


# Evaluations of OpenCL-written tsunami simulation on FPGA and comparison with GPU implementation

Fumiya Kono<sup>1</sup>  · Naohito Nakasato<sup>1</sup> ·  
Kensaku Hayashi<sup>1</sup> · Alexander Vazhenin<sup>1</sup> ·  
Stanislav Sedukhin<sup>1</sup>

Published online: 16 March 2018

© Springer Science+Business Media, LLC, part of Springer Nature 2018

**Abstract** When a tsunami occurred on a sea area, prediction of its arrival time is critical for evacuating people from the coastal area. There are many problems related to tsunami to be solved for reducing negative effects of this serious disaster. Numerical modeling of tsunami wave propagation is a computationally intensive problem which needs to accelerate its calculations by parallel processing. The method of splitting tsunami (MOST) is one of the well-known numerical solvers for tsunami modeling. We have developed a tsunami propagation code based on MOST algorithm and implemented different parallel optimizations for GPU and FPGA. In the latest study, we have the best performance of OpenCL kernel which is implemented tsunami simulation on AMD Radeon 280X GPU. This paper targets on design and evaluation on FPGA using OpenCL. The performance on FPGA design generated automatically by Altera offline compiler follows the results of GPU by several kernel modifications.

**Keywords** Tsunami · Method of splitting tsunami · OpenCL · GPU · Intel FPGA SDK for OpenCL SDK · FPGA

## 1 Introduction

Tsunami is a secondary natural disaster which follows after a submarine earthquake. The faster prediction of tsunami is strongly desired for disaster prevention. When an earthquake occurs, we can forecast tsunami propagation by using numerical simulations with an initial condition and the laws of physics governing the phenomenon.

---

✉ Fumiya Kono  
d8151104@u-aizu.ac.jp

<sup>1</sup> Graduate School of Computer Science and Engineering, The University of Aizu, Ikki-machi Tsuruga, Aizuwakamatsu, Fukushima 965-8580, Japan

However, such simulations should cover vast region by processing a large amount of computational data, and therefore, in the sequential computation, it is often difficult to complete the simulation faster than real time. Accordingly, large-scale and real-time simulations require massively parallel computing technologies with various parallel computing architectures, their programming models, and languages.

There are several previous works on high-performance tsunami simulations using new and modern computing systems based on the heterogeneous computing paradigm.

Imamura et al. [1] developed Tsunami package (TUNAMI-N1) with the staggered leap-frog scheme. Gidra et al. [2] evaluated parallelized TUNAMI-N1 code by CUDA on NVIDIA QUADRO FX 1700. They showed the results on various sizes of the ocean bathymetry data sets for 7200 time steps. For a  $1040 \times 668$  grid, they obtained 5.86x speedup as in comparison with sequential computation with a single processor.

Acuna and Aoki [3] used Tesla M2050 GPU to solve the shallow water equations for tsunami simulation. They used a numerical solution based on the CIP-CSL2 semi-Lagrangian scheme and the method of characteristics. They simulated tsunami over a large grid covering the entire Pacific ocean using a Tsubame 2.0 system with multiple GPUs. By using adaptive mesh refinement (AMR), they saved memory usage by 20–40%. Finally, they achieved 313 GFlops with a single GPU. Fujita [4] also reported his accelerated tsunami simulation on FPGA. He manually extracted large data flow graphs from the program and compiled it into FPGA circuits. The size of computation grid is  $1040 \times 668$ , and the simulation is conducted 7200 steps regarding one time step as 1 s. It was shown that FPGA tsunami simulation is 46 times faster than Intel core i7 processor at 2.93GHz.

In this research, we investigate parallel computing algorithms and architectures that are suitable for high-performance tsunami simulation based on the method of splitting tsunami (MOST) [5,6]. In the future, we will combine our parallelized code into the tsunami visualization tool [7] which is currently in development for the real-life applications such as where it is effective to put the tetrapods or breakwaters for reducing the damage generated by tsunami. Therefore, this research will contribute to make their experiments for modeling tsunami more faster with various initial conditions.

MOST, which is our target algorithm for acceleration, is one of the solvers for shallow water equations used for tsunami numerical simulation. The MOST algorithm can be considered as a combination of finite difference method and the Euler method for time integration. Our motivation for the acceleration of the MOST algorithm is to simulate tsunami propagation before tsunami actually arrives at the coastal area in real time. From the shallow water equations described in Sect. 2, we have the phase velocity of wave motion  $c$  as  $c = \sqrt{gH}$ , where  $g$  is the gravitational acceleration and  $H$  is the sea depth. For instance, the average sea depth in Pacific Ocean is known as 4000 m. In that case, the velocity of tsunami is about  $c = 712$  km/h. When the distance between the coastal area and the epicenter is 100 km, it takes about 8.5 min for tsunami arriving at the coastal area. For this case, a prediction based on numerical simulations must be conducted in shorter time than this time limit.

To speed up the simulations, we have parallelized MOST algorithm by using OpenMP, OpenACC, and OpenCL (Open Computing Language) [8] and evaluated their performance on Multi-core CPU and GPU. In that benchmarking, we have

obtained 185 GFlops which was the best performance by using OpenCL on AMD Radeon 280X GPU [9].

On the other hand, Nagasu et al. [10] designed the stream computing architecture and hardware for practical tsunami simulation. They introduced multiple stream processing element (SPE) arrays with parallel internal pipelines to exploit further available hardware resources. Their implementation with Arria 10 FPGA achieved the performance of 383 GFlops and the performance per power of 8.41 GFlops/W with six cascaded SPEs. Therefore, the dedicated implementation for Arria 10 FPGA shows higher performance than our best GPU implementation. The performance per power of the FPGA implementation is also better than the GPU implementation [10].

Meanwhile, there are some works to design FPGA accelerator by using OpenCL. OpenCL is one of the well-known framework for parallel programming on heterogeneous environments. It has versatility to compute on various devices including CPUs, GPUs, and reconfigurable systems such as FPGAs. With specific compilers, it is possible to generate hardware design for FPGAs automatically from OpenCL kernel without explicitly designing the hardware architecture. There are several studies working on FPGA design generating from OpenCL kernel.

Takei et al. [11] implemented FPGA accelerator of finite-difference time-domain (FDTD) method which is widely used in an electromagnetic simulation using OpenCL. They reported that the computation time of the FPGA design generated by OpenCL kernel was about 10 times faster than the computation by their GPU implementation.

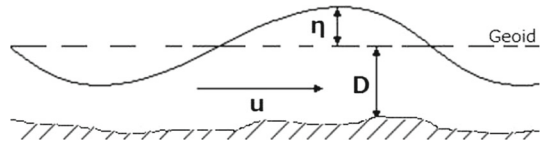
Tatsumi et al. [12] also implemented FPGA accelerator of the stereo correspondence matching. They exploited pipeline stages for Fourier transform efficiently for FPGA. Also, Waidyasooriya et al. [13] used the FPGA accelerator generated from OpenCL kernel to simulate molecular dynamics. Their hardware is implemented loop-pipelining, and it achieved over 4.6 times of speedup comparing with CPU by using only 36% of the Stratix V FPGA resources.

In more recent studies, Yinger et al. [14] presented the FPGA implementation for deep neural network as the application of matrix multiplication by writing OpenCL kernel. Wang et al. [15] also designed the FPGA accelerator for convolution neural networks by using OpenCL. Roozmeh and Lavagno [16] focused on the problem about high energy consumption and power dissipation for the modern datacenters. They presented the FPGA accelerator to speed up the join operation on the database.

Houtgast et al. [17] implemented highly efficient FPGA accelerator for the Smith–Waterman algorithm to find the optimal pairwise alignment in bioinformatics. They succeeded in implementing the same accelerator by writing only 90 lines of OpenCL kernel which is about 20% of their VHDL code.

As we can see, designing FPGA accelerators for various scientific applications by using OpenCL is now feasible. Nevertheless, since FPGA design design from OpenCL kernel is a technology appeared recently, the example of applications is not plenty yet against GPU implementation. In this paper, we focus to accelerate the MOST algorithm by using OpenCL. We have already developed OpenCL implementation of the MOST algorithm which was applied well-known spacial blocking. Here, we ported the OpenCL code and gave a several optimizations to our previous OpenCL kernel for the benchmarking on Arria 10 FPGA.

**Fig. 1** 1-D representation for wave propagation characteristics



This paper presents the evaluation and comparison of MOST algorithm written in OpenCL among four implementations:

1. Originally developed kernel by using spatial blocking on GPU as baseline;
2. Same kernel for GPU as baseline on FPGA design (without any optimization for FPGA);
3. Optimized kernel using shift registers for FPGA design;
4. Further optimized kernel to improve the parallelism by expanding the width of the data path.

The rest of this paper is organized as follows. In Sect. 2, the outline of MOST algorithm is given. Section 3 presents the description of the original MOST algorithm and parallelization by using spatial blocking algorithm. Section 4 shows the OpenCL implementation and its performance on several GPUs as baseline for following evaluations on FPGA. Section 5 shows the evaluation of OpenCL implementation on FPGA design generated automatically and its further optimizations. Section 6 shows the consideration and comparison of OpenCL implementation between GPU and FPGA. Finally, Sect. 7 concludes this paper with a mention of future work.

