Evaluating Reconfigurable Dataflow Computing Using the Himeno Benchmark

Yukinori Sato*†, Yasushi Inoguchi*, Wayne Luk‡ and Tadao Nakamura§

*Research Center for Advanced Computing Infrastructure, JAIST, Japan
†JST CREST, Japan
‡Dept. of Computing, Imperial College London, United Kingdom.
§Dept. of Information and Computer Science, Keio University, Japan

***Abstract*—Heterogeneous computing using FPGA accelerators is a promising approach to boost the performance of application programs within given power consumption. This paper focuses on optimizations targeting FPGA-based reconfigurable dataflow computing platform, and shows how they benefit an application. In order to evaluate them, we use the Himeno benchmark, which is a floating point computation kernel known to be bound by memory bandwidth. To understand the performance characteristics of the benchmark, we compare it with the current state-of-the-art implementation on GPUs. From the results, we find that our implementation with specialized dataflow pipelines outperforms the current state-of-the-art GPU implementations by making full use of memory locality.**

I. Introduction

Accelerator-based heterogeneous computing is widely spreading to offer significant boosts in performance and energy efficiency for applications. Examples of accelerators range from already-fabricated chips such as GPUs or IBM's Cell B.E. to reconfigurable logic fabrics such as FPGAs [1] [2]. Especially for designing highly efficient systems, designs with FPGA accelerators are a promising approach to build custom system appropriate for each application within a certain power, or cost budget.

While FPGA accelerators have potential to maximize performance within these limiting factors, the characteristics of each platform for particular application domains are required to be understood carefully. Each platform is characterized by the customizability and the reconfiguration capability of hardware architectures, system software and device themselves.

For instance, floating point data formats are traditionally considered not well-suited to implementing on FPGAs compared with already-fabricated chips due to their slower clock rates [3]. On the other hand, FPGA-based accelerators can customize memory access patterns for particular application domains. If the time consuming kernel is not computation-bound but memory-bound, memory optimization will be more significant factor than the clock rate or parallelism. By making full use of memory optimization, FPGA-based accelerators have potential to outperform GPU implementations [4]. Therefore, we should investigate the actual performance gain influenced by the characteristics of platform and application programs.

In this paper, we focus on reconfigurable dataflow engines on FPGAs provided by the Maxeler platform [5], [6]. In order to understand their characteristics for a particular application, we evaluate the performance and compare them with the current state-of-the-art implementation on GPUs. For evaluation, we use the Himeno Benchmark, which is composed of a floating point computation kernel by Jacobi method and widely known as a program that requires large memory bandwidth. Throughout the evaluation, we investigate the advantages of specialized hardware by reconfigurable dataflow engines in terms of memory locality and memory bandwidth. In addition, we demonstrate that all key optimizations can be done using a high-level language programming for the Maxeler platform.

The rest of this paper is organized as follows. Section 2 briefly explains the Himeno Benchmark. Section 3 provides an overview of the Maxeler platform. Then we present several optimization techniques, such as memory optimization, for designs targeting reconfigurable dataflow engines; these would be our first contribution. Section 4 shows how such techniques can be used in optimizing the Himeno Benchmark; these would be our second contribution. Section 5 covers an experimental framework, evaluation methodology and results; these would be our third contribution. Section 6 describes related work, and Section 7 concludes this paper.

II. The Himeno benchmark

The Himeno benchmark has been developed by Dr. Ryutaro Himeno to evaluate performance of incompressible fluid analysis code [7]. This benchmark program measures performance in solving the Poisson equation using the Jacobi iterative method. The Himeno benchmark is known to be highly memory intensive and bound by memory bandwidth [8]. Therefore, this benchmark has grown in popularity and has been used by the HPC community to evaluate the worst-case performance for bandwidth intensive codes [9].

