OpenCL-Based FPGA-Platform for Stencil Computation and Its Optimization Methodology

Hasitha Muthumala Waidyasooriya, *Member, IEEE*, Yasuhiro Takei, Shunsuke Tatsumi, and Masanori Hariyama, *Member, IEEE*

Abstract—Stencil computation is widely used in scientific computations and many accelerators based on multicore CPUs and GPUs have been proposed. Stencil computation has a small operational intensity so that a large external memory bandwidth is usually required for high performance. FPGAs have the potential to solve this problem by utilizing large internal memory efficiently. However, a very large design, testing and debugging time is required to implement an FPGA architecture successfully. To solve this problem, we propose an FPGA-platform using C-like programming language called open computing language (OpenCL). We also propose an optimization methodology to find the optimal architecture for a given application using the proposed FPFA-platform. According to the experimental results, we achieved 119 $\sim$ 237 Gflop/s of processing power and higher processing speed compared to conventional GPU and multicore CPU implementations.

Index Terms—OpenCL for FPGA, high performance computing, stencil computation, FDTD

1 Introduction

Stencil computation [1] is an iterative method where a grid is updated in each iteration according to a fixed computation pattern. As shown in Fig. 1, a stencil is a shape that consists of neighboring grid points called cells. The typical computation pattern is in the form of a sum of products. Stencil computation is widely used in scientific computations such as fluid dynamics [2] simulations, electromagnetic simulations [3], iterative solvers [4], [5], etc. It is one of the most researched subjects and yet requires improvements in many areas.

Many works have been done already to accelerate stencil computation using parallel processing capabilities of the recent multicore CPUs and GPUs. However, stencil computation has a small operational intensity [6] so that the performances are usually limited by the external memory bandwidth. The operational intensity is defined as the ratio of floating point operations to the total data movement. Temporal blocking is used in CPUs [7], [8] and GPUs [9], [10], [11] to increase the performance of the stencil computation. It is a technique that uses the cache or the internal memory to re-use the data between consecutive iterations without accessing the external memory. Recently, FPGA accelerators have been successfully used to increase the performance of stencil computation [12], [13]. The large amount of registers in the FPGAs are utilized to transfer the data of one iteration to the next internally without accessing the external memory. This method increases the operational intensity and also the processing speed.

Hardware design language (HDL) is usually used to design FPGA accelerators. HDL-based design requires an extensive knowledge about hardware and also a large design time due to clock-cycle-level simulations, testing and debugging. It is also very challenging to implement I/O interfaces such as memory and PCIe for the applications to transfer data and to communicate with the outside world (off-chip memory, host PC, etc). With these limitations, FPGAs are rarely used in actual scientific computation environments irrespective of their potentially superior performance compared to multicore CPUs and GPUs.

To overcome those severe disadvantages in HDL-based design, open computing language (OpenCL) for FPGA has been introduced [14]. The FPGA accelerator design is done using a C-like programming environment. The I/O interfaces are automatically generated. Moreover, OpenCL is a complete framework that includes firmware, software and device drivers to connect, control and transfer data to and from the FPGA. It supports different FPGA boards by the means of a board support package (BSP). It even comes with a software-based emulator for testing and debugging. There are some very recent works in [15], [16] that propose OpenCL-based FPGAs architectures. However, none of those are about stencil computation. One rare attempt in [17] to design an FPGA accelerator for stencil computations using OpenCL was not a great success due to low performance.

In our previous works in [18], we proposed an OpenCL-based FPGA accelerator for 2-D finite-difference time-domain (FDTD) computation. In this paper, we propose an OpenCL-based FPGA-platform for stencil computations that cover a wide range of applications. We also propose an optimization methodology to find the optimal architecture for any OpenCL compatible FPGA board. We have evaluated the proposed platform for both 2-D and 3-D stencil

• *The authors are with the Graduate School of Information Sciences, Tohoku University, Aoba 6-6-05, Aramaki, Aoba, Sendai, Miyagi 980-8579, Japan. E-mail: {hasitha, hariyama}@tohoku.ac.jp, {takei, s_tatsumi}@ecei.tohoku.ac.jp.*

Manuscript received 7 Mar. 2016; revised 20 Sept. 2016; accepted 29 Sept. 2016. Date of publication 4 Oct. 2016; date of current version 12 Apr. 2017.
Recommended for acceptance by R. Cumplido.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2614981

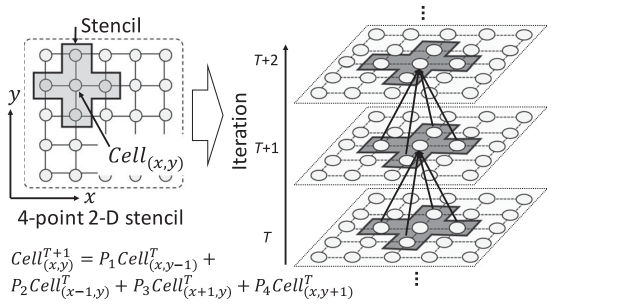
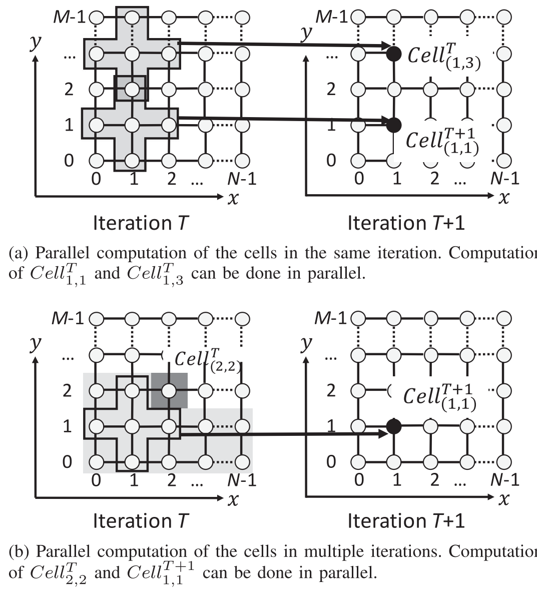

Stencil
y
x
$Cell_{(x,y)}$
4-point 2-D stencil
$Cell^{T+1}_{(x,y)} = P_1 Cell^{T}_{(x,y-1)} + P_2 Cell^{T}_{(x-1,y)} + P_3 Cell^{T}_{(x+1,y)} + P_4 Cell^{T}_{(x,y+1)}$
Iteration
T+2
T+1
T

Fig. 1. Stencil. Computation using a four-point 2-D stencil. The computations of the cells in a new iteration are done using the computation results of the cells in the previous iteration.



(a) Parallel computation of the cells in the same iteration. Computations of $Cell^T_{1,1}$ and $Cell^T_{1,3}$ can be done in parallel.

(b) Parallel computation of the cells in multiple iterations. Computations of $Cell^T_{2,2}$ and $Cell^{T+1}_{1,1}$ can be done in parallel.

Fig. 2. Computations in two iterations using an $N \times M$ grid.

computations in single-precision and achieved better performance compared to multicore CPUs and GPUs. We also achieved more than 60 percent of the peak performance provided by the FPGA. Although the proposed method can be applied for the double-precision computation, the performance using the current generation FPGAs is not good compared to the single-precision computation.

2 Related Works

Stencil computation is a grid-based iterative computation method that has a large number of parallel operations. Fig. 2 shows the computations in two iterations using an $N \times M$ grid. There is a data dependency for the computations among multiple iterations and no data dependency for the computations in the same iteration. Therefore, we can compute the cells in the same iteration in parallel as shown in Fig. 2a and we call this "cell-parallel" computation. The number of cells computed in parallel in the same iteration is called the "degree of cell-parallelism". Fig. 2b shows the computations of two consecutive iterations. To compute $Cell^{T+1}_{1,1}$ in the iteration $T+1$, data of its surrounding cells belongs to the iteration T are required. When the computation of $Cell^T_{2,2}$ is in progress in the iteration T, all the data required for the computation of $Cell^{T+1}_{1,1}$ are available. Therefore, the computations of $Cell^T_{2,2}$ and $Cell^{T+1}_{1,1}$ can be done in parallel. We call this "iteration-parallel" computation. The number of iterations computed in parallel is called the "degree of iteration-parallelism".

2.1 CPU/GPU Implementations

The most straightforward stencil computation method is to use cell-parallel iteration-serial computation. To implement this method successfully, we have to access the data in parallel. However, the grid size is usually very large and it is not possible to store the data of one whole iteration in the internal memory. Therefore, the data are written to the external memory and accessed in each iteration. As a result, the performance is restricted by the external memory bandwidth. The GPU implementations [19] give better results due to their large memory bandwidth compared to multicore CPUs.

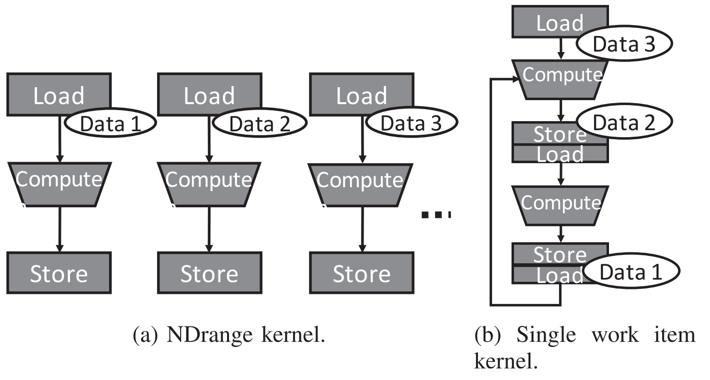
Temporal blocking can be used to remove the external memory bandwidth bottleneck by offloading some of the data to the cache or the internal memory. There are many temporal blocking techniques such as split tiling [10], overlapped tiling [9], [20], diamond tiling [21], time skewing [22], [23], multicore-aware wavefront temporal blocking [24], [25], etc. Different techniques can be suitable for different devices and significant improvements can be achieved for some applications.

