Efficient GPU implementations for the Conway's Game of Life

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

***Abstract*—The 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 or three neighbors are alive. The main contribution of this paper is to develop several acceleration techniques for simulating the Game of Life. The key techniques for the simulation is to store a block of cells in registers of 32 threads in a warp of a CUDA block and to perform multiple-step simulation. We use a warp shuffle instruction, which allows us to exchange data stored in registers of threads in a warp, to transfer the current states stored in registers of other threads necessary to compute the next states. Further, since multiple-step simulation is performed, the number of CUDA kernel calls can be decreased. The experimental results show that, the best configuration of our GPU implementation can perform 1024-step simulation of 16384×16384 cells in 0.163 seconds on GeForce GTX TITAN X GPU. The best sequential algorithm using a single core of Intel Xeon X7460 CPU runs 58.3 seconds. Hence, our best GPU implementation has achieved a speed-up factor of 357 over the CPU implementation.**

Keywords*—*Conway's Game of Life, cellular automaton, parallel algorithms, CUDA, GPGPU

I. Introduction

The GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1]–[4]. 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 [1], [5]–[10]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [11], [12], 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 [13], 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* [11]. 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 [9], [10], [12]–[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 [16]. Hence, we should minimize the memory access to the global memory to maximize the performance. Further, CUDA-enabled GPUs with Kepler [17] and Maxwell [18] architectures support *warp shuffle* instructions that directly exchanges data stored in registers of threads in the same warp [11]. 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 [19], [20].

The Conway's Game of Life was created by John Horton Conway, a mathematician at Gonville and Caius College of the University of Cambridge [21], [22]. 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:

1) (die) An alive cell becomes dead if it has fewer than two or more than three alive neighbors.
2) (born) A dead cell becomes alive if it has three alive neighbors.
3) (keep alive) An alive cell keeps alive if it has two or three alive neighbors.
4) (keep dead) A dead cell keeps dead if it has fewer than three or more than three dead neighbors.

Figure 1 illustrates the rules of the Game of Life. Originally the Conway's Game of Life assumes that the size of the 2-dimensional array is infinite. However, to store all the states in the memory, we assume that the universe is finite and the 2-dimensional array has $\sqrt{n} \times \sqrt{n}$ cells. Clearly, some neighbors of cells in the boundary of the 2-dimensional array do not exist. Sometimes, it is assumed that the states of such non-existent neighbors are always dead. In this paper, we assume that the

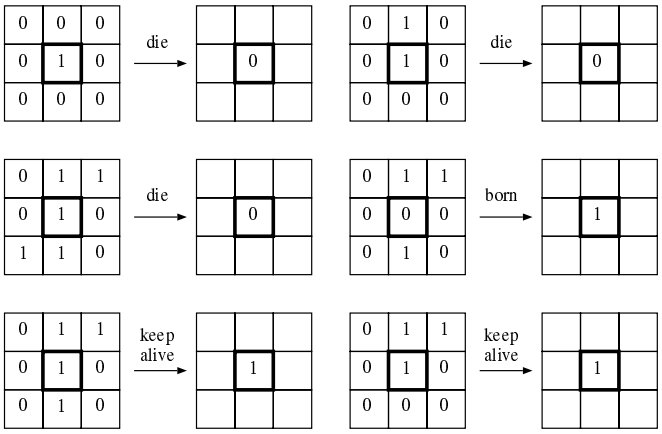

Fig. 1. The rules of the Game of Life

2-dimensional array is wrapper around to handle the boundary case. For example, the left neighbor of a cell in the leftmost column is the rightmost cell in the same row.

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 and a 32-bit integer. However, for accelerating the simulation, it makes sense to use *bit-per-cell* arrangement [23] 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 32 cells. A very sophisticated way to compute the next states of cells stored in a word by bitwise operations has been presented [24]. 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 [25].

We are interested in how we can accelerate the simulation of the Game of Life using CUDA-enabled GPUs. In many cases, simulation of the Game of Life means that 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 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.

