Automatic Optimization of OpenCL-Based Stencil Codes for FPGAs and Its Evaluation

Tsukasa Endo, Hasitha Muthumala Waidyasooriya,
Masanori Hariyama

Abstract

Recently, C-based OpenCL design environment is proposed to design FPGA (field programmable gate array) accelerators. Although many C-programs can be executed on FPGAs, the best c-code for a CPU may not be the most appropriate one for an FPGA. Users must have some knowledge about computer architecture in order to write a good OpenCL code. To solve this problem, we propose an automatic optimization method. We accurately predict the kernel performance using the log files generated at the initial stage of the compilation. Then we find the optimized FPGA architecture by searching all possible design parameters. We implement the proposed method to find the optimized architecture for stencil computation. According to the results, the design time has been reduced to 4% $\sim$ 8% of the conventional approach.

Keywords: OpenCL for FPGA, performance tuning, stencil computation, code optimization.

1 Introduction

FPGAs (field programmable gate arrays) are reconfigurable devices, where the user can change the architecture by a program. Usually, FPGAs are programmed using hardware description languages (HDL) such as Verilog [1] or VHDL [2]. However, this process is very time consuming and the programmer must have an extensive knowledge about the hardware design. Clock cycle-based simulations, timing and critical path analysis are required to design an FPGA accelerator. In addition, users have to design hardware for I/O controllers. Moreover, users also have to write device drivers and software in order to communicate and transfer data between an FPGA and a host CPU.

Recently, OpenCL for FPGA [3] is introduced to solve these problems. An FPGA accelerator can be designed by writing an OpenCL kernel in C-language [4]. The same kernel can be executed on different FPGA boards. Although writing an OpenCL kernel is easy, designing the optimal architecture is difficult. Works in [5, 6] considers several techniques such as using shift-registers, loop-unrolling, vectorization, using constant memory, using multiple compute units, etc to increase the processing speed of the OpenCL-based designs. However, it is difficult to determine the best technique for a particular FPGA board and

an application. Since there are many FPGA boards with different amount of resources and memory bandwidths, we may have to compile many different kernels by applying various combinations of different techniques. Since the compilation time is very large, the kernel design time could increase significantly. Moreover, neither of these works consider an optimization problem. Therefore, it is difficult to know whether the designed kernel is optimal.

The OpenCL-based design method can be divided into two stages: OpenCL to HDL code generation (stage 1) and HDL to FPGA bit-stream generation (stage 2). In our previous work in [7], we proposed a method to find a near-optimal architecture based on the predicted performance in stage 1. As a result, we can significantly reduce the design time in stage 2. In this paper, we extend our work to further reduce the design time by applying a binary search approach instead of the exhaustive search used in [7]. Moreover, we optimize the compiler options to increase the processing speed further. We provide a comprehensive evaluation of the proposed method by applying it for stencil computation kernels. We achieved the optimized architecture in 4% $\sim$ 8% of the design time compared to conventional methods.

2 OpenCL-based FPGA accelerator design

The conventional OpenCL-based FPGA accelerator design-flow is shown in Figure 1. It can be divided into three main phases. The emulation phase starts with a kernel program written in OpenCL. We can emulate the behavior of the kernel on a CPU to find whether the desired outputs are achieved. The next phase is the estimation phase. We compile the OpenCL kernel to get an HDL code, and we call it the “intermediate compilation”. After the intermediate compilation, offline compiler provides an estimated resource usage report. Usually, there is a small difference between the estimated resource usage and the actual resource usage. If the resource usage is acceptable we can proceed to the last phase, which is the performance tuning phase. We compile the HDL code to generate a bit-stream that is executable on an FPGA. We call this the “full compilation”. The full compilation takes many hours of compilation time, since it involves time consuming processes such as placement, routing, etc. After the full compilation, we get the actual area usage and the frequency information. Using those, we can determine whether the performance is acceptable or not. If the performance is not acceptable, we can identify the bottlenecks by recompiling the kernel for profiling. Then we can re-write the kernel to avoid the bottlenecks, and this whole process repeats again from the emulation phase.