2.2 FPGA Implementations

The cell-parallel computation is not suitable for FPGAs due to smaller external memory bandwidth. The overlapped tiling method has been implemented on FPGAs in [17], [26], [27]. However, the performances are restricted by the redundant computations in large overlapped regions. The most successful FPGA implementations such as [12], [13], and [28] use iteration-parallel computation by caring data between multiple iterations through shift-register arrays. Since FPGAs contain a large amount of registers compared to CPUs and GPUs, very long shift-register arrays can be implemented.

The research in [13] is the most comprehensive and recent work done on FPGA-based stencil computation. It proposes a scalable custom FPGA accelerator. It is re-programmable for different stencil computations using an instruction set. However, the data paths and the connections between the processing elements (PEs) are fixed. Therefore, re-programming it for complex stencil computations such as FDTD [3] could be a difficult task due to the data dependencies and the boundary conditions exist among electric and magnetic field computations. Additional pipeline registers are required to solve the data dependencies and conditional branches are required to implement various boundary conditions. To implement such changes, re-designing of the entire data path is required. Moreover, the evaluation in [13] only considers Jacobi computation, and the performance of the other types of stencil computations are not clear.

The research in [28] is another recent work that proposes an FPGA-based stencil computation accelerator and its optimization methodology. It uses a high level design tool called MaxCompiler [29]. However, [28] has not done any experiments to compare their results with any other works that use FPGAs, GPUs or CPUs. Moreover, MaxCompiler can be used in very limited number of FPGA environments that are usually the products of the Maxeler Technologies.



(a) NDrange kernel. (b) Single work item kernel.

Fig. 3. Different kernel execution models in OpenCL for FPGA. (a) Since the parallelism is exploited by executing the same operation on multiple data simultaneously, this model is suitable when there are no data dependencies. (b) Since the parallelism is exploited by executing multiple operations on different data simultaneously, this model is suitable when there are data dependencies.

3 OpenCL-Based Stencil Computation Architecture

OpenCL is a framework to write programs to execute across heterogeneous parallel platforms [30]. It views a system as a number of computing devices (OpenCL devices) connected to a host. The host is usually a CPU while the devices can be any of OpenCL capable CPUs, GPUs, FPGAs, etc. A device contains one or more compute units and such compute units contain one or more processing elements. Kernels are the functions that are executed on an OpenCL device. The unit of the concurrent execution of a kernel is called a work item. Work items are organized into work groups. The entire collection of work items is called the NDrange, where work groups and work items can be divided into different dimensions.

Since OpenCL-based FPGA design is a very recent topic, we briefly discuss how the "Altera offline compiler (AOC)" converts an OpenCL code in to an FPGA design. The most important and significant difference between OpenCL for FPGA and OpenCL for CPU/GPU is that, AOC distinguishes a kernel executed by a single work item from that of multiple work items. In FPGAs, a kernel executed with no reference to the work item ID, or a kernel declared with the "task" attribute is called a "single work item kernel". On the other hand, a kernel executed by multiple work items is called an "NDrange kernel". Fig. 3 shows these two types of kernels. The execution of NDrange kernel shown in Fig. 3a is similar to GPU processing. This is very effective when the work items are completely independent. However, when there are data dependencies, users have to explicitly insert barriers at different stages of the execution so that every work item must complete their operations up to the barrier stage, before proceeding to the next stage. This synchronization mechanism costs a lot of hardware and decreases the performance. In single work item kernels, the data dependent operations are performed one after the other as shown in Fig. 3b. The parallelism is achieved by pipelining. AOC analyses loops and generates pipeline stages automatically for different operations in the loop. When there are nested loops, we can unroll the inner-loops using "#pragma unroll" directive. When the inner-loops are unrolled, processing elements are generated for each loop-iteration for parallel processing. AOC also generates a separate hardware for each conditional branch so that the branches are

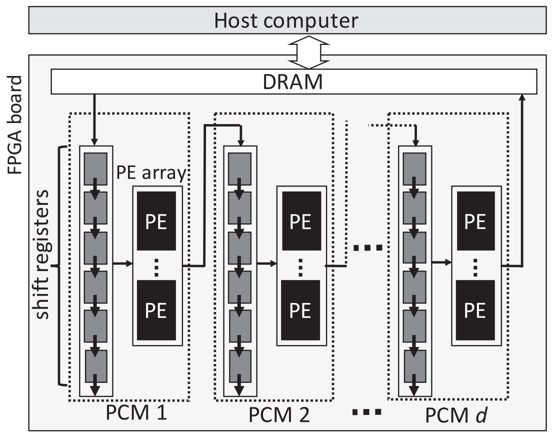


Fig. 4. The FPGA-platform for stencil computation. A PCM is a pipelined computation module and d PCMs are used to compute d iterations in parallel.

processed in parallel. This is a major difference compared to CPU or GPU based OpenCL implementations. We use single work item kernels since there are data dependencies among iterations as explained in Section 2.

In the proposed FPGA-platform, we mainly use iteration-parallel computation where multiple iterations are processed in parallel. The degree of iteration-parallelism is given by d. We can also use cell-parallel computation similar to GPUs and CPUs. Note that, only the adjacent cells are processed in parallel. The degree of cell-parallelism is given by P_{cell}. Increasing either d or P_{cell} increases the performance. We use vector data types such as $float2, float4, \ldots$ in the OpenCL code and process all data in a vector in one step using SIMD computation on FPGA. To process multiple cells, we have to access multiple data in parallel. That requires more external memory bandwidth. Therefore, we can only increase P_{cell} until the required memory bandwidth reaches its limit. Increasing P_{cell} beyond the maximum bandwidth does not increase the performance.

Fig. 4 shows the FPGA-platform for stencil computation. It consists of a DRAM and d "pipelined computation modules" (PCMs). Each PCM process the stencil computation belonging to one iteration. A PCM consists of shift-registers and multiple processing elements. The computation of a cell is done in a PE. Multiple PEs in a PCM compute multiple cells belonging to the same iteration in parallel. Shift-registers are used to transfer the computed data of one iteration to the next iteration. Multiple PEs are used to process P_{cell} number of cells per an iteration in parallel. Based on this platform, we design the architecture for a given stencil computation application by determining the values of d and P_{cell} and also the computation of a PE. Therefore, d and P_{cell} are the design parameters. Please note that, since we used only the OpenCL code and not the HDL code, we cannot confirm that the compiler-generated accelerator is exactly the same as the one in Fig. 4. However, referring the OpenCL for FPGA programming manual, optimization guide and other reference materials [31] and also observing the behavior of the accelerator, we can say that the architecture of the compiler-generated accelerator must be very similar to the one shown in Fig. 4.

Algorithm 1 shows the pseudo code of the stencil computation accelerator (computation kernel). Shift-registers are generated according to the code from lines 6 to 13. The computations in each PCM are defined from lines 14 to 29. All

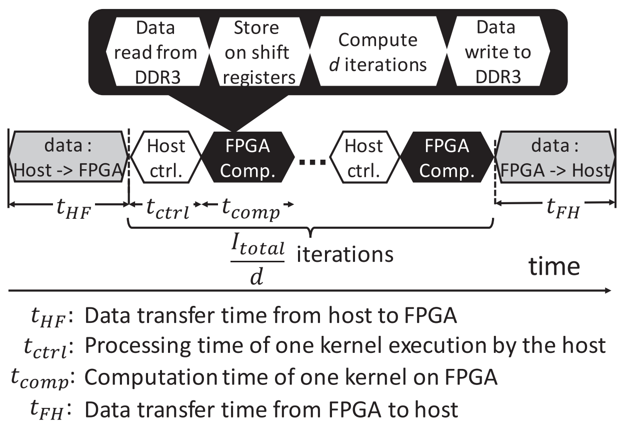

Data read from DDR3
Store on shift registers
Compute d iterations
Data write to DDR3
data : Host -> FPGA
Host ctrl.
FPGA Comp.
Host ctrl.
FPGA Comp.
data : FPGA -> Host
t_{HF}
t_{ctrl}
t_{comp}
t_{FH}
$\frac{I_{total}}{d}$ iterations
time
t_{HF}: Data transfer time from host to FPGA
t_{ctrl}: Processing time of one kernel execution by the host
t_{comp}: Computation time of one kernel on FPGA
t_{FH}: Data transfer time from FPGA to host

Fig. 5. Time chart of the stencil computation on FPGA.

the loops are unrolled completely to produce an output in every clock cycle. As shown in line 22, the computation of each PCM is delayed for a certain number of clock cycles until all required inputs are stored in the shift-registers.

