Parallelization of Tsunami Simulation based on CPU, GPU and FPGAs

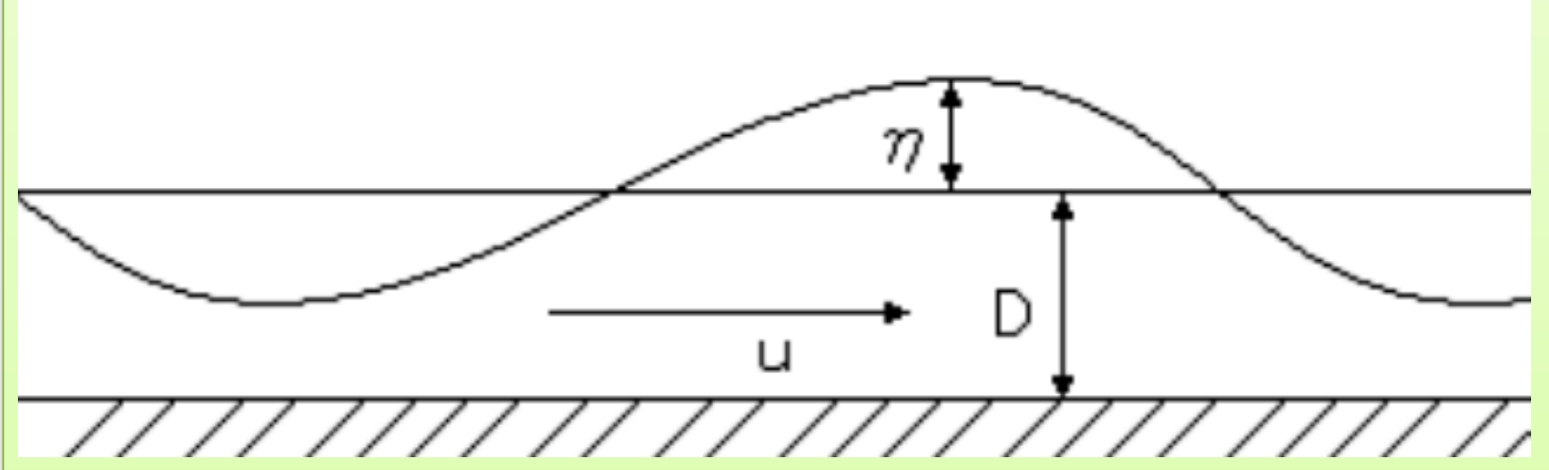
Fumiya Kono, Naohito Nakasato, Kensaku Hayashi, Alexander Vazhenin, Stanislav Sedukhin (University of Aizu, Japan)
Kohei Nagasu, Kentaro Sano (Tohoku University, Japan), **Vasily Titov** (NOAA Center for Tsunami Research, USA)

MOST (Method of Splitting Tsunami)

MOST is developed as a solution of shallow water equations for tsunami numerical simulations.

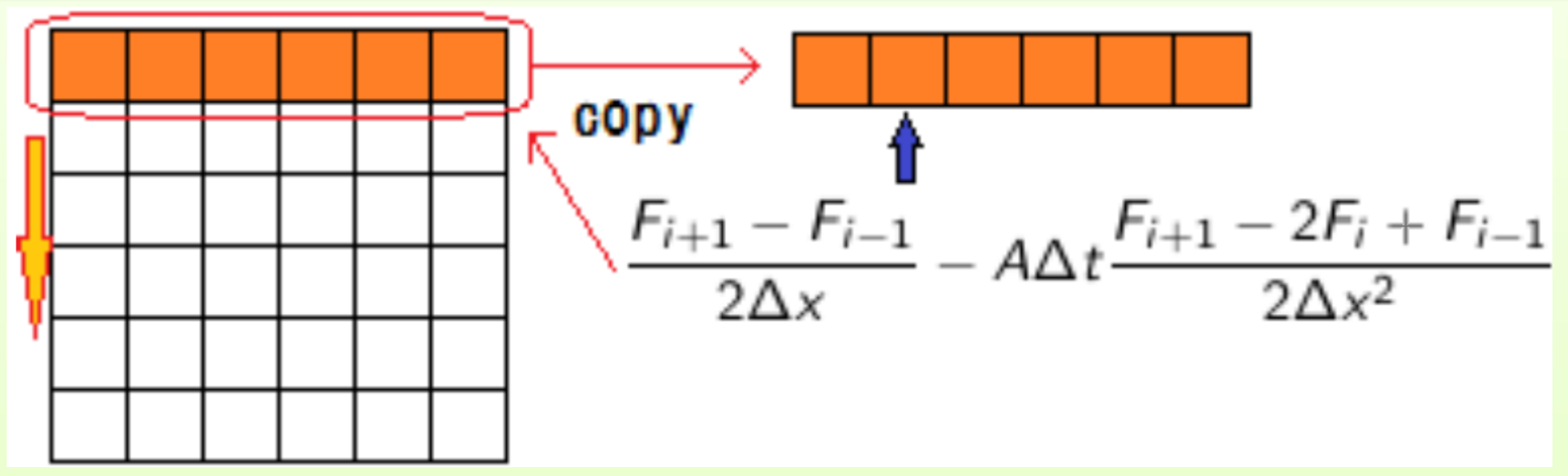
$$\begin{cases} H_t + (uH)_x + (vH)_y = 0 \\ u_t + uu_x + vu_y + gH_x = gD_x \\ v_t + uv_x + vv_y + gH_y = gD_y \end{cases}$$