As explained above, there is no concrete way for the user to know that the performance is optimal or even close to that. The profiling provides limited information about the bottlenecks of the kernel code. However, a different code could produce completely different performance. Therefore, the performance usually depends on the skill and the experience of the designer. Moreover, changing one part of the code could worsen the performance of the other parts. Therefore, it could be difficult to find how and which part should be corrected in order to increase the performance. In addition, the compilation takes many hours of processing time. If the compilation is done many times by re-writing the code, the accelerator design time could be increased significantly.

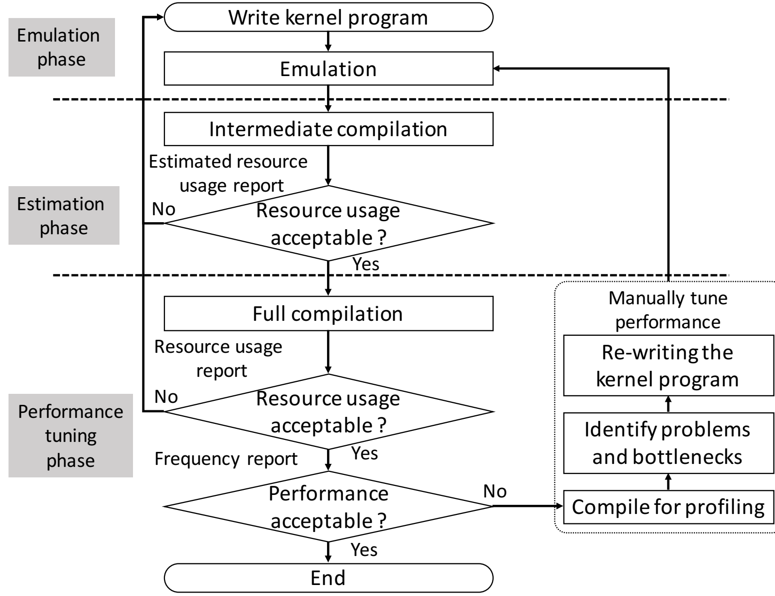


Figure 1: Conventional OpenCL-based FPGA accelerator design flow.

3 Automatic optimization methodology

3.1 Performance prediction of the OpenCL codes

In OpenCL for FPGA, There are two types of OpenCL kernels, single work-item kernels and NDRange kernels. The OpenCL codes of the NDRange kernels are quite similar to GPU kernels, and multiple work-items (corresponds to the threads in GPUs) are executed in a pipelined manner in an FPGA. The single work-item kernels follow a natural coding style similar to a typical c-program, where the loop-iterations are processed in a pipeline manner. In this paper, we consider only the single work-item kernels. However, similar approach could be used for NDRange kernels in future works. Generally, OpenCL code of a single work-item kernel consists of multiple loops. Listing 1 shows an example of an OpenCL kernel code. It has a hierarchical loop structure, where the outer loop contains multiple inner loops. After doing the first stage compilation, we get a log file that contain information such as the loop number, "initiation interval (II)", pipelining details, resource usage estimation, etc. Note that, II stands for the number of clock cycles between outputs. For example, if $II = 1$, an output is produced in every clock cycle.

We analyze the log file and the kernel code file to generate a block structure, where each block corresponds to a loop in the kernel code. Figure 2 shows such a block structure generated for the code in Listing 1. Each level in the block structure is associated with the loop hierarchy of the kernel code. A block is associated with a loop. A block contains information such as pipeline details, the number of loop-iterations, the number of clock cycles, etc. The number of clock cycles required by the loop is shown by BI in a block. It is defined by $BI = II \times$ iterations. Block information is used to predict the kernel performance.

```
__kernel void sample ( __global const float * restrict A,
                       __global const float * restrict B,
                       __global float * restrict C )
{
  float a, b;

  for(int i = 0; i < N; i++) {
    for(int j = 0; j < N; j++) {
      a += A[j];
    }

    for(int k = 0; k < M; k++) {
      b *= B[k];
    }

    C[i] = a + b;
  }
}
```

Listing 1: Loop structure of an OpenCL kernel

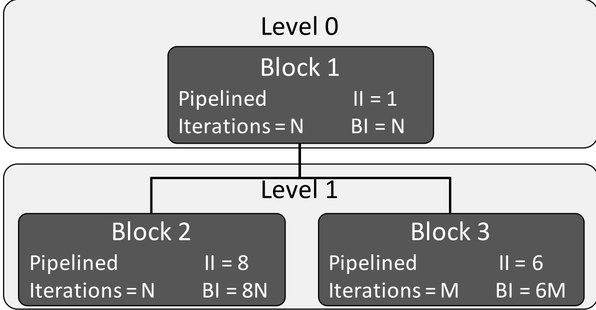


Figure 2: The block structure corresponds to the kernel in Listing 1.

