High-level Synthesis based on Parallel Design Patterns using a Functional Language

Morihiro KUGA*, Kansuke FUKUDA**, Motoki AMAGASAKI*,
Masahiro IIDA* and Toshinori SUEYOSHI*

* **Faculty of Advanced Science and Technology, Kumamoto University
2-39-1 Kurokami, Kumamoto, 860-8555 Japan

*{kuga, amagasaki, iida, sueyoshi}@cs.kumamoto-u.ac.jp}

ABSTRACT

Logic-circuit integration of a field-programmable gate array (FPGA) has grown considerably with improvements in semiconductor technology. High-level synthesis (HLS) is now widely used to implement complex FPGA applications to increase design efficiency, taking the place of typical register-transfer level (RTL) design. Most HLS tools support C-like languages such as C/C++, SystemC, OpenCL, and CUDA. However, coarse-grain parallelism from a C-like language cannot be extracted easily, hence some tools use explicitly parallel programming languages to design hardware. However, all such tools rely on the programmer to correctly parallelize and perform optimizations on the application, which often forces the programmer to acquire hardware-design knowledge.

In this work, we propose an HLS environment for FPGAs using the embedded domain-specific language (DSL) of Haskell as the design language. Haskell is a pure functional language that has the features of referential transparency and no side effects. Hence, it is better for mapping to hardware. Higher-order functions such as `map`, `zipWith`, and `reduce` are useful for allowing a parallel design pattern to automatically extract parallelism in the design. In our environment, the embedded DSL program is compiled to the LLVM intermediate representation (LLVM IR), which is then integrated into the open-source HLS tool, LegUp. LegUp synthesizes the LLVM IR to Verilog HDL, which can be merged with the FPGA design tools of the FPGA vendor.

The evaluation results show that our proposed implementation achieves 3.00 and 4.96 times speed-up in two benchmarks, array addition and summation of array, respectively, relative to a C-like language design.

1. INTRODUCTION

A field-programmable gate array (FPGA) is a programmable integrated circuit that can be used to realize application-specific hardware. It achieves substantially superior performance and lower power consumption compared with the conventional hardware architecture of a central processing unit (CPU), a graphics processing unit (GPU), and a digital signal processor (DSP). For the digital circuit design of an FPGA, which has many gate-level logical components, one generally uses register transfer levels (RTLs) in a hardware description language (HDL) such as Verilog HDL or VHDL. However, such hardware design is less efficient than software coding because the RTL description requires a cycle-accurate design. In recent years, to increase the design productivity for more complex system-on-a-chip (SoC) designs, many studies have been conducted into high-level synthesis (HLS) [1, 2], which enables more abstract design using C-like languages and which synthesizes RTL source code automatically. In particular, commercial HLS tools are becoming popular and hardware design techniques are changing from an RTL description to an untimed functional description. As for the present HLS, its performance is worse than that of RTL design if insufficient directives for behavioral synthesis are included. However, HLS is more efficient than other hardware design techniques in the sense that the HLS tool supports automatic architecture exploration; it searches for an efficient architecture that gives an acceptable trade-off between hardware resources and performance. Therefore, HLS facilitates searching for circuit designs, which are difficult to design in RTL. Many HLS tools are available, but there is an inherent difficulty: they extract the parallelism of logical circuit behavior to synthesize RTL code from procedural languages such as C. The problem is in extracting thread-level parallelism from an entire set of source code, which is necessary for highly efficient FPGA implementation [1]. Parallel processing is the biggest advantage of using an FPGA, therefore we must extract the maximum parallelism of an application. To solve these problems, most design tools extend a base language or use a specific parallel programming language. However, with all these tools, the programmer is forced to parallelize and optimize an application correctly.

In contrast, a domain-specific language (DSL) supports more abstract and application-specific designs than are possible using Verilog HDL or VHDL for RTLs. If the designer is knowledgeable about the hardware design and the DSL description, the design efficiency is higher than that of RTL design. However, the designer must be knowledgeable about the hardware design and must determine the circuit design using her/his design experience and her/his know-how.

In the present work, we propose an HLS environment for FPGAs using the Haskell embedded DSL as the design language. Haskell is a pure functional language with the features of referential transparency and no side effects, so it is a better fit for mapping to hardware. Higher-order functions

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2017) Bochum, DE, June 7-9, 2017.

such as `map`, `zipWith`, and `reduce` are useful for parallel design patterns that automatically extract any parallelism in the design. In addition, the design pattern is helpful for generating a hardware accelerator even if the programmer does not have the necessary hardware design skills.

