# Enhancing spatial parallelism on loop structure for FPGA

Yuka Sano[1], Taisuke Boku[2,1], Mitsuhisa Sato[3,4] , Miwako Tsuji[4], Norihisa Fujita[2,1] and Ryohei Kobayashi[2,1]

1: Degree Programs in Systems and Information Engineering, University of Tsukuba, Japan
2: Center for Computational Sciences, University of Tsukuba, Japan
3: Faculty of Health Data Science, Juntendo University, Japan
4: Center for Computational Science, RIKEN, Japan

## ❖ Introduction

### Background

- FPGA is focused on for an accelerating device for HPC systems.
- HLS (High Level Synthesis) programming framework to apply high level programming language such as OpenCL for FPGA implementation is attractive, however, more user-friendly languages for accelerating devices such as OpenACC are not popular yet.
- CCS (Center for Computational Sciences, University of Tsukuba) and R-CCS (Center for Computational Science, RIKEN) have been collaborating to develop the Omni-OpenACC compiler for GPU as a source-to-source compiler from OpenACC to OpenCL
- We propose the methods for optimization for FPGA in compiler level to exploit spatial parallelism toward high performance execution and implement the optimized OpenCL code generator for OpenCL to the Omni-OpenACC Compiler.

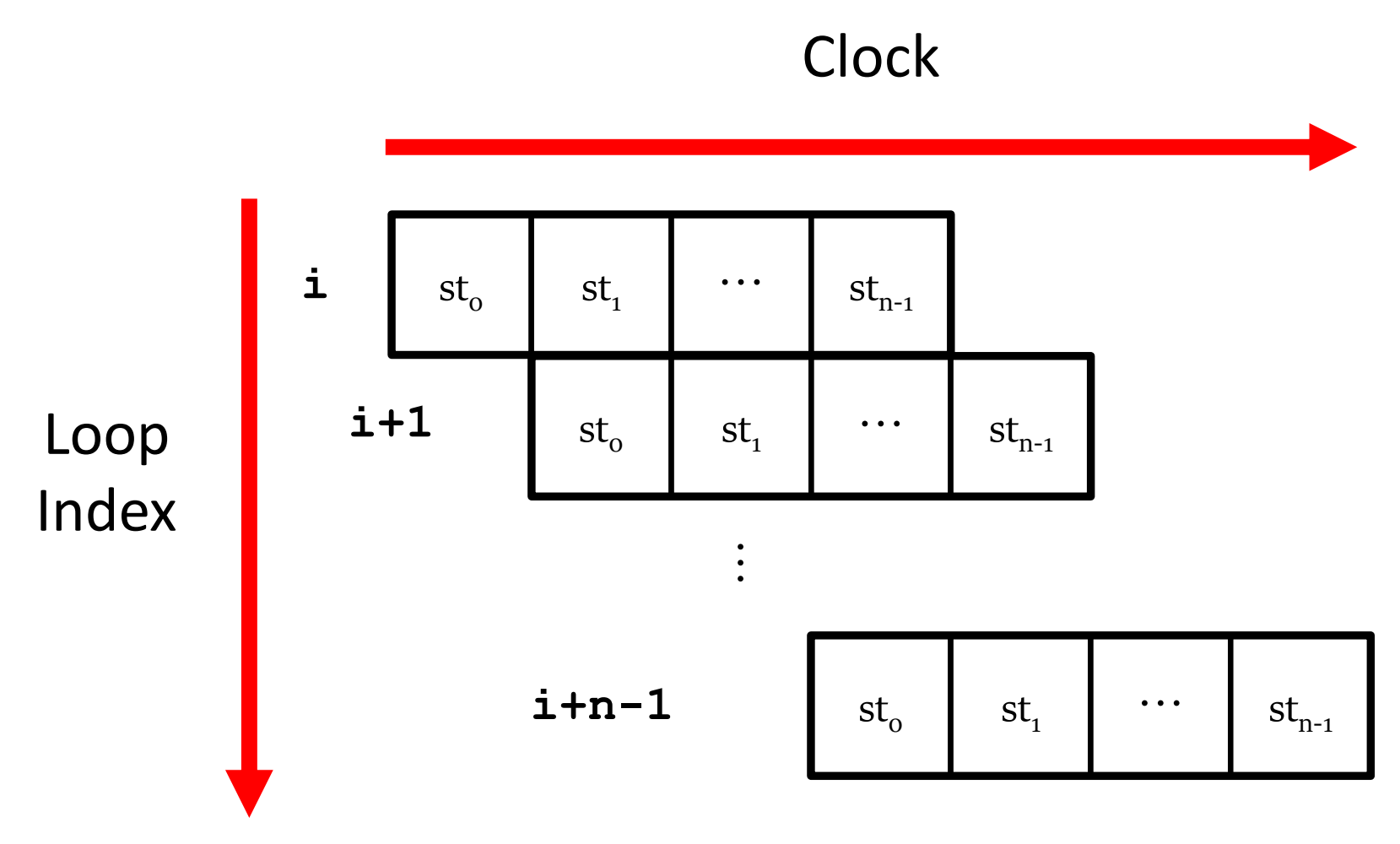
### Approach

- Enhancing current Omni-OpenACC compiler for GPU to generate the optimized OpenCL code especially for FPGA execution, we investigate the methods to exploit the spatial parallelism on the target code toward the maximum utilization of logic elements.
- Since the basic parallelism of FPGA is provided by pipeline execution, we apply several methods to increase the spatial parallelism in addition.
- We implement the below desclibed OpenACC directives and clauses.
  - ✓ Supports the spatial parallelism
  - ✓ Declare the memory allocation on BRAM
  - ✓ Communication among kernels