Partial differential equation which represents shallow water



1-D representation for wave propagation

η : wave height **D**: depth of the sea **H**: total wave height (H = η + D)
u, **v**: wave velocity component **g**: gravitational acceleration

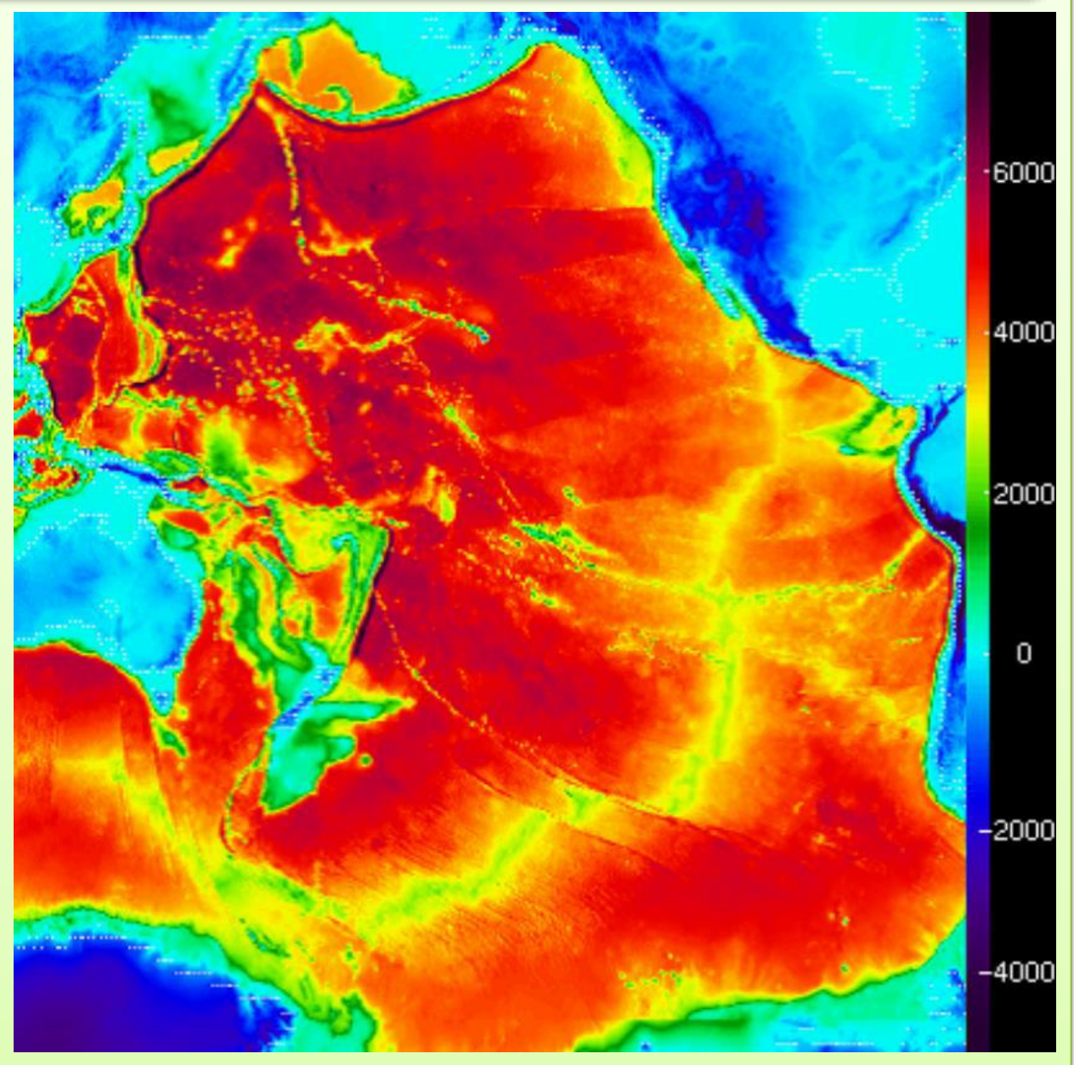


$$\frac{F_{i+1} - F_{i-1}}{2\Delta x} - A\Delta t \frac{F_{i+1} - 2F_i + F_{i-1}}{2\Delta x^2}$$

***Finite Difference Method* and *The Euler method* for time integration are used for MOST algorithm.**

For all rows or columns in the 2-D array :

- Copy every one row or column into the temporary 1-D array.
- Apply the 3 point central difference each element in the array.
- Copy back the results. (**velocity** and **wave height** are updated.)



For larger computation regions, (e.g. entire Pacific Ocean)