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 [26], [27], 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 [23]. Our implementations use the same technique. In addition, 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 this 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 [23] achieved 2.47×10^{10} updates per second on GeForce GTX 480 GPU. Our implementation performs 1024-step simulation of the Game of Life with 2^{28} cells in 0.163 seconds on GeForce GTX TITAN X GPU. Hence, it achieves 1.69×10^{12} updates per second and more than 68 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 GPUs is taking into account.

This paper is organized as follows. In Section II, we first briefly explain the GPU architecture and CUDA programming model to understand GPU implementations of the Game of Life. Section III defines the Game of Life formally, and Section IV shows basic techniques to accelerate the simulation of the Game of Life including bit-per-cell arrangement and simulation using bitwise logical operations. Section IV also shows a straightforward implementation using the basic techniques on GPUs using the global memory. In Section V, we present our new techniques for accelerating the simulation. We show that we can reduce the total number of bitwise operations if each thread computes the next states of cells in two words at the same time. We also present an idea of multiple-step simulation, which copies the states of cells to the shared memory and repeats the simulation several times. For further acceleration, we can store the states in registers of threads for multiple-step simulation, and the simulation can be performed by a warp shuffle instruction. Finally, Section VI shows experimental results. More specifically, we have implemented simulation algorithms of the Game of Life on the CPU and the GPU, and evaluated the running time. The experimental results show that our implementation is 357 times faster than the CPU implementation. Section VII concludes our work.

II. 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 [11] 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*, *the shared memory* and *the register file*. The GPU processor is connected to *an off-chip memory*. For example, GeForce GTX TITAN X has 16 SMs[1] with 192 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,

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

say several hundred clock cycles. The memory access latency of the shared memory is around 10 cycles [16] 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 to support scalability. We assume that CUDA Compute Capability 5.2, which is available for GeForce GTX TITAN X [28]. 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*, 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 [11], 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 terminates, 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, they 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 is `__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. 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* 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 permits 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.

III. Conway's Game of Life and an conventional implementation

The universe of *the 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 2-dimensional arrays such that u_0 stores the initial states, and each u_t ($t \geq 1$) is an array of cells 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 2-dimensional array is wrap 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, that is,