## 2 MOST: method of splitting tsunami

Firstly, we show the original MOST algorithm for the solution of shallow water equations. Shallow water equations which are nonlinear approximation of shallow water system are represented by following partial differential equations (PDEs) [5,6].

$$\begin{aligned}
 u_t + uu_x + vu_y + gH_x &= gD_x, \\
 v_t + uv_x + vv_y + gH_y &= gD_y, \\
 H_t + (uH)_x + (vH)_y &= 0.
 \end{aligned}
 \tag{1}$$

Here,  $H = H(x, y, t) = \eta(x, y, t) + D(x, y)$ , where  $\eta$  and  $D$  are the wave height and the depth profile (bathymetry), respectively,  $u$  and  $v$  are the wave velocity in each spatial coordinate,  $g$  is gravitational acceleration. Figure 1 schematically shows these quantities as an 1-D plot.

An alternative form of Eq. (1) is represented as follows:

$$\frac{\partial \mathbf{z}}{\partial t} + \mathbf{A} \frac{\partial \mathbf{z}}{\partial x} + \mathbf{B} \frac{\partial \mathbf{z}}{\partial y} = \mathbf{F},
 \tag{2}$$

where

$$\mathbf{z} = \begin{pmatrix} u \\ v \\ H \end{pmatrix}, \mathbf{A} = \begin{pmatrix} u & 0 & g \\ 0 & u & 0 \\ H & 0 & u \end{pmatrix},$$

$$\mathbf{B} = \begin{pmatrix} v & 0 & 0 \\ 0 & v & g \\ 0 & H & v \end{pmatrix}, \mathbf{F} = \begin{pmatrix} gD_x \\ gD_y \\ 0 \end{pmatrix}.$$

The numerical treatment of MOST is based on two auxiliary systems. Applying spatial decomposition to Eq. (2) along each coordinate, we get two auxiliary systems,  $\Phi = (u, 0, H)^T$  and  $\Psi = (0, v, H)^T$ , which depend only on one spatial variable such as

$$\begin{cases} \frac{\partial \Phi}{\partial t} + \mathbf{A} \frac{\partial \Phi}{\partial x} = \mathbf{F}_1, & 0 \leq x \leq X, & (3a) \\ \frac{\partial \Psi}{\partial t} + \mathbf{B} \frac{\partial \Psi}{\partial y} = \mathbf{F}_2, & 0 \leq y \leq Y, & (3b) \end{cases}$$

where

$$\mathbf{F}_1 = \begin{pmatrix} gD_x \\ 0 \\ 0 \end{pmatrix}, \mathbf{F}_2 = \begin{pmatrix} 0 \\ gD_y \\ 0 \end{pmatrix}.$$

MOST algorithm uses the method of characteristics for the numerical solutions. For the solution along  $x$ -coordinate, Eq. (3a) is transformed into following form:

$$\frac{\partial \mathbf{W}}{\partial t} + \mathbf{A}' \frac{\partial \mathbf{W}}{\partial x} = \mathbf{F}'_1, \tag{4}$$

where

$$\mathbf{W} = \begin{pmatrix} v \\ u + 2\sqrt{gH} \\ u - 2\sqrt{gH} \end{pmatrix}. \tag{5}$$

Here, all elements in  $\mathbf{W}$  are the Riemann invariants which are constants along the characteristic curves of the equation, and diagonal matrix  $\mathbf{A}'$  and  $\mathbf{F}'_1$  are expressed as following form:

$$\mathbf{A}' = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix}, \mathbf{F}'_1 = \begin{pmatrix} 0 \\ gD_x \\ gD_y \end{pmatrix}, \tag{6}$$

where  $\lambda_1, \lambda_2,$  and  $\lambda_3$  are eigenvalues of  $\mathbf{A}$ ,

$$\lambda_1 = u, \lambda_2 = u + \sqrt{gH}, \lambda_3 = u - \sqrt{gH}.$$

For the numerical solution of Eq. (4), the following finite difference method (FDM) and the explicit Euler method for time integration are applied as

$$\begin{aligned} & \frac{W_{i,j}^{n+1} - W_{i,j}^n}{\Delta t} + A' \frac{W_{i+1,j}^n - W_{i-1,j}^n}{2\Delta x} \\ & - A' \Delta t \frac{A'(W_{i+1,j}^n - W_{i,j}^n) - A'(W_{i,j}^n - W_{i-1,j}^n)}{2\Delta x^2} \\ & = \frac{F'_{i+1,j} - F'_{i-1,j}}{2\Delta x} - A' \Delta t \frac{F'_{i+1,j} - 2F'_i + F'_{i-1,j}}{2\Delta x^2}. \end{aligned} \tag{7}$$

Here,  $n$  denotes the  $n$ th computational step, and  $i, j$  corresponds to  $x, y$ -coordinates, respectively.  $\Delta t$  and  $\Delta x$  also denotes time step and grid resolution, respectively. The criterion of stability for the MOST algorithm can be written as the relationship between time step and grid resolution [18]:

$$\Delta t \leq \frac{\Delta x}{\sqrt{gH}}. \tag{8}$$

The actual calculation procedure for one time step is summarized as follows:

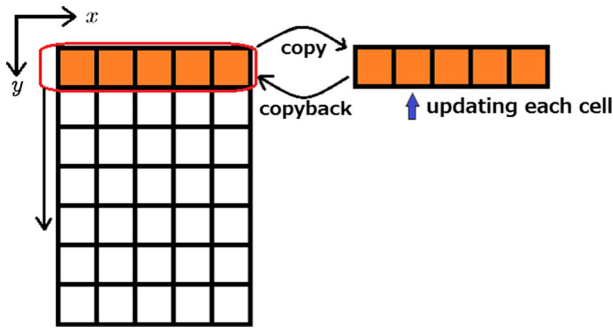
1.  $u, v,$  and  $H$  are transformed by Eq. (5).
2. Calculate the solution along  $x$ -coordinate by Eq. (7).
3. The variables are transformed back to the original variables  $u, v$  and  $H$ .
4.  $v, u$  and  $H$  are transformed by the equations corresponding to Eq. (5) for  $y$ .
5. Calculate the solution along  $y$ -coordinate.
6. The variables are transformed back to the original  $v, u$  and  $H$ .

In this procedure, we need 200 floating-point operations for updating one cell in total.

The accuracy of simulations based on the MOST algorithm generally depends on following three factors;

- Algorithm and program for calculating tsunami wave propagation,
- Accuracy of bathymetry data,
- Accuracy of generating initial wave displacement.

For the first factor, the original MOST is a second-order accurate in space and a first-order accurate in time. And it is standard and well-verified software used in sequential computation. In this paper, we applied various optimizations for parallelization to the original algorithm and found that there is no significant difference in the results due to such optimizations. On the other hand, we use single-precision (SP) floating-point operations in our evaluation. In the majority of Pacific Ocean, the sea depth is roughly 4000 m in average so that we consider SP arithmetic is sufficiently accurate. However, there are some areas whose sea depth is more than 10000 m like ocean trench. When the difference of sea depth for two adjacent cells is very large, we experienced the computation by SP arithmetic causing large numerical errors. For that case, we can easily switch to use double precision floating-point operations in our OpenCL-based parallelization of the MOST algorithm.



**Fig. 2** Procedure for data updating along  $x$ -direction characteristics

To simulate tsunami generated by an earthquake for practice, a deformation model of the sea floor [19] can be used to compute initial  $H$ ,  $u$  and  $v$ . The initial condition is modeled by parameters such as the epicenter (the point on the Earth's surface vertically above the earthquake source), the earthquake magnitude, and the distance between the earthquake source and epicenter. In this paper, for the bathymetry and initial wave displacement, we use flat bathymetry and simple initial wave displacement as we describe in Sect. 4. We believe they are not significant for our performance evaluation.

### 3 Algorithms for parallelization

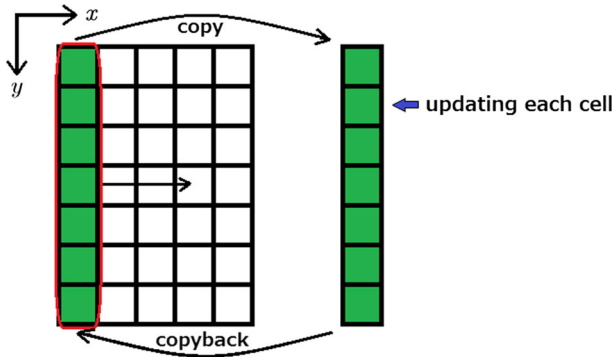
#### 3.1 Original computing algorithm

Before we present details of optimizations for MOST algorithm, we show the original computing algorithm of MOST presented in Sect. 2. Assume quantities such as  $D$ ,  $H$ ,  $u$ , and  $v$  are stored in the 2-D arrays. Inputting  $D$ ,  $H$ ,  $u$ , and  $v$  at the time step  $n = 0$ , we update  $H$  and  $u$ ,  $v$  on every time step. In the original MOST program, each datum is stored in the format of structure of array (SOA). Each quantity contains different 2-D arrays.

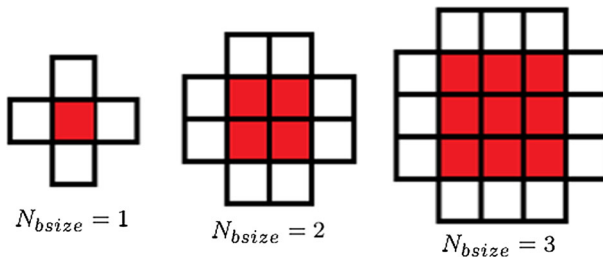
Each 2-D data array is updated by using the 1-D temporary array based on the scheme which we showed in Sect. 2. Figure 2 shows the procedure for data updating along longitude in one time step.