- Compensation due to the difference of latitude is required.（Due to the curvature of the Earth)

Purpose of this study and Solution

- We accelerate the performance of the tsunami simulation, MOST, to compute the elapsed time of a model **faster than real time**. (It is useful for tsunami prediction and evacuation.)

- We parallelize MOST algorithm written in C++.
  ⇒ We developed three optimized algorithms for parallelization.
- We parallelize our optimized codes using **OpenMP, OpenACC** and **OpenCL**.

We benchmarked our parallelized codes on

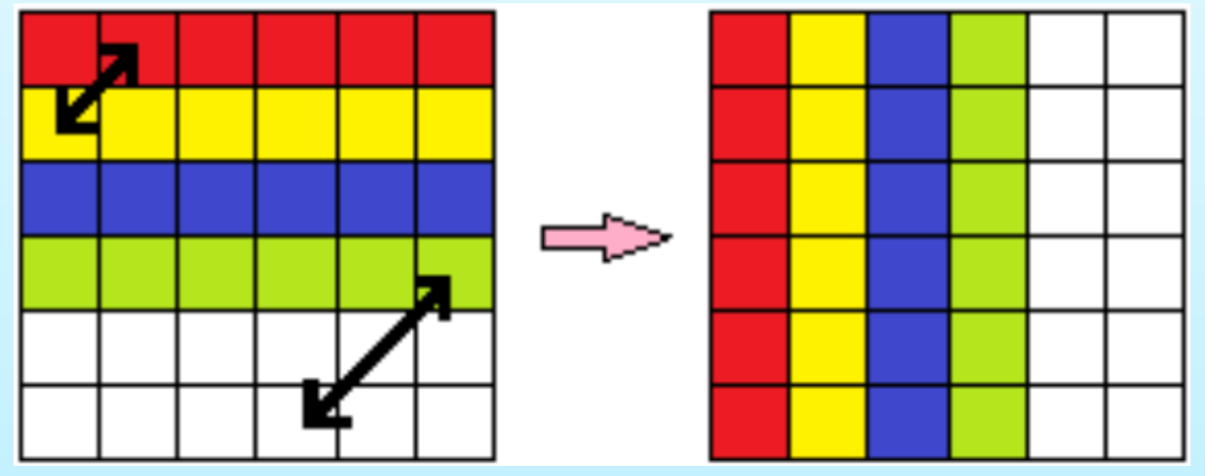
- **Intel Xeon (Multi-CPU systems)**
- **Intel Xeon Phi (Many Integrated Core Architecture, MIC)**
- **NVIDIA Tesla K20 (GPU)**
- **AMD Radeon HD7970 (GPU)**

Implementation on **FPGAs** (as a reference) is also presented.