The kernel of the Himeno benchmark is a linear solver for 3D pressure Poisson equation which appears in an incompressible Navier-Stokes solver:

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} + \alpha\frac{\partial^2 p}{\partial xy} + \beta\frac{\partial^2 p}{\partial xz} + \gamma\frac{\partial^2 p}{\partial yz} = \rho \qquad (1)$$

978-1-4673-2921-7/12/$31.00 ©2012 IEEE

```
for(n=0 ; n<nn ; ++n){

  for(i=1 ; i<imax-1 ; i++)
    for(j=1 ; j<jmax-1 ; j++)
      for(k=1 ; k<kmax-1 ; k++){
        s0 = a[0][i][j][k] * p[i+1][j  ][k  ]
          + a[1][i][j][k] * p[i  ][j+1][k  ]
          + a[2][i][j][k] * p[i  ][j  ][k+1]
          + b[0][i][j][k] * ( p[i+1][j+1][k  ] - p[i+1][j-1][k  ]
                            - p[i-1][j+1][k  ] + p[i-1][j-1][k  ] )
          + b[1][i][j][k] * ( p[i  ][j+1][k+1] - p[i  ][j-1][k+1]
                            - p[i  ][j+1][k-1] + p[i  ][j-1][k-1] )
          + b[2][i][j][k] * ( p[i+1][j  ][k+1] - p[i-1][j  ][k+1]
                            - p[i+1][j  ][k-1] + p[i-1][j  ][k-1] )
          + c[0][i][j][k] * p[i-1][j  ][k  ]
          + c[1][i][j][k] * p[i  ][j-1][k  ]
          + c[2][i][j][k] * p[i  ][j  ][k-1]
          + wrk1[i][j][k];

        ss = ( s0 * a[3][i][j][k] - p[i][j][k] ) * bnd[i][j][k];
        wrk2[i][j][k] = p[i][j][k] + omega * ss;
      }

  for(i=1 ; i<imax-1 ; ++i)
    for(j=1 ; j<jmax-1 ; ++j)
      for(k=1 ; k<kmax-1 ; ++k)
        p[i][j][k] = wrk2[i][j][k];

} /* end n loop */
```

Fig. 1. The outline of the computation kernel of the Himeno Benchmark.

The Poisson equation is discretized using finite-difference method, and a point-Jacobi method is employed to solve the equation. Figure 1 shows the main solver which applies a 19-point stencil computation to the 3D array p. Here, p is pressure, and it becomes the output of this computation. The *a, b, c* are coefficient matrices, the *wrk1* is a source term of Poisson equation, the *omega* is a relaxation parameter, the *bnd* is a control variable for boundaries and objects, and all of these variables are read-only variables in this region. The *wrk2* is a temporary working area for computation and used for updating the pressure p. Also, all of these arrays contain data represented in single precision floating point format.

This main solver is composed of one outermost loop indexed by n for the Jacobi iterations, and two triply-nested loops indexed by i, j, k. The body of this computational kernel originally involves 34 floating point calculations. Based on this fact, the performance of this benchmark is measured in FLOPS (FLoating-point Operations Per Second), where the total number of floating point operations is divided by the execution time.

III. Optimizing reconfigurable dataflow

This section presents several techniques for optimizing designs targeting reconfigurable dataflow engines. The dataflow computing paradigm is fundamentally different from computing with conventional instruction-based processors [5]. In conventional processors, after instructions and data are fetched from memory, operations specified by the instructions are performed and the results are written back to memory sequentially. However, each execution of an instruction requires a cyclic access for memory. As a result, this model incurs a large number of data transfers between processor and memory, which often becomes the performance bottleneck. Although techniques such as cache memory improve memory latency and bandwidth requirements to some extent, these are not a fundamental solution for this memory bottleneck.

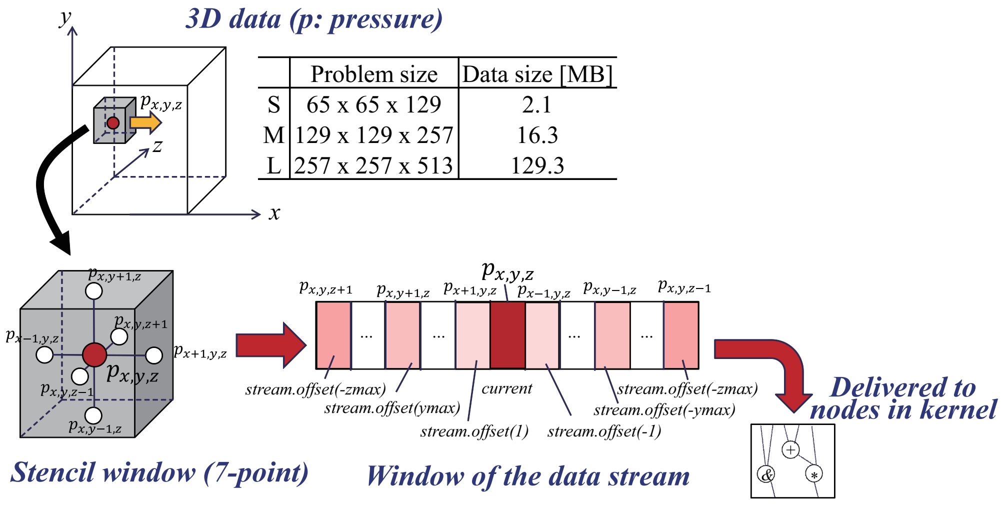
On the other hand, in the dataflow computing model, a time-consuming kernel to be accelerated is represented as a dataflow graph. Since each node of the dataflow graph is directly mapped into a functional unit on the FPGA, the computation is performed by forwarding intermediate results directly from one functional unit to the next units without a cyclic access for memory. Once the data are fed into the accelerator from the memory, we can perform chains of processing until all of processing on the dataflow is completed. Therefore, we can make use of inherent locality of dataflow computation, avoiding memory bottleneck.