The remainder of this paper is organized as follows. In section 2, we discuss related work and we focus on LegUp and Lava in particular as the base tools for our HLS environment. We give an overview of our HLS in section 3, and we describe the parallel design pattern that we use in our HLS in section 4. In section 5, we evaluate the ability of our HLS tools, and finally we conclude the work in section 6.

2. RELATED WORK

In recent years, research and commercialization of HLS tools for FPGAs have been actively undertaken, most of which support C-like languages such as C/C++, SystemC, OpenCL, and CUDA. Design tools such as Vivado HLS by Xilinx, LegUp [2], and ROCCC [3] use C/C++ and SystemC as the design languages. Unfortunately, it is difficult to extract parallelism from a widespread procedural language such as C [1]. Therefore, OpenCL-to-FPGA [4] and FCUDA [5] use OpenCL and CUDA, respectively, which are specific parallel programming languages. However, with all these tools, the programmer must describe the parallel programming and optimization, which requires knowledge about the hardware design.

2.1 LegUp

LegUp [2] is an open-source HLS tool developed by the University of Toronto. It uses ANSI-C, which has no extensions. LegUp is a synthesis tool for the back-end of LLVM [6], which is a compiler environment. The C program is compiled to the LLVM intermediate representation (LLVM IR) by Clang, which is the C front-end of LLVM. The LLVM IR then outputs Verilog HDL code through HLS processes such as allocation, scheduling, and binding. Any hot spots in the program are identified by the profiling and are accelerated by the synthesized hardware accelerator. The remainder of the program is compiled and its binary is executed by the MIPS processor implemented on the FPGA with the hardware accelerator.

2.2 Lava

Lava is a hardware design and a verification tool. It uses a functional-type HDL that is incorporated into the Haskell language. There are various versions of Lava: original Lava [7], Lava 2000 [8] from the same developer, Xilinx Lava [9] that is specialized for Xilinx's FPGA, Kansas Lava [10] that is a revised version of Lava 2000, and York Lava [11] that was developed for the Reduceron processor design. In the present paper, we use Lava 2000 to develop our tool.

In Lava, a circuit is described as a streaming accelerator that generates an output stream from an input stream. The accelerator is described as a function in the Haskell language. An application behavior is described by a set of functions. Therefore, there are fewer lines of source code than there would be in procedural languages such as C/C++ and SystemC because of the use of function expressions. In the Lava system, there are reusable primitive circuit components, much like a library or intellectual property (IP). Therefore, a function expression can be easily generated as a circuit structure by combining circuit parts. Function expressions are possible in Lava because of its structure-based circuit design, but Lava cannot explore architectures unlike typical HLS tools. Hence, each function expression is synthesized to only one RTL design.

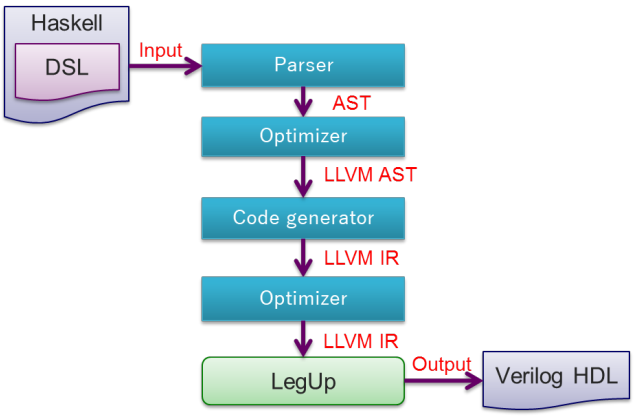


Figure 1: Design flow in the present HLS.

Therefore, our proposed HLS environment is ultimately intended to be an automatic tool to synthesize the RTL design from one original description, which should be suited to expressing constraints such as operating frequency and FPGA resource limitations, by combining Lava and LegUp.