Algorithm 1. Pseudo Code of the Stencil Computation Kernel

```
1  __kernel stencil (din, dout)
2      shiftreg[$d$][$size$];
3      result[$size$];
4      Loop iterations = $x$;
5      while $count \neq x$ do
6          #pragma unroll
7          for $i = (size - 1) \rightarrow i = 1$ do
8              #pragma unroll
9              for $j = 0 \rightarrow j = d$ do
10                 shiftreg[$j$][$i$] = shiftreg[$j$][$i - 1$];
11             end
12         end
13         shiftreg[0][0] = din[$count$];
14         #pragma unroll
15         for $j = 0 \rightarrow j = (d - 1)$ do
16             //Boundary conditions
17             $V_1$ = (condition 1) ? shiftreg[j][2] : 0.0;
18             $V_2$ = (condition 2) ? shiftreg[j][$N + 1$] : 0.0;
19             ...
20             //Computation
21             result[$j$] = $c_1.V_1 + c_2.V_2 + ...$;
22             if $count > (j + 1) \times latency$ then
23                 if $j == (d - 1)$ then
24                     dout[$count$] = result[$j$];
25                 else
26                     shiftreg[$j + 1$][0] = result[$j$];
27                 end
28             end
29         end
30         $count++$;
31     end
32 end
33
```

4 Optimization Methodology

4.1 Processing Time Estimation

Fig. 5 shows the time-chart of the stencil computation. The total processing time composed of the data transfer time, the computation time and the control overhead by the host. The total processing time (t_{total}) is given by Eq. (1) where I_{total} is the total number of iterations and d is the degree of iteration-parallelism. That is the number of iterations processed in the FPGA during a kernel execution

$$t_{total} = t_{HF} + \frac{I_{total}}{d} \times (t_{ctrl} + t_{comp}) + t_{FH}. \quad (1)$$

The measured data transfer times from host-to-FPGA and FPGA-to-host are given by t_{HF} and t_{FH} respectively. The data transfer is done only at the start and the end of the computation. The control overhead per kernel execution by the host is given by t_{ctrl}. These processing times do not depend on the design parameters d and P_{cell}. However, the computation time t_{comp} (or the kernel execution time) varies with the design parameters.

To estimate the computation time, we consider the following two scenarios.

1) Scenario 1: The memory access is faster than the computation. The computation pipeline does not stall due to the memory access such as waiting for the inputs and outputs.
2) Scenario 2: The memory access is slower than the computation. The computation pipeline stalls until the data are read from or written to the memory.

Therefore, it is important to identify the correct scenario that an accelerator belongs to. This can be done by comparing the required memory bandwidth of an accelerator with the maximum practical memory bandwidth of the FPGA board. The required memory bandwidth B_{req} is given by

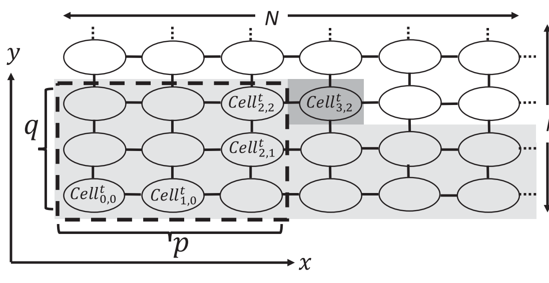
$$B_{req} = (S_{input} + S_{output}) \times P_{cell} \times f_{core}. \quad (2)$$

The sizes (in bytes) of the input and output data accessed in one clock cycle are given by S_{input} and S_{output} respectively. The clock frequency of the accelerator is given by f_{core} and the degree of cell-parallelism is P_{cell}. Note that, estimating f_{core} is also a difficult task and we will discuss this in Section 5. The maximum practical memory bandwidth B_{prac} is given by

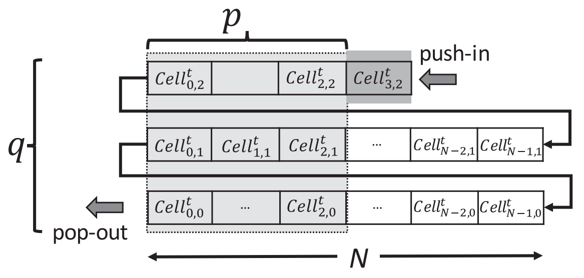
$$B_{prac} = S_{bus} \times f_{mem} \times N_{mem} \times E_{coef}. \quad (3)$$

The memory controller bus width (in bytes), the memory controller frequency and the number of memory banks (modules) accessed in parallel are given by S_{bus}, f_{mem} and N_{mem} respectively. The efficiency coefficient of the memory controller is given by E_{coef}. The theoretical memory bandwidth of the FPGA board is $S_{bus} \times f_{mem} \times N_{mem}$. Since no memory controller has 100 percent efficiency, the practical maximum memory bandwidth is smaller than the theoretical one. To calculate E_{coef}, we can simply divide the theoretical processing time by the measured processing time to access a large amount of data. If $B_{req} < B_{prac}$, the accelerator belongs to scenario 1. If $B_{req} \geq B_{prac}$, the accelerator belongs to scenario 2. The computation time of one kernel execution is given by Eq. (4), where $t_{comp}^{Scenario1}$ and $t_{comp}^{Scenario2}$ are the computation times in scenarios 1 and 2 respectively

$$t_{comp} = \begin{cases} t_{comp}^{Scenario1} & when\ B_{req} < B_{prac} \\ t_{comp}^{Scenario2} & when\ B_{req} \geq B_{prac} \end{cases}. \quad (4)$$



(a) $N \times M$ grid and $p \times q$ stencil. Computations are done by scanning cells from left-to-right and down-to-up.



(b) $(q-1)N+p+1$ shift-registers are required to store data.

Fig. 6. Temporary data storage required in the computation of $Cell^{t+1}_{1,1}$.

We first explain how to estimate the computation time per kernel execution when the accelerator belongs to scenario 1. We consider an $N \times M$ grid and a $p \times q$ stencil as shown in Fig. 6a. For the simplicity, we consider a rectangular stencil. Since a stencil computation is completed in every clock cycle after the pipeline is filled, the computation time equals to $\left\{\frac{1}{P_{cell}} \times N \times M + \text{pipeline latency}\right\}$ clock cycles. The degree of cell-parallelism is given by P_{cell}. The pipeline latency refers to the number of clock cycles required to write the first computation result to the external memory. To estimate the pipeline latency, let us consider the different tasks we have to go through in order to do the stencil computation. As shown in Fig. 5, data are read from the external memory, and go through a series of shift-registers and PEs corresponding to multiple iterations. After that, the computed data are written to the external memory. The number of clock cycles spent in the shift-registers is the dominant component of the pipeline latency. Usually, the length of the shift-registers is several thousands, while the computation cycles are less than a few tens. Since the memory access is required only at the start and the end, it is very small compared to the time spent on shift-registers. Therefore, we consider only the number of clock cycles spent on the shift-registers to estimate the pipeline latency.

Fig. 6b shows how the sequentially read data are stored in shift-registers. For example, to compute $Cell^{t+1}_{1,1}$, we have to store all data in $Cell^t_{0,0} \sim Cell^t_{2,2}$. Another register is required to store the newly read value of $Cell^t_{3,3}$. Therefore, $(q-1)N+p+1$ shift-registers are required per a PCM to compute a $p \times q$ stencil. The computation of the first cell ($Cell^{t+1}_{0,0}$) is delayed by $\frac{(q-1)}{2} \times N + \frac{p+1}{2} + 1$ cycles. When P_{cell} number of data are read sequentially and d iterations are computed, the pipeline latency is $d\left(\frac{(q-1)N+p+1}{2 \times P_{cell}} + 1\right)$. The computation time in scenario 1 is given by

$$t^{Scenario1}_{comp} = \frac{1}{P_{cell} \times f_{core}} \times \left\{d\left(\frac{(q-1)N+p+1}{2} + P_{cell}\right) + N \times M\right\}. \tag{5}$$

We next explain how to estimate the computation time per kernel execution when the accelerator belongs to scenario 2. In this scenario, an output is not produced in every clock cycle due to slow memory access. For example, let us assume that B_{req} is twice as large as B_{prac}. In this case, two clock cycles are required to access one data set (input and output). Therefore, an output is written in every two clock cycles, and the processing time is two times larger than that in scenario 1. In general case, the processing time is B_{req}/B_{prac} times larger than that in scenario 1. The computation time in scenario 2 is given by

$$t^{Scenario2}_{comp} = \frac{B_{req}}{B_{prac}} \times \frac{1}{f_{core} \times P_{cell}} \times \left\{d\left(\frac{(q-1)N+p+1}{2} + P_{cell}\right) + N \times M\right\}. \tag{6}$$

Note that, if we substitute B_{req} and B_{prac} from Eqs. (2) and (3) to Eq. (6), we can rewrite it as

$$t^{Scenario2}_{comp} = \frac{S_{input} + S_{output}}{S_{bus} \times f_{mem} \times N_{mem} \times E_{coef}} \times \left\{d\left(\frac{(q-1)N+p+1}{2} + P_{cell}\right) + N \times M\right\}. \tag{7}$$

From Eq. (7), we can see that $t^{Scenario2}_{comp}$ depends on f_{mem}, and it does not depend on f_{core}. Therefore, if $B_{req} \geq B_{prac}$, the computation time per kernel execution only depends on the memory access and we cannot decrease it by increasing parallel computations or the clock frequency (f_{core}).

4.2 Resource Estimation

The resource constraint is given by Eq. (8). The amount of resources used by the accelerator and the maximum resources available in the FPGA are given by R_{design} and R_{fpga} respectively. The resources could be logic elements, block RAMs (internal memory), DSPs (multipliers), registers, etc

$$R_{design} \leq R_{fpga}. \tag{8}$$

The accelerator is composed of PCMs and I/O interfaces used to communicate with DRAM and host computer. The amount of resources used by the PCMs increases with the degree of iteration-parallelism. The I/O interfaces are composed of DRAM controllers, PCIe controllers and their data paths. Those are pre-determined in the "board support package" given by the FPGA board vendor. Therefore, the amount of resources used by the I/O interfaces is a constant for a given board. As a result, the amount of resources used by the accelerator is given by

$$R_{design} = d \times R_{PCM} + R_{base}. \tag{9}$$

The amount of resources used by a PCM and I/O interfaces are given by R_{PCM} and R_{base} respectively.