If the parent block m of level t is denoted by p^t_m and its child blocks at level $t+1$ are denoted by $c^{t+1}_1 ... c^{t+1}_n$, the number of clock cycles required to execute the parent block ($p^t_m[cycle]$) is given by Eq.(1). Note that, $c^{t+1}_n[BI]$ is the number of clock cycles required by the loop of the child block c^{t+1}_n. When the parent block is pipelined, child blocks are executed in parallel so that the number of clock cycles depend on the largest *BI* of the child blocks. When the parent block is not pipelined, child blocks are executed in one by one so that the number of clock cycles depend on the sum of *BI* of all child nodes. This process is continued for all parents from the bottom level to the top level (level 0). The total number of clock cycles equals to the sum of the number of clock cycles in the parent nodes of level 0.

$$p^t_m[cycle] = \begin{cases} max\left\{c^{t+1}_1[BI], ..., c^{t+1}_n[BI]\right\} \times p^t_m[BI] & \text{if } p^t_m \text{ is pipelined} \\ \sum^n_{k=1} c^{t+1}_k[BI] \times p^t_m[BI] & \text{if } p^t_m \text{ is not pipelined} \end{cases} \quad (1)$$

For example, the number of clock cycles of the kernel code in Listing 1 is given by follows, assuming $N = 100$ and $M = 200$.

$$max\left\{8 \times 100, 6 \times 200\right\} \times 100 = 120,000$$

Using this method, we can predict the number of clock cycles of any OpenCL kernel code. If we assume the clock frequency is constant, we can compare the number of clock cycles

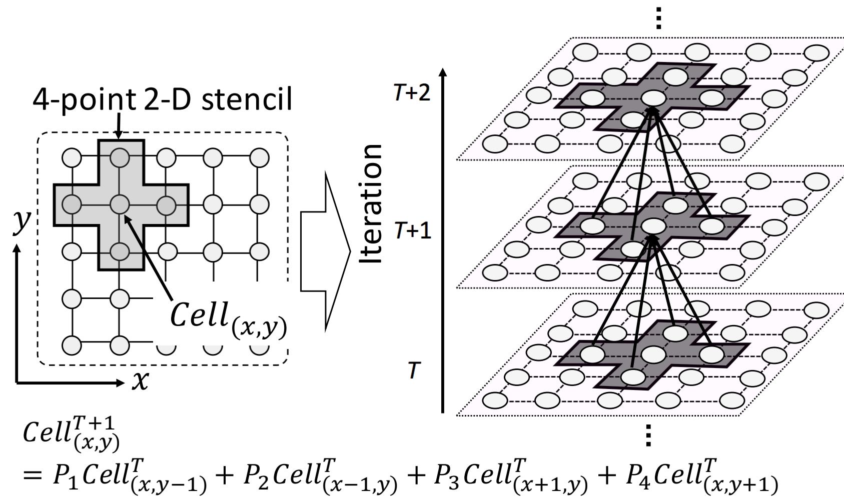


Figure 3: Stencil computation using a 2-D 4-point stencil.

required by different codes and select the one with the minimum number of cycles as the best one.

3.2 Stencil computation kernels

Stencil computation [8] is an iterative computation method used in many fields such as fluid dynamics [9], electromagnetic simulations [10], etc. It is a well studied problem and its FPGA oriented architectures have been proposed in many previous works such as [11, 12, 13]. However, there are many different stencil computation applications and many different FPGA boards, so that finding the optimal architecture for a given application and an FPGA board is a difficult and time consuming problem.