In this case, updating is conducted row-by-row in the following steps. First, the data of the selected row are copied from the 2-D array to a 1-D temporary array. Second,  $H$  and  $u$ ,  $v$  are transformed into Riemann invariants. Third, FDM and the Euler method are applied to each cell in the 1-D temporary array. Fourth, Riemann invariants  $H$  and  $u$ ,  $v$  which were transformed previously are reverted. Finally, the updated data in the 1-D temporary array are copied back to the 2-D array. The update along longitude is finalized by applying this procedure for all rows.

Afterward, the processing of 2-D data is implemented along latitude. As shown in Fig. 3, this procedure is very similar to the update along longitude. In this case, the computations are conducted on every column. Thus,  $H$  and  $u$ ,  $v$  in the 2-D array are updated in one time step. Importantly, the algorithm has a high-probability cache



**Fig. 3** Procedure for updating along  $y$ -direction characteristics



**Fig. 4** Extracting cells which should be updated in a spatial block from the entire 2-D array (in case of central  $N_{\text{blocksize}} \times N_{\text{blocksize}}$  cells are updated)

**Table 1** Number of loading and storing data required for updating  $N_{\text{blocksize}} \times N_{\text{blocksize}}$  cells in stencil computation

$N_{\text{blocksize}}$	1	2	3	$m$
Num of load	5	12	21	$m^2 + 4m$
Num of store	1	4	9	$m^2$

miss in the 2-D array for every data copied into 1-D array due to C/C++ row- or longitude-wise storage for planar data in memory.

### 3.2 Algorithm with spatial blocking

In our GPU implementation, spatial blocking is applied to the original MOST algorithm in order to obtain high level of parallelism on GPU. The data in the 2-D array is firstly divided into spatial blocks, and updated every spatial block, respectively.

Let  $N_{\text{blocksize}}$  be the block size for each spatial block. As shown in Fig. 4, a spatial block is extracted to update central  $N_{\text{blocksize}} \times N_{\text{blocksize}}$  cells in the 2-D arrays. To update a block with  $N_{\text{blocksize}} \times N_{\text{blocksize}}$  data cells,  $(N_{\text{blocksize}} + 2) \times (N_{\text{blocksize}} + 2)$  cells are actually used since halo is required to update boundary cells in the block.

In this figure, the red-colored cells are updated by the stencil computation, and the other cells represent halo. Table 1 is the summary of the number of loading and storing data actually required to process the stencil computation. In case of  $N_{\text{blocksize}} = m$ ,



the number of cells updated by one stencil computation is  $C = fm^2$ , where  $f$  is the coefficient for the computation, and the total number of memory reference is  $M = 2m^2 + 4m$ . Therefore, the computational intensity  $C/M$  is given as

$$\frac{C}{M} = \frac{fm^2}{2m^2 + 4m} = \frac{f}{2 + \frac{4}{m}},$$

It is clear that  $m = 1$  gives us the highest level of parallelism. In contrast, larger  $m$  is desirable for the higher computational intensity. Both the high parallelism and high computational intensity are required for high-performance computation on GPU due to availability of massively hardware parallelism. The optimal  $N_{\text{bsize}}$  that the parallelism and the computational intensity are compatible on each GPU is different. We examine the optimal  $N_{\text{bsize}}$  for each hardware in the following benchmarking.

## 4 Implementation and evaluation on GPUs

We parallelized MOST algorithm based on spacial blocking described in Sect. 3. In this section, we present the performance evaluation of our OpenCL implementation for GPUs.

Throughout the present work, we measured the execution time of our code for 300 time steps. This particular choice of the number of time steps is just for evaluation in this paper. For practical simulations, much more number of time steps are sometimes required. However, we have confirmed that our implementation is scalable for any number of computation steps, and the number of computation steps does not affect the performance and its evaluation.

The size of the 2-D array is  $2581 \times 2879$  which is equal to the existing bathymetry size of entire Pacific Ocean used by the original MOST program. For the simplicity, in this benchmarking, we used a simple flat bathymetry where  $D$  is constant everywhere as  $D = 2500\text{m}$  in the computation grid. We generate the initial wave at the center of the computational grid as a cosine wave with the peak height of 10 m.

For the treatment of boundary condition in the MOST algorithm, the reflecting boundary is applied for the boundary between sea and land and the open boundary, at which wave passes through to outside of the computation domain, is applied for the edges of the computation domain. In our evaluation, we have no land inside the computation domain and only apply the open boundary condition at all edges.

In this section, we show the specification of performance benchmarking, implementation, and performance evaluation on GPUs, respectively.

### 4.1 GPUs for performance benchmarking

Our MOST algorithm was written in C++ so that we used g++ (ver. 4.8) compiler for benchmarking on GPU. The following AMD GPUs and NVIDIA GPU are used in this performance evaluation: Radeon R9 280X, FirePro W8100, W9000 (see Tables 2, 3), and Tesla K20c (see Table 4).

**Table 2** Hardware specification of AMD GPU, Radeon

GPU	Radeon R9 280X
Num of GPU cores	2048
Clock frequency	1020 MHz
Memory size	3 GB
Global cache size	16 KB
Memory bandwidth	288 GB/s
Peak perf. (SP)	4.1 Tflops

**Table 3** Hardware specification of AMD GPU, FirePro

GPU	FirePro W8100	FirePro W9100
Num of GPU cores	2560	2816
Clock frequency	824 MHz	930 MHz
Memory size	8 GB	16 GB
Global cache size	16 KB	16 KB
Memory bandwidth	320 GB/s	320 GB/s
Peak perf. (SP)	4.2 Tflops	5.2 Tflops

**Table 4** Hardware specification of NVIDIA GPU, Tesla

GPU	Tesla K20c
Num of GPU cores	2496
Clock frequency	706 MHz
Memory size	5.12 GB
L2 cache	12 KB
Memory bandwidth	208 GB/s
Peak perf. (SP)	3.5 Tflops

The last row in these tables shows theoretical peak performance of single-precision (SP) arithmetic operation in each architecture.

## 4.2 Performance evaluation of GPU implementation

As described in Sect. 3, our OpenCL kernel for MOST algorithm is based on spatial blocking. Before starting the computation, the memory spaces are allocated for the variables used in the computation on GPU. After that, quantities such as  $D$ ,  $H$ ,  $u$ , and  $v$  which are stored in the 2-D arrays and some constants such as gravitational acceleration  $g$ , and size of the spatial block  $m$  are all transferred to global memory on GPU. Based on the number of spatial blocks which are computed in parallel, the number of threads (work items) is determined. Regarding of the efficiency of parallel computation on GPU, the total number of threads is set as multiples of 128.

Figure 5 is the overview of our OpenCL kernel as baseline. This kernel is called by every thread (work item) running on GPU. In OpenCL kernel, predefined functions

```

1  __kernel void most_sweep_2d(
2  __global float* qa, __global float* ua,
3  __global float* va, ..., const float GA){
4  float dw_g[block_size][block_size]; //spatial blocks for D
5  float qw_g[block_size][block_size];
6  ...
7
8  int g_xid = get_global_id(0);
9  int g_yid = get_global_id(1);
10 int g_w   = get_global_size(0);
11 int blo   = g_yid*g_w + g_xid;
12
13 if (blo >= nt) return; //nt: number of threads
14 itr_lon = blo % block_nx * block_size;
15 itr_lat = blo / block_nx * block_size;
16 itr_lon_end = itr_lon + block_size + 2;
17 itr_lat_end = itr_lat + block_size + 2;
18
19 //Generate the spatial block
20 block_j=0;
21 for (j = itr_lat; j < itr_lat_end; j++){
22   block_i=0;
23   for (i = itr_lon; i < itr_lon_end; i++){
24     itr_i = i; itr_j = j;
25     dw_g[block_j][block_i] = GET(da,itr_j,itr_i);
26     qw_g[block_j][block_i] = GET(qa,itr_j,itr_i);
27     uw_g[block_j][block_i] = GET(ua,itr_j,itr_i);
28     vw_g[block_j][block_i] = GET(va,itr_j,itr_i);
29     block_i++;
30   }
31   block_j++;
32 }
33 //Transformation to Riemann invariants
34 for(j = 0; j < block_size+2; j++) {
35   for (i = 0; i < block_size+2; i++){
36     u1[j][i] = uw_g[j][i] + 2.0*sqrt(GA*qw_g[j][i]);
37     q1[j][i] = uw_g[j][i] - 2.0*sqrt(GA*qw_g[j][i]);
38     v1[j][i] = vw_g[j][i];
39   }
40 }
41 //Stencil computation described in Section. 3 follows.
42 ...

```

**Fig. 5** Overview of baseline OpenCL (Code GPU) implementation

such as `get_global_id()` are provided to identify the threads. In our case, we used them to assign each thread to process the specific spatial block.

Lines 19 to 32 in Fig. 5 show that the portion of the kernel copies the data such as  $D$ ,  $H$ ,  $u$ , and  $v$  which are required to process stencil computation from 2-D arrays in global memory. Actually, the data in global memory are stored as 1-D array. The macro-function `GET()` is defined to convert the data in 1-D array in global memory to 2-D array in private memory whose name ends with `_g` suffix which expresses the spatial block. As we can see, the format of storing data for the stencil computation is SOA.