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

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

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

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

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

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

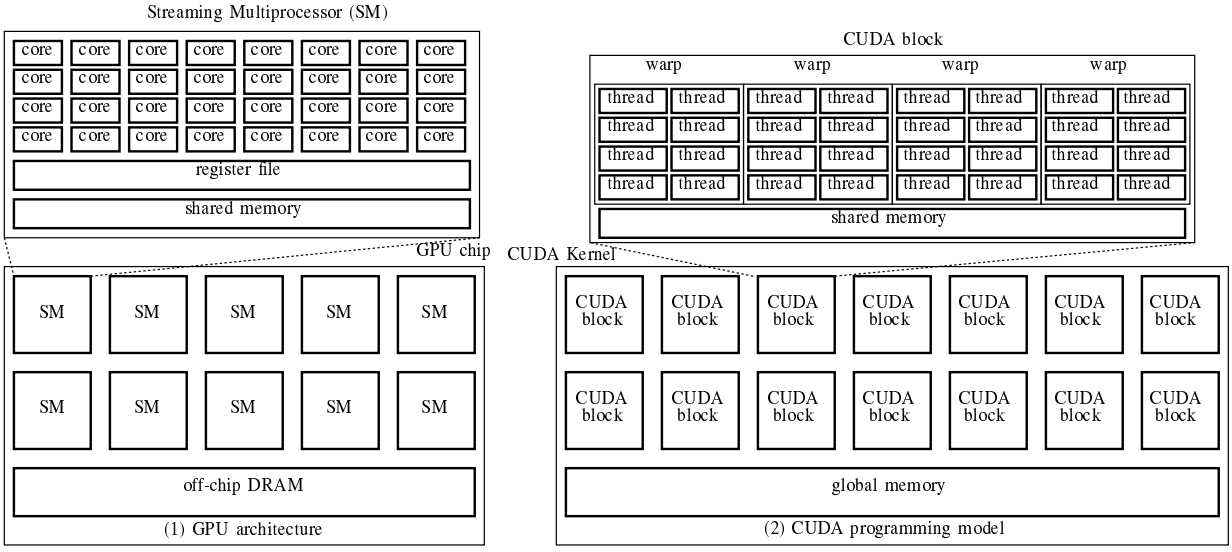

Fig. 2. GPU architecture and CUDA programming model

state of each cell is stored in a word of the memory, such as a 32-bit integer or an 8-bit character. We assume that the initial states of cells are stored in the global memory of the GPU. 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. 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 it evaluates formulas (1) and (2) to compute 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 $g_2(i,j)$ 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.

IV. Bit-per-cell arrangements, bitwise summing technique, and one-step simulation

The main purpose of this section is to show an efficient simulation of the Game of Life. The idea is to arrange the state of each cell in a bit of a word, and compute the next state by bitwise operations. These techniques has been presented and the CPU implementation has been shown in [24].

A. Bit-per-cell arrangements

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. As illustrated in Figure 3, consecutive 32 cells in the same row is arranged in a 32-bit word. In general, d consecutive cells in the same row is 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. We can have two address modes, *row-major* and *column-major*, to map addresses to these words. As shown in the figure, the row-major/column-major bit-per-cell arrangement maps addresses to words in row-major/column-major order, respectively. Since CUDA supports 32-bit and 64-bit words, it makes sense to set $d = 32$ or $d = 64$ when we implement the bit-per-cell arrangement in GPUs.

Note that we should use the column-major bit-per-cell arrangement although most of existing implementations use the row-major. For example, in a GPU implementation that we will show later, a block of 32×32 cells are operated in the same time. If we use the row-major order as illustrated in Figure 3 (1), the leftmost top block is arranged in stride addresses 0, 4, 8, ..., and 124. On the other hand, memory access is destined for coalesced addresses 0, 1, 2, ..., and 31, if we use the column-major order as illustrated in Figure 3 (2).

B. Bitwise summing technique

To simulate 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 fundamental bitwise operations such as bitwise OR and bitwise AND. The original idea has been shown in [24]. We extend this idea for further acceleration.

Suppose that we have three d-bit words A, B, and C and we want to compute the sum of each bit. More specifically, let $A = a_{d-1}a_{d-2}\cdots a_0$, $B = b_{d-1}b_{d-2}\cdots b_0$, $C = c_{d-1}c_{d-2}\cdots c_0$. The goal is to compute the bitwise sum of A, B, and C, that is, the sum y_ix_i such that $y_i \cdot 2 + x_i = a_i + b_i + c_i$ for all i $(0 \leq i \leq d-1)$. *The bitwise summing technique* computes such x_i and y_i as two words $X = x_{d-1}x_{d-2}\cdots x_0$, and $Y = y_{d-1}y_{d-2}\cdots y_0$. We will show that two words X and Y can be computed simply by bitwise XOR ($\oplus$), bitwise AND ($\wedge$), and bitwise OR ($\vee$). By simulating the full adder, we can compute X and Y in 7 bitwise binary operations as

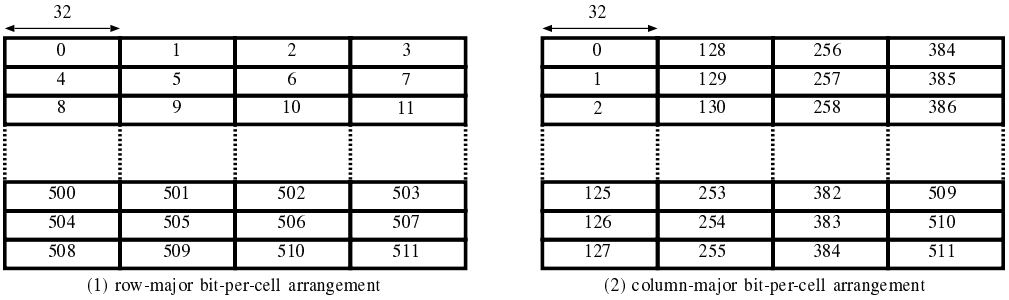

Fig. 3. Bit-per-cell arrangements of 128×128 array of 32-bit words

follows:

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

We can further reduce the number of bitwise operations to 5 if we use a temporal word T as follows:

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

To compute the next states of d cells stored in a d-bit word, the states of $2d+6$ neighboring cells are necessary as illustrated in Figure 4 (1), where $d = 4$. We store neighboring cells in eight words $A, B, \ldots, H$ as illustrated in Figure 4 (2), which are used to compute the next state of word I. For this purpose, we compute the bitwise sums as shown in Figure 4 (3) 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. More specifically, $(I \wedge I_2) \vee I_3$ is the next state of each cell in I. 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 how I_2 and I_3 are computed. 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 of I are as follows:

[Algorithm SINGLE-WORD]

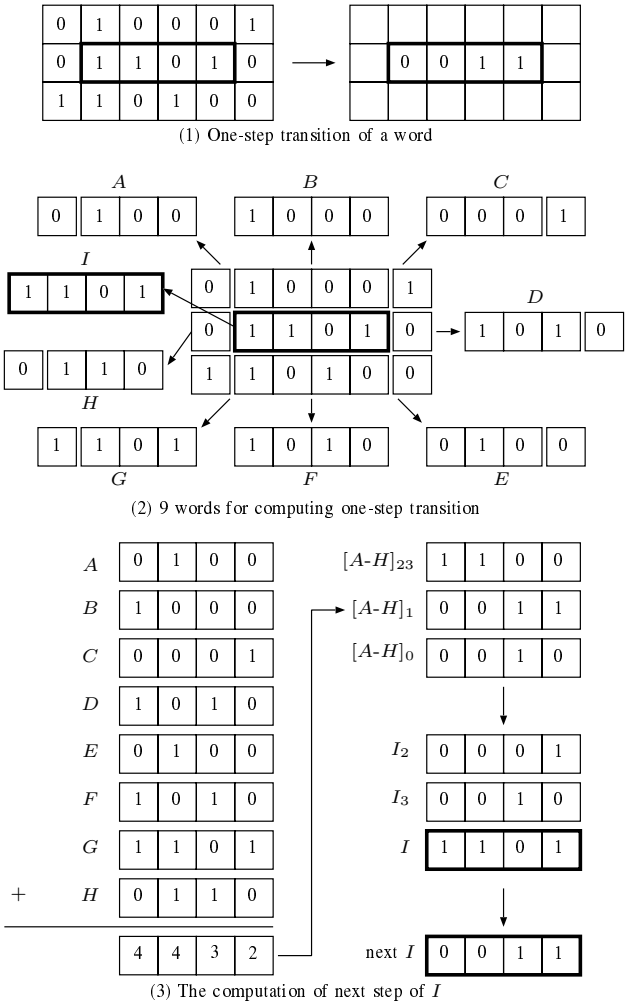

Fig. 4. The computation of the next states of 4 cells in a 4-bit word by Algorithm SINGLE-WORD

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 $4 \times 2 + 2 \times 5 + 9 + 5 + 2 = 34$. 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 $\frac{n}{64 \cdot 32}$ 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 $A, B, \ldots, H$ from these words, and compute 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.

V. Our acceleration techniques

The main purpose of this section is to show our new ideas and techniques for further acceleration of simulation of the Game of Life.