Algorithms for parallelization

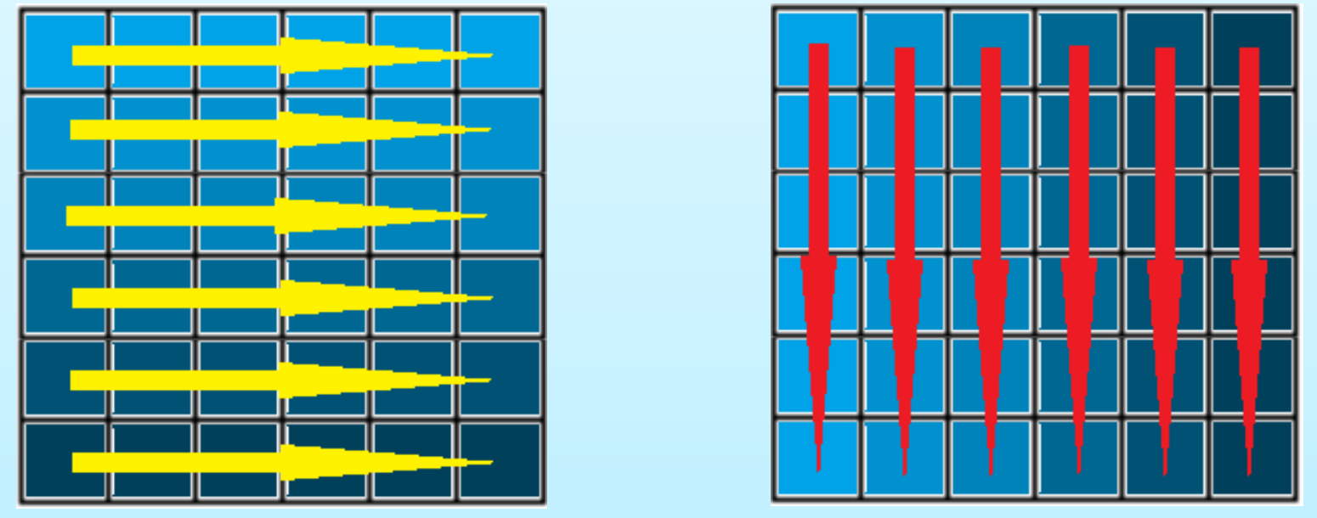
Let an original code be **Code 1**. We made some optimizations based on **Code 1**.

Code 2: Matrix Transposition



- Reduce cache miss rate when every column is copied to the temporary array.
(Due to row-wise storing data in C/C++)
- Matrix transposition is conducted twice before and after calculation along latitude (y-direction).

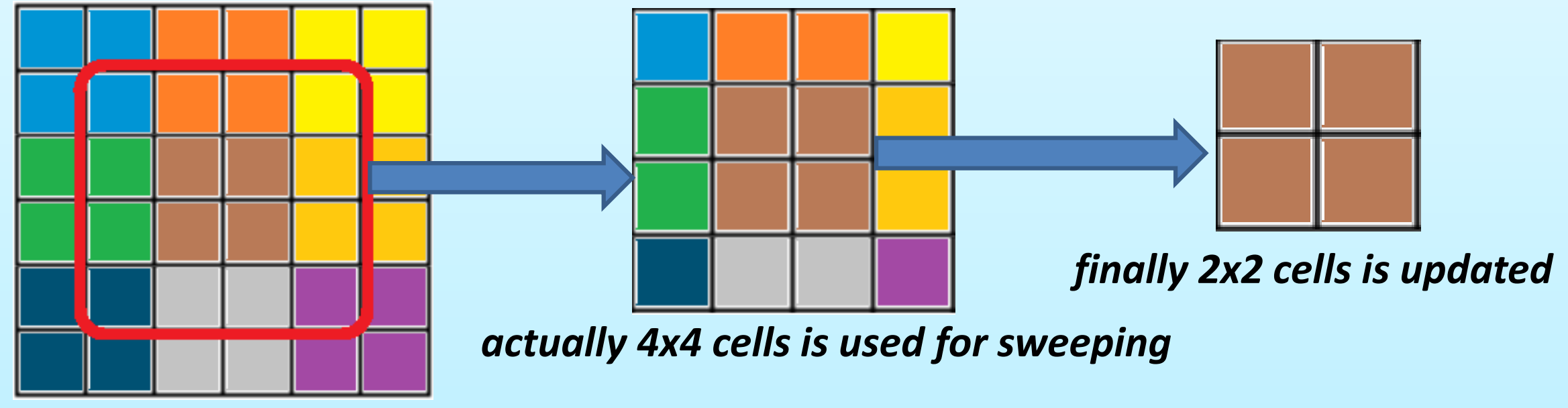
Code 3: Update the 2-D array all once



Sweep along longitude (x-direction) ⇒ along latitude (y-direction)

- Copy entire 2-D array into the temporary 2-D array.
- Sweep every row in the temporary array.
- Sweep every column in the temporary array.
- Copy back to the original 2-D array.