**Table 5** Computation time of original OpenCL kernel on AMD FirePro W8100, W9100, Radeon R9 280X, and NVIDIA Tesla K20c Unit: (s)

$N_{\text{bsize}} \times N_{\text{bsize}}$	W9100	W8100	Radeon	Tesla
$1 \times 1$	16.20	19.61	2.95	11.72
$2 \times 2$	10.72	10.34	2.41	11.67
$4 \times 4$	9.53	9.86	6.88	18.07
$8 \times 8$	12.83	13.58	12.47	19.15

**Table 6** Performance of original OpenCL kernel on AMD FirePro W8100, W9100, Radeon 280X, and NVIDIA Tesla K20c Unit: (GFlops)

$N_{\text{bsize}} \times N_{\text{bsize}}$	W9100	W8100	Radeon	Tesla
$1 \times 1$	27.52	22.74	151.13	38.02
$2 \times 2$	41.59	43.12	185.00	38.20
$4 \times 4$	46.78	45.22	64.80	24.67
$8 \times 8$	34.75	32.83	35.75	23.28

Here, private memory is one of the memory on GPU which is assigned to each thread individually, and basically allocated to registers. Generally, the variables declared in the OpenCL kernel without any prefix are attempted to store in private memory. Nevertheless, it is expected that the size of private memory is not sufficient to store variables used in our MOST implementation shown in Fig. 5. Some variables spilled from registers are stored in global memory.

After the data copy is finished, computation in the spatial block including transformation to Riemann invariants and update by Euler method follows.

### 4.3 Performance evaluation on GPU

Tables 5 and 6 show the computation time and performance of OpenCL code which was originally implemented as baseline on each GPU. The computation time is converted into GFlops by considering that there is 200 floating-point arithmetic operations for updating one cell in the stencil.

The optimal value of  $N_{\text{bsize}}$  for the computation is different for each architecture. The performance on NVIDIA Tesla GPU has a peak when  $N_{\text{bsize}} = 1$  or 2.  $N_{\text{bsize}} = 4$  is optimal for the computation on AMD FirePro GPU, and the performance depends on the version of GPU. AMD Radeon GPU achieved the best performance among all architectures which we evaluated. The optimal  $N_{\text{bsize}}$  on Radeon GPU is  $N_{\text{bsize}} = 2$  and the computation is finished within 2.5 s and its performance is 185GFlops given by multiplying the number of grid points (e.g.,  $2581 \times 2879$ ), the number of floating-point operations (200), the computation time steps (300), and the inverse of computation time.

Regarding of the specifications such as GPU clock frequency and single-precision floating-point operations per second, AMD FirePro W9100 GPU (see Table 3) was expected to achieve the best performance in each implementation. Though both Radeon and FirePro GPUs are devices produced by AMD and its partners, both GPUs are

designed for different purposes [20]. The difference between Radeon and FirePro series was seen by using CodeXL, a performance profiling tool.

In case of the original kernel, cache hit rate is reached to 60–70%, and 138 vector registers were used on Radeon GPU. Besides, vector ALU instructions are processed in more than 85% of computation time on GPU which is nearly optimal value. Conversely, on FirePro GPU series, cache hit is less than 10%, and only 97 vector registers were used. In terms of vector ALU instructions, they were processed in about 15% of computation time on GPU. Furthermore, we detected that memory stall and write stall occurred about 25% of computation time on FirePro GPU.

The difference in performance on Radeon and FirePro GPUs is originated in the difference in generated instructions for each GPU architecture. Since Radeon and FirePro GPUs are targeted for consumer market and professional graphics market, respectively, the device drivers which are responsible to emit the machine instructions are different. We examined the machine instructions for both GPU and found that a way to load data from global memory is different in each case.

For Radeon with the device driver OpenCL 1.2 AMD-APP version 1729.3 targeting consumer graphics, we found that it explicitly use texture cache to load data from global memory. The texture cache is highly effective for loading read-only data from global memory. It gives us high cache hit rate as we found. For FirePro with the device driver OpenCL 2.0 AMD-APP version 1642.5 targeting professional compute and graphics, we found it does not use the texture cache. Accordingly, cache hit rate is as low as 10%. At the moment, we cannot explicitly use texture cache on FirePro. An alternative way to mitigate this problem in OpenCL is to explicitly use local memory which is shared by work items in the same local work group for caching the data from global memory. It should also improve the performance of Tesla K20c. In our recent work [21], we evaluated the performance of optimized kernels using local memory for GPUs.

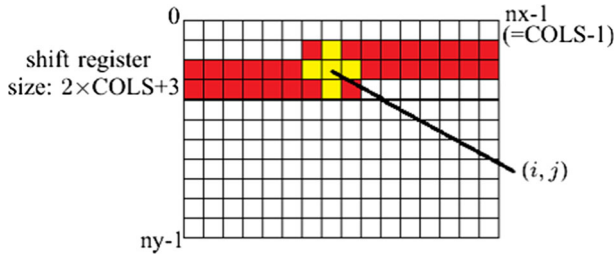
## 5 Implementation and evaluation on FPGA

In this section, we present the implementations and evaluation of tsunami simulations on FPGA design. We modify and further optimize the OpenCL kernel implemented for GPUs to accommodate architecture of FPGA.

### 5.1 FPGA for performance benchmarking

For the benchmarking of OpenCL code on FPGA, we use specific compilers to design hardware automatically from OpenCL kernel. We used the compiler `aoc` (Intel FPGA SDK for OpenCL, 64-Bit Offline Compiler, Quartus 16.0.2). In this paper, we show the result of performance benchmarking on a DE5a-Net Arria 10 FPGA board which has two independent DDR3 memories.

We show the benchmarking of four OpenCL kernels: original code as baseline which was previously mentioned (Code GPU), the optimization for FPGA shown later (Code SR), and another optimization to expand the width of the data path (Code MC1 and MC2). Code MC1 and MC2 presents the technique to improve the parallelism on



**Fig. 6** The data held by shift register which are required for stencil computation when the quantities at  $(i, j)$  element are updated

one pipeline. The benchmarking on FPGA is also conducted under the same initial condition as on GPU.

## 5.2 Optimization by using shift register and its performance on FPGA

Cache system is the element of on-chip memory for loading and storing data efficiently. Spatial blocking which we applied for MOST algorithm is assumed to use cache memory (or local memory) for efficient memory access. Therefore, we cannot obtain the high performance on FPGA by using the algorithm which depends on cache memory.

As an optimization for FPGA, there is a way to write OpenCL kernel so that shift registers are used for loading and storing data on FPGA [22]. In every clock cycle, a new data are shifted into the array shown in Fig. 6. Assume COLS is the number of columns of entire computation domain, we use the shift register whose size is  $2 \times \text{COLS} + 3$  for  $3 \times 3$  stencil. After inserted sufficient number of data to the shift register for updating the central element of the stencil, the computation is started. In this implementation, the parallelism between each loop iteration is extracted and loop-pipelining is generated by the compiler.

Figure 7 shows the overview of this implementation. This kernel is written as executed with a single thread which is known as task-parallel programming.

The 1-D arrays named `urows`, `grows` and others which have `rows` suffix represent the shift register, which is stored to the private memory in the format of SOA. Lines 7 to 17 show the implementation of shift registers. New data are shifted into the buffer every cycle. By unrolling, this loop allows the compiler to infer a shift register. In addition, by unrolling every loop in the kernel, the compiler attempts to pipeline and enable multiple iterations of every loop to execute concurrently.

Table 7 summarizes the resource utilization of hardware design automatically generated from our two OpenCL kernels on DE5a-Net Arria 10 FPGA. The second column is the specifications of the FPGA design. The third column named Code GPU is the resource utilization of the original kernel developed for GPUs (Fig. 5), and the last column named Code SR is the resource utilization of which was generated from the optimized kernel using the shift registers (Fig. 7), respectively. Here, the original kernel is compiled for FPGA design assumed  $N_{\text{bsize}} = 1$ .

```

1 //used as shift register (SOA)
2 float urows[2 * COLS + 3], qrows[2 * COLS + 3];
3 ...
4 int count = -(2 * COLS + 3);
5
6 //computation domain : nx*ny
7 while (count != nx*ny){
8 //shift register
9 #pragma unroll
10 for (int i = 0; i < COLS * 2 + 2; i++) {
11 urows[i] = urows[i + 1];
12 qrows[i] = qrows[i + 1];
13 ...
14 }
15 urows[COLS*2+2] = (count >= 0 && count < nx*ny)?ua[count]:0;
16 qrows[COLS*2+2] = (count >= 0 && count < nx*ny)?qa[count]:0;
17 ...
18
19 //Generate the spatial block from shift register
20 #pragma unroll
21 for(int jb = 0; jb < 3; jb++){
22 #pragma unroll
23 for (int ib = 0; ib < 3; ib++){
24 dw_g[jb][ib] = drows[jb*COLS+ib];
25 qw_g[jb][ib] = qrows[jb*COLS+ib];
26 ...
27 }
28 }
29
30 //Stencil computation follows
31 u1[1][1] = uw_g_block[1][1] - t1*((3.0*uw_g_block[1][0]
32 +qw_g_block[1][0] + 3.0*uw_g_block[1][2] + qw_g_block[1][2])
33 / 8.0*(uw_g_block[1][2] - uw_g_block[1][0]) - d2
34 + d1*(3.0*uw_g_block[1][1]+qw_g_block[1][1]) /4.0
35 - dt/32.0*(3.0*uw_g_block[1][1] + qw_g_block[1][1])*
36 ...
37
38 //store updated values
39 int updatepoint = count -(COLS+1);
40 if (count >= 2*COLS+2){
41 qa_out[updatepoint] = (u1[1][1] - q1[1][1])*
42 (u1[1][1] - q1[1][1]) / (16.0*GA);
43 ...
44 }
45 count++;
46 }

```

**Fig. 7** Overview of OpenCL implementation by using shift registers (Code SR)

We have conducted the performance benchmarking of these two OpenCL kernels. First, the computation of Code GPU whose clock frequency is 242.24MHz takes 2.5 hours for 300 steps. As we mentioned, this was implemented for GPUs and spatial blocking is a technique to utilize cache memory effectively. The original OpenCL kernel can run on FPGA design, which is far from the sufficient performance.

