Evaluating Reconfigurable Dataflow Computing Using the Himeno Benchmark

Yukinori Sato (JAIST, JST CREST)
Yasushi Inoguchi (JAIST)
Wayne Luk (Imperial College London)
Tadao Nakamura (Keio University)

2012 International Conference on ReConFigurable Computing and FPGAs
5th December, 2012

Introduction

Acceleration using FPGA

- An approach to build custom system appropriate for each application
- Realize custom system within a certain power, or cost budget
- Customization and optimization must be performed

Customization and optimization strategies must be made based on the characteristics of platform and the type of applications

Characteristics of platforms	**Hardware** *Reconfiguration capability, Customizability, Device itself*	**Software** *System software, HW compiler, API, run-time, library*
Types of applications	**Computation-bound apps** The clock rate or parallelism should be increased	**Memory-bound apps** Optimizations for memory is the most significant factor

In this paper

We focus on FPGA accelerator provided by Maxeler, and evaluate its performance for a memory-intensive application

- We present several optimization techniques
  - Targeting for Maxeler's reconfigurable dataflow engines
  - In terms of memory locality and memory bandwidth

  We demonstrate all key optimizations can be done by high-level programming for the Maxeler platform

- For evaluation, we use the Himeno Benchmark
  - Widely known as a memory-bound FP program
- We compare it with the implementation on GPUs

The Himeno Benchmark

- Developed by Dr. Ryutaro Himeno, RIKEN, Japan
- A highly memory intensive application kernel
  - Becoming popular in the HPC community to evaluate the worst-case performance for memory bandwidth intensive codes
  - The kernel is a linear solver for 3D pressure Poisson equation which appears in a Navier-Stokes solver:

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} + \alpha \frac{\partial^2 p}{\partial xy} + \beta \frac{\partial^2 p}{\partial xz} + \gamma \frac{\partial^2 p}{\partial yz} = \rho$$

  - This Poisson equation is solved using the Jacobi iterative method
  - The performance of this is measured in FLOPS (FLoating-point Operations Per Second)

Source code of the Himeno benchmark

```
1   for(n=0 ; n<nn ; ++n){
2
3     for(i=1 ; i<imax-1 ; i++)
4       for(j=1 ; j<jmax-1 ; j++)
5         for(k=1 ; k<kmax-1 ; k++){
6           s0 = a[0][i][j][k] * p[i+1][j  ][k  ]
7             + a[1][i][j][k] * p[i  ][j+1][k  ]
8             + a[2][i][j][k] * p[i  ][j  ][k+1]
9             + b[0][i][j][k] * ( p[i+1][j+1][k  ] - p[i+1][j-1][k ]
10                              - p[i-1][j+1][k ] + p[i-1][j-1][k ] )
11            + b[1][i][j][k] * ( p[i  ][j+1][k+1] - p[i ][j-1][k+1]
12                              - p[i ][j+1][k-1] + p[i ][j-1][k-1] )
13            + b[2][i][j][k] * ( p[i+1][j  ][k+1] - p[i-1][j ][k+1]
14                              - p[i+1][j  ][k-1] + p[i-1][j ][k-1] )
15            + c[0][i][j][k] * p[i-1][j ][k  ]
16            + c[1][i][j][k] * p[i  ][j-1][k ]
17            + c[2][i][j][k] * p[i  ][j  ][k-1]
18            + wrk1[i][j][k];
19
20          ss = ( s0 * a[3][i][j][k] - p[i][j][k] ) * bnd[i][j][k];
21          wrk2[i][j][k] = p[i][j][k] + omega * ss;
22        }
23
24    for(i=1 ; i<imax-1 ; ++i)
25      for(j=1 ; j<jmax-1 ; ++j)
26        for(k=1 ; k<kmax-1 ; ++k)
27          p[i][j][k] = wrk2[i][j][k];
28
29  } /* end n loop */
```

(Only the primary loop region)

19-point stencil computation to the 3D array *p*

p is pressure,
a, b, c are coefficient matrices,
wrk1 is a source term of Poisson equation,
omega is a relaxation parameter,
bnd is a control variable for boundaries and objects,
wrk2 is a temporary working area for computation
Also, data are represented in single precision floating point format.

Loop nest structure

Outermost loop by *n*

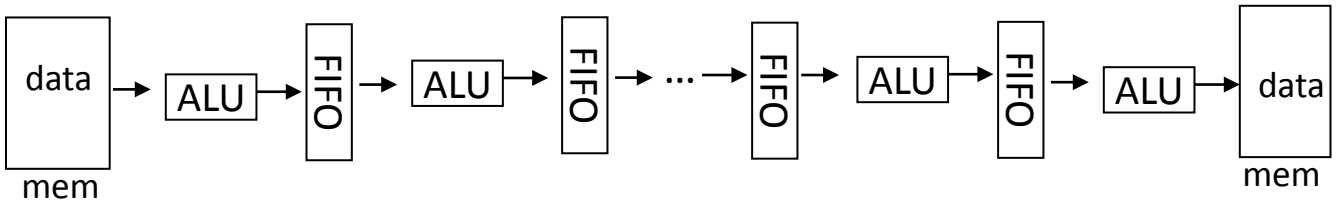
Triply nested loop by *i,j,k*

Triply nested loop by *i,j,k*

The body of this computational kernel involves 34 FP ops
So, FLOPS can be measured by dividing it by the total execution time

