that minimizes the running time. For simplicity, we assume that the cost for computing the next state 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 patch of size $m \times m$ is $t + c$. Hence, the cost of T-step simulation of $\sqrt{n} \times \sqrt{n}$ cells is:

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

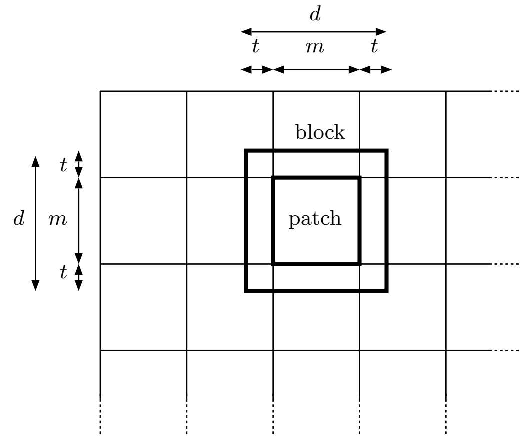

Fig. 7. A $d \times d$ patch and an $m \times m$ block in a large 2-dimensional array

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

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

Clearly, t is an increasing function of c. 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 8. For example thread 1 is responsible for updating 128 cells 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 [19]. 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 a 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 conflicts as illustrated in Figure 9 (1). If 32 threads access rows 2, 4, 6, ..., 32, then two memory access operations are performed to the same bank as illustrated in the Figure. To avoid such bank conflicts, we use shift arrangement as illustrated in Figure 9 (2), in which elements are shifted by one in every two rows.

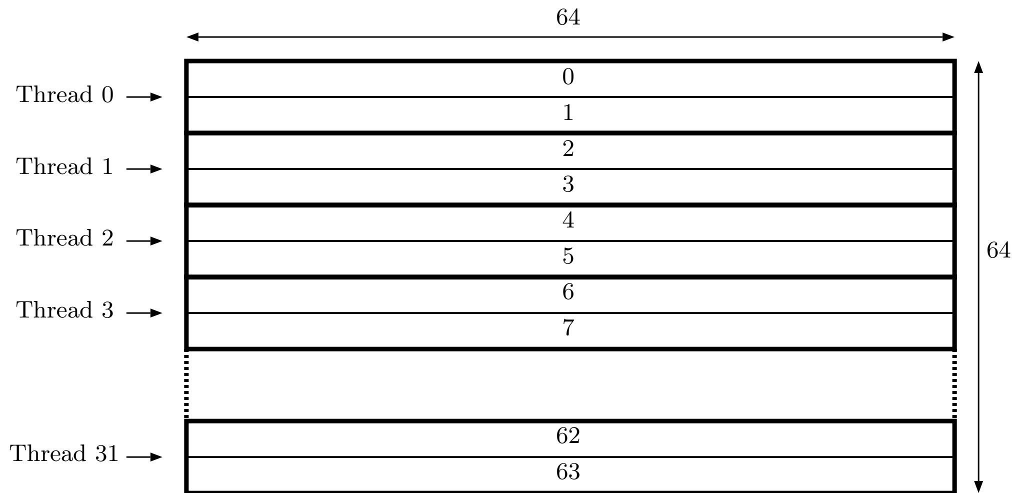

Fig. 8. Words accessed by threads executing Algorithm DOUBLE-WORD

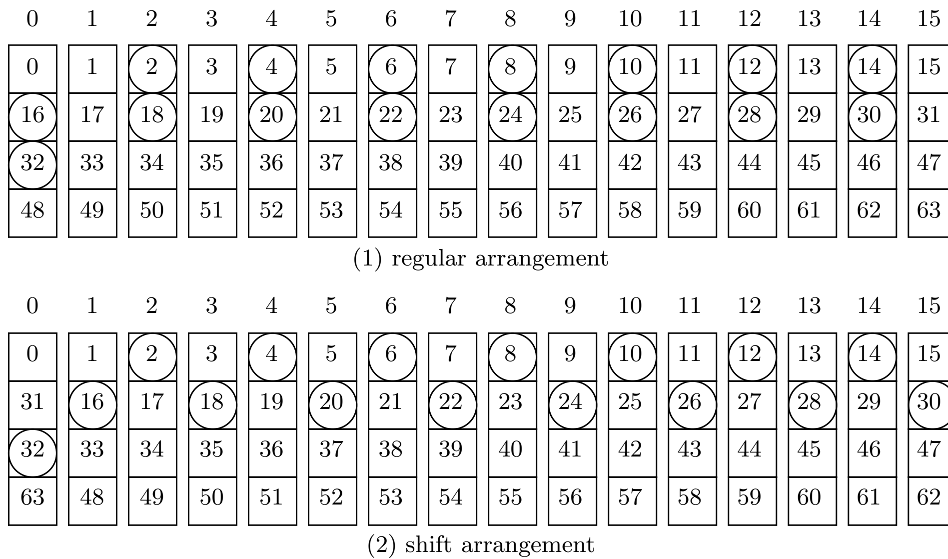

Fig. 9. Regular arrangement and shift arrangement

7. Further acceleration using warp shuffle

The memory access latency of the shared memory is not small [21]. 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, 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 instruction, which copies registers of threads in the same warp, as illustrated in Figure 10. 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.