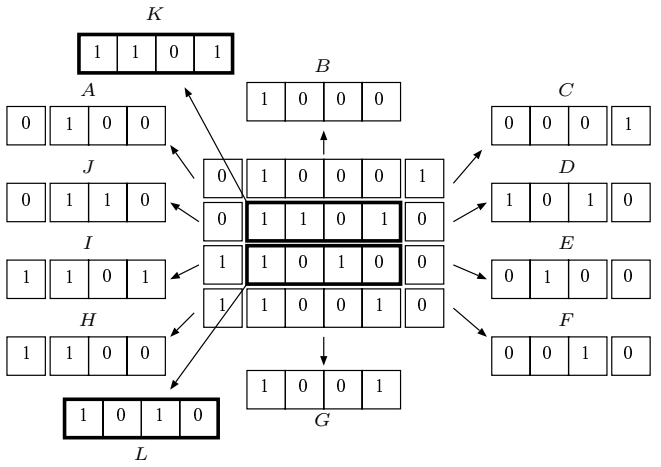

Fig. 5. Illustrating 12 words for computing next states of cells in two words by Algorithm DOUBLE-WORD

A. 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 $(L, [D\text{-}K]_{23}, [D\text{-}K]_1, [D\text{-}K]_0])$.

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. $(L, L_2, L_3) \leftarrow (I, [D\text{-}K]_{23}, [D\text{-}K]_1, [D\text{-}K]_0])$

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.

B. 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 2-dimensional array, 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.

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

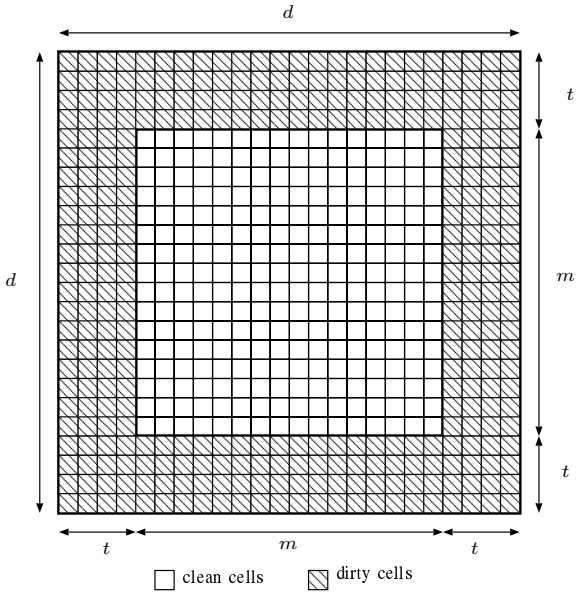

Fig. 6. Dirty cells

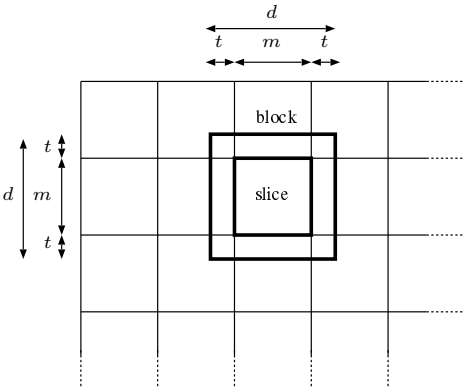

Fig. 7. A $d \times d$ slice and an $m \times m$ block in a large 2-dimensional array

Clearly, we should use the column-major bit-per-cell arrangement because cells in a block are arranged in neighboring addresses. More specifically, the d or $2d$ words in the global memory storing $d \times d$ cells are stride if we use the row-major bit-per-cell arrangement. On the other hand, the $2d$ words are in two consecutive addresses of length d each if the column-major bit-per-cell arrangement is used.

If Algorithm DOUBLE-WORD is implemented using the shared memory as it is, memory access to the shared memory has bank conflicts. The shared memory of Maxwell architecture has 32 memory banks with 32-bit width [29]. 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 with 64-bit width. Further, if each of 32 threads in a warp access to a 64-bit number in the shared memory, 16 threads in the first half warp try to access them, and then 16 threads in the second half warp do the same thing. Suppose that 64 64-bit numbers that constitute a block (Figure 8 (1)) and let $a(i)$ $(0 \leq i \leq 63)$ be the i-th 64-bit number. is stored in the 16 banks of the shared memory as it is (Figure 8 (2)). If Algorithm DOUBLE-WORD is executed using 32 threads in a warp, the first warp may access 64-bit numbers $a(2), a(4), a(6), \ldots, a(32)$. As we can see in Figure 8 (2), two numbers are in the same bank.

