Parallelization of Tsunami Simulation on CPU, GPU and FPGAs

Fumiya Kono *, Naohito Nakasato, Kensaku Hayashi, Alexander Vazhenin, Stanislav Sedukhin
Kohei Nagasu †, Kentaro Sano, Vasily Titov ‡
*University of Aizu, Japan
†Tohoku University, Japan
‡NOAA Center for Tsunami Research, USA

I. Introduction

After 2011 off the Pacific coast of Tohoku Earthquake, people in Japan have the interest in disaster prevention and its earlier prediction. In this poster, we evaluated our tsunami simulation using various parallelization techniques on different parallel architectures. Our target program for acceleration is MOST (Method of Splitting Tsunami) [1] [2] which is a shallow water equations solver for numerical simulation of tsunami.

We parallelized the C/C++ code based on MOST. Besides, we optimized this code in several ways. We show the performance evaluation of some parallelized variants.

In addition, we are developing MOST hardware accelerator implemented with Field-Programmable Gate Arrays (FPGAs) for high-performance and power-efficient simulation. We present the preliminary evaluation of our FPGA implementation.

II. MOST and its implementation for parallelization

MOST solves the shallow water equations represented by following partial differential equations

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

Here, $H = H(x, y, t) = \eta(x, y, t) + D(x, y, t)$, where η and D are the wave height and the depth of water, respectively, u and v are the wave velocity in each direction, g is gravitational acceleration.

In the calculation steps, we adopted finite difference method to discretize each parameters in the 2-D arrays. Especially, the Euler method is applied for time integration.

We first rewrote the original MOST program in Fortran into a code in C++ language for parallelization (Code 1) and then applied three parallel optimizations techniques. First optimization is the insertion of matrix transposition before and after updating along latitude to diminish cache miss (Code 2). Second technique is the merging of the calculation for each row or column to update the 2-D array all at once in one step (Code 3). Finally, third technique is applying a spatial blocking to the second optimization (Code 4). Entire 2-D array is divided into spatial blocks in advance and the calculation is conducted for every spatial block.

OpenMP directives are inserted into the most outer loop of every multiple nested loops for one step calculation. OpenACC directives are used to parallelize Code 4 for every spatial block. Furthermore, we implemented OpenCL kernel to parallelize Code 4.

III. Evaluation

We have conducted the benchmarking of our parallelized MOST and evaluated the performance. We first compared the calculation time of 300 steps on multi-CPU systems and Many Integrated Core Architecture (MIC) (see Table I) with OpenMP parallelization.

CPU	Intel Xeon	Xeon Phi (MIC)
Num of cores	8	60
Num of threads	16	240
Clock frequency	2.6 GHz	1.052 GHz
Memory size	32 GB	8 GB
Single Precision	0.666 Tflops	2.022 Tflops

TABLE I. Execution environment (multi-CPUs, MIC)

The grid size of calculation is 2581×2879. This is derived from the bathymetry size of entire Pacific ocean used by the original MOST program. First, we present the computing time of Code 1 and Code 2 in Table II.

	Code1		Code2	
Threads	Xeon	MIC	Xeon	MIC
1	290.71	8330.41	261.32	5339.13
4	223.83	8020.50	96.78	1487.54
16	227.75	7812.61	51.73	522.81
64	—	7770.91	—	292.94
240	—	8164.10	—	268.65

TABLE II. Computing time of Code 1 and Code 2, Unit: (sec.)

Code1 is the same algorithm as original MOST and simply paralelized with OpenMP directives described in Section 2. It took 2.3 hours to finish the calculation by serial execution on MIC and this is much slower than the multi-CPU system. Generally, for one thread, the performance of MIC is lower than that of other multi-CPU systems due to the memory access speed. Code 1 shows no performance improvement in parallel processing due to massive memory bandwidth requirement in the original algorithm.

On the other hand, Code 2 showed higher performance on MIC. Matrix transposition solved performance drop originated from memory access pattern in the update for column direction.

	Code3		Code4	
Threads	Xeon	MIC	Xeon	MIC
1	471.96	13276.12	357.18	5958.12
4	122.82	3478.07	88.12	1551.61
16	41.22	522.81	26.57	403.20
64	—	292.94	—	121.75
240	—	136.84	—	70.49

TABLE III. COMPUTING TIME OF CODE 3 AND CODE 4, UNIT: (SEC.)

The another result of benchmarking is shown in Table III. Code 3 updates the 2-D array of results all at once in one step to obtain higher parallelism. Code 4 is applied a spatial blocking based on Code 3 and its calculation is parallelized every block.