Here, we briefly describe an overview of Maxeler's dataflow engines [5]. In the Maxeler platform, we develop the kernel code by the hardware compiler called MaxCompiler. The Maxeler platform utilizes Java as a meta-programing language for implementation. Hence, Java extensions are used for hardware descriptions. Using a meta-program that describes the structure of dataflow as an input, MaxCompiler generates the .max file which contains the FPGA bitstream. Also, data exchange between a host CPU and a dataflow engine is performed using a run-time library API called MaxCompilerRT.

In the Maxeler system, data streaming and execution are performed as follows. The dataflow engine is initialized and the .max file generated by MaxCompiler is loaded from the CPU to the engine, configuring the FPGA. After input data are streamed from CPU memory onto the FPGA chip, these are processed by forwarding intermediate results from one functional unit to another where the results are needed, without ever being written to the off-chip memory until the chain of processing is complete. After all processing on the dataflow engine is completed, the final output data are transferred to CPU memory.

Using a high-level hardware description language and compiler by the Maxeler platform, we attempt to optimize reconfigurable dataflow especially focusing on memory locality and memory bandwidth. Here, we present strategies of optimization in the following four aspects.

Optimization (A): We optimize a kernel design to minimize memory bandwidth demands by explicitly forwarding data directly from operation to operation. In the dataflow computing model, nodes of a dataflow graph are directly arranged into cascaded functional units implemented on hardware. Then, we can use the locality of processing as much as possible by keeping intermediate results within a pipeline. In addition, since the dataflow model implies fine-grained parallelism appeared in dataflow graph, we can make use of more parallelism inherent in applications that are traditionally hard to be parallelized by the instruction-based models. Also, dataflow engines can make use of temporal parallelism by pipelining dataflow paths. Using heavily pipelined structures, we can overlap chains of processing once the host memory provides only initial source data.



	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

Fig. 2. The `stream.offset` and each point of stencil computation. Here, we depict 7-point out of 19-point stencil in the Himeno benchmark.

Optimization (B): We optimize kernel designs using the characteristics of applications so that the data would be organized into highly regular streams. Here, we target loops that can be implemented implicitly by the stream of data for an acceleration kernel. Because these loop structures are just a mechanism for addressing the data in the array, the array based accesses can be translated into one-dimensional sequential accesses without explicit loop structures. The regularity of the flow of data is strongly emphasized so that deeply pipeline processing can be performed without fatal pipeline stalls. In kernels, not to break the regularity of data stream, random access to global memory is not supported and only limited forms of control flow are provided.

Optimization (C): We optimize kernel designs to fully utilize hardware resources as much as possible. In the dataflow computing model, nodes of a dataflow graph are directly mapped into functional units on hardware. Therefore, we attempt to create copies of the hardware to maximize parallelism. Here, we refer the body of a custom dataflow pipeline as a pipe. By unrolling independent loops, we can build multiple parallel pipes in a kernel. This multiple pipe implementation is a scalable design pattern to increase the performance of a dataflow engine.