To avoid bank conflicts, we use the shift arrangement as illustrated in Figure 8 (3). In the shift arrangement, the second row and the fourth row are shifted by one. We can confirm that $a(2), a(4), a(6), \ldots, a(32)$ are arranged in distinct banks. The first warp also may access, sets of 16 numbers

- $a(0), a(2), a(4), \ldots, a(30)$, and
- $a(1), a(3), a(5), \ldots, a(31)$.

We can confirm that 16 numbers in each set are in distinct banks. Thus, we can avoid bank conflicts by the shift arrangement.

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. 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 slice of size $m \times m$ is $t + c$. Hence, the cost of T-step simulation of $\sqrt{n} \times \sqrt{n}$ cells is:

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

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

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

Clearly, 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.

C. Further acceleration using warp shuffle

The memory access latency of the shared memory is not small [16]. 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 Subsection V-B, 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 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.

VI. Experimental results

The main purpose of this section to show the performance of algorithms for Game of Life.

We have evaluated the running time of 1024-step simulation for a 16384×16384 $(2^{14} \times 2^{14})$ array. The array is wrap-around in the sense that cells in the top-row and the bottom-row are neighbor. Also,the leftmost-column and the rightmost-column are neighbor. We have used GeForce GTX TITAN X and Intel Xeon X7460 CPU (2.66GHz) for the experiment. GeForce GTX TITAN X has 24 streaming multiprocessors with 128 cores each. Sequential algorithms are executed as they are using a single thread running Intel Xeon X7460 CPU. We may accelerate these sequential algorithms using multiple threads and/or AVX resources. However, these acceleration techniques for Xeon CPU are out of scope of this work, because our goal is not to compare the capability of NVIDIA GPU and Intel Xeon CPU. The speedup factors of GPU implementations over CPU implementations shown by our experiments are just for reference purpose.

Table I shows the running time of straightforward implementations, for the word-per-cell and the bit-per-cell. 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 16384×16384 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 CPU computes the next state of every cell 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.

We have used 64-bit unsigned long long integers for the bit-per-cell arrangement. Hence, $16384 \times 16384 = 2^{28}$ cells are implemented in $16384 \times 256 = 2^{22}$ words. To see the difference of performance of Algorithms SINGLE-WORD and DOUBLE-WORD, we have implemented both algorithms. To compute the next states of all cells, the CPU executes SINGLE-WORD 2^{22} times. It also need to execute DOUBLE-WORD 2^{21} times for 1-step simulation. To compute the next states of all cells by Algorithm SINGLE-WORD, a CUDA kernel of 2^{17} CUDA blocks of 32 threads each is invoked. Also, for 1-step simulation by Algorithm DOUBLE-WORD, 2^{16} CUDA blocks of 32 threads each are used. These 1-step simulations are repeated 1024 times for 1024-step simulation.