Code 4: Spatial blocking based on Code 3



Process of an update the blown colored block (in the case of every block is updated)

- Separate entire 2-D array into **spatial blocks**.
- Apply the process of **Code 3** to every block.
- To update ***N*** x ***N*** cells, (***N+2***) x (***N+2***) cells is actually required.
(Neighbor cells are used to update each cell in the block)

This implementation gives us **the highest level of parallelism on GPU**.

Performance Benchmarking

- We parallelized Codes 1 to 4 using **OpenMP** and Code 4 using **OpenACC** and **OpenCL**.
- Codes 1 to 3 were parallelized by inserting the directive to the **most outer loop**. (Computations which sweeps every row are parallelized.)
- Code 4 with the spatial blocking technique is parallelized on **every spatial block**.
- In the implementation by OpenACC on GPU, we tuned the size of **vector** and **worker** directives. (We tried different combination of those parameters to obtain the best performance.)

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 (AVX512)	0.666 Tflops	2.022 Tflops

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

Execution environments for our benchmarking

We measured computing time of

- Code 1 to 4 parallelized by OpenMP on Multi-CPU systems and MIC.
- Code 4 parallelized by OpenACC on NVIDIA GPU and OpenCL on AMD GPU.

Total computing steps : 300
Size of computing domain : 2581x2879
(This is derived from the bathymetry data for entire Pacific Ocean used for original MOST program.)

Evaluation and Results

	Code1		Code2		Code3		Code4	
Threads	Xeon	MIC	Xeon	MIC	Xeon	MIC	Xeon	MIC
1	290.71	8330.41	261.32	5339.13	471.96	13276.12	357.18	5958.12
4	223.83	8020.50	96.78	1487.54	122.82	3478.07	88.12	1551.61
16	227.75	7812.61	51.73	522.81	41.22	522.81	26.57	403.20
64	—	7770.91	—	292.94	—	292.94	—	121.75
240	—	8164.10	—	268.65	—	136.84	—	70.49

Results of benchmarking on Multi-CPU systems and Many Integrated Core Architecture (OpenMP)

For Code4, when block size is ***32x32***, we obtained **the best performance**.