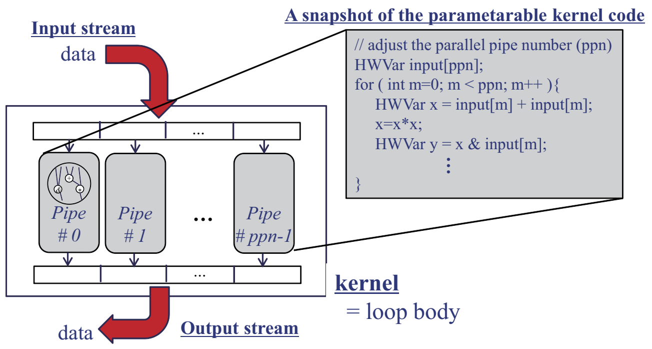
Optimization (D): We optimize communication between a kernel and CPU host code. After identifying the regions that should be accelerated, the code for the accelerator is partitioned from the application program. Because the amount of communication depends on how we partition the kernel code from the application program, we must investigate program partitioning strategy carefully. Here, we note that only the time-consuming portion of code is off-loaded to the accelerator. Then, most of the lines of code in a program will still run on the CPU, and the accelerator productivity augments the performance of the system by porting only the critical part of the code.

IV. Optimizing the Himeno Benchmark

In the dataflow computing model, we must optimize a kernel design applying domain knowledge and experience for an application. These range from the mathematics and the algorithm down to the physical hardware [6], and these are applied to realize highly efficient data streaming. In this Section, we focus on the Himeno benchmark as a case study for application-hardware codesign.

The Himeno benchmark inherently has a high degree of parallelism but requires large memory bandwidth as discussed in Section II. To mitigate heavy memory traffic of a stencil program, we perform Optimization (A) to make full use of locality of memory access as much as possible. Here, we cascade multiple sequential points of the stencil computation into a kernel to keep more intermediate results within pipelines. Since sequential points of a stencil computation often access the same data, we can reuse points that appeared in the previous computations. As a result, input data can be used effectively for sequential computations before outputting data back to memory. Since cascading of stencil point computations increases the amount of computations per memory access, we can reduce memory bandwidth demands.

Next, we perform Optimization (B) so that all data structures are converted into an 1-dimensional data stream. In order to create a window for stencil computation, we need to access values at different points other than the current point in the data stream. Figure 2 shows how we obtain the values other than the current position of the stream. Here, we use `stream.offset` provided by the Maxeler's kernel compiler API. To feed data into a kernel, the one dimensional data stream is buffered in a window. By specifying a relative distance from the current stream data, we can obtain the past or the future data. In the Maxeler platform, the MaxCompiler transforms the `stream.offset` into a FIFO buffer on the FPGA.



```
// adjust the parallel pipe number (ppn)
HWVar input[ppn];
for ( int m=0; m < ppn; m++ ){
    HWVar x = input[m] + input[m];
    x=x*x;
    HWVar y = x & input[m];
            ⋮
}
```

Fig. 3. An overview of a parameterizable kernel design.

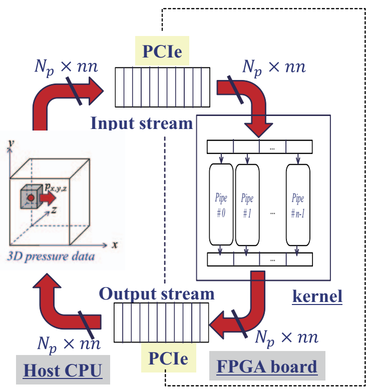


Fig. 4. Data stream via PCIe bus (PCIe-**).

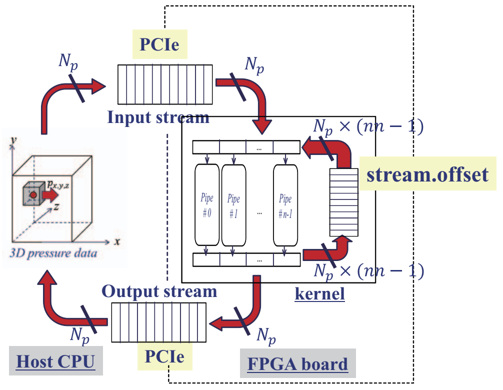


Fig. 5. Data stream via an internal buffer (nn-itr-**).

Next, we perform Optimization (C) to adjust parallelism based on the resource utilization. Here, we adjust the number of pipes by varying the degree of cascading of stencil point computations, where a pipe corresponds to a stencil point computation. In the Maxeler platform, it is possible to write parameterizable kernel designs by means of specifying the number of pipes as a variable. Figure 3 shows an overview of a parameterizable kernel design. In the Maxeler platform, hardware variables derived from Java extensions for MaxCompiler are only used for generating hardware of dataflow engines while Java variables can be used as constants in hardware. Using Java variables as parameters for the pipe configuration, we can transparently translate the parameterized looped kernel to unrolled multiple parallel pipes.