A PCM is composed of PEs and a shift-register array. When P_{cell} increases, the number of PEs and their resources

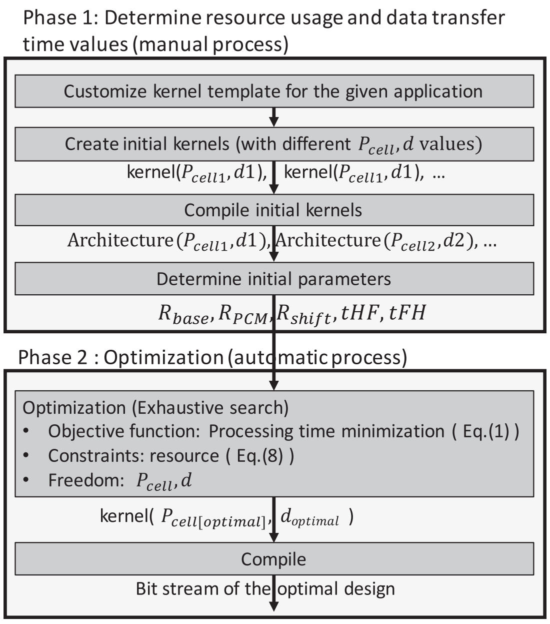

Phase 1: Determine resource usage and data transfer time values (manual process)
Customize kernel template for the given application
Create initial kernels (with different P_{cell}, d values)
kernel($P_{cell1}, d1$), kernel($P_{cell1}, d1$), ...
Compile initial kernels
Architecture ($P_{cell1}, d1$), Architecture ($P_{cell2}, d2$), ...
Determine initial parameters
$R_{base}, R_{PCM}, R_{shift}, tHF, tFH$
Phase 2 : Optimization (automatic process)
Optimization (Exhaustive search)
Objective function: Processing time minimization (Eq.(1))
Constraints: resource (Eq.(8))
Freedom: P_{cell}, d
kernel($P_{cell[optimal]}, d_{optimal}$)
Compile
Bit stream of the optimal design

Fig. 7. The FPGA architecture design flow. The phase 1 is done manually while the optimization in phase 2 is done automatically.

increase. However the amount of resources used by the shift-register array does not change with P_{cell}. It only depends on the grid size of the stencil computation. Therefore, the amount of resources used in a PCM is determined by Eq. (10), where R_{shift} and R_{PE} are the resources used by the shift-register array and PEs respectively

$$R_{PCM} = P_{cell} \times R_{PE} + R_{shift}. \tag{10}$$

4.3 Processing Time Minimization

By using Eqs. (1), (4), (5), and (7), the total processing time can be written in the form as

$$t_{total} \propto \begin{cases} \frac{I_{total}}{P_{cell} \cdot f_{core}} \left(\frac{(q-1)N}{2} + \frac{NM}{d} \right) & B_{req} < B_{prac} \\ \frac{I_{total}}{f_{mem}} \left(\frac{(q-1)N}{2} + \frac{NM}{d} \right) & B_{req} \geq B_{prac} \end{cases}. \tag{11}$$

From Eq. (11), we can say that the total processing time can be reduced by increasing the degree of iteration-parallelism. For large grid sizes and small d, $\frac{NM}{d}$ is very large compared to $\frac{(q-1)N}{2}$. Therefore, the processing time is decided by $\frac{NM}{d}$ and the processing speed linearly increases with d. When d increases, the term $\frac{NM}{d}$ gradually decreases and becomes comparable with $\frac{(q-1)N}{2}$. At this time, although the processing speed increases with d, the relationship is not a linear one. When d is very large, the term $\frac{NM}{d}$ moves towards zero. Therefore, the processing time becomes almost a constant and we cannot increase the processing speed further by increasing d. If there is no resource constraint, the minimum processing time is achieved when d equals I_{Total}. Therefore, even the work in [13] claims of linear processing speed increase, it is only true when the grid size is very large and the degree of iteration-parallelism is small. Moreover, if $B_{req} < B_{prac}$, we can decrease the processing time by increasing P_{cell}.

The design flow is shown in Fig. 7. It has two phases. In phase 1, we determine the resource usage, the data transfer times and the control overhead practically by compiling a few kernels that have different d and P_{cell} values. The data transfer times between the host and the FPGA (T_{HF}, T_{FH}) and the control overhead t_{ctrl} can be determined by measuring them directly. The resource usage values ($R_{base}, R_{PE}, R_{shift}$) can be determine by using Eqs. (9) and (10). For this purpose, we have to compile at least two kernels. The phase 1 takes a few hours to complete due to the large compilation time. In phase 2, we solve the optimization problem to find the optimal architecture that has the minimum processing time under the resource constraint. We do an exhaustive search on a CPU for all the values of d and P_{cell}. The range of the values of P_{cell} is very small. Therefore, the search time does not increase exponentially and a solution can be found in a few seconds even the range of d is very large. After the optimal architecture is found, we compile it to get the bit stream for the FPGA. This compilation also takes a few hours. Proposed optimization method drastically reduces the FPGA accelerator design time. Otherwise, we have to compile many different architectures and measure their processing times to find the optimal solution, which would take days to months.

The synthesis flow proposed in [28] that uses MaxCompiler is very similar to ours. However, [28] reports that there are many mismatches between the estimation and the measured results. There are cases that some of the optimized kernels are failed in the compilation process due to resource constraint violation. As a result, the designer has to compile the next best solution until a successful compilation is achieved. The reason could be that the designed architecture in MaxCompiler may not fully agree with the resource estimation in [28]. However, the optimized kernel provided by our design flow is always passes the compilation process and the estimated results match with the measured ones. We will discuss the accuracy of the optimization methodology in Section 5.

5 Evaluation

For the evaluation, we use two FPGA boards, three GPUs and two CPUs. FPGAs are configured using Quartus 16.0 with OpenCL SDK. GPUs are programmed using CUDA 7.5 and the multicore CPUs are programmed using Intel C compiler 2016 (Intel Parallel Studio XE 2016). The operating system is CentOS 6.7. We used the stencil computation examples shown in Table 1. All computations are done for 15,360 iterations.

5.1 Accuracy of the Estimation

Fig. 8 shows the measured resource usage for Laplace equation computation using different design parameters. Fig. 8a shows the logic resource usage measured by adaptive logic modules (ALMs). The amount of ALMs increases linearly with d. It also increases with P_{cell}. When P_{cell} increases, more PEs are used to compute data in parallel and that requires more logic resources. Fig. 8b shows the memory usage. It increases linearly with d, but remains the same for P_{cell}. For example, when P_{cell} is two, the lifetime of the data in the shift-registers are reduced by half. Therefore, a shorter pipeline is required. However, two of such pipelines are required for parallel processing. Therefore, the internal memory usage does not change. Fig. 9 shows the estimated

TABLE 1
FPGA Boards Used for the Evaluation

Application	Computation
Laplace equation 4,096 × 32,768 grid	$0.25\left(V^t_{i,j-1} + V^t_{i-1,j} + V^t_{i+1,j} + V^t_{i,j+1}\right)$
2-D 5-point Jacobi 4,096 × 32,768 grid	$c_1.V^t_{i,j-1} + c_2.V^t_{i-1,j} + c_3.V^t_{i,j} + c_4.V^t_{i+1,j} + c_5.V^t_{i,j+1}$
2-D 9-point Jacobi 4,096 × 32,768 grid	$c_1.V^t_{i-1,j-1} + c_2.V^t_{i,j-1} + c_3.V^t_{i+1,j-1} + c_4.V^t_{i-1,j} + c_5.V^t_{i,j} + c_6.V^t_{i+1,j} + c_7.V^t_{i-1,j+1} + c_8.V^t_{i,j+1} + c_9.V^t_{i+1,j+1}$
2-D FDTD 1,024 × 16,384 grid	$Ez^t_{i,j} = Ez^{t-1}_{i,j} - C1_{i,j}.\left(Hx^{t-\frac{1}{2}}_{i,j+\frac{1}{2}} - Hx^{t-\frac{1}{2}}_{i,j-\frac{1}{2}}\right) + C2_{i,j}.\left(Hy^{t-\frac{1}{2}}_{i+\frac{1}{2},j} - Hy^{t-\frac{1}{2}}_{i-\frac{1}{2},j}\right)$ $Hx^{t+\frac{1}{2}}_{i,j+\frac{1}{2}} = Hx^{t-\frac{1}{2}}_{i,j+\frac{1}{2}} - C3_{i,j}.\left(Ez^t_{i,j+1} - Ez^t_{i,j}\right)$ $Hy^{t+\frac{1}{2}}_{i+\frac{1}{2},j} = Hy^{t-\frac{1}{2}}_{i+\frac{1}{2},j} - C4_{i,j}.\left(Ez^t_{i+1,j} - Ez^t_{i,j}\right)$
3-D 7-point Jacobi 128 × 128 × 8,192 grid	$c_1.V^t_{i,j-1,k} + c_2.V^t_{i-1,j,k} + c_3.V^t_{i,j,k-1} + c_4.V^t_{i,j,k} + c_5.V^t_{i+1,j,k} + c_6.V^t_{i,j+1,k} + c_7.V^t_{i,j,k+1}$

and measured ALM usages. The estimation is highly accurate and gives very similar results compared to the measured ones. We achieved similar results for the other values of the design parameters and also for the other types of resources.