Reconfigurable dataflow engine

- Dataflow graph is directly mapped into FPGA
- Computations are performed by forwarding intermediate results directly from one functional unit to the next units
- Memory transfer operations can be dramatically reduced compared with instruction-based processor



An overview of dataflow processing:
Its datapath becomes a deeply pipelined structure

Using inherent locality of dataflow computation, we attempt to avoid memory bottleneck

An outline of Maxeler's dataflow engine

A high-level HW synthesis platform

- Kernel code is generated by MaxCompiler
  - Use Java as a meta-programing language
    - The structure of dataflow graph is described
    - Java extensions are used for hardware descriptions
  - MaxCompiler generates FPGA bitstream
  - Data exchange between a host CPU and a dataflow engine is performed using a run-time library API (called MaxCompilerRT)
- The bitstream file (.max) is loaded from the CPU to the engine, and the FPGA is configured to the design

Using a high-level description language for Maxeler platform, we optimize design focusing on memory locality and bandwidth

O. Pell and O. Mencer, "Surviving the end of frequency scaling with reconfigurable dataflow computing," ACM *SIGARCH Comput. Archit. News*, vol. 39, no. 4, pp. 60–65, Dec. 2011.

Optimization strategies for dataflow engine

We present the following 4 optimization strategies

Optimization (A):
Directly forwarding data

Making use of fine-grained parallelism in dataflow graph, temporal parallelism by pipelining dataflow paths

Optimization (B):
Organizing highly regular data stream

The regularity of the flow of data is strongly emphasized so that deeply pipeline processing can be performed without fatal pipeline stalls

Optimization (C):
Utilizing available HW resources

By unrolling independent loops, we build multiple parallel pipes in a kernel

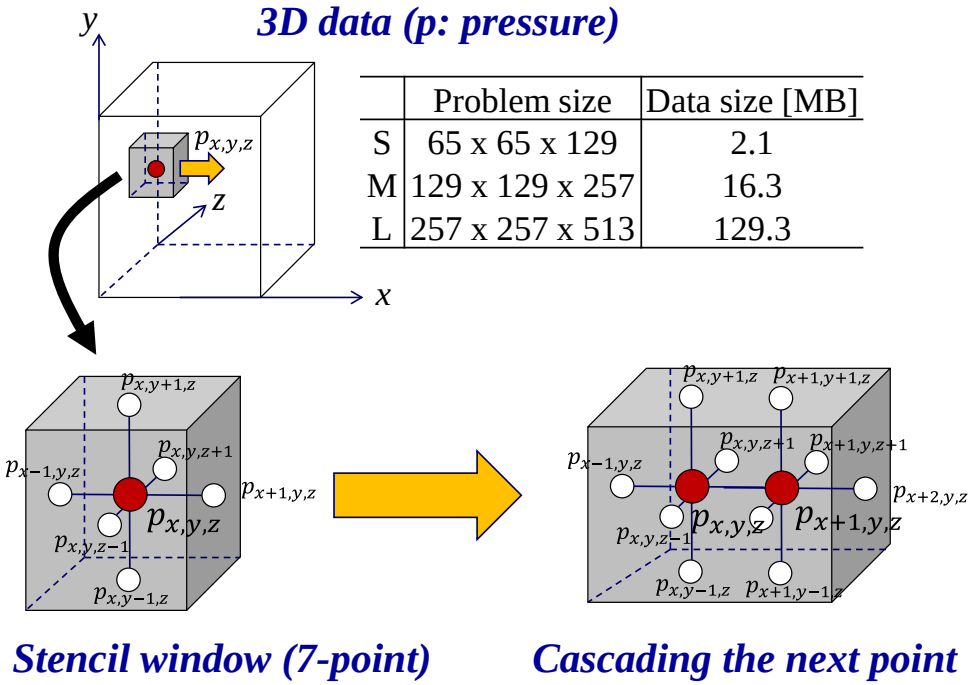
Optimization (D):
Communication btw CPU and FPGA

When partitioning the code regions for an FPGA accelerator, the amount of communication must be investigated carefully

Based on these 4 strategies together with domain knowledge and experience for apps, we optimize the kernel

Optimizing the Himeno benchmark (1)

- Perform Optimization (A) to improve locality of memory access
  - We cascade multiple sequential points of the stencil computation
  - This enables to keep more intermediate results within pipelines and mitigates heavy memory traffic of a stencil program