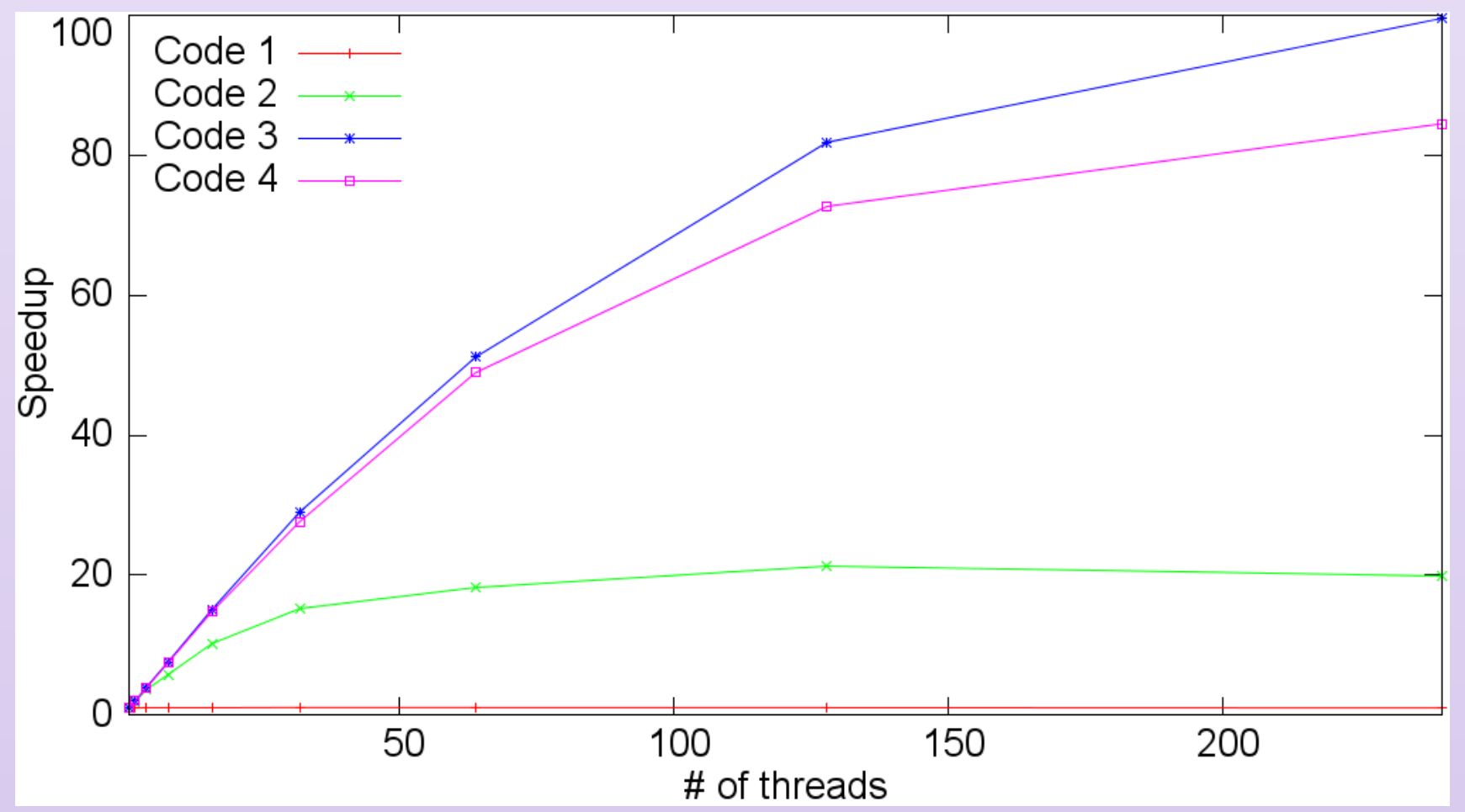
Computing time for **300 steps**
(Unit : sec.)

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

Results of benchmarking on Tesla GPU (OpenACC)

MIC obtained *lower* performance than multi-CPU systems in any cases due to its **memory access speed**.

- On **multi-CPU systems** (Intel Xeon), speedup is **proportional to the number of threads**.
- On **MIC**, speedup depends on the implementations.
  - **Code1** did not speedup using parallelization.
  - **Code2** obtained **20x** speedup at maximum and computing times are also shorten.
    ⇒ Matrix transposition **improved the cache hit ratio** for accessing the 2-D array.
  - **Code3** shows the best speedup (**100x**) against serial execution.
    ⇒ The number of data copy to the temporary array is minimized in this implementation.
  - Computation of **Code4** **is the fastest** **(about 1 minute)** of 4 codes (with 32x32 block).
    ⇒ This is expected greater performance on GPU.



Speedup of parallelized codes against serial execution on MIC

Device	Computing time
Xeon	71.95
MIC	207.20
Tesla	15.46
Radeon	3.20

Results of benchmarking on CPU and GPU (OpenCL)
(Computing time of 1x1 block is shown which achieved the best performance.)

- **64x64** block obtained the best performance.
- 300 steps for tsunami propagation can be calculated within approximately **18 seconds** by **OpenACC** codes.
- **Data transfer between CPU and GPU is minimized**.
  ⇒ When computation is started, the results are not transferred until the output routine is called.

OpenCL code **on AMD GPU is the fastest** for all implementations.

- In this implementation, 1x1 block is the optimal value for fast computation.
- Larger block size gives performance drop.
- Calculation on AMD GPU can be finished within approximately **3 seconds**.