We explain the stencil computation architecture briefly. Figure 3 shows the computation of a 4-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. Figure 4 shows the stencil computation architecture proposed in previous work [13]. It has multiple pipelined computation modules (PCMs) where each PCM processes one iteration. One stencil computation is done in one PE and multiple PEs are used to compute multiple stencils in parallel. Shift registers are used to carry forward the result of one iteration to the next PCM that computes the next iteration.

Listing 2 shows a stencil computation kernel written in OpenCL. The number of PEs in a PCM and the number of PCMs are denoted by *nPE* and *nPCM* respectively. Those are the two design parameters of the stencil computation architecture. Increasing *nPE* requires multiple computations done in parallel. As a result, more resources and more data are required. Since more data are required in a clock cycle, the required bandwidth is increased. Increasing *nPCM* increases the amount of resources.

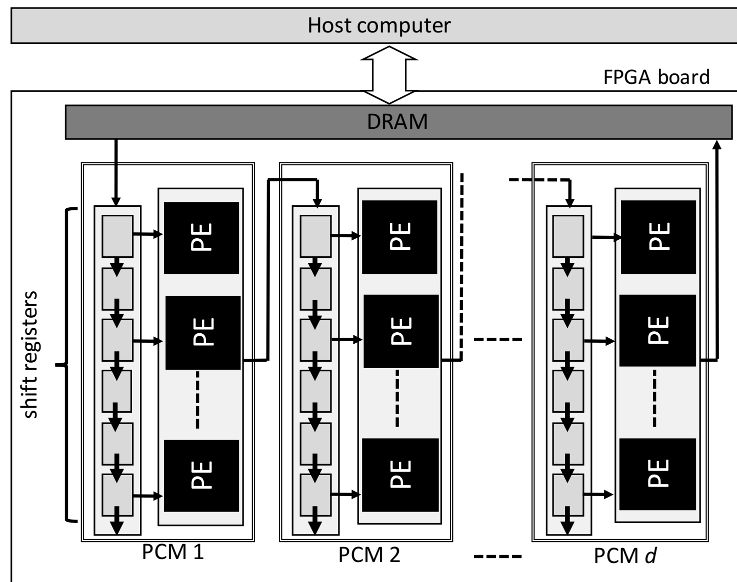


Figure 4: Stencil computation architecture proposed in [13].

```
#define N ((HEIGHT*WIDTH +nPCM*(WIDTH+2))/nPE)

__kernel void stencil( global float * restrict frame_in,
                       global float * restrict frame_out )
{
  float rows[nPCM][...];

  //manually unroll loop by nPE
  for (int count=0, count<N, count++)
  {
    #pragma unroll
    for (int i+..; i>0; --i) {
      #pragma unroll
      for (int j=0; j<nPCM; j++ ) {
        rows[j][i] = rows[j][i-1];
      }
    }
    rows[0][0] = frame_in[count];

    #pragma unroll nPCM
    for (int j=0; j<nPCM-1; j++) {
      //computation
      rows[j+1][0] = ...;
    }

    //computation of the final iteration
    frame_out[...] = ...;
  }
}
```

Listing 2: Stencil computation kernel

3.3 Automatic optimization for Stencil codes

We consider the following optimization problem to find the OpenCL kernel with the minimum processing time.

Table 1: Resource constraints.

Constraints
85% of the total logic modules
90% of the total registers
80% of the total memory blocks
100% of the total DSPs
90% of the global memory bandwidth

Objective function :

Minimization of the total number of clock cycles.

Constraints :

i. Resource utilization.

ii. Global memory bandwidth.

Freedom :

i. Number of PCMs (*nPCM*).

ii. Number of PEs per a PCM (*nPE*).

The objective function is computed according to Eq.(1). As shown in Listing 2, there are multiple loops in the stencil computation code. However, all the inner loops are completely unrolled, so that only the outer loop is considered to determine the number of clock cycles. Therefore, the number of clock cycles are "$II \times$ the number of loop iterations". Note that, to increase the number of PEs, we manually unroll the outer loop by a factor of *nPE*. This reduces the number of clock cycles by a factor of *nPE* as shown in the first line of the stencil code. If the clock frequency is a constant, the number of clock cycles is relative to the processing time.