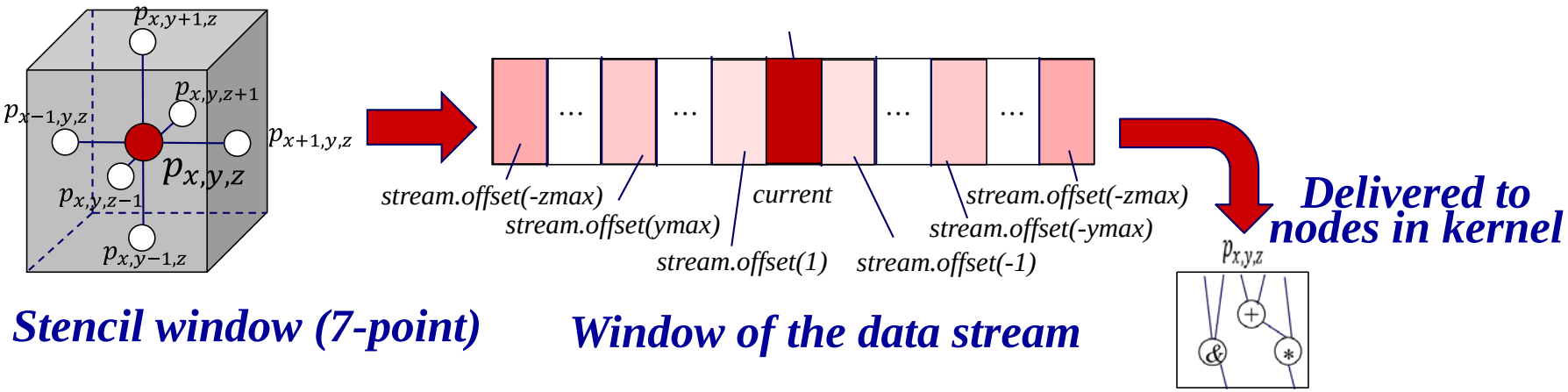
	Problem size	Data size [MB]
S	65 x 65 x 129	2.1
M	129 x 129 x 257	16.3
L	257 x 257 x 513	129.3

An overview of cascading:

Since sequential points of a stencil computation often access the same data, we can reuse points appeared previously

Optimizing the Himeno benchmark (2)

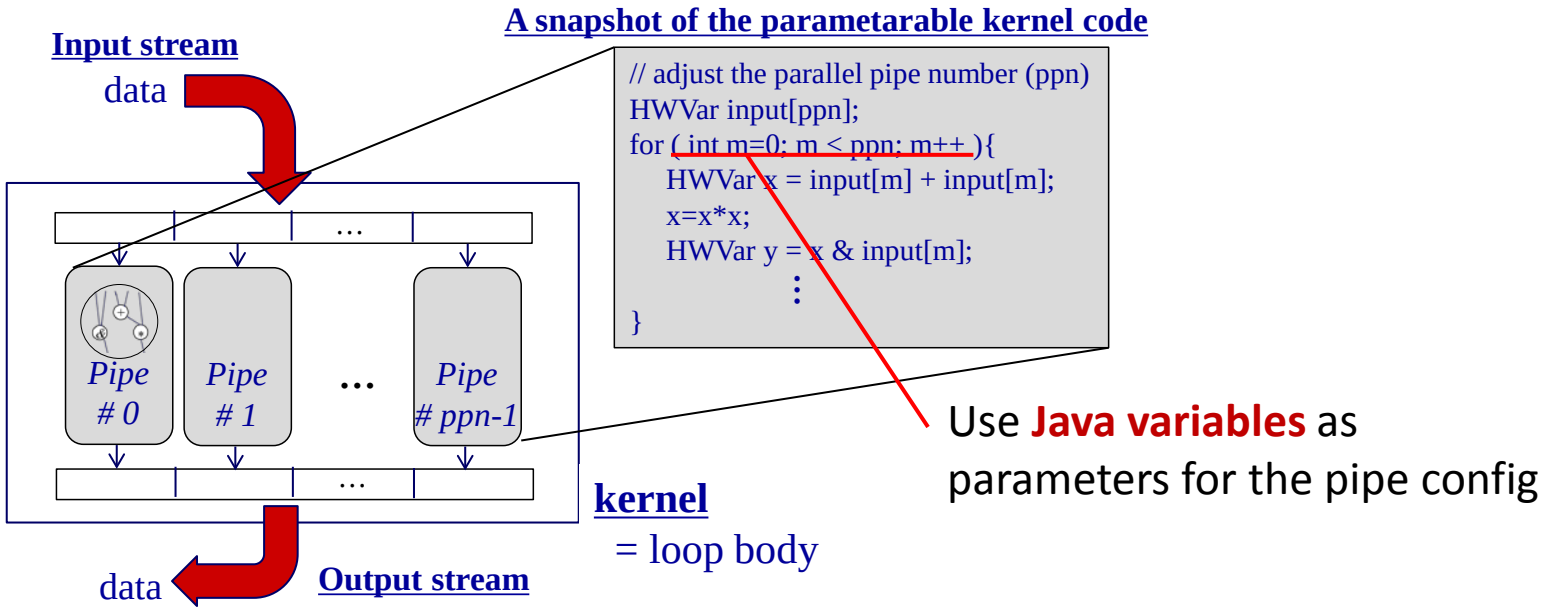
- Perform Optimization (B) by converting data stream into 1-D
  - To create a window for stencil computation, we need to access the adjacent values in a 3D array structure
  - We use stream.offset (Maxeler API) to feed data into a kernel
  - MaxCompiler transforms the stream.offset into an FIFO buffer on FPGA



Converting 3-D data accesses into 1-D stream:
By specifying a relative distance from the current stream data, we can obtain the past or the future data.