Fig. 10 shows the clock frequency against the logic utilization percentage. When the logic utilization is less than 75 percent, the clock frequency is around 300 MHz. From 75 percent to 85 percent of utilization, the clock frequency drops from 300 to 270 MHz. From 85 to 90 percent of utilization, the clock frequency stays around 270 MHz. Architectures of over 90 percent logic utilization are difficult to compile while maintaining a sufficient clock frequency. Therefore, we consider 90 percent logic utilization as the resource constraint. The lower and the higher limits of the clock frequency depend on the FPGA board. According to the results in Fig. 10, the difference between the frequency limits can be considered as small. Moreover, we use the assumption that, when d increases, the processing time decreases until the architecture reaches the resource limit. Since the frequency is a function of the resource usage, architectures that use a lot of resources have a low frequency. Usually, the optimal architecture requires a lot of resources to perform a large amount of parallel computations. Therefore, the optimal architecture tends to have a low frequency. Since our goal is to find the optimal architecture, we use the lower frequency limit to estimate the processing time.

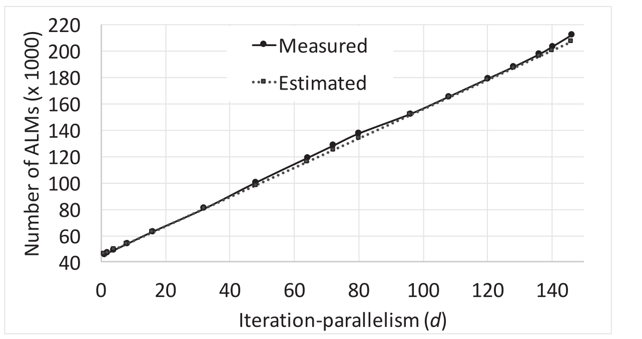


Fig. 9. Estimated and measured logic (ALM) usage for Laplace equation computation when $P_{cell} = 1$.

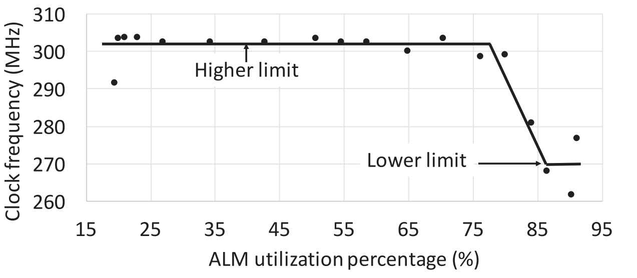
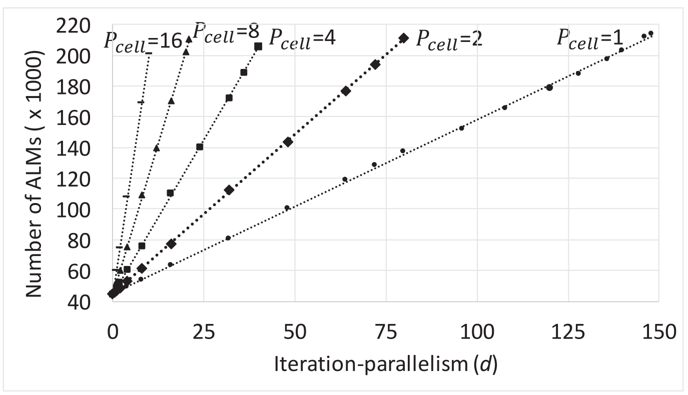
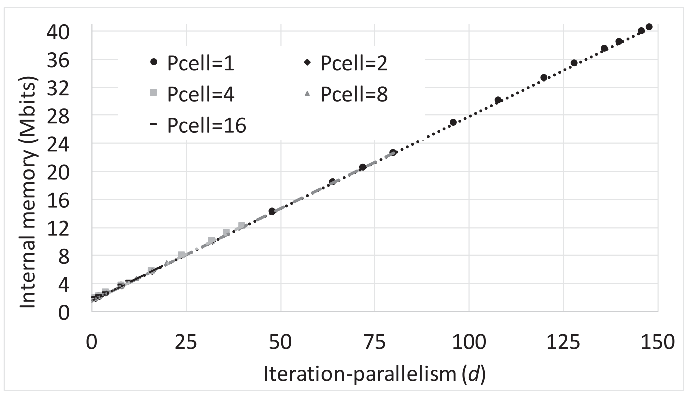


Fig. 10. Clock frequency against resource utilization percentage for DE5 board. The higher and the lower frequency limits are 300 and 270 MHz respectively.

Fig. 11 shows the estimated and the measured processing times. Figs. 11a, 11b, and 11c shows the processing times when P_{cell} is 1, 2 and 8 respectively. Those architectures belong to the scenario 1 explained in Section 4.1. Fig. 11d shows the processing time when P_{cell} is 16 and those architectures belong to the scenario 2. The processing time estimation is done using both 270 and 300 MHz frequency limits. According to the results, the measured processing time stays in between the two curves of estimated processing times. When d is large, the measured processing time gets closer to the estimated one based on 270 MHz. This shows that, it is safe to use only the lower limit of the frequency to estimate the processing time to find the optimal architecture.

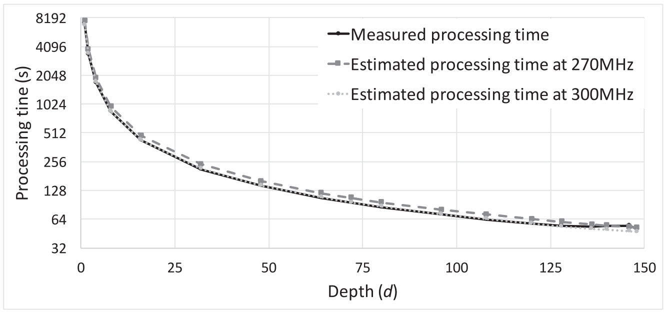


(a) Logic element (ALM) usage.

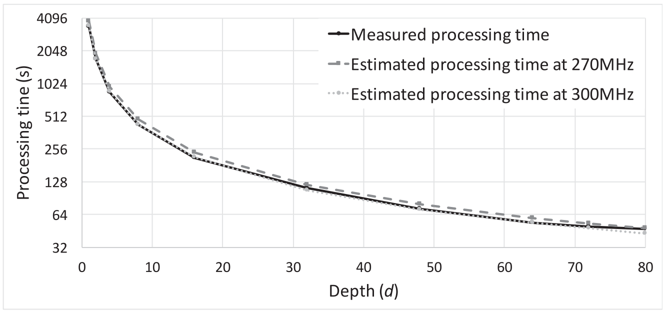


(b) Internal memory usage.

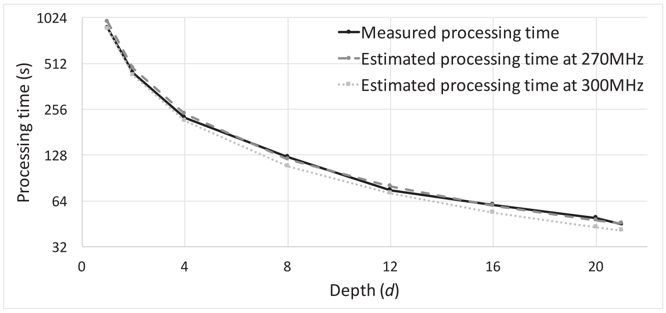
Fig. 8. Measured resource usage for Laplace equation computation using different design parameters.



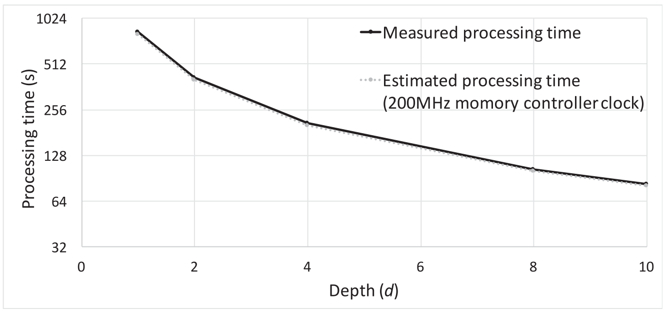
(a) $P_{cell} = 1$.



(b) $P_{cell} = 2$.



(c) $P_{cell} = 8$.



(d) $P_{cell} = 16$.

Fig. 11. Estimated and measured processing time for Laplace equation computation.

TABLE 2
Comparison of Different Architectures That Use the Same Amount of Logic Resources But Have Different Design Parameters

Design parameters		Resource usage (%)				Frequency	Processing time (s)	
P_{cell}	d	Logic elements	Internal memory	Registers	DSPs	(MHz)	Measured	Estimated
1	146	90.0	76.1	40.7	0	262	54.1	52.9
2	80	89.9	43.2	40.2	0	277	47.4	47.9
4	40	87.6	23.2	38.8	0	285	46.9	47.8
8	21	89.7	13.6	39.4	0	283	44.8	45.6
16	10	85.7	8.2	38.4	0	268	83.2	80.7

TABLE 3
Specifications of the FPGA Boards Used for the Evaluation

FPGA board	DE5	395-D8	395-AB
FPGA	5SGXEA7N2F45C2	5SGSED8N2F46C2LN	5SGXEABN2F46C2LN
ALMs (adaptive logic modules)	234,720	262,400	359,200
Registers	938,880	1,049,600	1,436,800
Internal memory	50.0 Mbits	50.1 Mbits	51.5 Mbits
DSP	256	1,963	352
Peak performance	196.0 Gflop/s	1,502.9 Gflop/s	269.5 Gflop/s
External memory frequency	DDR3 1,600 MHz	DDR3 1,066 MHz	DDR3 1,066 MHz
External memory size	4 GB	32 GB	32 GB
External memory bandwidth	25.6 GB/s	34.1 GB/s	34.1 GB/s

Table 2 shows the comparison of the architectures obtained using different design parameters. The sample application is Laplace equation and the resource constraint is 90 percent of the total ALMs (ALM is the critical resource). According to the results, the processing time changes for different design parameters. The optimal architecture is found when P_{cell} equals 4 and d equals 40. The estimated and measured processing times are very similar so that the optimal architecture can be obtained by the estimation.