Let the number of block size be b. We found that in case of b=32 achieved the best performance. Therefore, the result of Code4 shown in Table III adopted bxb blocks. As we can see, MIC showed lower performance than multi-CPU systems for all cases. It takes 3.8 hours to finish sequential calculation of Code 3 on MIC. However, it achieved 100x speedup when 240 threads are used. Moreover, Code 4 showed the highest performance on each architecture. It takes approximately 1 minute for calculation on MIC and 30 seconds on the multi-CPU system.

Furthermore, we measured computing time of Code 4 using OpenACC or OpenCL on two GPU architectures (see Table IV).

GPU	Tesla K20	Radeon HD7970
Num of GPU cores	2496	2048
Clock frequency	706 MHz	925 MHz
Memory size	5 GB	3 GB
Single Precision	3.52 Tflops	3.79 Tflops

TABLE IV. EXECUTION ENVIRONMENT (GPUS)

Our OpenACC code is translated into NVIDIA GPU code by the compiler so that we show the computing time on NVIDIA Tesla K20 (see Table V). For better performance, we adjusted the size of vector and worker directives in OpenACC, and found that b=64 achieved the best performance. We also minimized data transfer between CPU and GPU. Before starting computation, data transfer from CPU to GPU is conducted once. When computation is started, the results are not transferred until the output routine is called. As a result, it takes 18.35 seconds to complete the calculation. This is faster than that on multi-CPU systems.

Block size	Computing time
4x4	72.98
16x16	31.15
24x24	20.86
32x32	20.87
64x64	18.35

TABLE V. COMPUTING TIME OF CODE 4 ON TESLA K20, UNIT: (SEC.)

Besides, we also show the performance of our OpenCL code (see Table VI). OpenCL has capability running on various parallel environment. Therefore, we conducted the benchmarks on various architectures including AMD Radeon GPU.

Regarding OpenCL parallelization, we also tuned the block size. We found that in case of b=1 achieved the best performance. Therefore, the results on Table VI show in that case.

Device	Computing time
Xeon	71.95
MIC	207.20
Tesla	15.46
Radeon	3.20

TABLE VI. COMPUTING TIME OF CODE 4 USING OPENCL, UNIT: (SEC.)

It takes approximately 72 and 207 seconds on Xeon and MIC, respectively. Both performance are inferior to OpenMP implementation of Code 4. OpenCL code could not achieve high performance on multi-core or many core systems.

On the other hand, it achieved great performance on GPU. It takes approximately 15 seconds on Tesla GPU. Hence, the performance of OpenCL code is higher than that of OpenACC code.

On Radeon GPU, we have the highest performance in our implementation such that the computing time is approximately 3 seconds. In this implementation, we do not have optimization of OpenCL code yet such as using shared memory on GPU. There is a possibility that the performance is improved with optimization.

IV. IMPLEMENTATION OF MOST ON FPGAS

As an alternative way of parallelization, we are developing a hardware accelerator of MOST with FPGAs, which have potential for high-performance and power-efficient computation. The hardware accelerator utilizes both the temporal parallelism and the spatial parallelism by deeply pipelining a data path and deploying more pipelines, respectively.We have accomplished to implement 1-D shallow water equation solver on ALTERA Stratix V FPGA [3] .

Besides, we designed FPGA-based stream computation of tsunami simulation based on MOST to solve the 2-D shallow water equations using deeply-pipelined stream processing element (SPE). With the total computing cycles at 150 MHz, we obtained the computing time for the FPGA-based simulation. The single SPE on FPGA took 14.9 seconds for 300 time steps with the 2581x2879 grid. The power consumption of the FPGA board was 28.1 W during the computation.

REFERENCES

[1] Vasily V. Titov and F. I. Gonzalez, "Implementation and testing of the Method of Splitting Tsunami (MOST) model", "NOAA Technical Memorandum ERL PMEL-112" (1997)

[2] M. Lavrentiev-jr, A. Romanenko, V. Titov and A. Vazhenin,"High-Performance Tsunami Wave Propagation Modeling",Parallel Computing Technologies Lecture Notes in Computer Science volume 5698, pp.423-434 (2009)

[3] K. Sano, F. Kono, N. Nakasato, A. Vazhenin and S. Sedukhin,"Stream Computation of Shallow Water Equation Solver for FPGA-based 1D Tsunami Simulation", Highly Efficient Accelerators and Reconfigurable Technologies (2015)