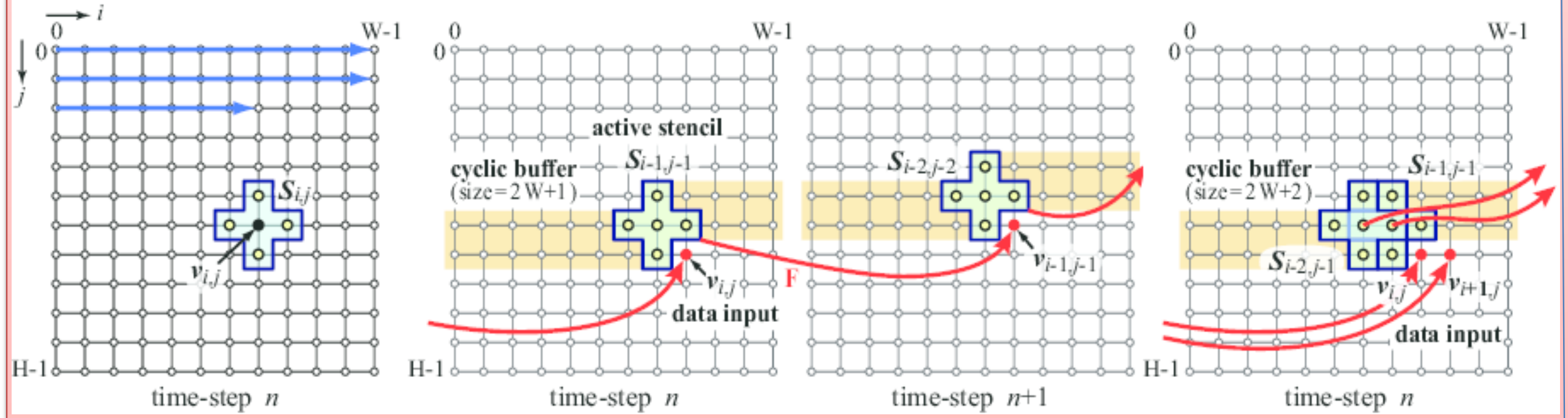
Implementation of MOST on FPGAs

FPGAs (Field-Programmable Gate Arrays) has potential to achieve high-performance and power-efficient computation.

We are designing FPGA-based hardware accelerator of MOST which depends on

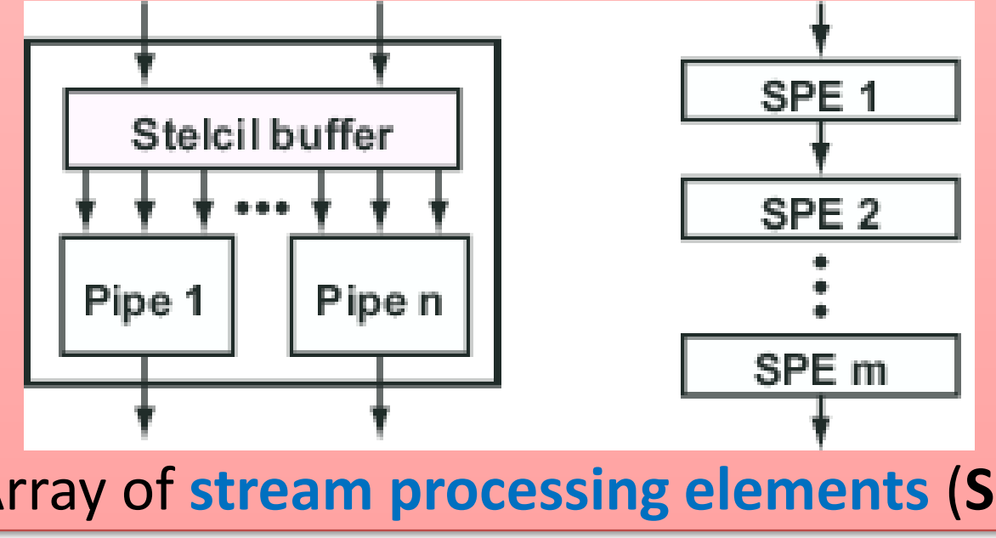
- **temporal parallelism by pipelining** to increase the number of operations per cycle
- **spatial parallelism by deploying more pipelines** operating in parallel