**Table 7** Resource Usage on DE5a-Net Arria 10 FPGA generated automatically from OpenCL kernels

	Device limit	Code GPU	Code SR
Logic	427,200	115,274 (27%)	73,521 (17%)
I/O pins	992	161 (16%)	161 (16%)
DSP blocks	1518	345 (23%)	351 (23%)
Memory bits	55,562,240	3,552,586 (6%)	5,398,798 (10%)
RAM blocks	2713	432 (16%)	502 (19%)

**Table 8** Number of floating-point operators on FPGA generated automatically from Code SR

Adder	Multiplier	Divider	Sqrt	Madder	Others	Total
74	43	22	8	22	32	201

In contrast, Code SR, the optimized kernel, implemented shift registers were well pipelined and exploited the loop parallelism by the compiler, and successive iterations launched every cycle on FPGA. That resulted in the performance improvement. The computation time of Code SR is 10.53 s which is 827 times faster than Code GPU.

The number of floating-point operators on FPGA generated from Code SR is shown in Table 8, and totally 201 operators are used. We used the compile option `-mad-enable` to extract multiply-add operations from OpenCL kernel, then the fifth column shows the number of multiply-add operations actually generated by the compiler. The number of operators in sixth column represents 6 `fpext` and 26 `fptrunc` operations which seems to be generated as auxiliary operators for Divider or Sqrt. For the performance evaluation on FPGA design, we assume that 200 operators are used, which is the same number of floating-point operations on GPU.

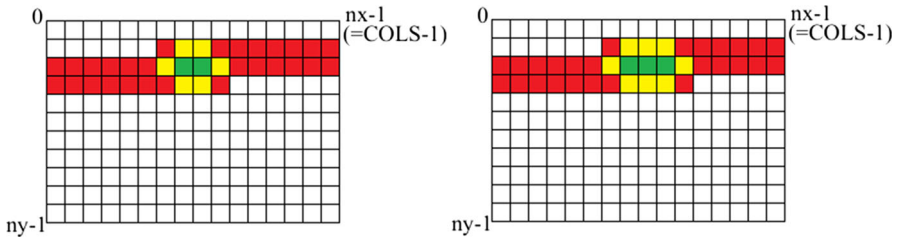
The clock frequency of generated FPGA design is 248.63 MHz. Then, we can estimate the peak performance  $0.248\text{GHz} \times 200 = 50$  GFlops. The actual performance obtained from the computation time is 42.3 GFlops which is given by the same arithmetic as obtaining the performance on GPU. This is 85% of the hardware peak performance.

We conducted the performance profiling of the kernel code of Code SR by using Altera Dynamic Profiler for OpenCL. It is confirmed that the kernel occupancy and bandwidth efficiency of data transmission kept almost 100% in the computation. However, storing data in global memory which is shown at Line 43 in Fig. 7 causes memory stall at most 8 %, and this leads to the performance drop.

### 5.3 Multiple computations techniques on one pipeline stage for increasing parallelism and its performance on FPGA

By estimating from Table 7, we expect that our device can implement at most 4 calculation pipelines. It is multiple SIMD-like operations by widening the width of data path in the same stage.





**Fig. 8** Shift register holding the data required for stencil computation in case of updating two or three cells (green-colored cells) in one computation step

**Table 9** Resource Usage on DE5a-Net Arria 10 FPGA generated from Code MC1

	Device limit	$N_{buf} = 2$	$N_{buf} = 4$
Logic	427,200	76,319 (18%)	100,450 (24%)
I/O pins	992	161 (16%)	161 (16%)
DSP blocks	1518	628 (41%)	1206 (79%)
Memory bits	55,562,240	10,269,450 (18%)	10,719,434 (19%)
RAM blocks	2713	754 (28%)	849 (31%)

In that case, the performance of OpenCL kernel on FPGA is expected to approach to one of on GPU. In our implementations, the number of the pipelines is equal to the number of data inserted into the shift register and updated on it in one computation step.

Figure 8 shows the example of designing shift registers for updating two or three cells in one computation step. The size of 1-D array which represents shift register varies relative to the number of cells updated in one computation step.

Here, let  $N_{buf}$  be the number of cells updated in the pipeline stage in one computation step, the length of shift register can be represented as  $2 \times COLS + (2 + N_{buf})$ . This can be implemented by changing each loop condition in Fig. 7 that is very similar to changing  $N_{bsize}$  for stencil computation on GPU. By compiling the code applied, this changes with `aoc`, the loops are unrolled, and multiple SIMD-like operations are generated.

Let this kernel code be Code MC1. We have conducted the benchmarking under the same condition, and generated designs for  $N_{buf} = 2$  and 4 are summarized in Table 9. Code MC1 especially consumes the resource of memory bits on FPGA by increasing the number of computations on one pipeline stage. Table 10 shows the computing time of Code MC1. Though Code MC1 gives the correct results, the performance was significantly worsened even if increasing the number of pipelines.

In this benchmarking, our implementation for both GPU and FPGA design is designed based on the spatial blocking algorithm. As mentioned before, spatial blocking is expected to use cache memory for efficient computation. The purpose of using kernel codes based on spatial blocking for this benchmarking is to compare the performance of similar OpenCL codes on different architectures. Since there is no longer room to optimize the kernel code based on spatial blocking for further high per-

**Table 10** Computation time and hardware clock frequency of Code MC1 (generated for  $N_{\text{buf}} = 2$  to 4)

$N_{\text{buf}}$	Computation time (s)	Frequency (MHz)
2	800.31	236.67
3	563.19	225.73
4	439.55	222.91

formance on FPGA design, we reconstruct the design of OpenCL kernel for FPGA design.

Figure 9 is an overview of the new OpenCL kernel named code MC2 without spatial blocking. The major modification in Code MC2 is reducing memory accesses in the kernel code.

Figure 10 illustrates the difference of memory accesses between Code MC2 and previous implementations. In Code MC1, the data in the shift register are first copied to spatial block represented by a 2-D array and the stencil computation is processed every computation step. In contrast, Code MC2 is designed to update the data in the shift registers directly (without copying to other memory spaces). In case of MOST algorithm, the transformation to Riemann invariants is conducted for the stencil computation. That appears twice before and after the update by finite difference method in a 2-D array every computation step. Code MC2 conducts the transformation only once per one data when the data inserted into the shift register. Namely, in this implementation, the shift register always has the data which has already transformed for MOST algorithm, and the computation of finite difference can be conducted to the area of stencil on the shift register. After the update, only the data which will be stored in global memory should be reverted.

In addition, the data structure is also replaced. The data such as  $D$ ,  $q$  and  $u$ ,  $v$  for the stencil computation is loaded from global memory as the format of SOA. In Code MC2, they are stored to the private memory as the format of array of structure (AOS). In our case, the structure has members which are used for the stencil computation for updating each cell, and they are aligned closely each other on the memory. That is expected to improve the efficiency of memory access. On the other hand, we also tried to store the data as SOA which is the same style as previous FPGA and GPU implementations. However, the OpenCL kernel implemented the data structure as SOA for Code MC2 failed to generate FPGA design correctly due to enormous memory usage during compilation against our compilation environment.

Besides, there are several division operations by a constant in the original MOST program. In order to reduce the number of floating-point operators of divider on FPGA design, we substituted the multiplication of inversion for division.

For the performance benchmarking, we measured the computation time of each FPGA device with  $N_{\text{buf}}$  of pipelines. Table 11 shows the computation time of Code MC2 in case of  $N_{\text{buf}} = 1$  to 5.