In addition to parallelism adjusted by the parameterizable pipe design, how the data are streamed into an FPGA is a key to achieve high performance. Therefore, we perform Optimization (D) by building three scenarios for the stream communication. The simplest implementation is that all of stream input and output are communicated via the PCIe bus as shown in Figure 4. Here, we refer to this design as 'PCIe-**' (** shows the number of pipes). In this design, the stream of data on the accelerator is driven by the triply-nested *i,j,k* loop. Then, we activate the accelerator once per iteration of the outermost loop driven by n. After a set of output data for the iteration is stored in the host CPU memory, we feed it to PCIe bus as the input data for the next accelerator activation. Here, we implement this data switch by exchanging the pointers of input and output data regions.

While this design is straightforward, the data transfer via PCIe bus might be the bottleneck. In Figure 4, the amount of data transferred via each communication path is presented; the N_p represents the data size of p. The nn is the number of iterations of the outermost loop n corresponding to line:1 of the source code in Figure 1. Here, we can understand that there are a large amount of communication must be done in this design.

Figure 5 shows the design that makes use of iterative behavior of the kernel. Here, we refer to this design as 'nn-itr-**'. In order to avoid the bottleneck by the data transfer, we feed the output data of the outermost loop iteration directly to their input without data transfers between the host CPU memory. For keeping the values from the previous iteration of the outermost loop indexed by n, we use `stream.offset` for inserting a special buffer. Then, data stream from the previous iteration of the outermost loop is fed into the pipes directly. In this design, data stream is seamlessly fed into the pipes once the host CPU transfers the initial data stream. Therefore, the performance of this design is expected to be high. However, this design requires a special buffer for keeping the values from the previous iteration, and this consumes BRAMs equal to the size of array p. Using this buffer, we can reduce the communication between CPU and FPGA up to N_p.

To apply larger data sets, we attempt to use on-board DRAM devices. While DRAM memory access latency is slower than that of on-chip BRAM memory in the FPGA, on-board DRAM can provide several orders of magnitude larger amount of memory. Figure 6 shows the kernel design that uses on-board DRAM device. Here, we refer to this design as 'DRAM-**'. In this design, the data stream on the accelerator is driven by the triply-nested *i,j,k* loop similar to the 'PCIe-**' design while the 'DRAM-**' design can make use of higher bandwidth of

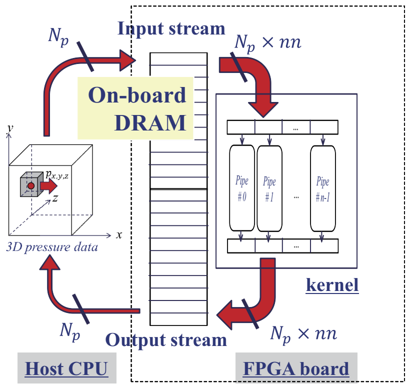


Fig. 6. Data stream via on-board DRAM (DRAM-**).

on-board DRAM memory compared with that of PCIe bus.

Also, the accelerator is activated once per iteration of the outermost loop driven by n. The kernel reads the input data from on-board DRAM and writes the output data to the same location of the DRAM. Then, iterations of the outermost loop are processed without any data transfer to the host CPU memory once the initial input data are loaded into the on-board DRAM memory. After all of the kernel computations are completed, the final output data are transferred to host CPU memory. Therefore, we can dramatically reduce the amount of communication between FPGA and CPU just like the case of 'nn-itr-**'. The disadvantage of this design is that we have to implement memory address generators on the FPGA chip. This consumes resources on the FPGA chip and sometimes the logic for DRAM access becomes a part of the critical path.

V. Performance Evaluation

A. Methodology

We evaluate our design using the Maxeler MaxWorkstation, which is composed of a MAX3 acceleration card and Intel Core i7-870 2.93GHz CPU. The MAX3 acceleration card contains a Virtex-6 SX475T FPGA, 24GB DDR3 memory and PCI express gen2 x8 interface. We use the MaxCompiler version 2012.1 to generate kernel codes. Host codes are compiled using gcc.4.1.2 with '-O3' option. Also, we set the clock frequency of kernels on FPGA as 100MHz, which is the default value of this platform.