Optimizing the Himeno benchmark (3)

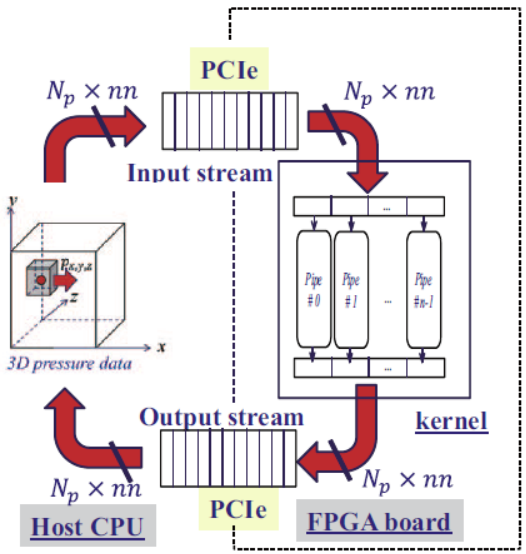
- Perform Optimization (C) by configuring the # of pipes
  - We adjust **the # of pipes** to maximize parallelism and resource utilization of an FPGA chip
  - Here, we refer the body of a custom dataflow pipeline as a **pipe**
  - By unrolling loop iterations based on degree of cascading of stencil computations, we realize the parameterized kernel



Parameterized kernel design:
We can transparently translate the parameterized loop to unrolled multiple parallel pipes

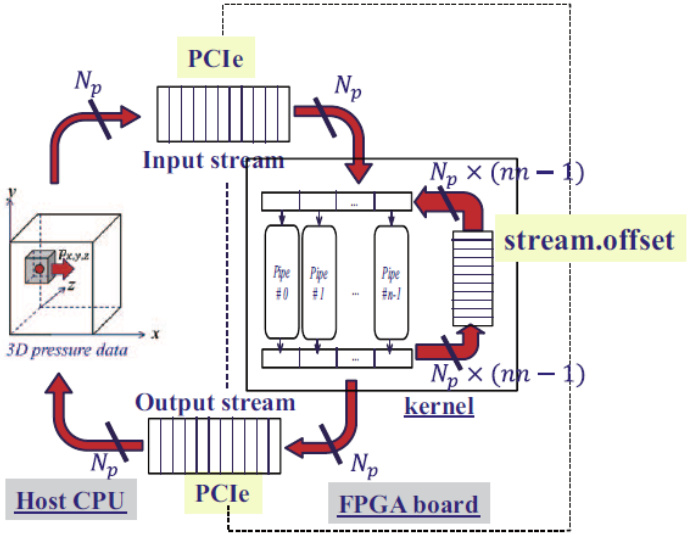
Optimizing the Himeno benchmark (4)

- Perform Optimization (D) by building 3 scenarios for the stream communications



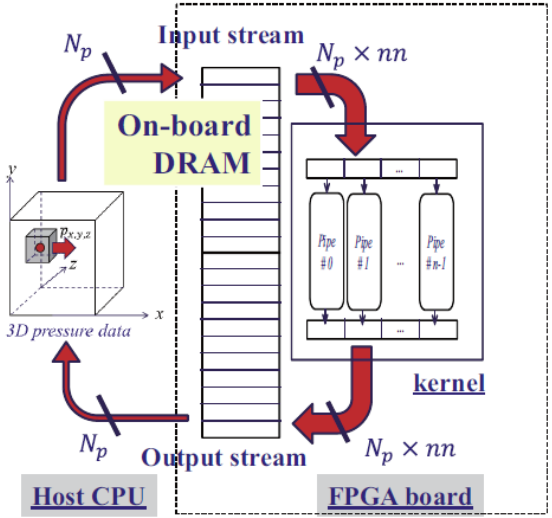
PCIe–** design

- Data stream via PCIe bus
- Simple
- But data transfer via PCIe bus will be bottleneck



nn–itr–** design

- Data stream via internal buffer
- Consume BRAM equal to the size of array '*p*'



DRAM–** design

- Data stream via on-board DRAM
- Require memory adr gen

Performance evaluation

Methodology

Maxeler MaxWorkstation

- Intel Core i7-870 2.93GHz CPU, CentOS
- The MAX3 acceleration card
  - Virtex-6 SX475T FPGA, 24GB DDR3 memory
  - PCI express gen2 x8 interface
- MaxCompiler version 2012.1
- Generate host code using gcc.4.1.2 with '-O3'