To compute the resource utilization, we consider all major resources of the FPGA, such as logic blocks, memory blocks, DSPs, etc. Different FPGAs contain different amount of resources. However, it may not be possible to achieve 100% utilization of all resources, due to placement difficulties on an FPGA. Moreover, routing becomes difficult for large designs and the clock frequency could drop considerably. Therefore, we often use less than 100% utilization of resources in order to predict performance accurately. The resource constraints we used in this paper are shown in Table 1. It is possible to achieve this resource utilization without compromising on the clock frequency. According to our experience in many works such as [13, 14, 15], large accelerators use around 80% of the FPGA resources. Therefore, we can say that such large accelerators can be implemented, while satisfying the resource constraint.

Figure 5 shows the proposed automatic optimization methodology. It has two stages. In stage 1, the performances of the stencil computation kernels with different design parameters are predicted. For each *nPE* value, we search for the optimum *nPCM* that provides the smallest number of clock cycles. This process is done automatically using a python-based program. According to the findings in [13], if *nPE* is a constant, the largest *nPCM* value provides the best performance. This is because, the clock frequency remains the same for

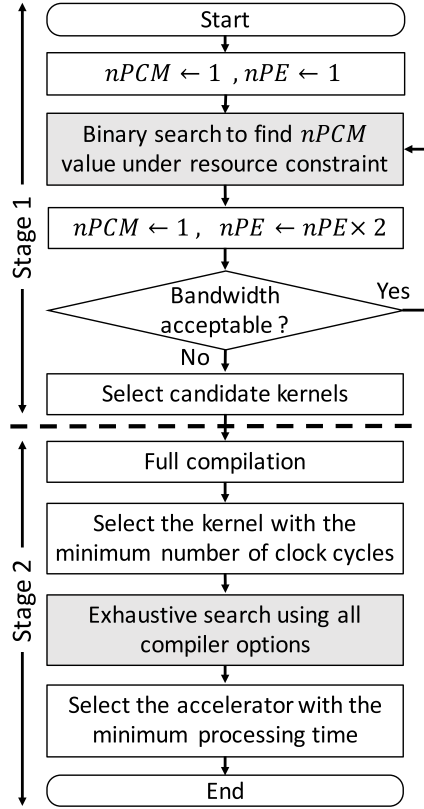


Figure 5: The proposed automatic optimization methodology.

different *nPCM* values. However, we cannot say which *nPE* value would provide the best accelerator architecture. This is because, slightly changing *nPE* can have a significant affect on *nPCM* and also the length of the pipeline. For example, if we reduce the *nPE* value by half, we can double the *nPCM* value, while maintaining the same degree of parallelism. According to our practical experience, offline compiler may not be able to maintain a high clock frequency for very large pipelines. Therefore, accelerators with different *nPE* could have different clock frequencies. Without knowing those clock frequencies, we may not be able to compare the performance of different accelerators. After stage 1, we will get a few candidate kernels where one of those is the optimal one. They have different *nPE* values and for each *nPE*, *nPCM* is optimized. In stage 2, we perform the full compilation on all candidate kernels. After that, we can find the best one by evaluation the processing time of each. Since only a few kernels are compiled in stage 2, the design time is small. After the best kernel is selected, we reduce its processing time further by optimizing the compiler options.

In stage 1, we use a binary search method, compared to the exhaustive search used in our previous work [7]. The binary search method is shown in Figure 6. At the beginning, we use $nPCM = 1$ and do the intermediate compilation. If it meets the resource constraints, we estimate the performance. Then we double the *nPCM*s and do the intermediate compilation again. If a kernel violate the resource constraints, we reduce the *nPCM*s. If there are n kernels, the exhaustive search requires n compilations, while binary search requires only $log(n)$ compilations. As a result, compilation time in stage 1 can be reduced significantly.