5.2 Evaluation of the Optimization Methodology

We use board support packages (BSPs) of three FPGA boards, DE5 [32], 395-D8 [33] and 395-AB [34] to evaluate the optimization methodology. Table 3 shows the specifications of the boards. A large amount of DSPs is available in 395-D8 while a large amount of ALMs is available in 395-AB. The peak performances of the FPGAs are calculated according to the method given in [35].

Table 4 shows the optimal architectures found using the proposed optimization methodology for different stencil

TABLE 4
Comparison of the Design Parameters of the Optimal Architectures of Different Applications

Example	DE5				395-D8				395-AB			
	architecture		Constrained	f_{core}	architecture		Constrained	f_{core}	architecture		Constrained	f_{core}
	P_{cell}	d	resource	(MHz)	P_{cell}	d	resource	(MHz)	P_{cell}	d	resource	(MHz)
Laplace equation	8	21	ALM	283	4	44	ALM	253	8	32	ALM	248
2-D 5-points Jacobi	2	25	DSP	296	4	27	ALM	256	4	17	DSP	248
2-D 9-points Jacobi	1	28	DSP	283	4	14	ALM	234	2	19	DSP	226
2-D FDTD	1	44	Memory	291	2	30	ALM	227	2	34	Memory	247

computations. The optimal design parameters are quite different from application to application. Even for the same application, the optimal design parameters are quite different for different FPGA boards. Since the computations of the applications and the resources on the boards are different, we cannot use the same architecture for different applications or on different boards. Using the proposed optimization methodology, we can find the optimal architecture for any FPGA board and for many stencil computation applications. The clock frequency of the accelerator is f_{core} and it varies with different FPGAs.

The required external memory bandwidth (B_{req}) of the optimal architecture of each application is shown in Fig. 12. We used 395-D8 FPGA board for this evaluation. The performance of the FDTD computation is constrained by the memory bandwidth due to $B_{req} \geq B_{prac}$. In all other applications, $B_{req} < B_{prac}$ and the performances are not constrained by the memory bandwidth. Note that, d has no relationship with the memory bandwidth so that we can increase the performance by increasing d to the resource limit, even for the memory bandwidth bounded applications.

5.3 Comparison with Other Devices and Methods

Table 5 shows the processing time comparison among FPGAs, multicore CPUs and GPUs. We used two FPGA boards, DE5 and 395-D8 and the stencil computation architecture is optimized for each board. (Note that, we could not measure the performance of 395-AB FPGA board since we have only the BSP and not the actual board). All applications are optimized for CPUs and GPUs using temporal blocking. We used both non-blocking and temporal blocking methods for all applications and select the fastest implementation on each device. The performances of both FPGAs are better than those of CPUs. The difference between the performances of FPGAs and GPUs greatly varies with the application. Both 395-D8 FPGA and GTX960 GPU provide

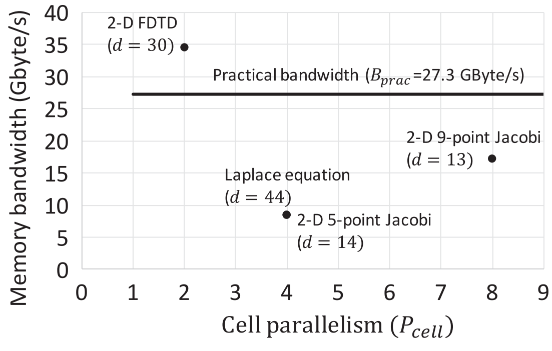


Fig. 12. Required memory bandwidth of the optimal architectures of different applications using 395-D8 FPGA board.

similar performance for 2-D nine-point Jacobi stencil. Since GPUs perform an addition and a multiplication in one clock cycle, the performance are better for the applications that have nearly 1:1 addition-to-multiplication ratio such as 2-D nine-point stencil. However, 2-D FDTD and Laplace equation have an addition-to-multiplications ratios of 2:1 and 3:1 respectively. As a result, the performances of the GPUs are not high. Unlike GPUs, FPGAs are re-configurable devices where we can design the most suitable architecture for a given application considering its operations. We can use the resources efficiently by designing computing units that do only the required operations. As a result, we can get consistent performances in FPGAs for a wide range of applications.

Fig. 13 shows the CPU and GPU performances compared to the performance limit obtained by the roofline model. We use 80 percent of the theoretical bandwidth to calculate the roofline model performance. Experimental results and previous works [11] show that it is extremely hard to obtain more than 80 percent of the theoretical bandwidth. The operational intensity is calculated by dividing the total computations in an iteration by the total data amount accessed. For example, the operational intensity of 2-D five-point Jacobi is calculated as follows:

$$\frac{4{,}048 \times 32{,}768 \times 9}{4{,}048 \times 32{,}768 \times sizeof(float) \times 2 \text{ [read \& write]}} = 1.125.$$

According to the results, we achieved around 90 percent of the roofline model performance limit for both CPUs without using any temporal or spatial blocking techniques. When the temporal blocking is used, the performances of some applications exceed the roofline limit. The Maxwell architecture based GTX960 GPU gives around 80 percent of the roofline limit without using any blocking techniques. The performances of the older Fermi and Kepler based GPUs are smaller than the newer ones. We achieved extremely good results for GTX960 and relatively good results for Kepler architecture based GTX760 by using temporal blocking. Since temporal blocking reduces the external memory access, the operational intensity is increased. As a result, a large performance improvement is obtained. According to these results, we can say that a fair comparison is done in Table 5 since we have shown optimized performance for all FPGAs, GPUs and CPUs.

Fig. 14 shows the performances of different devices measured in Gflop/s. We achieved 133~181 Gflop/s using DE5 and 143~237 Gflop/s using 395-D8. We achieved 73~228 Gflop/s in GTX960. The performances of the other GPUs are smaller than 122 Gflop/s. As shown in Table 5, the clock

TABLE 5
Comparison with GPUs and CPUs

		FPGA		GPU			Multicore CPU	
		DE5	395-D8	C2075	GTX760	GTX960	E5-1650 v3	i7-4960x
Specifications	Number of cores[1]	-	-	448	1,152	1,024	6	6
	Core clock frequency (MHz)	≈270	≈230	1,150	980	1,127	3,500	3,600
	Memory bandwidth (GB/s)	25.6	34.1	144	192	112	59.7	51.2
	Peak performance (Gflop/s)	196	1,502.9	1,030.4	2,257.9	2,308.1	672	345.6
Processing time (s)	Laplace equation	45.3	46.9	232.4	176.5	111.7	258.9	260.2
	2-D 5-point Jacobi	139.2	78.0	263.5	213.3	113.0	262.8	281.5
	2-D 9-point Jacobi	259.8	164.6	391.9	289.2	153.1	339.7	419.9
	2-D FDTD	20.5	21.6	95.1	71.6	41.3	265.1	290.7

[1] *GPU: CUDA cores, Multicore CPU: CPU cores.*

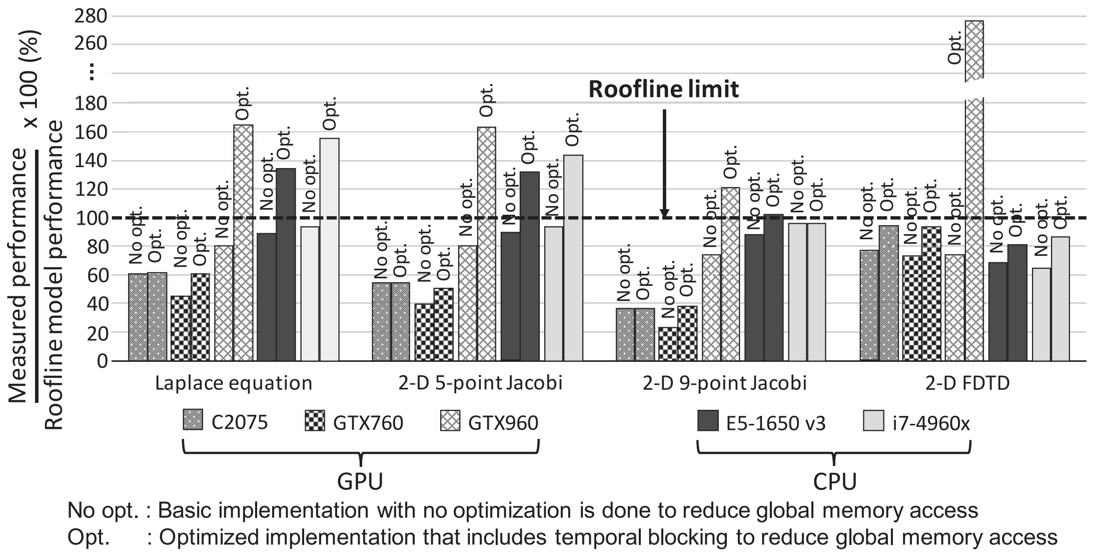


Fig. 13. GPU and CPU performances against the roofline model performance. The roofline model performance is based on 80 percent of the theoretical memory bandwidth.

frequency, the external memory bandwidth and the peak performance of FPGAs are many times smaller compared to those of GPUs. However, the performances are better than or equals to GPUs due to the efficient implementation of the iteration-parallel computation.