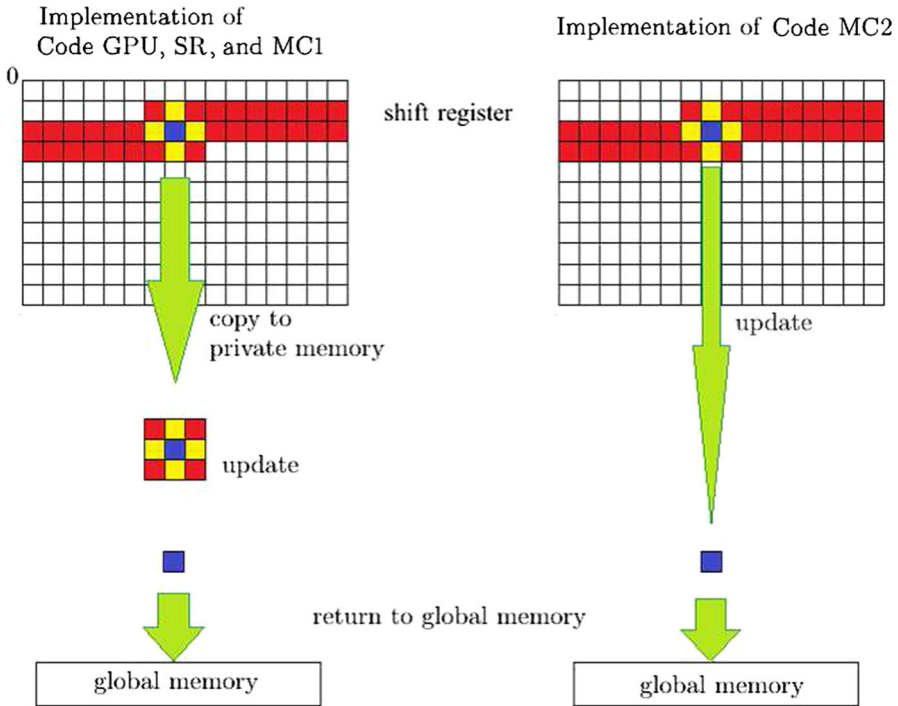
In this implementation, we obtained the performance improvement by changing  $N_{\text{buf}}$ . Though the performance is expected to increase proportional to  $N_{\text{buf}}$ ,  $N_{\text{buf}} = 4$  gives the peak whose computation time is about 6.5 s for this implementation.

```

1 //for shift register
2 typedef struct{
3   float urows, qrows, vrows;
4   float u1, q1;
5   ...
6 }SR;
7
8 #define GET(p,j,i) p[(j)*nx + (i)]
9 #define COLS 2583
10 #define BUFFER_SIZE 4
11
12 __kernel void most_sweep_2d(float *ua, *qa, ... ){//from global
13   SR shift[2 * COLS + 2 + BUFFER_SIZE]; // stored as A0S
14   int count = -(2 * COLS + 2 + BUFFER_SIZE);
15   double d1, d2, t1;
16
17   // computation domain : nx*ny
18   while (count != nx*ny ){
19     //update the shift register
20     #pragma unroll //shifting BUFFER_SIZE in the shift register
21     for (int i = 0; i < COLS * 2 + 2; i++) {
22       shift[i] = shift[i+BUFFER_SIZE];
23     }
24
25     #pragma unroll
26     for(int j = 0 ; j <BUFFER_SIZE ; j++){
27       int loadpoint = count+j;
28       shift[COLS * 2+2+j].urows =
29         (loadpoint >= 0 && loadpoint < nx*ny) ? ua[loadpoint]:0;
30       shift[COLS * 2+2+j].qrows =
31         (loadpoint >= 0 && loadpoint < nx*ny) ? qa[loadpoint]:0;
32       ... // the same for va, da
33
34       //Transformation to Riemann invariants
35       float a = 2.0 * sqrt(GA * shift[COLS * 2+2+j].qrows);
36       shift[COLS * 2+2+j].u1 = shift[COLS * 2+2+j].urows +a;
37       shift[COLS * 2+2+j].q1 = shift[COLS * 2+2+j].urows -a;
38       shift[COLS * 2+2+j].v1 = shift[COLS * 2+2+j].vrows;
39       ...
40     }
41
42     //update by FDM on the shift register
43     #pragma unroll
44     for (int i = 0; i < BUFFER_SIZE; i++){
45       //index for operation on the stencil
46       int im = COLS+i;
47       int ic = COLS+1+i;
48       int ip = COLS+2+i;
49
50       u1[ic] = urows[ic] - t1*((3.0f*urows[im] + qrows[im] +
51       3.0f*urows[ip] + qrows[ip]) * 0.125f*(urows[ip] - urows[im]) -
52       ...
53
54       int updatepoint = count -( COLS + 1);
55       //After the shift register is filled with sufficient data
56       //for stencil computation
57       if (count >= 2 * COLS + 2 && ... ) { //store
58         float umq = shift[ic].ut1 - shift[ic].qt1;
59         qa_out[updatepoint] = (umq * umq) / (16.0 * GA);
60         va_out[updatepoint] = (shift[ic].ut1 + shift[ic].qt1)*0.50;
61         ua_out[updatepoint] = shift[ic].vt1;
62       }
63       count+=BUFFER_SIZE;
64     }
65 }

```

**Fig. 9** Overview of OpenCL kernel further optimized for FPGA design (Code MC2)



**Fig. 10** Illustration for the difference of memory reference between Code MC2 and others

**Table 11** Computation time and hardware clock frequency of Code MC2 (generated for  $N_{buf} = 1$  to 5)

$N_{buf}$	Computation time (s)	Frequency (MHz)
1	12.49	201.57
2	8.79	201.97
3	7.55	197.22
4	6.49	198.33
5	8.19	201.57

In this implementation, aoc can successfully exploit the loop parallelism well and successive iterations launched every cycle in any  $N_{buf}$ . The problem for computing with pipelines seen in Code MC1 is solved. This modification also affected the compilation time. It took from 12 hours to one day for any previous kernels to generate hardware by aoc, that became less than half. It is critical for examining with various  $N_{buf}$ .

Nevertheless, the memory stall for storing to global memory still remains. Besides, the clock frequency of FPGA design generated from this OpenCL kernel reaches just 200 MHz for any  $N_{buf}$ . They are the factor for degrading the performance of this new FPGA design. When  $N_{buf} = 1$ , the performance is actually dropped comparing with Code SR.

**Table 12** Resource Usage on DE5a-Net Arria 10 FPGA, generated from Code MC2

	Device limit	$N_{\text{buf}} = 1$	$N_{\text{buf}} = 2$	$N_{\text{buf}} = 3$	$N_{\text{buf}} = 4$	$N_{\text{buf}} = 5$
Logic	427,200	75,128 (18%)	105,994 (25%)	145,972 (34%)	168,093 (39%)	178,663 (42%)
I/O pins	992	161 (16%)	161 (16%)	161 (16%)	161 (16%)	161 (16%)
DSP blocks	1518	223 (15%)	421 (25%)	618 (41%)	817 (54%)	1143 (75%)
Memory bits	55,562,240	3,842,891 (7%)	7,212,887 (13%)	7,311,115 (13%)	7,408,100 (13%)	7,521,329 (14%)
RAM blocks	2713	372 (14%)	542 (20%)	573 (21%)	589 (22%)	605 (22%)

**Table 13** Number of floating-point operators on FPGA generated from Code MC2

Adder	Multiplier	Divider	Sqrt	Madder	Others	Total
371	174	74	8	8	218	753

Here, we show the specification of a FPGA design generated from Code MC2 for  $N_{\text{buf}} = 1$  to 5. Table 12 shows the resource usage on FPGA. For the comparison, the resource usage generated in  $N_{\text{buf}} = 1$  to 5 is shown at each column. The usage of DSP blocks are increased as  $N_{\text{buf}}$  is increased. Other resource usages are keeps or increased slowly relative to DSP blocks usage. Table 13 is the number of floating-point operators in case of  $N_{\text{buf}} = 4$  which gives the best performance in our FPGA implementation. It is considered as four times of total operators for the update of one cell.

In spite of being also used `-mad-enable` option for compilation, the number of Madder was decreased comparing with Table 9 which represents the hardware of Code SR. Substituting multiplication of inversion for division also did not directly promote to decrease the number of divider operators.

Here, we also estimate the performance of this FPGA design in case of  $N_{\text{buf}} = 4$ . The clock frequency of this hardware is 198.33 MHz. In this case, we can obtain the peak performance of this hardware,  $0.198 \text{ GHz} \times 4 \times 200 \sim 160 \text{ GFlops}$ , which is the multiplication of clock frequency, the number of pipelines, and the number of floating-point operations per pipeline. Actual performance of this design under the same evaluation as previous is 68.7 GFlops. The performance for multiple computations on the pipeline stage are at most 43% of hardware peak performance. In the current implementation, there is still 20% memory stall of computation in the kernel, which is the critical bottleneck to improve the performance further.

Nagasu et al. [23] were working to design the custom hardware for MOST accelerator. Their implementation is different from our automatically generated hardware; exploiting not only spatial but also temporal parallelism in which computations for multiple timesteps are cascaded. Comparing our Code SR using shift registers with their implementation on Stratix V 5SGXA7 FPGA, the performance of our implementation is approximately same as the 160 MHz MOST accelerator in case of 1 SPE is implemented.

In the latest implementation by Nagasu et al. [10], they evaluated the performance and power consumption of their dedicated FPGA implementation of the MOST algorithm on the same Arria 10 FPGA. In addition, they presented a performance model applied both spatial and temporal parallelism. Specifically, the performance model was constructed for the case  $(n, m)$ , where  $n$  and  $m$  are spatial and temporal parallelism, respectively. Our best implementation Code MC2 with  $N_{\text{buf}} = 4$  without temporal parallelism corresponds to  $(n, m) = (4, 1)$  in their model. According to their model, we see that the required memory bandwidth (BW) is proportional to  $n$ . With  $n = 4$ , the required BW is 28.8 GB/s, while the theoretical BW of our hardware is 17 GB/s with two 4GB DDR3-1066 SODIMM. In fact, they found a large gap between the sustained performance and the theoretical performance for  $n > 1$  given  $m = 1$  (see Figure 19 in [10]). Since the best design with  $(n, m) = (1, 6)$  archived 383 Gflops in their paper,

they concluded that the temporal parallelism is the most effective optimization for the acceleration of the MOST algorithm.