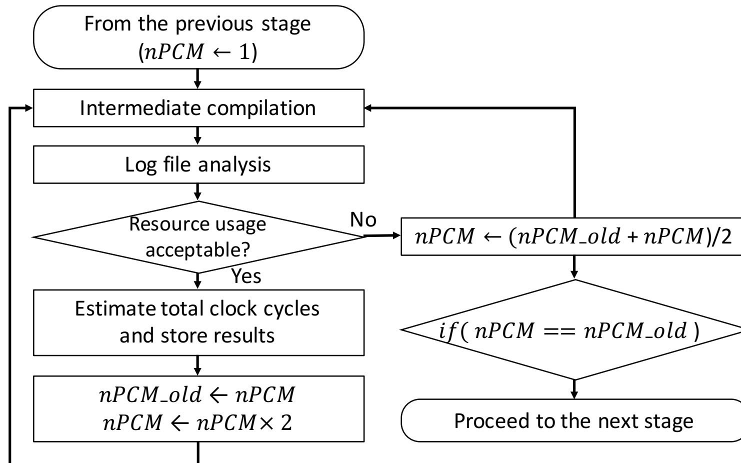


Figure 6: Binary search.

Table 2: Resources of the FPGA boards used for the evaluation.

Board name	DE5	DE5a
FPGA	5SGXEA7N2F45C2	10AX115N2F45E1SG
Logic modules	234,720	427,200
Internal memory	50.00 Mbits	65.6 Mbits
Memory blocks	2560	2713
DSP	256	1518
Theoretical bandwidth	25.6GB/s	25.6GB/s

To improve the performance further, we can optimize the compiler options. Compiler options determine how the kernel is compiled. Some examples of compiler options are, “non-interleaving”, “fpc” and “fp-relaxed”. Using the option “non-interleaving” generates an accelerator that does not use memory interleaving. Using the option “fpc” allows fused floating-point operations. The option “fp-relaxed” is used to rearrange the computation order. Choosing the most efficient compiler options reduces the resource utilization and increases the clock frequency. Figure 7 shows how to optimize compiler options. We select the kernel with the smallest processing time, and re-compile it after changing the compiler options. We try all combinations of compiler options and select the accelerator with the minimum processing time. Since the re-compilation is done for a single kernel, the compilation time is not large.

4 Evaluation

For the evaluation, we use two FPGA boards, DE5 [17] and DE5a [18]. The resources of the FPGA boards are shown in Table.2. FPGAs are configured using Intel FPGA SDK for OpenCL 16.1 [16]. Each example uses a $4096 \times 32,768$ grid. Computations are done for 1,632 iterations.

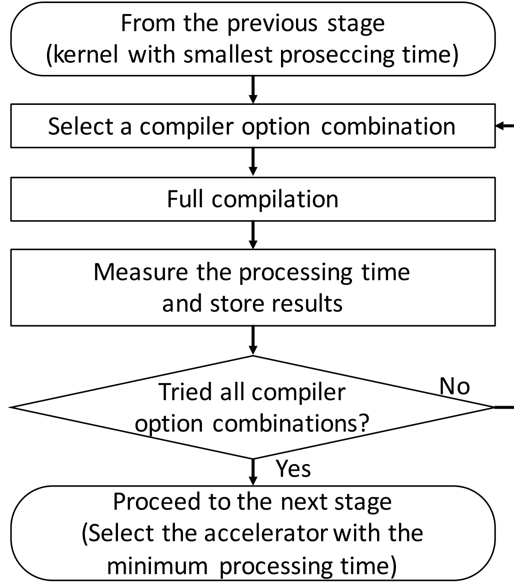


Figure 7: Optimizing the compiler options.

Table 3: Estimated and measured number of clock cycles for different *nPE* while increasing *nPCM* to the maximum.

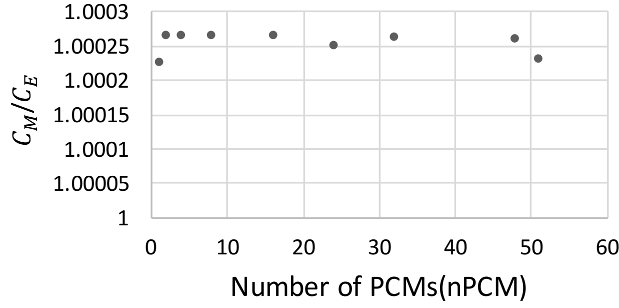
nPE	**Estimated cycles** (C_E)	**Measured cycles** (C_M)	**Ratio of difference** (C_M/C_E)	**Frequency** (MHz)
1	4,301,655,232	4,302,639,000	1.0002	297
2	4,432,567,524	4,434,804,934	1.0005	298
4	4,565,077,184	4,569,681,156	1.001	304
8	4,564,241,600	4,629,408,652	1.014	295