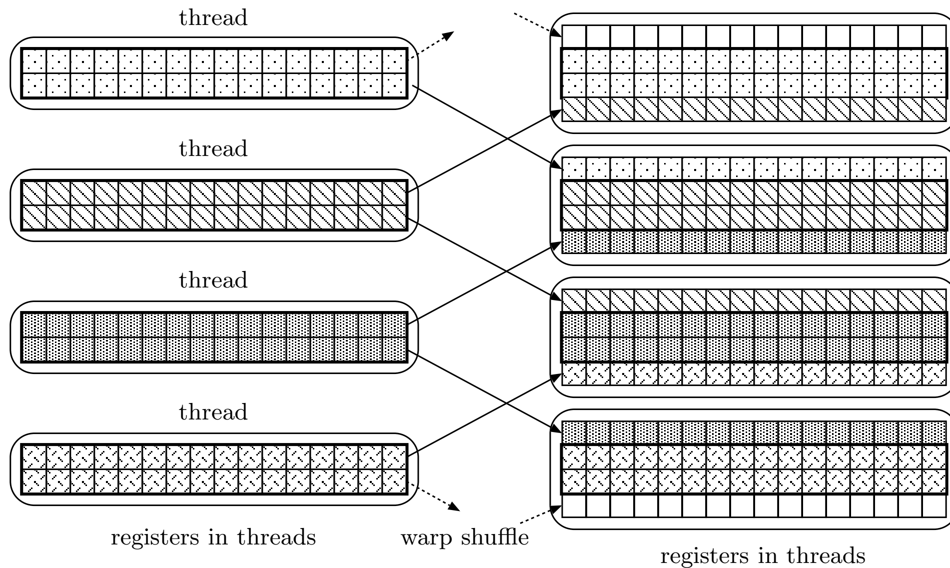

Fig. 10. Copying words storing cells using a warp shuffle instruction

8. 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.

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. For this purpose, similarly to multi-step simulation shown in Section 6, 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 specifically, 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.

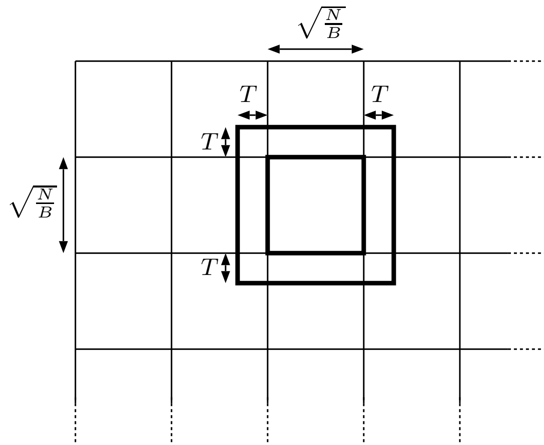

Fig. 11. Partition of a large universe of size $\sqrt{N} \times \sqrt{N}$ into B sub-universes of size $\sqrt{\frac{N}{B}} \times \sqrt{\frac{N}{B}}$

9. Experimental results

The main purpose of this section 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 cells. In other words, a 2-dimensional array of $16K \times 16K$ unsigned characters are used and evaluated formulas (1) and (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

specifically, 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 states of cells by formulas (1) and (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 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 $1K \cdot 16K \cdot 16K/921.7 \approx 0.298 \times 10^9$ updates per second.

Table 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 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 a t-step transition.

To find 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 $\frac{16K}{m} \times \frac{16K}{m}$ patches 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 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 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, implementations using Algorithm DOUBLE-WORD are faster than the corresponding implementations of Algorithm SINGLE-WORD. From the table, 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 1). Hence, multi-step simulation by the GPU can accelerate the computation by a speedup factor of 3.

Table 3 shows the running time for 16K-step simulation of 512K×512K cells. The universe of 256G cells (that is, 32Gbytes) is stored in the SSD. We have partitioned the universe into 16 sub-universes of size 128K×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 8-step, 64×64 block, DOUBLE-WORD algorithm, which is the best configuration from Table 2. Simula-

tion 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 $T = 8K$. We have evaluated the running time for 16K-step simulation of 512K× 512K cells using a Core i7 CPU for T-step simulation. We found that the total running time is minimized when $T = 128$ using Algorithm DOUBLE-WORD, 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 3. The running time (in seconds and 10^9 updates per second) for 16K-step simulation of 512K× 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

10. Conclusion

This paper 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 in 32/64-bit word (integers) and the next states are computed by the Bitwise Parallel Bulk Computation (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) multi-step simulation is performed to reduce the overhead of data transfer and invoking CUDA kernel. The experimental results show that the simulation of Conway's Game of Life is efficiently accelerated using these techniques. Further, we have presented an implementation for the simulation of a very large universe stored in the SSD.