Experimental conditions

- Set FPGA operating frequency as 100MHz (system default)
- Constant array values are generated using ternary-if logic

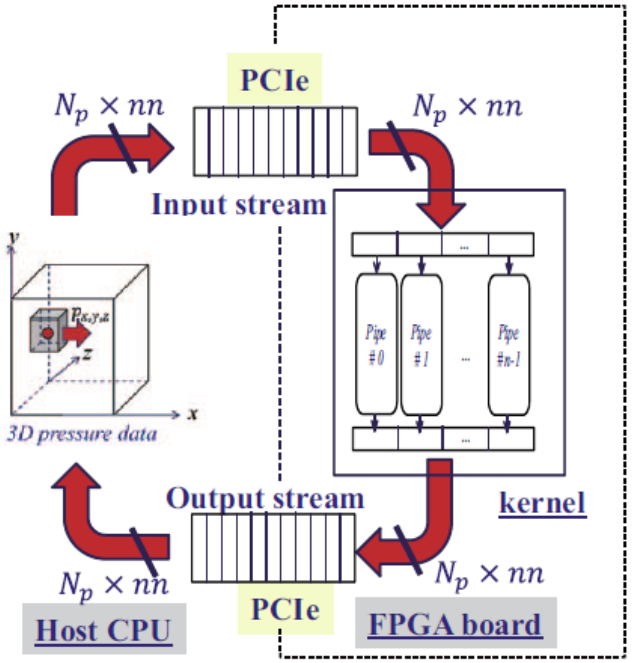
Evaluation results

Verification of our designs

- Compare the output values of *p* from the FPGA with those of the original CPU implementation across all of design
- As a result, we confirmed that all of these two are completely the same ⟶ [accurate implementation]
- Also, we could implement our design smoothly

The productivity of Maxeler platform with support for IEEE floating point format enables *accurate* and *smooth* implementations

Stream communication via PCIe



Performance result

# of pipes	score [GFLOPS]
1	2.70
2	4.96
4	8.29
8	8.33

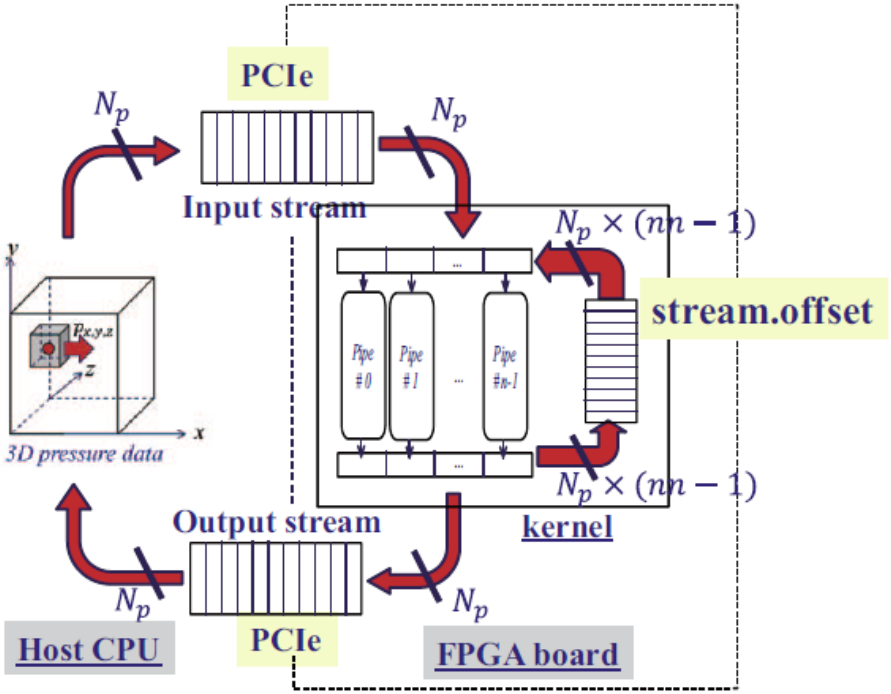
PCIe- desing**

- Communication via PCIe bus
- The data stream is driven by the triply-nested i,j,k loop
- Simple implementation
- But data transfer via PCIe bus will be bottleneck

- From the result, we find that the perf. is not increased even if we scale the # of pipes
- The data transfer via the PCIe bus becomes the bottleneck

Data stream via an internal buffer

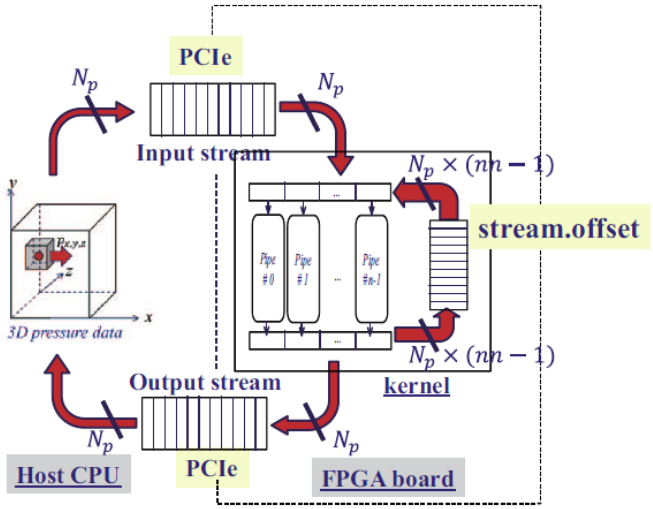
In order to avoid the data transfer overhead,
we attempt to feed output data directly to the input