Fig. 15 shows the "effective to peak performance ratio (EPR)" given by Eq. (12). Although GPUs have a very high peak performance, less than 10 percent is utilized by the stencil computation. Compared to that, FPGA in the DE5 board uses over 68 percent of its peak performance. Such a high utilization is achieved by designing the most suitable architecture for a given application considering its operations. However, 395-D8 FPGA provides less than 15 percent of its peak performance. The peak performance of 395-D8 is large due to the large amount of DSP units. However, DSPs can be used only for multiplications, and we need more multiplications in an application to extract the full potential of 395-D8

$$EPR = \frac{\text{Actual performance}}{\text{Peak performance}} \times 100\%. \quad (12)$$

Table 6 compares the proposed method with the work in [11]. It is one of the latest and very comprehensive work

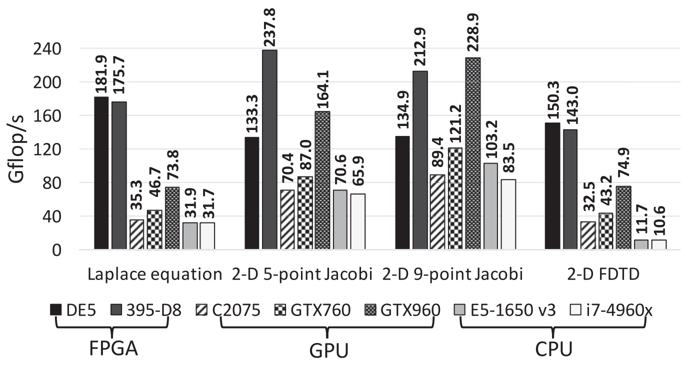


Fig. 14. Comparison of the performance of FPGAs, CPUs and GPUs in Gflop/s.

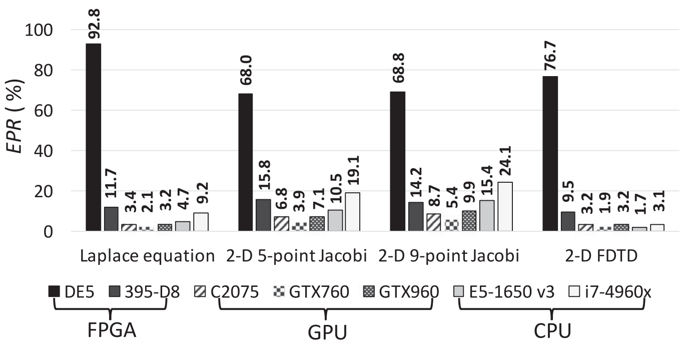


Fig. 15. Comparison of EPR.

TABLE 6
Comparison with GPUs Using 3-D Seven-Point Jacobi Stencil in Single-Precision Floating Point

	Device	This paper (FPGA)		Work in [11] (GPU)	
		DE5	395-D8	M2075	K20X
Specifications	Memory bandwidth (GB/s)	25.6	34.1	150.0	250.0
	Peak performance (Gflop/s)	196	1,502.9	1,030.2	3,950.0
Measured results	Performance (Gflop/s)	111.3	193.3	131.4	287.1
	EPR (%)	59.3	12.8	12.7	7.2
	performance/roofline × 100 (%) [1]	-	-	63.4	83.1

[1] *Roofline model performance is calculated under the assumption that the achievable memory bandwidth is smaller than 80 percent of the theoretical maximum.*

done on stencil computation acceleration using GPUs. We evaluate the performance of 395-D8 FPGA using 3-D seven-point Jacobi stencil in single-precision. The same stencil computation is used in [11]. We achieved 192 Gflop/s in FPGA while [11] achieved 131.4 and 287.1 Gflop/s using M2075 and K20X GPUs respectively. The performance of 395-D8 FPGA is 1.4 times larger compared to M2075 GPU. However, the performance is 1.4 times smaller compared to K20X. Note that, K20X is a high-end GPU, where the external memory bandwidth and peak performance are 7.3 and 2.6 times larger compared to 395-D8. Therefore, the better performance in K20X is understandable. However, we could expect significantly improved performance using next generation Stratix 10 FPGAs where the amount of resources are more than 10 times larger compared to the current generation Stratix V FPGAs.

Table 7 shows the performance of 3-D stencil computation using 3-D seven-point Jacobi stencil. In 3-D stencils, we have to store the data of a few planes instead of a few lines in the case of 2-D stencils. As a result, the register and the internal memory requirements could be larger than those for the 2-D stencils. Even for the same grid size, the amount of resources required depends on the smallest plane. For a $128 \times 128 \times 8{,}192$ grid, the smallest plane is only 128×128 large. Therefore, the internal memory is not a bottleneck and we achieved over 193 Gflop/s. When the smallest plane is 256×128, more memory is required. As a result, the frequency is reduced due to the large resource usage. When the plane size is further increased to 256×256, the internal memory becomes a bottleneck. Therefore, we have to reduce the number of PCMs in order to fit the architecture on to the FPGA board. Since the number of PCMs corresponds to the degree of parallelism, the processing speed decreases drastically. Using multiple FPGAs in a deep pipeline may solve this problem. This method has been demonstrated using HDL-based manual designs in [13] and shows a near linear performance increase against the number of FPGA boards. In OpenCL, it is possible to use the channel attribute to manage the connections among multiple FPGAs. The SFP+ ports that have over 100 Gbps bandwidth can be used for inter FPGA data transfers. Such methods should be considered in future works.

Table 8 shows the performance of double precision stencil computation. The older GTX580 GPU used in [36] gives better performance compared to the FPGAs. Double-precision performance in FPGA is less than 25 percent of the single-precision performance. The current generation FPGAs (Stratix V series), does not contain dedicated floating-point units. Multiplications are done in DSPs and the additions are done using ALMs. Due to the large ALM requirement for double precision computation, the performances are reduced. However, the next generation FPGAs such as Stratix 10 and Aria 10 devices contain dedicated floating-point units and that will significantly reduce the ALM requirement and increase the processing speed.

The GPU optimization considers many different parameters such as shared memory, warp size, external memory bandwidth, cache size, registers, threads per block, etc. Therefore, the programmer's skill and experience are required for a good implementation. Similarly, writing the optimized OpenCL code for FPGAs also depends on the programmer. However, if the applications are restricted to stencil computation, writing the optimized code is very easy using the proposed method. In fact, we used almost the same kernel and host codes on different FPGA boards in this paper. The only difference is the two design parameters, d and P_{cell} which are defined as constants in a header file. Moreover, even for different applications, the changes done to the kernel code is minimum such as changing the computation equation and writing the boundary conditions. Therefore, if we use the proposed method, the programmer's responsibility is minimum and anyone can find the optimal architecture for any OpenCL capable FPGA board.

TABLE 7
Performance of the 3-D Stencil Computations Using 395-D8 FPGA

Grid size	Number of PCMs	Bottleneck resource	Frequency (MHz)	Performance (Gflop/s)
$128 \times 128 \times 8{,}192$	9	ALM	230.3	193.3
$256 \times 128 \times 4{,}096$	9	ALM	196.2	167.7
$256 \times 256 \times 2{,}024$	6	Memory	180.1	111.5

TABLE 8
Comparison with a GPU Using Double-Precision Stencil Computation

Application	This paper (FPGA)		Work in [36] (GPU)
	DE5	395-D8	GTX580
3-D 7-point Jacobi	27.2	40.7	50.0
2-D 5-point Jacobi	27.3	40.9	49.5

TABLE 9
Performance of Other FPGA-Based Approaches Using 3-D Stencils in Single Precision

Previous work	FPGA	Performance
Work in [37] (3-D Jacobi using manual design)	Virtex-4 XC4VLX200	103 Gflop/s
Work in [38] (3-D FDTD using MaxCompiler)	Virtex-6 XC6VSX475T	19.5 Gflop/s

TABLE 10
Comparison with the Work in [13] Using 2-D Four-Point Jacobi Stencil Computation

	FPGA board	Performance (Gflop/s)	*EPR* (%)
Work in [13]	DE3 (Stratix III)	28.94	87.5
This paper	DE5 (Stratix V)	119.82	61.1

Table 9 shows the performances of the other recent FPGA based approaches. A custom accelerator is designed using Verilog HDL in [37] while the MaxCompiler (a high level design tool using a C-like language) is used in [38]. The performance are 103 and 19.5 Gflop/s respectively for single-precision computations. Compared to those, we achieved higher performances upto 193 Gflop/s for 3-D and upto 212 Gflop/s for 2-D stencils. Note that, [38] has given the performance in number of cells computed per second and we convert it to Gflop/s by multiplying it with the number of computations per a cell.

Table 10 compares our method with [13] that proposes a custom accelerator designed using HDL. We achieved 61.1 percent *EPR* compared to 87.5 percent in [13]. However, we believe that this *EPR* is sufficient for a software-based design where the design time is only a few hours. Compared to that, HDL-based design flow usually requires many months of design time. The higher Gflop/s value in the proposed method is quite obvious since we used a larger and advanced FPGA.