Waidyasooriya et al. [24] presented the optimization methodology for general stencil computations. They also suggested to exploit temporal parallelism for the performance improvement in stencil computation. Our current implementation only exploits the spatial parallelism. Though the MOST algorithm is basically classified to five-point stencil computation, the stencil for MOST algorithm is much more complicated than the stencil computations presented in Waidyasooriya et al. [24]. Thereby, we have not appropriately implemented the temporal parallelization by using Intel FPGA SDK. We will improve our OpenCL kernels for the spatial and temporal parallelism with reference to their implementation.

## 6 Discussion

In this section, we first summarize our results of performance benchmarking presented in this paper and compare them to other related works. Additionally, we discuss the applicability of our GPU and FPGA implementation to the real-time tsunami simulation.

### 6.1 Summary of performance benchmarking and comparison with other works

We here summarize the hardware specification and benchmarking results of each implementation for GPU and FPGA in Tables 14 and 15. For the comparison on each platform, the computation time of different optimized kernels as shown in Sect. 5 is converted to the performance as floating-point operations per second (FLOPS).

Consequently, in OpenCL implementation, the original kernel computing on Radeon 280X achieved the best performance in our latest study. GPU has high peak performance for floating-point operations. In this paper, we especially presented the baseline implementation for GPU to compare with FPGA implementation. The performance on GPU is expected to get even higher by explicitly using efficient memory system represented by local memory and texture memory on GPU. However, its high power consumption often issues as the disadvantage of GPU. In our computation, Radeon GPU consumes 12.5W in the idle status without computing and 184.9W in the computing status, respectively [10].

The OpenCL kernel for computing on GPU also can be executed on FPGA design. By adopting the shift registers and loop unrolling, the OpenCL kernel on FPGA achieved the approximately same performance as the original kernel on FirePro GPU. Furthermore, increasing the number of computations on the pipeline stage also contributed the performance improvement.

FPGA is known that the power consumption is much lower than GPU on the same computation. In Nagasu et al. [10], their FPGA board consumes 25–30 W in the idle status and 29.1–45.5 W in the computing status, respectively. Therefore, the performance per power of FPGA accelerator is approximately eight times higher than that of GPU implementation. With little modification of kernel code, OpenCL kernels developed for GPU can be executed on FPGA design and obtain the same or high

**Table 14** Summary of hardware specification presented in this paper, GPU and FPGA generated from OpenCL kernels

GPU	Num of cores	Frequency (MHz)	Memory (GB)	Bandwidth (GB/s)	Peak perf. (TFlops)
Radeon R9 280X	2048	1020	3	288	4.1
FirePro W8100	2560	824	8	320	4.2
FirePro W9100	2816	930	16	320	5.2
Tesla K20c	2496	706	5	208	3.5
FPGA	Frequency (MHz)	Logic	DSP	Memory bits	RAM
Limit for DE5a-Net	–	427,200	1518	55,562,240	2713
Code GPU ( $N_{\text{buf}} = 1$ )	242.24	115,274	345	3,552,586	432
Code SR ( $N_{\text{buf}} = 1$ )	248.63	73,521	351	5,398,798	502
Code MC1 ( $N_{\text{buf}} = 4$ )	222.91	100,450	1206	10,719,434	849
Code MC2 ( $N_{\text{buf}} = 4$ )	198.33	168,093	817	7,408,100	589



**Table 15** Summary of performance achievement in this paper by OpenCL implementation for GPU and FPGA Unit: (GFlops)

$N_{\text{bsize}} \times N_{\text{bsize}}$	W9100	W8100	Radeon	Tesla
GPU				
$1 \times 1$	27.52	22.74	151.13	38.02
$2 \times 2$	41.59	43.12	185.00	38.20
$4 \times 4$	46.78	45.22	64.80	24.67
$8 \times 8$	34.75	32.83	35.75	23.28
$N_{\text{buf}}$	FPGA (Code GPU)	FPGA (Code MC1)	FPGA (Code MC2)	
FPGA				
1	0.05	42.34 (Code SR)	35.70	
2	–	0.56	50.72	
3	–	0.79	59.05	
4	–	1.01	68.70	
5	–	–	54.43	

performance as GPU computation. Designing hardware architecture and logic circuits appropriately is difficult and takes much time. Actually the Verilog HDL code generated from our OpenCL kernel consisted from totally more than 600,000 lines, that can be relieved by writing several hundred lines of OpenCL kernel code with the specific compiler. In our case for the MOST algorithm, the performance was improved by the kernel modification to increase the number of computations per a pipeline. That can be achieved by unrolling a `for` loop and storing them as Array of Structure in the OpenCL kernel. Therefore, OpenCL programmers have an additional environment of the application, though implementing optimal OpenCL kernel for FPGA design generation requires several trials.

## 6.2 Estimation of the applicability for real-time simulation

Here, we estimate the applicability of our GPU and FPGA implementation for the real-time tsunami simulation by using currently obtained results. As the practical evaluation, we evaluate our implementations based on the phase velocity derived from the shallow water equations. The phase velocity of tsunami is obtained as  $c = \sqrt{gH}$ , where  $g$  is the gravitational acceleration and  $H$  is the sea depth. Given the distance between the coastal area and the epicenter  $D$ , the numerical simulation must be finished within  $D/c$ .

The estimation is presented by referring following past disasters, the 2011 Tohoku earthquake and tsunami in Japan. In this case, the epicenter is located at  $D = 180$  km from the coastal region.

Assuming the average sea depth  $H$  is 1500 m, we obtain the phase velocity  $c = 436$  km/h. Computing the arrival time of tsunami under these conditions, we find that

**Table 16** Estimation of the application of our OpenCL computation on Radeon GPU and FPGA for the tsunami in Tohoku, Japan, 2011

Case	Radeon (Code GPU)		FPGA (Code MC2)	
Covered domain ( $L \times L$ )	200 × 200 km <sup>2</sup>			
$N$	2581			
Grid resolution $\Delta x$	77.48 m			
Required time for updating 1 cell	1.08 × 10 <sup>-9</sup> s		2.91 × 10 <sup>-9</sup> s	
Constant $\alpha$ in Eq. (9)	1.0	0.5	1.0	0.5
Upper limit of $\Delta t$ (s)	0.64	0.32	0.64	0.32
Used $\Delta t$ (s)	0.50	0.25	0.50	0.25
Estimated computation time (s)	23.33	46.66	62.83	125.66

is approximately 27 min. In this earthquake, tsunami has actually arrived at the coastal area of Fukushima with 30 min. This estimate is fairly correct.

We then estimate the computation time for this required situation by using our OpenCL kernel on Radeon GPU (Code GPU) and FPGA (Code MC2). The computation time is calculated by assuming the computation domain and total grids  $N \times N$ , and determining  $\Delta t$ . Here, we consider the computation domain which covers  $L \times L$  km<sup>2</sup> area where we set  $L = 200$ . To simplify the calculation, assume the computation domain is covered by the square grid (2581 × 2581). For the accurate simulation, we must choose appropriate value for time step  $\Delta t$  to hold Eq. (8) shown in Sect. 2. Here, we modify Eq. (8) as follows to multiply the constant for reliability  $\alpha$  ( $0 < \alpha \leq 1$ ) on the right-hand side and use it in the evaluation.

$$\Delta t \leq \alpha \frac{\Delta x}{\sqrt{gH}}. \quad (9)$$

Table 16 presents the condition for the accurate simulation and the estimated time for tsunami simulation under that condition. In this case,  $\Delta x$  is given as 200 km/2581 = 77.48 m.

When we set  $\alpha = 1.0$  as the optimistic estimation which gives the strict limit for  $\Delta t$  as shown in Eq. (8),  $\Delta t$  must be smaller than 0.64 s. If we choose  $\Delta t = 0.5$  s, 3240 total computation steps are required to simulate tsunami for 1620 s in real time. Using the elapsed time for updating 1 cell which is obtained from our benchmarking results on GPU and FPGA presented in previous sections (fifth row in Table 16), we can estimate the computation time for this situation. As shown in ninth row, the computation time on each hardware is estimated at most 70 s. Those are much shorter than 1620 s in real time.

To ensure the higher reliability of our simulation for the practical application, we have another estimation with  $\alpha = 0.5$ . In this case, the upper limit for  $\Delta t$  is about 0.30. When we use  $\Delta t = 0.25$  s, 6480 steps are required to simulate the tsunami for 1620 s in real time. The computation time of each environment under this condition is still shorter than 1620 s in real time. Therefore, the computation by using either OpenCL

kernels for Radeon GPU or FPGA is applicable to the forecast for this situation. Note that the performance of our implementation is independent of each constant.