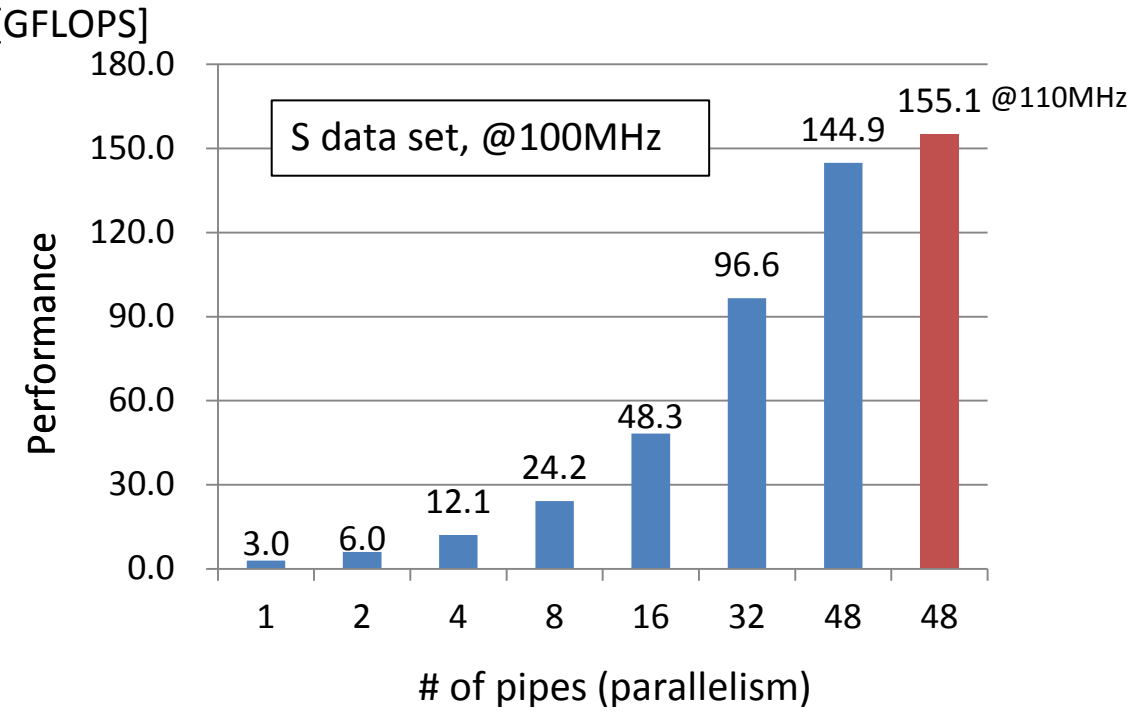
- The data stream is driven by the outermost loop
- For keeping the values from the previous loop iteration, we use **stream.offset** for inserting a special buffer
- Once the host CPU transfers the initial data stream, data stream from the previous iteration is fed into the pipes directly

nn-itr- design**

The results obtained by data stream via an internal buffer



nn-itr-** design



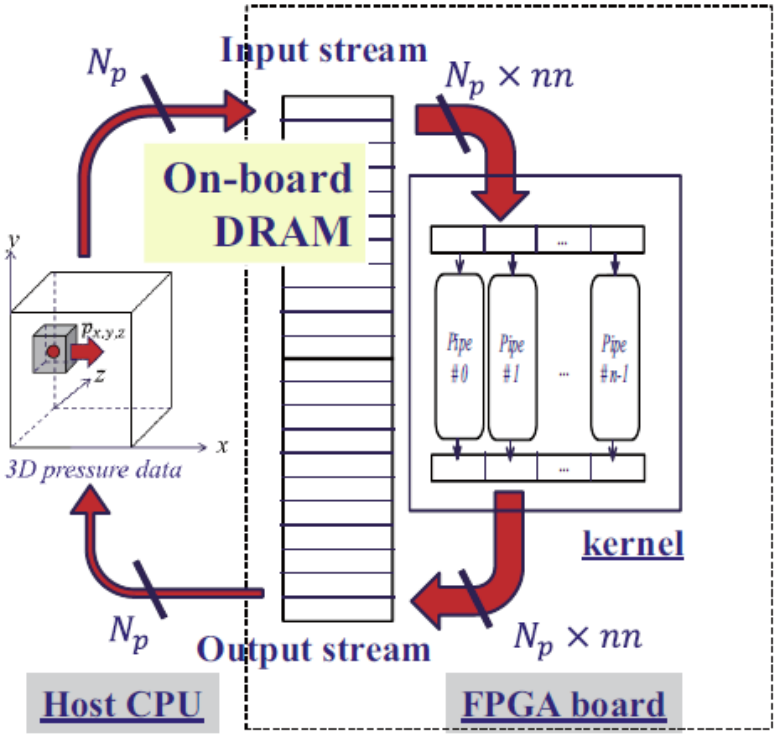
Evaluation Results

- This design can linearly increase its performance when we scale the # of pipes
- The data transfer bandwidth bottleneck can be avoided
- We successfully build and run the design with 48 pipes
- The 48-pipe design with 110MHz achieves 155.1 GFLOPS