3. HARDWARE DESIGN BY FUNCTIONAL LANGUAGE

High-level-synthesis tools are widely used to implement complex FPGA applications to increase the design efficiency, taking the place of typical RTL design. The C programming language is well known, and most HLS tools support C-like languages. It was hoped that C-like language design would increase the design efficiency with abstract descriptions and by generating hardware designs that were correct and more scalable. However, there are many problems with HLS in a procedural language. For example, a procedural language is essentially targeted at sequential processing, whereas the hardware essentially treats parallel processing. Therefore, it is difficult for the synthesizer to estimate the parallelism and generate the correct hardware structure. Specifically, the designer must describe the problem such that the synthesizer can generate the correct hardware, and must insert some compiler directives to indicate the parallelism of the program and an adequate hardware structure. Moreover, an HLS tool is black-box software, and the synthesized HDL code is less readable and does not correspond to the original code. Therefore, the designer has to check the hardware behavior with a timing chart and must repeat the optimizing work to generate an adequate hardware by revising the source code.

Functional languages (specifically, pure functional languages) have the feature of referential transparency. A function with referential transparency has the following properties:

i) it returns the same result for the same inputs, and

ii) the function causes no observable changes before or after it is called; in other words, there are no side effects.

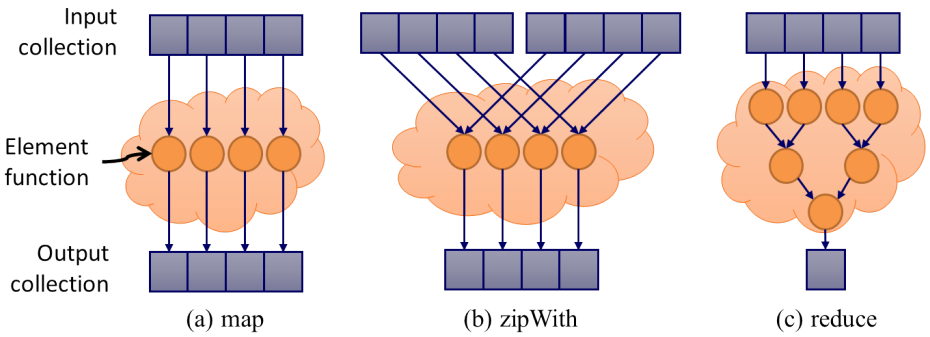


Figure 2: Parallel design pattern.

In a functional language, because all functions are referentially transparent, each function can be parallelized without side effects even if it is a complex function. Therefore, the functional expression can be represented directly by its dataflow graph, and the dataflow can be mapped directly to the hardware function. We use the embedded DSL of the functional language Haskell as our HLS design language. Our DSL supports function definition, the `Let` statement, the `If` statement, the `For` statement, arithmetic operations, comparison operations, the "extern" syntax, and the data types `int` and `float`. It also supports the `map`, `zipWith`, and `reduce` parallel design patterns, which are described in section 4.2.

The design flow of our HLS environment is shown in Fig. 1. Firstly, for the embedded DSL source code in Haskell that was described by the application programmer, its syntax is analyzed by the parser and an abstract syntax tree (AST) is generated. Parallel design patterns such as `map`, `zipWith`, and `reduce` in AST translate the optimized streaming process by the first optimizer and stores the information to LLVM AST. Next, LLVM AST is converted to LLVM IR by the code generator. At this point, LegUp converts the LLVM IR information to synthesizable code. Furthermore, the second optimizer attempts to optimize at least the tiling. After that, the final LLVM IR becomes the input to LegUp, which generates the data path and its control-state machine in accordance with LLVM IR. In addition, LegUp treats the parallelism in the basic block. Finally, LegUp generates Verilog HDL code for implementation into the FPGA using the FPGA design tool.

4. PARALLEL DESIGN PATTERN

4.1 Control structure

Before discussing parallel design patterns, we introduce three control structures in a structured theorem [12], the combinations of which can describe all algorithms.

4.1.1 Sequence

Sequence is the list of tasks that are executed in a specific order. In general, the task order is ordered by "source code order" in a procedural programming language. Each task is completed before the next task begins. This sequence order must be kept even if there is no data dependency between the tasks. Therefore, if there is a side effect during the tasks, the final results become deterministic. The functional language excludes "source code order" limitation; the function order is determined only by the data dependency between the functions.

4.1.2 Selection

In selection, condition `c` is evaluated first, task `a` is executed if the condition is `true`, and task `b` is executed if the condition is `false`.