Finally, we remark how smaller  $\Delta x$  we can use to improve the accuracy of simulation. In other words, we estimate how larger  $N$  we can use in our computation environment for tsunami forecasting. Let the computation domain be  $L \times L$  divided by  $\Delta x \times \Delta x$  grids and the total simulation time be  $T$  with the time step of  $\Delta t$ , respectively. We obtain the total number of grids computed by our stencil computation as

$$\frac{T}{\Delta t} \times N^2 = \frac{T}{\Delta t} \times \left( \frac{L}{\Delta x} \right)^2. \quad (10)$$

When the computation time required for updating one cell in one time step is  $f$ , the total computation time  $T_{\text{comp}}$  is given as following.

$$T_{\text{comp}} = \frac{T}{\Delta t} \times \left( \frac{L}{\Delta x} \right)^2 \times f \quad (11)$$

Assuming  $T_{\text{comp}} = T$ , we find the lower limit for  $\Delta x$  by using Eq. (9).

$$\Delta x = \sqrt[3]{\frac{\sqrt{gH} \times L^2 f}{\alpha}} \quad (12)$$

Thus, we can calculate  $\Delta x$  and  $N$  by using this formula and estimate the computation time for particular simulation.

In case of the computation on Radeon GPU (by using Code GPU), we obtain the limit of  $\Delta x$  as  $\Delta x = 21.88$  m by substituting  $L = 200$  km,  $H = 1500$  m,  $f = 1.08 \times 10^{-9}$  s, and  $\alpha = 0.5$ , respectively. To cover the computation domain with this  $\Delta x$ ,  $N = L/\Delta x$  is at most 9139. On the other hand, in case of the computation on FPGA (by using Code MC2), we obtain the limit of  $\Delta x$  as  $\Delta x = 30.45$  m by using  $f = 2.91 \times 10^{-9}$  s. In this case,  $N = L/\Delta x$  is at most 6568.

## 7 Conclusion

We developed our tsunami simulation codes based on the MOST algorithm applied spatial blocking and parallelized them by OpenCL. OpenCL kernels can be executed not only on GPUs but also on FPGAs and other architectures. The best result of performance benchmarking on GPU with a  $2581 \times 2879$  computation grid is currently that OpenCL code with  $2 \times 2$  spatial blocking takes approximately 2.41 s (185.0 GFlops) on AMD Radeon R9 280X GPU for 300 time steps.

In this paper, we aimed to achieve the high performance on FPGA design by using the different OpenCL kernels from GPU implementation. Here, we used the compiler supported by Intel FPGA SDK for generating the FPGA design automatically that enables us to write OpenCL kernel the same way as for GPUs. Though the same kernel as developed for GPU can be executed on FPGA, it did not achieve the expected

performance. The performance of an optimized kernel implemented shift registers reaches to the original kernel running on FirePro GPU. In addition, we reconstructed the GPU kernel code for FPGA implementation, and our latest OpenCL kernel named Code MC2 is able to support the SIMD-like operations to increase parallelism on FPGA design. In case of implementing four computations per a pipeline, our optimized kernel on FPGA design achieved 6.49 s (68.7 GFlops) under the same condition as evaluations on GPU.

However, we have to write OpenCL kernel in the specific ways in order to obtain the high-performance hardware as MOST accelerator. In our current study, memory stall, especially during the storing to global memory, interrupts the fluent computation in OpenCL kernel, and that makes the performance with Code MC2 away from hardware peak. We will research the coding techniques in OpenCL which is translated to hardware computing efficiently. In particular, there is room to improve the performance by exploiting temporal parallelism presented by other papers.

On the other hand, the current implementation achieves sufficient performance in terms of applicability for the real-time simulation. In the case of the 2011 Tohoku earthquake and tsunami in Japan, our computation is more than 20 times faster than real time. In the future, we extend our MOST program to compute on the nested grids which has the high-resolution grids computed precisely for the intensive area cooperating with other studies [18]. In that case, it is required the several variations of computation kernel for each computation domain. It is much easier and takes fewer time to customize OpenCL kernel than manually designing FPGA.

Finally, we note that our OpenCL implementation is also applicable to distributed-memory clusters. One possible application is that we simulate different models concurrently on individual nodes in such cluster. Additionally, our OpenCL implementation can be easily used in modeling one large computation in parallel for various GPU clusters. In fact, we are currently working on the optimization of our parallel implementation for GPU clusters using Message Passing Interface (MPI). Furthermore, although we currently have no available distributed cluster with FPGA, our OpenCL kernel should easily work on such clusters available in near future. For instance, Amazon Elastic Compute Cloud is offering a compute instance with FPGA (called F1 instance). We will evaluate the performance of our MPI+OpenCL implementation on the F1 instance in future publications.

## References

1. Tsunami Engineering Laboratory, International Research Institute of Disaster Science, Tohoku University (2015). <http://www.tsunami.civil.tohoku.ac.jp/hokusai3/E/index.html>. Accessed 15 Dec 2015
2. Gidra H, Haque I, Kumar N, Sargurunathan M, Gaur MS, Laxmi V, Zwolinski M, Singh V (2011) Parallelizing TSUNAMI-N1 using GPGPU. In: High Performance Computing and Communications (HPCC), pp 845–850
3. Acuna MA, Aoki T (2014) AMR multi-GPU accelerated tsunami simulation. In: The 1st International Conference on Computational Engineering and Science for Safety and Environmental Problems, pp 708–710
4. Fujita M (2015) Tsunami simulation on FPGA/GPU and its analysis based on statistical model checking. <http://cmacs.cs.cmu.edu/seminars/slides/fujita3.pdf>. Accessed 1 Nov 2015

5. Titov VV (1989) Numerical modeling of tsunami propagation by using variable grid. In: Proceedings of the IUGG/IOC International Tsunami Symposium, pp 46–51. Computing center Siberian Division USSR Academy of Sciences, Novosibirsk, USSR
6. Titov VV, Gonzalez FI (1997) Implementation and testing of the method of splitting tsunami (MOST) model. NOAA Technical Memorandum ERL PMEL-112
7. Takano S, Hayashi K, Vazhenin A, Marchuk A (2015) Hybrid tsunami modeling infrastructure: tsunami source data and bathymetry editor. In: International Workshop on Applications in Information Technology (IWAIT-2015), pp 21–24
8. The Khronos Group (2016) OpenCL. <http://www.khronos.org/opencl/>. Accessed 31 Jan 2016
9. Kono F, Nakasato N, Hayashi K, Vazhenin A, Sedukhin S, Nagasu K, Sano K, Titov V (2015) Parallelization of tsunami simulation on CPU, GPU and FPGAs. In: The International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), poster paper no 82 (2 pages)
10. Nagasu K, Sano K, Kono F, Nakasato N (2017) FPGA-based tsunami simulation: performance comparison with GPUs, and roofline model for scalability analysis. *J Parallel Distrib Comput* 106:153–169
11. Takei Y, Waidyasooriya H, Hariyama M, Kameyama M (2014) Design of an FPGA-based FDTD accelerator using OpenCL. In: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp 371–375
12. Tatsumi S, Hariyama M, Miura M, Ito K, Aoki T (2015) OpenCL-based design of an FPGA accelerator for phase-based correspondence matching. In: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp 90–95
13. Waidyasooriya H, Hariyama M, Kasahara K (2016) Architecture of an FPGA accelerator for molecular dynamics simulation using OpenCL. In: IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)
14. Yinger J, Nurvitadhi E, Capalija D, Ling A, Marr D, Krishnan S, Moss D, Subhaschandra S (2017) Customizable FPGA OpenCL matrix multiply design template for deep neural networks. In: 2017 International Conference on Field Programmable Technology (ICFPT), Melbourne, Australia, pp 259–262
15. Wang D, Xu K, Jiang D (2017) PipeCNN: an OpenCL-based open-source FPGA accelerator for convolution neural networks. In: 2017 International Conference on Field Programmable Technology (ICFPT), Melbourne, Australia, pp 279–282
16. Roozmeh M, Lavagno L (2017) Implementation of a performance optimized database join operation on FPGA-GPU platforms using OpenCL. In: IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), Linköping, pp 1–6
17. Houtgast E, Sima VM, Al-Ars Z (2017) High Performance streaming Smith–Waterman implementation with implicit synchronization on intel FPGA using OpenCL. In: 2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE), Washington, DC, pp 492–496
18. An G, Marchuk K, Hayashi A, Vazhenin P (2015) Trans-boundary realization of the nested-grid algorithm for trans-pacific and regional tsunami modeling. *Bull Novosib Comput Center Ser Math Model Geophys* 18:35–47
19. Okada Y (1985) Surface deformation due to shear and tensile faults in a half-space. *Bull Seismol Soc Am* 75:1135–1154
20. AMD (2015) What the difference between AMD Radeon and AMD FirePro graphics cards? <http://support.amd.com/en-us/search/faq/84>. Accessed 31 Oct 2015
21. Kono F, Nakasato N, Hayashi K, Vazhenin A, Sedukhin S (2017) Performance evaluation of tsunami simulation using OpenCL on GPU and FPGA. In: IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc-17)
22. ALTERA (2017) Intel FPGA SDK for OpenCL programming guide. [https://www.altera.com/content/dam/altera-www/global/en\\_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf](https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf). Accessed 15 Mar 2017
23. Nagasu K, Sano K, Kono F, Nakasato N (2016) Parallelism for high-performance tsunami simulation with FPGA: spatial or temporal? In: The 24th IEEE International Symposium on Field-Programmable Custom Computing Machines
24. Waidyasooriya HM, Takei Y, Tatsumi S, Hariyama M (2017) OpenCL-based FPGA-platform for stencil computation and its optimization methodology. *IEEE Trans Parallel Distrib Syst* 28(5):1390–1402