We use the source code of Himeno benchmark, static allocate version implemented in C language. There are several input data set in the Himeno benchmark. We use Small (S), Medium (M), Large (L) data set in this evaluation. The problem size and the data size of each data set are shown in Figure 2. For kernel code, we implement our design as follows. To save memory footprint on an FPGA board, we generate constant values using `ternary-if` logic based on i,j,k indexes, and then only the array p is the data transferred to the on-board memory or FPGA in our design. To remove unnecessary parts from the acceleration code, we do not calculate the *gosa* value appeared in the original source code. The reason is that this value can be calculated by an extra

TABLE I
Performance via PCIe communication (PCIe-**).

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

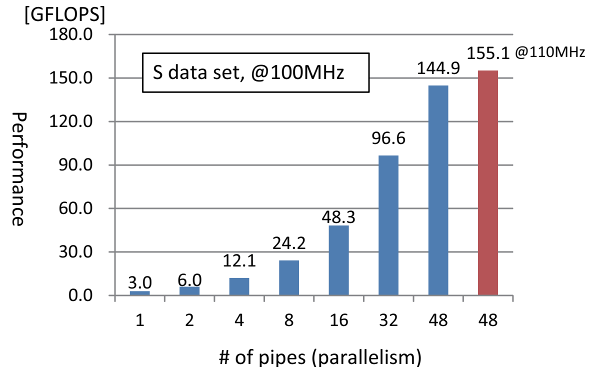


Fig. 7. Results of the design using an internal buffer (nn-itr-**).

invocation of the outermost loop n and this calculation should be done as a post-processing step on the CPU host code.

B. Results

1) Verification of our designs: First of all, we compared the output values of p from accelerator with those from the original CPU implementation. Then we confirmed that all such comparisons showed that the two are the same. We presume that support for the IEEE floating point format provided by Maxeler contributes to accurate and smooth implementations on an FPGA.

2) Stream communication via PCIe: Table I shows the performance obtained by the stream communication via PCIe bus. Here, the results for the S data set are shown. From the results, we find that the performance is not increased even if we scale the number of pipes when the data transfer via the PCIe bus becomes the bottleneck. Also, there are situations that each pipe in a kernel is not running with the maximum overlap. These occur once at the beginning and once at the end for an activation of the accelerator.

3) Streaming via an internal buffer: Figure 7 shows the design using an internal buffer. From the results, we find that this design can linearly increase its performance when we scale the number of pipes. Here, we successfully build and run the design with 48 pipes, but cannot build designs larger than 48 pipes. In the 48-pipe design, we can achieve 144.9 GFLOPS.

Then, we attempt to build designs with higher clock frequencies. As a result, we can build and run the 48-pipe design with 110MHz and this achieves 155.1 GFLOPS. We note that we can avoid not only the data transfer bandwidth bottleneck,

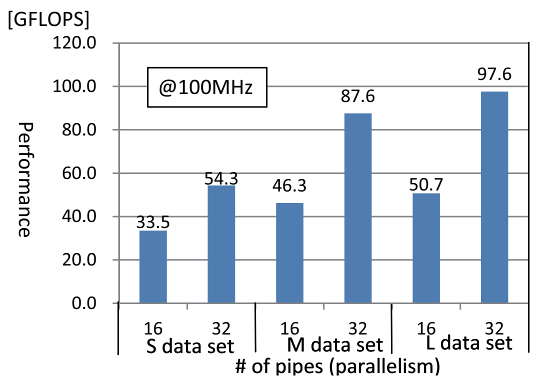


Fig. 8. Results of the design using on-board DRAM (DRAM-**).

but also the pipeline start-up overheads for an activation of the accelerator. This is because a new input steam is directly fed from the buffer soon after the current stream has completed.

4) Streaming via on-board DRAM: Figure 8 shows the design using on-board DRAMs. From the results, we find that this design can accept larger data sets and increase its performance according to the number of pipes. Here, we successfully build and run the design with 32 pipes. In the 32-pipe design, we can achieve 97.6 GFLOPS in the case of the L data set. Here, we cannot build the 48 pipe designs just like the case of 'nn-itr-48'. This is because we have to implement memory address generators on the FPGA in addition to the kernel logic.