References

[1] A. Adamatzky, *Game of Life Cellular Automata* (Springer, 2015).

[2] M. Bailey and S. Cunningham, A hands-on environment for teaching GPU programming, *Proc. of SIGCSE Technical Symposium on Computer Science Education*, (ACM, 2007), pp. 254–258.

[3] N. Brunie, S. Collange and G. Diamos, Simultaneous branch and warp interweaving for sustained GPU performance, *Proc. International Symposium on Computer Architecture*, (ACM, 2012), pp. 49–60.

[4] M. Fisher, Conway's Game of Life on GPU using CUDA, http://www.marekfiser.com/Projects/Conways-Game-of-Life-on-GPU-using-CUDA (Mar 2013).

[5] T. Fujita, K. Nakano and Y. Ito, Bitwise parallel bulk computation on the gpu, with application to the cky parsing for context-free grammars, *to apper in Proc. of International Parallel and Distributed Processing Symposium Workshops*, (IEEE CS Press, May 2016).
[6] M. Gardner, Mathematical games: The fantastic combinations of John Conway's new solitaire game "life", *Scientific American* **223** (Oct. 1970) 120–123.
[7] W. W. Hwu, *GPU Computing Gems Emerald Edition* (Morgan Kaufmann, 2011).
[8] 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* **Vol. E96-D** (Dec. 2013) 2617–2625.
[9] A. Kasagi, K. Nakano and Y. Ito, An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation, *Proc. of International Conference on Parallel Processing (ICPP)*, (IEEE CS Press, Oct. 2013), pp. 1–10.
[10] A. KASAGI, K. NAKANO and Y. ITO, Offline permutation on the cuda-enabled gpu, *IEICE TRANSACTIONS on Information and Systems* **E97-D** (Dec. 2014) 3052–3062.
[11] D. Man, K. Uda, H. Ueyama, Y. Ito and K. Nakano, Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs, *International Journal of Networking and Computing* **1** (July 2011) 260–276.
[12] MathWorks, Stencil operations on a GPU, https://www.mathworks.com/examples/parallel-computing/398-stencil-operations-on-a-gpu (2015).
[13] K. Nakano, Optimal parallel algorithms for computing the sum, the prefix-sums, and the summed area table on the memory machine models, *IEICE Trans. on Information and Systems* **E96-D**(12) (2013) 2626–2634.
[14] K. Nakano, Simple memory machine models for GPUs, *International Journal of Parallel, Emergent and Distributed Systems* **29**(1) (2014) 17–37.
[15] NVIDIA Corporation, NVIDIA CUDA C best practice guide version 3.1 (2010).
[16] NVIDIA Corporation, NVIDIA's next generation CUDA compute architecture: Kepler GK110, whitepaper (2012).
[17] NVIDIA Corporation, GeForce GTX TITAN X (2015), http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x.
[18] NVIDIA Corporation, NVIDIA CUDA C programming guide version 7.0 (Mar 2015).
[19] NVIDIA Corporation, Tuning CUDA applications for Maxwell (2015).
[20] NVIDIA Corportaion, NVIDIA GeForce GTX980 whitepaper (2014).
[21] S. Okamoto, Y. Ito, K. Nakano and J. L. Bordim, Thorough evaluation of GPU shared memory load and store instructions,, *Proc. of International Symposium on Computing and Networking*, (IEEE CS Press, Dec. 2015), pp. 614–616.
[22] K. S. Perumalla and B. G. Aaby, Data parallel execution challenges and runtime performance of agent simulations on GPUs, *Proc. of Spring Simulation Multiconference*, (Society for Computer Simulation International, 2008), pp. 116–123.
[23] V. Podlozhnyuk, Image convolution with CUDA (July 2007).
[24] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk and W. mei W. Hwu, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, *Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, (ACM, 2008), pp. 73–82.
[25] P. Steffen, R. Giegerich and M. Giraud, GPU parallelization of algebraic dynamic programming, *Proc. of International Conference on Parallel Processing and Applied Mathematics: Part II*, (Springer, Sept. 2009), pp. 290–299.
[26] N. Tsuda, Acceleration of Game of Life by the bit operation (bit-board), in Japanese,

http://vivi.dyndns.org/ tech/games/LifeGame.html (December 2012).
[27] A. Uchida, Y. Ito and K. Nakano, Fast and accurate template matching using pixel rearrangement on the GPU, *Proc. of International Conference on Networking and Computing*, (IEEE CS Press, Dec. 2011), pp. 153–159.
[28] Y. Yang, Z. Guan, H. Sun and Z. Chen, Accelerating RSA with fine-grained parallelism using GPU, *Proc. of Information Security Practice and Experience (LNCS)*, **9065**, (Springer, 2015), pp. 454–468.