4.1.3 Iteration

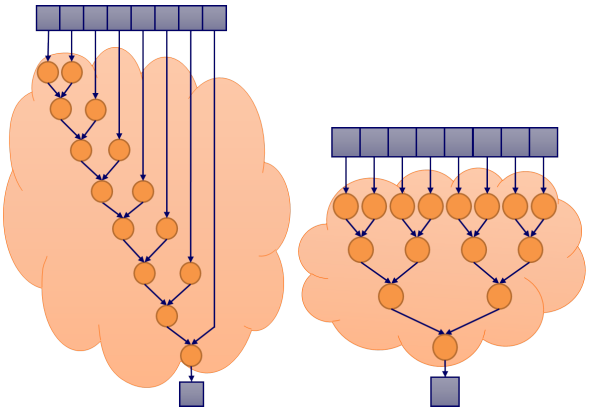
In iteration, condition `c` is evaluated first and task `a` of the loop block is executed if the condition is `true`, whereupon condition `c` is evaluated once again. This process is repeated until the condition becomes `false`. One of the problems when parallelizing the iteration pattern is that there is a possibility of some data dependency between loop iterations. According to the data dependency, the loop iterations can be parallelized via methods such as "parallel for."

4.2 Parallel design pattern

The parallel design pattern shown in Fig. 2 extends the control structure described in section 4.1 for parallel processing. A parallel design pattern can easily represent a parallel processing structure. This was treaded in the related works [13, 14].

4.2.1 map

The `map` pattern shown in Fig. 2(a) substitutes for the specific repetition of the serial program. All repetition is independent, and when the number of iterations is known in advance, all calculations depend only on the index value, and the loop that reads the data gives the input for the indexes the collection. `map` transforms the set by the function. Specifically, it applies a function (hereinafter referred to as an element function) to all elements of the collection in parallel. Each element function accesses separate data elements. Each parallel transformation with this element function is called an instance of that element function. There must be no side effects caused by executing all the instances of `map` in arbitrary order in the element function that is used in `map`. With this independence, it is not necessary to synchronize among the separate elements of `map`, and maximum parallelism is achieved. Generally, side effects of the element function must be considered, because it is difficult to judge automatically whether any occur, but Haskell is a functional language that does not permit side effects, so the requirements on `map` are satisfied.



(a) Sequential. (b) Parallel (Tree).

Figure 3: Implementation of 8-input `reduce`.

4.2.2 *zipWith*

As shown in Fig. 2(b), `zipWith` resembles `map`. There are two input collections in `zipWith`, and an element function outputs one new result from each element pair from the two input collections. For example, one would use `map` when squaring each element in an array, and would use `zipWith` when adding each element in two arrays. The former element function is the unary function $f(x) = x^2$ or $f(x) = x * x$, and the latter element function is the binary function $g(x, y) = x + y$.

4.2.3 *reduce*

As shown in Fig. 2(c), `reduce` calculates a single result from all input collections using the aggregating function $f(x, y) = x \circ y$ in the same way as summing all the elements of an input. It is possible to implement `reduce` in various ways, assuming that the elements can be pairwise combined. The implementation of `reduce` is shown in Fig. 3. Figure 3(a) is a simple sequential algorithm for the case of an eight-input collection. To parallelize `reduce`, the operation order must be changed from that of the sequential algorithm. There are various ways to change an order, but any such algorithm depends on the associative or commutative properties of the aggregating function.

Associativity is a property of the binary operator "$\circ$," and satisfies the following formula:

$$(f \circ g) \circ h = f \circ (g \circ h).$$

If the aggregate function is associative, then the aggregate for all neighbor pairs can be calculated in arbitrary order, like the binary tree shown in Fig. 3(b). Also, when associativity is satisfied, the binary operator "$\circ$" can have the commutative property. The aggregate function is said to satisfy the commutative property when the two operands can be swapped in calculating the binary operator "$\circ$" without affecting the outcome. That is:

$$f \circ g = g \circ f.$$

5. EVALUATION

In this section, we evaluate the advantage of parallelization using the parallel design pattern described in section 4.2 with the embedded DSL of the Haskel functional language.