## ❖ Enhancing spatial parallelism on FPGA

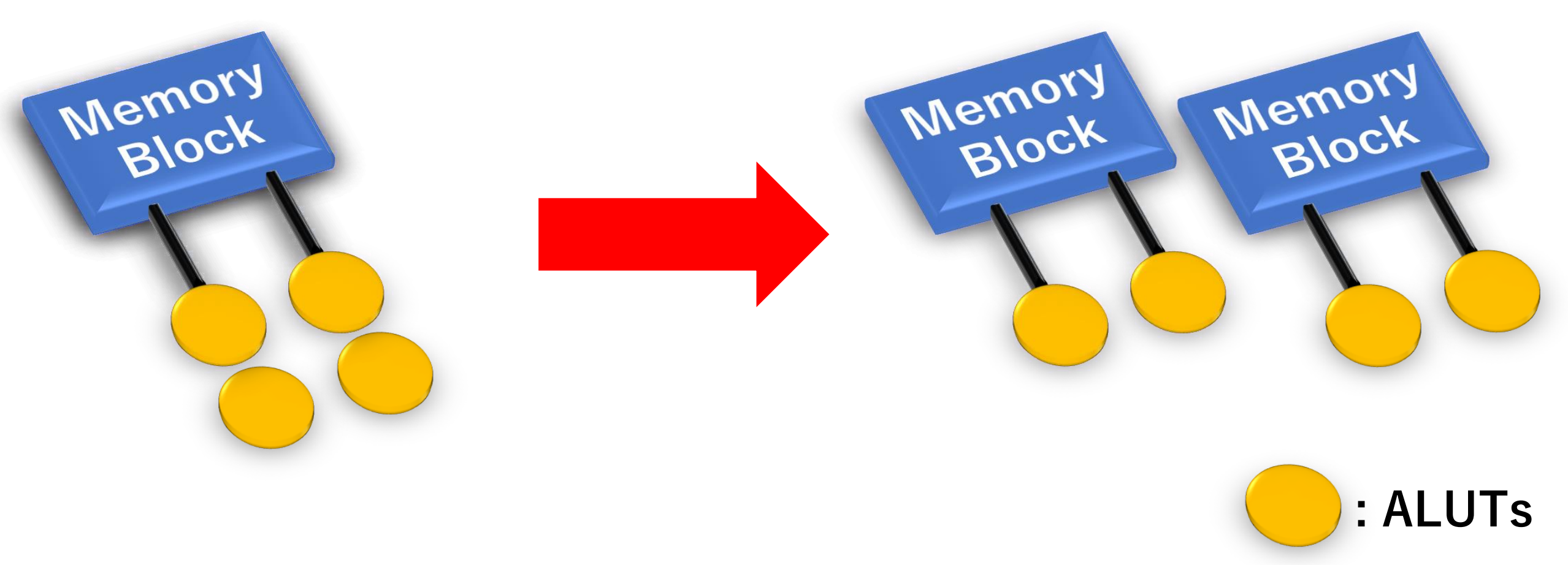
### Pipelining

- Multiple operations are executed in a single clock, so that the temporal parallelism is increased.
- “#pragma ivdep” informs the Intel compiler there is no loop dependency.
- The most important thing is making loops “Initiation Interval (II)” = 1
- In the implementation, kernels without functions for ND-Range kernels can be pipelined.