We note that there is start-up overhead of the kernel pipeline for an activation of the accelerator because we have to wait until the complete output stream is written to memory before starting a new iteration of the outermost loop. Especially in the case that the data size is not so large, the kernel computation driven by the triply-nested *i,j,k* loop is not long enough to amortize the pipeline overhead for its start-up and ending. Then, the overhead is more significant than the implementation for larger data set in the S data set.

On the other hand, the L data set size is large enough to amortize the pipeline overhead. Therefore, the 32-pipe design with the L data set can achieve almost the same performance as the 32-pipe design using internal buffers in Figure 5. So, the design with the L data set is considered to avoid the pipeline overhead and memory bottleneck while the DRAM memory access latency and bandwidth are limited compared with these of on-chip BRAM memory.

C. Resource utilization of each design

Table II shows the actual number of resources in Virtex-6 SX475T FPGA, and the resource utilization of each design. Here, we represent input data set in the parentheses after the name of each design.

From the results, it is observed that the amount of resources is increased according to the number of pipes. Also, there are variations among three designs if we compare them with the same number of pipes. We find that the design 'PCIe-**'

TABLE II
RESOURCE UTILIZATION.

	LUTs	FFs	BRAMs	DPSs
Virtex6 SX475T	297600	595200	1064	2016
PCIe-1 (S)	4.0%	2.3%	3.1%	1.2%
PCIe-2 (S)	5.3%	3.3%	3.2%	2.1%
PCIe-4 (S)	8.2%	5.1%	3.2%	3.9%
PCIe-8 (S)	14.5%	9.2%	4.0%	7.4%
nn-itr-1 (S)	6.3%	3.9%	52.5%	0.9%
nn-itr-2 (S)	9.3%	6.0%	53.0%	1.8%
nn-itr-4 (S)	13.3%	8.9%	51.6%	3.6%
nn-itr-8 (S)	24.0%	16.0%	52.3%	7.1%
nn-itr-16 (S)	31.1%	19.4%	53.5%	14.3%
nn-itr-32 (S)	51.3%	33.2%	51.0%	28.9%
nn-itr-48 (S)	75.9%	48.9%	53.7%	43.2%
DRAM-16 (S)	36.9%	24.6%	15.2%	14.3%
DRAM-32 (S)	64.3%	41.3%	22.1%	28.6%
DRAM-16 (M)	35.9%	23.9%	19.8%	14.6%
DRAM-32 (M)	64.7%	41.5%	24.8%	28.6%
DRAM-16 (L)	35.3%	23.9%	36.6%	14.6%
DRAM-32 (L)	64.6%	41.2%	42.1%	28.6%

can be realized with the smallest resources compared with the other designs implementing the same number of pipes.

For the 'nn-itr-**' design, the `stream.offset` is used to realize the buffer. Since it consumes BRAMs on the FPGA equal to the size of array *p* in addition to the BRAMs for the 19-point stencil window, the amount of BRAM resources is larger than the other designs. Here, we note that we cannot implement the designs other than the S data set because the required capacity of BRAMs exceeds that in the actual resource.

For the 'DRAM-**' design, we have to add resources for memory address generators. The resource requirements for memory address generators increase the resource usages for LUTs and FFs. Also, when using larger data set, we need to keep its larger stencil window within an FPGA chip. Therefore, the usage of BRAM resources is increased according to the size of data set.

D. Comparison to the GPU implementation

We attempt to compare the performance of the Himeno benchmark implemented on the state-of-the-art platforms such as GPUs. As a baseline, the performance of *CPU (1-core)* is obtained on the host CPU using a binary code generated by Intel Compiler 11.1 with '-fast' option. In the *GPU+pgcc* implementation, we insert directives for GPUs into the original source code and generate a binary code by PGI C compiler 11 with using '-fast -ta=nvidia' option. Then, the code is executed on a Tesla M2050 GPU. In the *CPU+CUDA* implementation, data are the ones presented in the paper by Philips *et. al* [8]. Also, we pick up the highest performance results of our designs for the FPGA-based implementation.