6 Conclusion

We proposed an OpenCL-based FPGA-platform for stencil computation. The proposed FPGA-platform reduces the design, testing and debugging times significantly compared to custom HDL-based accelerator design. The same program code can be reused by recompiling it for any OpenCL capable FPGA board, irrespective of the FPGA type or I/O resources such as different external memory specifications. The proposed architecture is designed to utilize iteration-parallelism instead of cell-parallelism to minimize the external memory access. It contains deep pipelines to carry the computation results between iterations. We achieved maximum of 14.1 and 5.1 times larger processing speeds compared to CPU and GPU implementations respectively. We also achieved 119 $\sim$ 237 Gflop/s of performances.

Acknowledgments

This work is supported by MEXT KAKENHI Grant Number 15K15958.

References

[1] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner, “Compiling stencils in high performance fortran,” in *Proc. ACM/IEEE Conf. Supercomputing*, 1997, pp. 1–20.
[2] G. Karniadakis and S. Sherwin, *Spectral/hp Element Methods for Computational Fluid Dynamics*. London, U.K.: Oxford Univ. Press, 2013.
[3] K. S. Yee, “Numerical solution of initial boundary value problems involving Maxwell’s equations in isotropic media,” *IEEE Trans. Antennas Propag.*, vol. 14, no. 3, pp. 302–307, May 1966.
[4] W. M. Kahan, “Gauss-Seidel methods of solving large systems of linear equations,” Ph.D. dissertation, Univ. Toronto, Toronto, ON, 1958.
[5] L. A. Hageman and D. M. Young, *Applied Iterative Methods*. North Chelmsford, MA, USA: Courier Corporation, 2012.
[6] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.
[7] S. M. F. Rahman, Q. Yi, and A. Qasem, “Understanding stencil code performance on multicore architectures,” in *Proc. 8th ACM Int. Conf. Comput. Frontiers*, 2011, pp. 30:1–30:10.
[8] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes, “Multicore-optimized wavefront diamond blocking for optimizing stencil updates,” *SIAM J. Sci. Comput.*, vol. 37, no. 4, pp. C439–C464, 2015.
[9] J. Meng and K. Skadron, “A performance study for iterative stencil loops on GPUs with ghost zone optimizations,” *Int. J. Parallel Program.*, vol. 39, no. 1, pp. 115–142, 2011.
[10] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, “Split tiling for GPUs: Automatic parallelization using trapezoidal tiles,” in *Proc. 6th Workshop Gen. Purpose Processor Using Graph. Process. Units*, 2013, pp. 24–31.
[11] N. Maruyama and T. Aoki, “Optimizing stencil computations for NVIDIA Kepler GPUs,” in *Proc. 1st Int. Workshop High-Performance Stencil Comput.*, 2014, pp. 89–95.
[12] W. Luzhou, K. Sano, and S. Yamamoto, “Domain-specific language and compiler for stencil computation on FPGA-based systolic computational-memory array,” in *Reconfigurable Computing: Architectures, Tools and Applications*. Berlin, Germany: Springer, 2012, pp. 26–39.
[13] K. Sano, Y. Hatsuda, and S. Yamamoto, “Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 695–705, Mar. 2014.
[14] T. S. Czajkowski, et al., “OpenCL for FPGAs: Prototyping a compiler,” in *Proc. Int. Conf. Eng. Reconfigurable Syst. Algorithms*, 2012, Art. no. 1.
[15] S. Tatsumi, M. Hariyama, M. Miura, K. Ito, and T. Aoki, “OpenCL-based design of an FPGA accelerator for phase-based correspondence matching,” in *Proc. Int. Conf. Parallel Distrib. Process. Techn. Appl.*, 2015, pp. 90–95.
[16] N. Suda, et al., “Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks,” in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 16–25.
[17] Y. Takei, H. M. Waidyasooriya, M. Hariyama, and M. Kameyama, “FPGA-oriented design of an FDTD accelerator based on overlapped tiling,” in *Proc. Int. Conf. Parallel Distributed Process. Techn. Appl.*, 2015, pp. 72–77.
[18] H. M. Waidyasooriya and M. Hariyama, “FPGA-based deep-pipelined architecture for FDTD acceleration using OpenCL,” in *Proc. 15th IEEE/ACIS Int. Conf. Comput. Inform. Sci.*, 2015, pp. 109–114.
[19] J. M. Cecilia, J. M. García, and M. Ujaldón, “CUDA 2D stencil computations for the Jacobi method,” in *Proc. 10th Int. Conf. Appl. Parallel Sci. Comput.*, 2010, pp. 173–183.
[20] J. Guo, G. Bikshandi, B. B. Fraguela, and D. Padua, “Writing productive stencil codes with overlapped tiling,” *Concurrency Comput. Practice Experience*, vol. 21, no. 1, pp. 25–39, 2009.
[21] V. Bandishti, I. Pananilath, and U. Bondhugula, “Tiling stencil computations to maximize parallelism,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 40:1–40:11.
[22] D. Wonnacott, “Achieving scalable locality with time skewing,” *Int. J. Parallel Program.*, vol. 30, no. 3, pp. 181–221, 2002.
[23] R. Strzodka, M. Shaheen, D. Pajak, and H. P. Seidel, “Cache accurate time skewing in iterative stencil computations,” in *Proc. Int. Conf. Parallel Process.*, 2011, pp. 571–581.

[24] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, “Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization,” in *Proc. 33rd Annu. IEEE Int. Comput. Softw. Appl. Conf.*, 2009, vol. 1, pp. 579–586.

[25] M. Wittmann, G. Hager, and G. Wellein, “Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory,” in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops PhD Forum*, 2010, pp. 1–7.

[26] A. A. Nacci, V. Rana, F. Bruschi, D. Sciuto, I. Beretta, and D. Atienza, “A high-level synthesis flow for the implementation of iterative stencil loop algorithms on FPGA devices,” in *Proc. 50th Annu. Des. Autom. Conf.*, 2013, pp. 52:1–52:6.

[27] G. Deest, N. Estibals, T. Yuki, S. Derrien, and S. Rajopadhye, “Towards scalable and efficient FPGA stencil accelerators,” in *Proc. 6th Int. Workshop Polyhedral Compilation Techn.*, 2016, pp. 1–11.

[28] K. Dohi, K. Okina, R. Soejima, Y. Shibata, and K. Oguri, “Performance modeling of stencil computing on a stream-based FPGA accelerator for efficient design space exploration,” *IEICE Trans. Inf. Syst.*, vol. E98-D, no. 2, pp. 298–308, 2015.

[29] MaxCompiler. (2016). [Online]. Available: https://www.maxeler.com/products/software/maxcompiler/

[30] The open standard for parallel programming of heterogeneous systems, (2015). [Online]. Available: https://www.khronos.org/opencl/

[31] Altera SDK for OpenCL, (2016). [Online]. Available: https://www.altera.com/products/design-software/embedded-software-devel

[32] Altera development and education boards. (2016). [Online]. Available: https://www.altera.com/support/training/university/boards.html#de5

[33] Nallatech 395-with stratix V D8. (2016). [Online]. Available: http://www.nallatech.com/store/uncategorized/395-d8/

[34] Nallatech 395-AB-with stratix V AB. (2016). [Online]. Available: http://www.nallatech.com/store/pcie-accelerator-cards/395-ab/

[35] Achieving one TeraFLOPS with 28-nm FPGAs, 2010. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/zh_CN/pdfs/literat

[36] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, “High-performance code generation for stencil computations on GPU architectures,” in *Proc. 26th ACM Int. Conf. Supercomputing*, 2012, pp. 311–320.

[37] M. Shafiq, M. Perics, R. de la Cruz, M. Araya-Polo, N. Navarro, and E. Ayguadé, “Exploiting memory customization in FPGA for 3D stencil computations,” in *Proc. Int. Conf. Field-Programmable Technol.*, 2009, pp. 38–45.

[38] K. Okina, R. Soejima, K. Fukumoto, Y. Shibata, and K. Oguri, “Power performance profiling of 3-D stencil computation on an FPGA accelerator for efficient pipeline optimization,” *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 4, pp. 9–14, 2015.

Hasitha Muthumala Waidyasooriya received the BE degree in information engineering, the MS degree in information sciences, and the PhD degree in information sciences from Tohoku University, Japan, in 2006, 2008, and 2010 respectively. He is currently an assistant professor in the Graduate School of Information Sciences, Tohoku University. His research interests include reconfigurable computing, processor architectures for big-data applications, and high-level design methodology for VLSIs. He is a member of the IEEE.

Yasuhiro Takei received the BE degree in electronic engineering, the MS degree in information sciences, and the PhD degree in information sciences from Tohoku University, Japan, in 2011, 2013, and 2016 respectively. His research interest include heterogeneous multicore processor architectures.

Shunsuke Tatsumi received the BE degree in information engineering and the MS degree in information sciences from Tohoku University, Japan, in 2012 and 2014, respectively. He is currently working toward the PhD degree in the Graduate School of Information Sciences, Tohoku University. His research interests include reconfigurable computing and image processing.

Masanori Hariyama received the BE degree in electronic engineering, the MS degree in information sciences, and the PhD degree in information sciences from Tohoku University, Japan, in 1992, 1994, and 1997, respectively. He is currently a professor in the Graduate School of Information Sciences, Tohoku University. His research interests include real-world applications such as robotics and medical applications, big data applications such as bio-informatics, high-performance computing, VLSI computing for real-world application, high-level design methodology for VLSIs, and reconfigurable computing. He is a member of the IEEE.