4.1 Accuracy of the processing time estimation

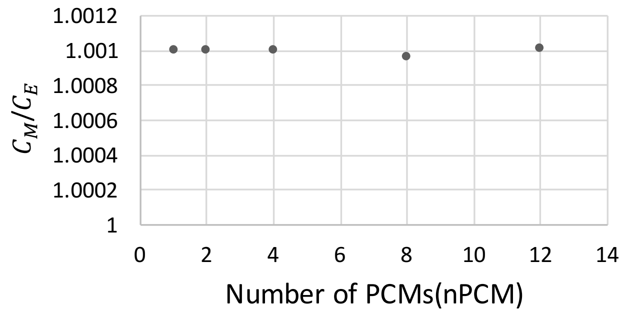
Figure 8 shows the ratio of measured cycle to estimated clock cycle for different parameter values. In Figure 8(a), $nPE = 1$ and *nPCM* changes from 1 to 51. In Figure 8(b), $nPE = 4$ and *nPCM* changes from 1 to 12. The estimated clock cycles are obtained using Eq.(1) as explained in section 3.3. The measured clock cycles are obtained by multiplying the processing time by the clock frequency. The ratio of difference is very close to 1. Table 3 shows the estimated and measured clock cycles for different *nPE* values. The evaluation is done for 2-D 5-point stencil on DE5 board. In this evaluation, we increased *nPCM* to the largest possible value that satisfy the constraints. The ratio of difference is very close to 1. According to these results, we can say that the estimation is very accurate.

4.2 Automatic optimization

The optimization is done using two design parameters, and two compile options. Table 4 shows the optimal parameters using different stencil computation examples and different FPGA boards. We can see that the optimal parameters are quite different. As a result,



(a) $nPE = 1$.



(b) $nPE = 4$.

Figure 8: Ratio of measured cycles to estimated clock cycles.

the optimal accelerator architecture is also different. This shows that the optimization is necessary to achieve the full potential of an FPGA board for a given application.

Table 5 shows the reduction of the amount of intermediate compilations using the binary search. The evaluation is done using DE5a FPGA board. According to the results, over 76% of the compilations can be reduced. This will reduce the accelerator design time.

Table 6 shows the comparison of the total design times of the previous work [7] and the proposed method using DE5a board. The total design time is reduced by upto 55.1%. This design time reduction is achieved by using a binary search to reduce the number of compilations in the stage 1.

Table 7 shows the comparison of the total design time of the conventional method explained in section 2, and the proposed method. The evaluation is done using DE5 board. In the conventional method, all possible stencil computation kernels have to be compiled

Table 4: Optimal parameters obtained for different applications and FPGAs

Application	DE5a (Arria 10)				DE5 (Stratix V)			
	nPE	nPCM	fpc	fp-relaxed	nPE	nPCM	fpc	fp-relaxed
Laplace equation	4	112	on or off	on	8	18	off	on
2-D 5-point Jacobi	4	57	on or off	on	1	51	on	on
2-D 9-point Jacobi	2	72	on or off	off	4	7	on	off

Table 5: Comparison of the number of compilations in the stage 1.

Application	Number of compilations		Reduction percentage (%)
	Previous work [7] (exhaustive search)	Proposed method (binary search)	
Laplace equation	372	47	87.3
2-D 5-point Jacobi	285	48	83.1
2-D 9-point Jacobi	191	45	76.4

Table 6: Comparison of the total design times of the previous work [7] and the proposed method.

Application	Total design time (hours)		Reduction percentage (%)
	Previous work [7] (exhaustive search)	Proposed method (binary search)	
Laplace equation	235.5	105.7	55.1
2-D 5-points Jacobi	190.1	90.9	52.2
2-D 9-points Jacobi	103.5	75.3	27.2

in both stages and evaluated to find the best accelerator. The proposed method reduces the number of compilations in stage 1 by using a binary search. It also reduces the number of compilations in stage 2 by predicting the performance accurately using the compilation results of stage 1. The design time of the proposed method is 4% $\sim$ 8% of that of the conventional method.