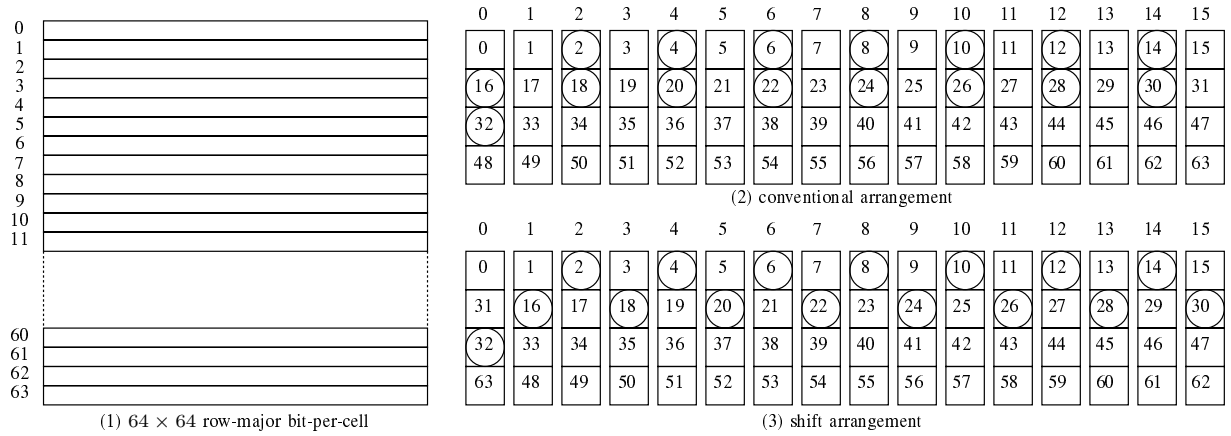

Fig. 8. The padding technique to avoid bank conflicts

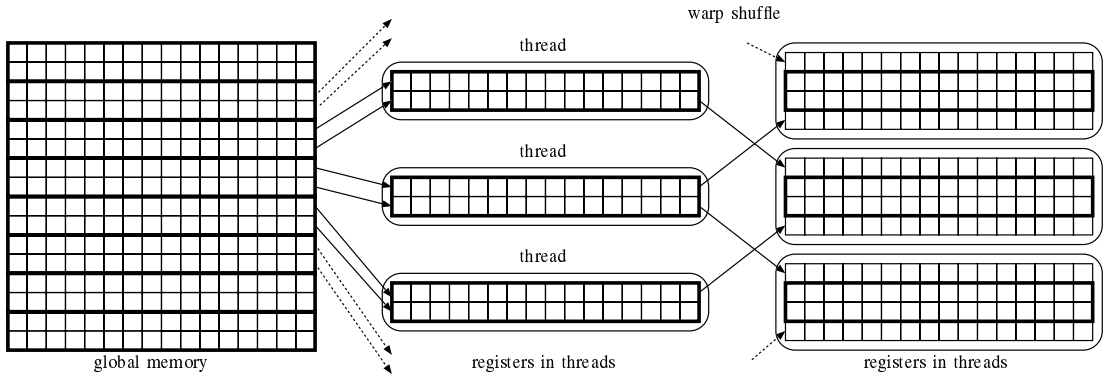

Fig. 9. Copying words storing cells using a warp shuffle instruction

TABLE I. THE RUNNING TIME (IN SECONDS) OF CPU IMPLEMENTATION AND GPU IMPLEMENTATION (GLOBAL MEMORY)

	word-per-cell	bit-per-cell	
		SINGLE-WORD	DOUBLE-WORD
Intel Xeon	2151	84.8	58.3
Nvidia GPU	119.3	0.574	0.672
speed-up	111	147	86.8

From Table I, the GPU implementations can accelerate the computation with a speed-up factor of more than 100 for the word-per-cell arrangement and the bit-per-cell arrangement using Algorithm SINGLE-WORD. 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 20 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 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 one 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 are used for 64 words, we execute SINGLE-WORD twice or DOUBLE-WORD once to compute 1-step transition. Each thread repeat this t times to complete 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 16384×16384 is partitioned into $\frac{16384}{m} \times \frac{16384}{m}$ 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 II. THE RUNNING TIME (IN SECONDS) OF GPU IMPLEMENTATIONS OF MULTIPLE-STEP SIMULATION

	GPU (shared memory)			GPU (register+warp shuffle)		
	32×32 block	64×64 block		32×32 block	64×64 block	
steps	SINGLE-WORD	SINGLE-WORD	DOUBLE-WORD	SINGLE-WORD	SINGLE-WORD	DOUBLE-WORD
2	0.607	0.439	0.385	0.543	0.390	0.379
4	0.482	0.301	0.237	0.386	0.216	0.194
8	0.850	0.356	0.248	0.545	0.197	0.163
16	-	0.762	0.511	-	0.377	0.295

Table II shows the running time of 1024-step simulation of the Game of Life with 16384×16384 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 shared memory implementations except for a few exceptions. From the table, 8-step simulation with 64×64 block using Algorithm DOUBLE-WORD runs 0.163 seconds, which is the minimum over all implementations that we have developed. Further, in Subsection V-B, we have shown that t must be in $[0, \frac{d}{6}]$ by theoretical analysis. We can see that the value t that minimize the computing time in this range.