Due to the size of BRAM, configurations with larger data set cannot be implemented

Data stream via on-board DRAM

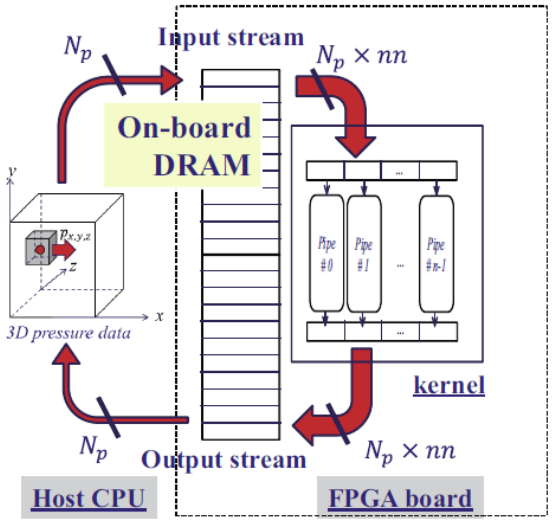
In order to apply larger data sets,
we attempt to use on-board DRAM devices



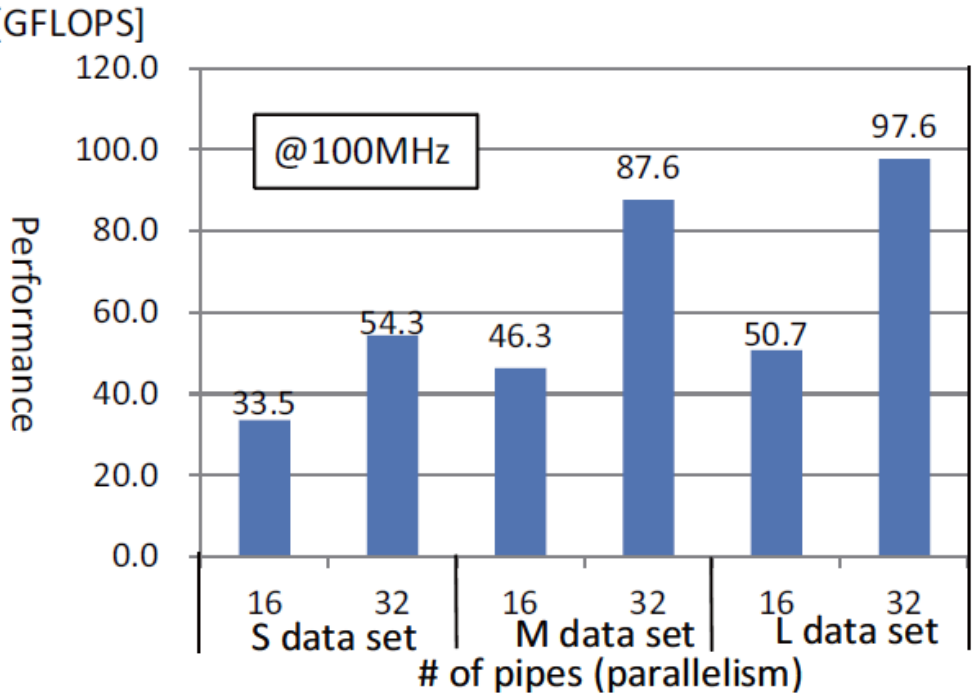
DRAM-** design

- The data stream is driven by the triply-nested i,j,k loop similar to the 'PCIe-**' design
- This design can make use of higher on-board DRAM memory bandwidth compared with the PCIe design
- Iterations are processed without any data transfer to the CPU memory once the initial input data are loaded