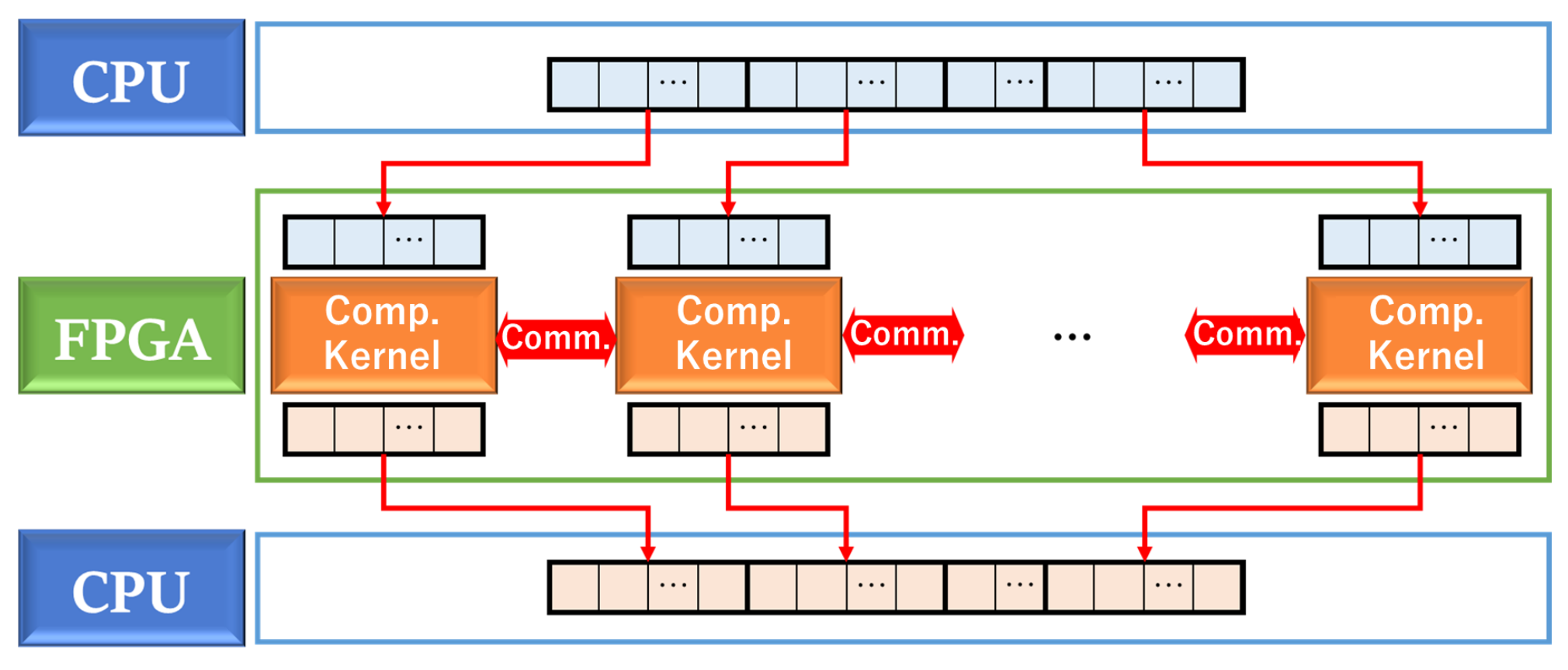
### Loop unrolling

- ”N” times of loop iteration execution are performed in a single clock cycle.
- “#pragma unroll N” specifies the loop unrolling.
- Loops with a loop-carry dependency such as reduction can keep II by using the temporal intermediate variables.
- Loop unrolling can cause a memory replication on BRAM, so BRAM capacity becomes the limiting factor of loop unrolling.
  - ✓ Our target FPGA (Intel Stratix10 H-tile) has totally 229 Mbits of BRAM capacity (single block‘s capacity is 20Kbit)
  - ✓ Much smaller than DRAM (DDR4-2400) with 64GB
- In the implementation, “#pragma acc loop unroll(N)” informs the Omni-OpenACC Compiler to generate an unrolled loop.