5.1 Experimental method and its environment

The benchmark program is described in the embedded DSL of Haskell proposed in section 3 and in C. The former is compiled to LLVM IR and LegUp generates Verilog HDL code from our HLS environment, and the latter is synthesized from C code to Verilog HDL by LegUp. In both, the target FPGA device is a Cyclone V SoC 5CSEMA5F31C6 on an Altera DE1-SoC board with a target clock period of 20 ns (50 MHz). Verilog HDL code is generated by LegUp 4.0 and latency is measured by an HDL simulator, ModelSim Altera 10.3d. If the HLS tool can extract the parallelism from the benchmark, two, four, or eight element functions are implemented as the hardware accelerator. We measure the processing time latency to evaluate the advantage of our HLS environment.

5.2 Benchmark program

We use two simple benchmark programs.

5.2.1 *Array addition*

"Array addition" adds each array element in two arrays. The generated C and Haskell code is shown in Figs. 4(a) and 4(b), respectively. In the C code, it appears as if each iteration is calculated sequentially by using a `for` statement. However, there is no dependency between any of the iterations. Therefore, this loop matches the `zipWith` parallel design pattern.

```
void array_add (int n,
     int a[n],int b[n],int c[n]) {
  int i;
  for( i=0 ; i<n ; i++ ) {
    a[i] = b[i] + c[i];
  }
}
```

(a) C code.

```
array_add b c = zipWith (+) b c
```

(b) Haskell code.

Figure 4: Array addition.

5.2.2 *Sum of array*

"Sum of array" calculates the sum of all the elements in an array. The C and Haskell codes are shown in Figs. 5(a) and 5(b), respectively. In the C code, it appears as if each iteration is calculated sequentially by using the `for` statement shown in Fig. 3(a). However, the addition operator "+" has associativity. Therefore, this calculation can be operated in parallel with the tree structure shown in Fig. 3(b).

5.3 Evaluation results and considerations

5.3.1 *Array addition*

Firstly, we show the evaluation results for the "Array addition" benchmark. Figure 6 shows the processing time latencies of the C and Haskell DSLs. In the Haskell DSL, the hardware accelerator has two element functions. The values on the horizontal axis indicate the array size. The results show that the Haskell DSL is 2.20–2.40 times faster than the C version. As for the associative property, in the

```c
int array_sum (int n, int a[n]) {
  int sum = 0;
  int i;
  for( i=0 ; i<n ; i++ ) {
    sum += a[i];
  }
  return sum;
}
```

(a) C code.

```
array_sum a = reduce (+) a
```

(b) Haskell code.

Figure 5: Sum of array.

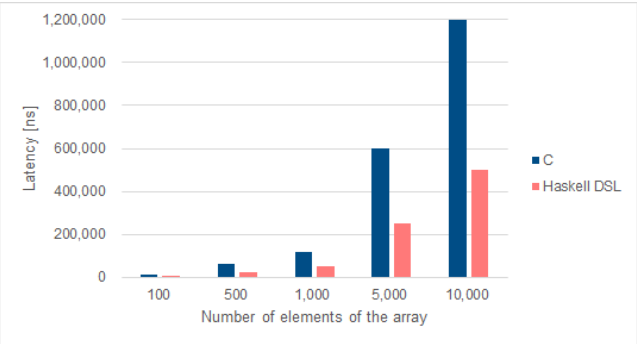


Figure 6: Latency of "array addition" (2-parallel).

Haskell DSL, the parallelism of the benchmark is extracted adequately.

In contrast, the C code shown in Fig. 4(a) appears to have the associative property in the `for` loop body. However, the following function call causes a side effect:

```c
int a[10];
array_add(10, a+1, a, a);
```

In this function call, there is a data dependency between iterations in relation to the array `a`. This possibility of overlapping forces the sequential processing.

Because there are no side effects in the element function in Haskell and there is no data dependency between iterations, all instances can be executed in parallel. In this way, the parallel design pattern can eliminate the regulation of the presupposed sequential execution. The HLS tool can recognize the parallelism in the program by using the parallel design pattern.

Figure 7 shows the processing time latency of the C and Haskell DSLs. In the Haskell DSL, the hardware accelerator has four element functions. The results show that the Haskell DSL is 2.67–3.00 times faster than the C version in relation to the associative property.

Figure 8 shows the processing time latencies of the C and Haskell DSLs. In the Haskell DSL, the hardware accelerator has eight element functions. The results show that the Haskell DSL is 2.34–3.00 times faster than the C version in relation to the associative property.