The results obtained by data stream via on-board DRAM



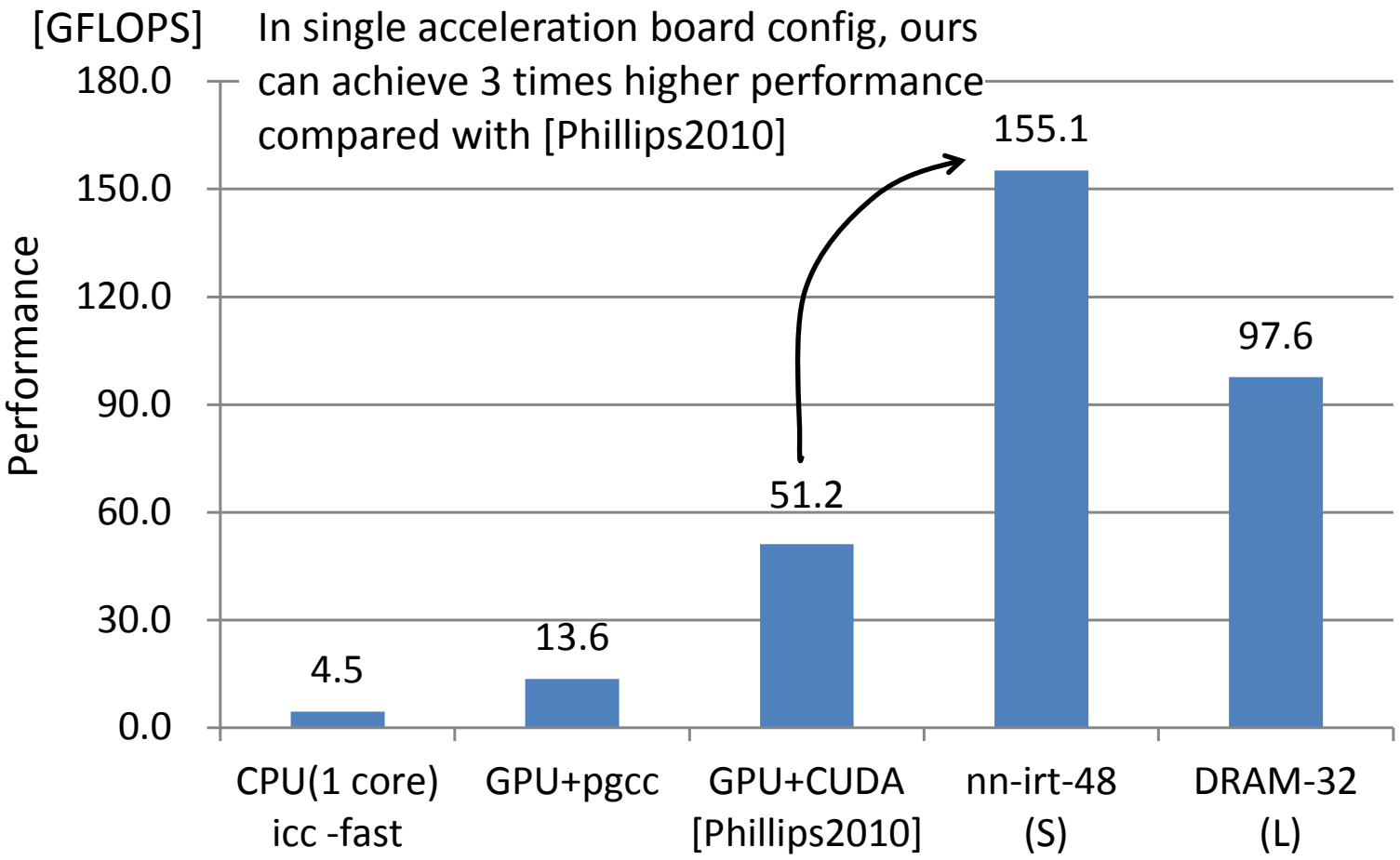
DRAM-** design



Evaluation Results

- This design can accept larger data sets (up to L: 129.3MB) and increase its performance according to the # of pipes
- We cannot build the 48 pipe designs due to HW overhead for memory address generators
- We can see start-up overhead of the kernel pipelinein smaller data set
- 32-pipe design with L data set can achieve 97.6 GFLOPS

Comparison with the state-of-the-art GPU



[Phillips 2010] E. Phillips and M. Fatica, "Implementing the himeno benchmark with CUDA on GPU clusters," in *IEEE Int'l Symp. on Parallel Distributed Processing*, 2010, pp. 1 –10.

Conclusions

We describe a memory bound application program on an FPGA accelerator based on a reconfigurable dataflow computing platform

- We have presented several optimization techniques for designs targeting reconfigurable dataflow engines
  - especially focus on memory locality and memory bandwidth
  - Also present how such techniques can be used in optimizing the Himeno Benchmark that requires large memory bandwidth
- From the results of evaluation
  - We have confirmed that specialized dataflow pipeline designs contribute to improving the actual performance of applications
  - Found that our implementation can outperform the recent GPU implementations in their achieved GFLOPS
  - Demonstrated that we can achieve competitive performance gain without low-level HDL coding

Future work & Acknowledgment

Current and future work

- Extending the set of optimization strategies and benchmarks for our approach
- Exploring methods for automating the application of such optimization strategies
- Studying how our approach can be improved to support power and energy optimization

Acknowkedgment

This work was supported in part by the JSPS ”Institutional Program for Young Researcher Overseas Visits”, by UK EPSRC, by the European Union Seventh Framework Programme under Grant agreement number 248976, 257906 and 287804, by the HiPEAC NoE, by the Maxeler University Program, and by Xilinx.