Figure 9 shows the performance achieved by each platform. Here, all results for accelerators are obtained using a single accelerator card. From the results, we can see that our 'nn-itr-48' implementation outperforms the *GPU+CUDA* implementation by a factor of 3 while the data set is limited to the S

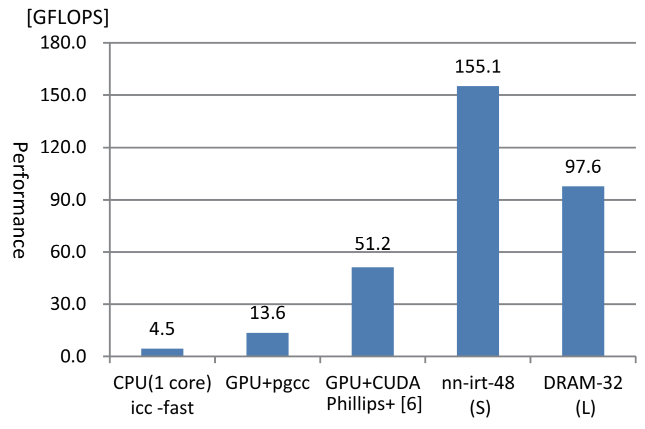


Fig. 9. Performance comparison to the GPU implementation.

data set. Also, our 'DRAM-32' implementation outperforms the *GPU+CUDA* implementation by a factor of 1.9 when we compare these of the L data set.

In addition, these results demonstrate that we can achieve better performance compared with other accelerator platforms even if we use high-level hardware description language and high-level synthesis for an FPGA implementation. This might imply higher level optimization and customization supported by high-level languages are becoming more important to achieve significant performance gain when using accelerators.

VI. Related works

As far as we know, this is the first paper that compares the performance of FPGAs and GPUs using intensively optimized application code based on the Himeno benchmark. While intensive optimization and performance tuning are time consuming tasks, these are critical for designs with accelerators. Without knowledge of characteristics for them, we cannot estimate the expected performance for each platform.

Sano *et al.* implement a 2D stencil computation on FPGA arrays [10]. While this can achieve 260 GFLOPS by their scalable streaming array, they use 9 FPGA boards to implement it. Also, as their design is a stand-alone implementation without host-accelerator communication, all the portions of the application must run on the FPGA boards.

VII. Conclusions

In this paper, 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, which especially focus on memory locality and memory bandwidth. Also we have shown how such techniques can be used in optimizing the Himeno Benchmark. From the results of evaluation, we have confirmed that the advantages of specialized dataflow pipelines contribute to improving the actual performance of applications that requires large memory bandwidth. We also have found that our implementation can outperform the current state-of-the-art GPU implementations in their achieved GFLOPS. In addition, we have demonstrated that we can achieve competitive performance gain without resorting to low-level HDL coding.

Current and future work includes extending the set of optimization strategies and benchmarks for our approach, exploring methods for automating the application of such optimization strategies, and studying how our approach can be improved to support power and energy optimization.

Acknowledgment

This work was supported in part by the JSPS "Institutional Program for Young Researcher Overseas Visits", by UK Engineering and Physical Sciences Research Council, by the European Union Seventh Framework Programme under Grant agreement number 248976, 257906, 287804 and 318521, by the HiPEAC Network of Excellence, by the Maxeler University Program, and by Xilinx.

References

[1] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?" in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 225–236.

[2] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with GPUs and FPGAs," in *IEEE Symp. on Application Specific Processors*, 2008, pp. 101–107.

[3] J. Richardson *et al.*, "Comparative analysis of HPC and accelerator devices: Computation, memory, I/O, and power," in *Fourth Int'l Workshop on High-Performance Reconfigurable Computing Technology and Applications*, 2010, pp. 1–10.

[4] B. Betkaoui, D. Thomas, and W. Luk, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *Int'l Conf. on Field-Programmable Technology*, 2010, pp. 94–101.

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

[6] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and O. Mencer, "Beyond traditional microprocessors for geoscience high-performance computing applications," *IEEE Micro*, vol. 31, no. 2, pp. 41–49, Mar. 2011.

[7] http://accc.riken.jp/HPC/HimenoBMT/index-e.html.

[8] 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.

[9] S. Matsuoka, T. Aoki, T. Endo, A. Nukada, T. Kato, and A. Hasegawa, "GPU accelerated computingfrom hype to mainstream, the rebirth of vector computing," in *SciDAC 2009, Journal of Physics: Conference Series 180*, 2009.

[10] K. Sano, Y. Hatsuda, and S. Yamamoto, "Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth," in *IEEE Int'l Symp. on Field-Programmable Custom Computing Machines*, 2011, pp. 234–241.