VII. CONCLUSION

The main purpose of this paper presents several techniques for accelerating the simulation of the Conway's Game of Life. In particular, we have presented techniques of (1) sharing the sum computation for two words, (2) multiple-step simulation, and (3) register with warp shuffle instructions. The best implementation performs 1024-step simulation of 16384×16384 cells in 0.163 seconds on GeForce GTX TITAN X GPU. This implies that it achieves 1.69×10^{12} updates per second, which is more than 68 times faster than previously presented best implementation.

REFERENCES

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

[2] V. Podlozhnyuk, "Image convolution with CUDA," July 2007.

[3] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2011, pp. 153–159.

[4] Y. Zhang, J. L. Recker, R. Ulichney, I. Tastl, and J. D. Owens, "Plane-dependent error diffusion on a GPU," in *Proc. SPIE*, vol. 8295, Jan. 2012.

[5] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Nov. 2010, pp. 279–280.

[6] P. Steffen, R. Giegerich, and M. Giraud, "GPU parallelization of algebraic dynamic programming," in *Proc. of International Conference on Parallel Processing and Applied Mathematics: Part II*, Sept. 2009, pp. 290–299.

[7] 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," in *Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73–82.

[8] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 320–326.

[9] 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. Vol. E96-D, no. 12, pp. 2617–2625, Dec. 2013.

[10] ——, "An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation," in *Proc. of International Conference on Parallel Processing (ICPP)*, Oct. 2013, pp. 1–10.

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

[12] ——, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[13] 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*, vol. 1, no. 2, pp. 260–276, July 2011.

[14] 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*, vol. E96-D, no. 12, pp. 2626–2634, 2013.

[15] ——, "Simple memory machine models for GPUs," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 1, pp. 17–37, 2014.

[16] S. Okamoto, Y. Ito, K. Nakano, and J. L. Bordim, "Thorough evaluation of GPU shared memory load and store instructions,," in *Proc. of International Symposium on Computing and Networking*, Dec. 2015, pp. 614–616.

[17] NVIDIA Corporation, "NVIDIA's next generation CUDA compute architecture: Kepler GK110, whitepaper," 2012.

[18] NVIDIA Corportaion, "NVIDIA GeForce GTX980 whitepaper," 2014.

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

[20] Y. Yang, Z. Guan, H. Sun, and Z. Chen, "Accelerating RSA with fine-grained parallelism using GPU," in *Proc. of Information Security Practice and Experience (LNCS)*, vol. 9065, 2015, pp. 454–468.

[21] M. Gardner, "Mathematical games: The fantastic combinations of John Conway's new solitaire game "life"," *Scientific American*, vol. 223, pp. 120–123, Oct. 1970.

[22] A. Adamatzky, *Game of Life Cellular Automata*. Springer, 2015.

[23] M. Fisher, "Conway's Game of Life on GPU using CUDA," Mar 2013. [Online]. Available: http://www.marekfiser.com/Projects/Conways-Game-of-Life-on-GPU-using-CUDA

[24] N. Tsuda, "Acceleration of Game of Life by the bit operation (bit-board)," December 2012, in Japanese. [Online]. Available: http://vivi.dyndns.org/tech/games/LifeGame.html

[25] MathWorks, "Stencil operations on a GPU," 2015. [Online]. Available: https://www.mathworks.com/examples/parallel-computing/398-stencil-operations-on-a-gpu

[26] K. S. Perumalla and B. G. Aaby, "Data parallel execution challenges and runtime performance of agent simulations on GPUs," in *Proc. of Spring Simulation Multiconference*, 2008, pp. 116–123.

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

[28] NVIDIA Corporation, "Geforce GTX TITAN X," 2015. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x

[29] ——, "Tuning CUDA applications for Maxwell," 2015.