The parallel design pattern can specify the parallelism in the source code. However, the efficiency of parallelizing is

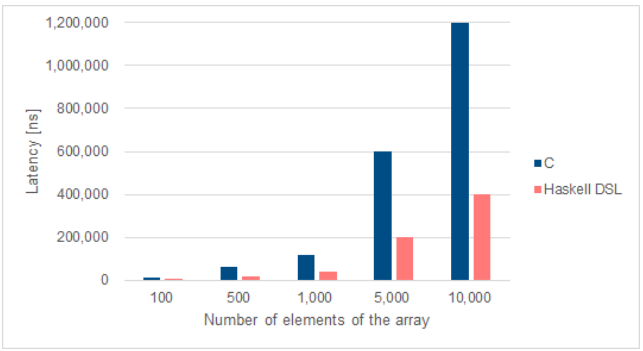


Figure 7: Latency of "array addition" (4-parallel).

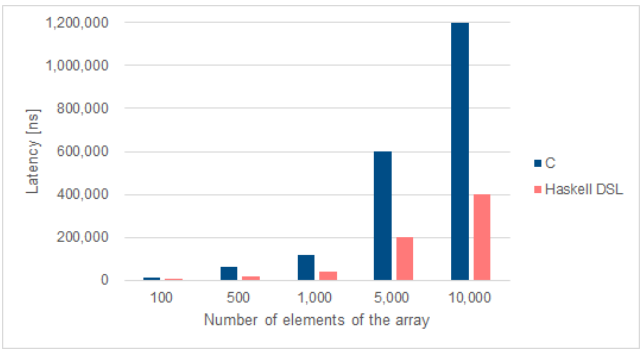


Figure 8: Latency of "array addition" (8-parallel).

not good when the computation load is small in each element function. To realize a scalable implementation, the best degree of parallelism must be explored with the tiling method, for example. Actually, although there are twice as many hardware elements in the architecture shown in Fig. 8 as in that shown in Fig. 7, the latency is not improved by a factor of 2. This means that the memory-access contentions that occur reflect simultaneous read and write accesses because of a lack of hardware resources of the memory port, for example.

5.3.2 *Sum of array*

Next, we show the evaluation results for the "Sum of array" benchmark. Figure 9 shows the processing time latencies of the C and Haskell DSLs. In the Haskell DSL, the hardware accelerator has two element functions. The results show that the Haskell DSL is 2.02–2.49 times faster than the C version. The C code shown in Fig. 5(a) appears to have a data dependency in relation to the next iteration loop, so that each iteration cannot be executed in parallel. However, if summation is performed with the tree structure shown in Fig. 3(b), this loop can be parallelized. In the Haskell code shown in Fig. 5(b), the programmer can explicitly hint the parallelism intended to the compiler by `reduce` in the parallel design pattern.

Figure 10 shows the processing time latencies of the C and Haskell DSLs. In the Haskell DSL, the hardware accelerator has four element functions. The results show that the Haskell DSL is 2.82–4.96 times faster than the C version in relation to the associative property.

Figure 11 shows the processing time latencies of the C and Haskell DSLs. In the Haskell DSL, the hardware accelera-

tor has eight element functions. The results show that the Haskell DSL is 1.97–4.92 times faster than the C version in relation to the associative property.

The compiled LLVM IR design from the Haskell DSL is correctly executed without a problem. This indicates that the synthesizer can freely select the order of calculation to obtain the best performance. To realize a scalable implementation, the best degree of parallelism must be explored in the same way as for "array addition." Actually, although there are twice as many hardware elements in the architecture shown in Fig. 10 than in the one shown in Fig. 11, the latency is not improved by a factor of two for the same reason as for "Array addition."

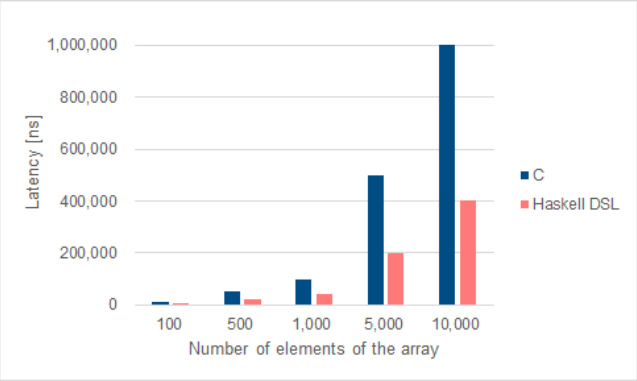


Figure 9: Latency of "sum of array" (2-parallel).