Iterative stencil computation for stream computing architecture



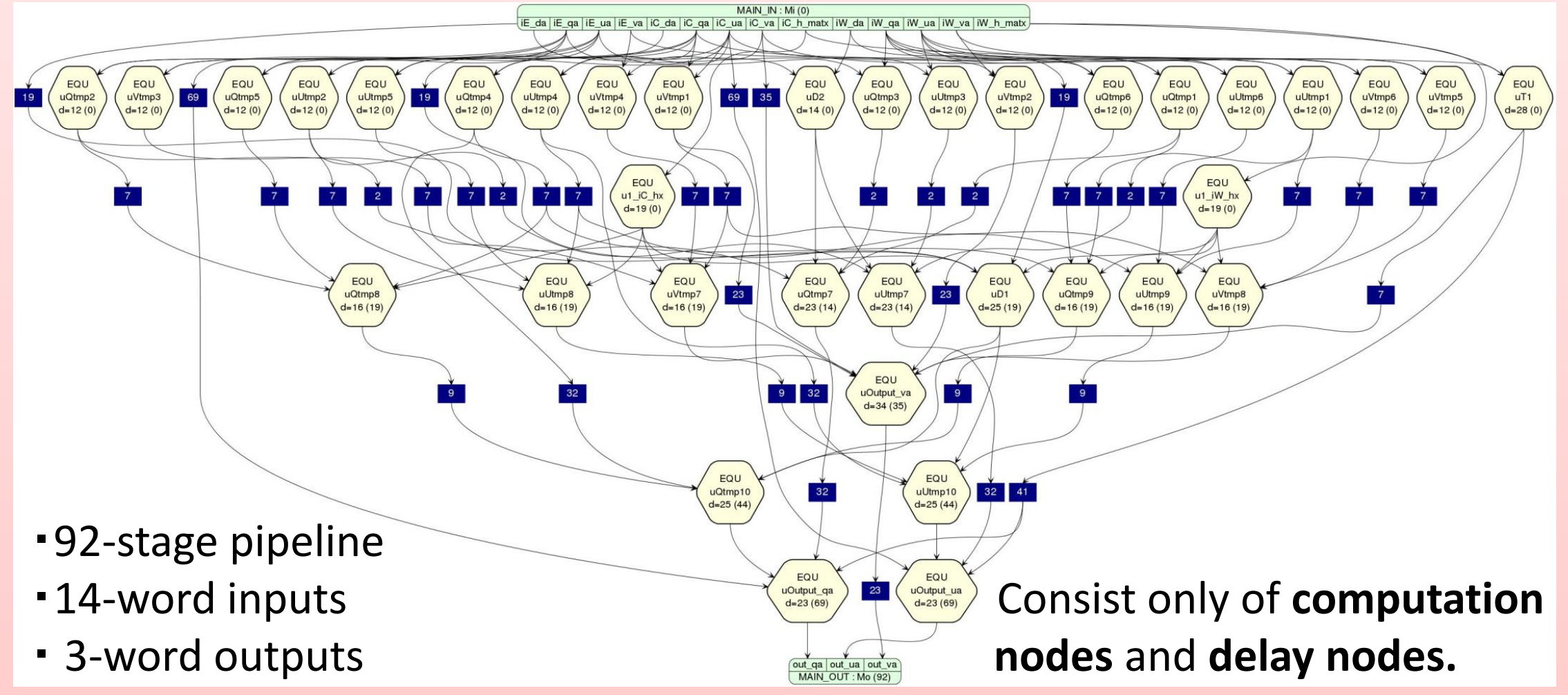
2D stencil computation ($v_{i,j}$ is updated.) *Pipelined stencil computation with cyclic buffers (3x3 stencil)* *2D stencil computation with spatial parallelism*

A stream-computing architecture generalized for **spatially** and **temporally** parallel stencil computation



Array of **stream processing elements** (**SPEs**)

The data-flow graph of the stencil computation submodule (for x-direction)



- 92-stage pipeline
- 14-word inputs
- 3-word outputs

Consist only of **computation nodes** and **delay nodes.**

Implementation of the **cascaded SPEs with DE5-NET board**

- ALTERA Stratix V 5SGXA7 FPGA
- Two DDR3 PC12800 SDRAMs, PCI-Express (PCIe) 3.0 interface.
- A peak bandwidth is 25.6 GB/s
- The platform has the three clock domains (250 MHz for PCIE interface, 200 MHz for DDR3 controllers, and 150 MHz for MOST accelerator).

Computing time at 150MHz for 300 steps with the 2581x2879 grid :

- 1 SPE on FPGA takes **14.9 seconds** at **28.1 W**.
(measured power of the entire FPGA board, single precision)
- 2 SPEs are expected to take 7.4 seconds.
(2 SPEs require 105% resources of FPGA, and need more optimization to fit them to FPGA.)

FPGAs can achieve high performance per power. We will evaluate it with the next generation FPGAs, Arria10, with hard floating-point cores.