### Multiple kernels

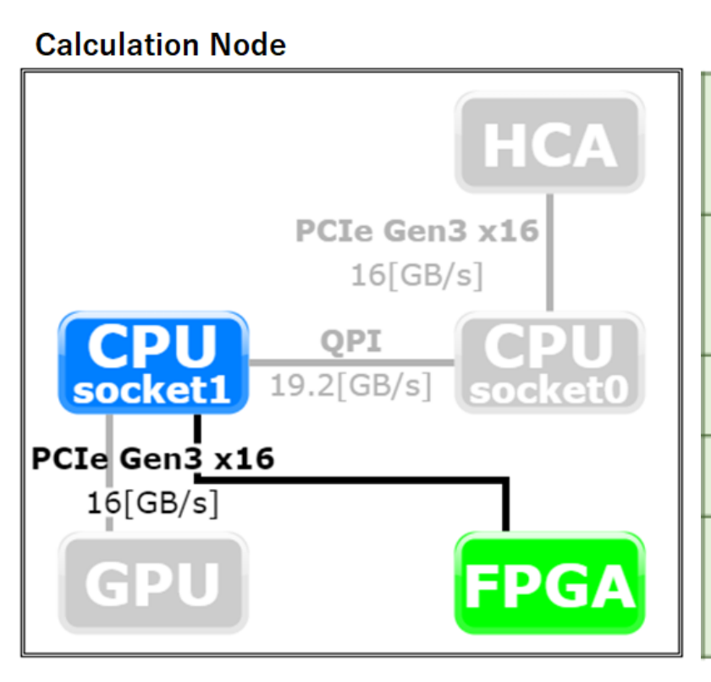
- Similar to the domain decomposition on distributed memory systems.
- To increase the spatial parallelism limiting BRAM capacity increase (by loop unrolling), distribute the workload into multiple kernels.
- The communication between them is required to synchronize the result data by partial computation spread to multiple kernels (as like as MPI programs).
- In the implementation, “#pragma acc parallel num_kernels(K)” informs the Omni-OpenACC Compiler to generate multiple kernels, and “#pragma acc loop mulker_length(SIZE)” informs it to divide a loop among kernels.