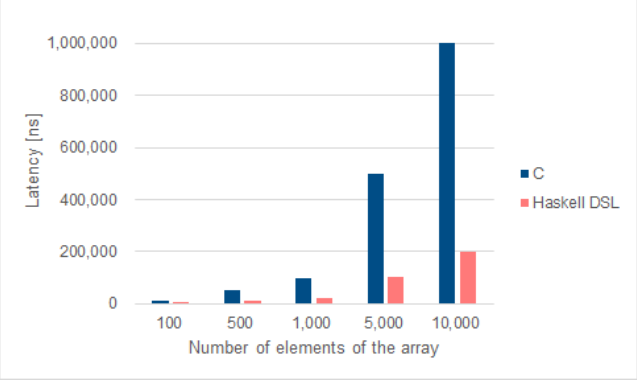


Figure 10: Latency of "sum of array" (4-parallel).

6. CONCLUSION

We proposed an HLS environment for FPGAs using the embedded DSL of Haskell as the design language. Haskell is a pure functional language and a better fit for mapping to hardware. Higher-order functions such as `map`, `zipWith`, and `reduce` are useful for allowing the parallel design pattern to automatically extract parallelism in the design. The evaluation results show that our proposed implementation using eight element functions is 3.00 and 4.96 times faster than with C-like language design in the two benchmarks, array addition and sum of array, respectively.

As future work, we will deal with loop tiling to achieve a more scalable mapping. We will also utilize the back-end log of the simulation and the implementation results to optimize the hardware design.

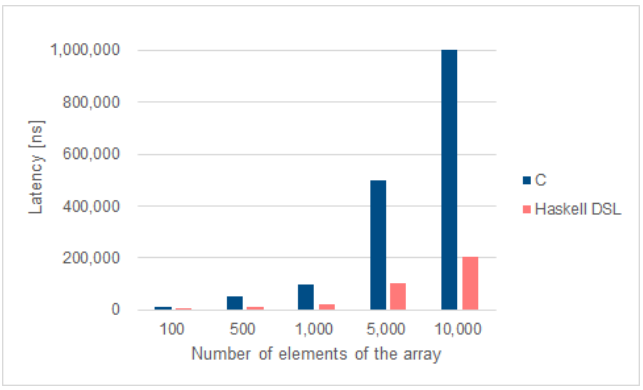


Figure 11: Latency of "sum of array" (8-parallel).

7. REFERENCES

[1] J. Cong, et.al, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[2] A. Canis, et.al, "LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems," *ACM TECS*, 13(2):24:1–24:27, 2013.

[3] J. Villarreal, et.al, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," *IEEE FCCM*, pp.127–134, 2010.

[4] T. S. Czajkowski, et.al, "From OpenCL to High-Performance Hardware on FPGAs," *Field Programmable Logic and Applications (FPL)*, pp.531–534, 2012.

[5] A. Papakonstantinou, et.al, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," *IEEE Symp. on Application Specific Processors (SASP)*, pp.35–42, 2009.

[6] The LLVM Compiler Infrastructure project, `http://www.llvm.org/`.

[7] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in haskell," *ACM SIGPLAN Notices*, 34(1):174–184, 1998.

[8] K. Claessen and M. Sheeran, "A Slightly Revised Tutorial on Lava: A Hardware Description and Verification System," 2007.

[9] S. Singh, et.al, "Lava and jbits: From hdl to bitstream in seconds," *IEEE FCCM*, pp.91–100, 2001.

[10] A. Gill, et.al, "Introducing Kansas Lava," *Int'l Symp. on Implementation and Application of Functional Languages (IFL)*, pp.18–35, 2009.

[11] M. Naylor and C. Runciman, "The reduceron reconfigured and re-evaluated," *Journal of Functional Programming*, 22(4-5):574–613, 2012.

[12] C. Böhm and G. Jacopini, "Flow diagrams, turing machines and languages with only two formation rules," *Comm. of the ACM*, 9(5):366–371, 1966.

[13] R. Prabhakar, et.al, "Generating Configurable Hardware from Parallel Patterns," *ACM ASPLOS*, pp.651–665, 2016.

[14] D. Koeplinger, et.al, "Automatic Generation of Efficient Accelerators for Reconfigurable Hardware," *Int'l Symp. on Computer Architecture (ISCA)*, pp.115–127, 2016.