5 Conclusion

In this paper, we proposed an automatic optimization method for OpenCL kernels. We predict the performance of an OpenCL kernel by analyzing the log files and resource utilization reports. We use the proposed optimization methodology to implement stencil computation kernels. According to the evaluation, the performance prediction is very accurate, and the optimized design can be found in significantly smaller design time compared to the con-

Table 7: Comparison of the design time.

Application	**Design method**	**Time (hours)**
Laplace equation	conventional	528.0
	proposed	19.5
2-D 5-points Jacobi	conventional	198.1
	proposed	10.0
2-D 9-points Jacobi	conventional	161.5
	proposed	12.3

ventional method. In future, it could be possible to enhance the proposed method for other types of applications and also for NDRange kernels.

Acknowledgment

This work is supported by MEXT KAKENHI Grant Number 16K124040.

References

[1] S. Brown and Z Vranesic, "Fundamentals of digital logic design with Verilog Design", 2007.

[2] S. Brown, "Fundamentals of digital logic design with VHDL Design", 2008.

[3] T.S., Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D.P. Singh, S.D. Brown, "OpenCL for FPGAs: Prototyping a compiler", International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), pp. 3-12, 2012.

[4] The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl/`, 2015.

[5] Q. Jia and H. Zhou, "Tuning Stencil Codes in OpenCL for FPGAs", IEEE 34th International Conference on Computer Design (ICCD), pp. 249-256, 2016.

[6] K. Krommydas, R. Sasanka, W. Feng, "Bridging the FPGA programmability-portability Gap via automatic OpenCL code generation and tuning", IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 213-218, 2016.

[7] T. Endo, H.M. Waidyasooriya, M. Hariyama, "Automatic Optimization of OpenCL-Based Stencil Codes for FPGAs". In: Lee R. (eds) Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. SNPD 2017. Studies in Computational Intelligence, Vol 721. Springer, Cham, 2017.

[8] G. Roth, J. Mellor-Crummey, K. Kennedy, R.G. Brickner, "Compiling Stencils in High Performance Fortran", Proceedings of the 1997 ACM/IEEE conference on Supercomputing, pp.1-20, 1997.

[9] G. Karniadakis, S. Sherwin, "Spectral/hp Element Methods for Computational Fluid Dynamics", Oxford University Press, 2013.

[10] K.S. Yee, "Numerical solution of initial boundary value problems involving Maxwells equations in isotropic media", IEEE Transactions on Antennas and Propagation, Vol.14, No.3, pp.302-307, 1966.

[11] K. Sano, Y. Hatsuda, S. Yamamoto, "Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth", IEEE Transctions on Parallel and Distributed Systems, Vol.25, No.3, pp.695-705, 2014.

[12] K. Dohi, K. Okina, R. Soejima, Y. Shibata, K. Oguri, "Performance Modeling of Stencil Computing on a Stream-Based FPGA Accelerator for Efficient Design Space Exploration", IEICE Transactions on Information and Systems, Vol.E98-D, No.2, pp.298-308, 2015.

[13] H.M. Waidyasooriya, Y. Takei, S. Tatsumi and M. Hariyama, "OpenCL- Based FPGA-Platform for Stencil Computation and Its Optimization Methodology," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no.5, pp.1390-1402, 2017

[14] H.M. Waidyasooriya, M. Hariyama and K. Kasahara, "Architecture of an FPGA Accelerator for Molecular Dynamics Simulation Using OpenCL," Proc. 15th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2016), pp.115-119, 2016.

[15] H.M. Waidyasooriya, M. Hariyama and K. Kasahara, "An FPGA Accelerator for Molecular Dynamics Simulation Using OpenCL," International Journal of Networked and Distributed Computing, Vol. 5, No. 1, pp.52-61, 2017.

[16] Intel FPGA SDK for OpenCL, `https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html`, 2016.

[17] Terasic, DE5-Net FPGA Development Kit, `http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English\&CategoryNo=158&No=526`

[18] Terasic, DE5a-Net Arria 10 FPGA Development Kit, `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=231&No=970&PartNo=2`