## ❖ Performance evaluation with Omni-OpenACC

### Evaluation testbed

PPX (Pre-PACS-X) system in CCS, University of Tsukuba

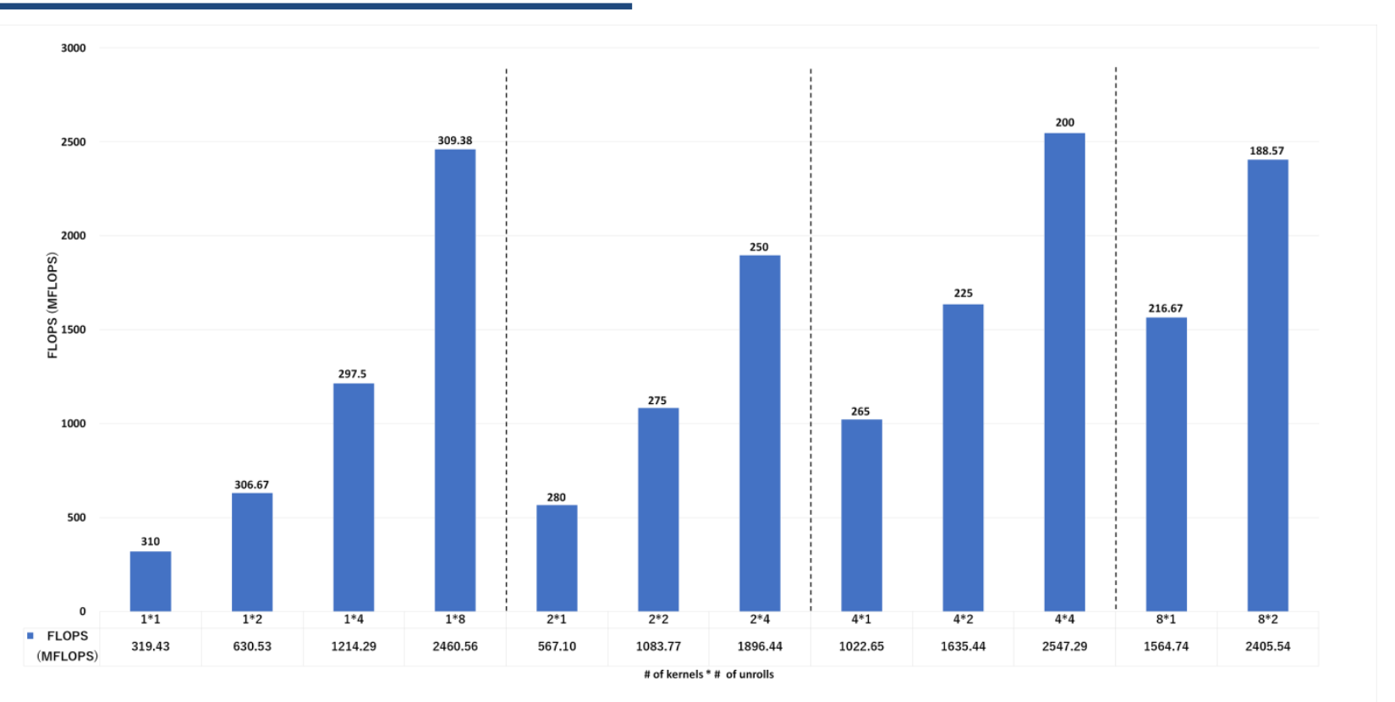


| | |
|---|---|
| CPU | Intel Xeon E5-2690 v4 @ 2.60GHz 14cores × 2sockets |
| FPGA | BittWare 520N (Intel Stratix 10 GX2800) × 1 |
| Host OS | CentOS 7.9 |
| C Compiler | gcc 4.8.5 |
| FPGA Toolkit | Intel FPGA SDK for OpenCL 19.4.0 Build 64 Pro Edition |

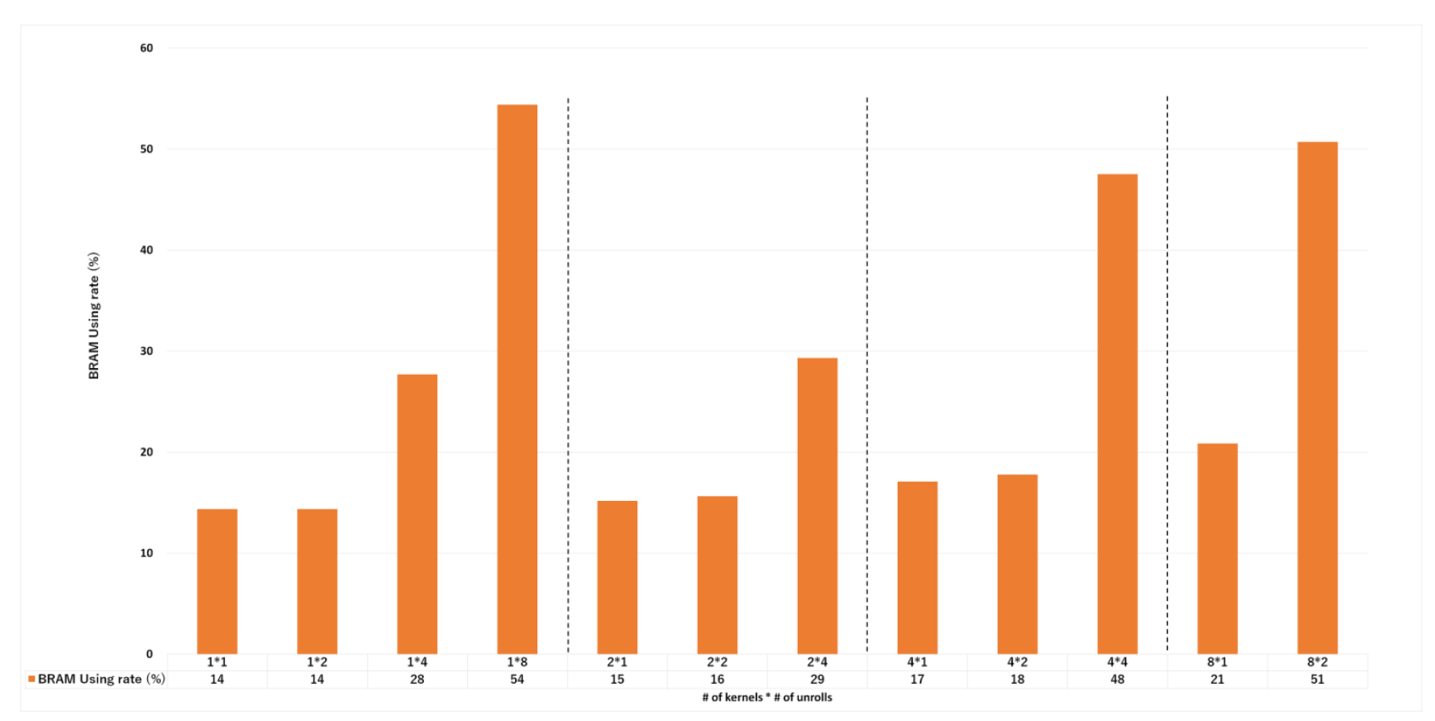
Benchmark: matrix solver by CG method

- ✓ matrix size: 3000x3000
- ✓ sparsity (non-zero elements): 4 %
- ✓ a constant count of loop iteration (without convergence check)
- ✓ on multi-kernel implementation, scatter/gather communication is performed between kernels over Intel ”channel”
- ✓ sources written in C + OpenACC
- ✓ all arrays on the BRAM while the calculation

### Evaluation result



- The highest FLOPS number is achieved with 4 kernels and 4 times unrolling.
- Keeping the same degree of spatial parallelism, multiple kernels cannot enhance the performance.



- Comparing with the same degree of spatial parallelism (degree ≤ 8), BRAM utilization is lower on multiple kernel solutions than unrolling method.
- When there is no remaining loop generated by Loop Unrolling, unrolls = 2 can increase spatial parallelism without another BRAMs.

### Discussion

- Loop unrolling is efficient for FLOPS, but it increases the BRAM capacity proportional to the unrolling depth to limit it, especially there are remaining loops generated by Loop Unrolling.
- Multiple kernel method lowers the execution clock frequency even though it saves the BRAM capacity, so it is necessary to choose appropriate kernels and unrolls for whole the application.

## ❖ Conclusion and future work

- CCS and R-CCS at RIKEN collaborate for development of Omni-OpenACC Compiler optimized for FPGA, and we implement the OpenCL code generator in the test environment.
- It is required to investigate more complicated applications than a simple CG method for wide variety of HPC benchmarks and applications to generate more optimized OpenCL code and provide the lack of directives and clauses.
- We develop a unified compiler for OpenACC compilation both for GPU and FPGA simultaneously.

ACKNOWLEDGEMENT This work was supported in part by the ``Next Generation High-Performance Computing Infrastructures and Applications R&D Program'' (Development of Computing Communication Unified Supercomputer in Next Generation) of MEXT. This research was also supported in part by the Multidisciplinary Cooperative Research Program in CCS, University of Tsukuba, and JSPS KAKENHI, Grant Number 21H04869. We also thank Dr. Naruhiko Tan of NVIDIA for his advice on OpenACC optimization を We also thank the Intel University Program for providing the hardware and software.