Electronic Preprint for Journal of Information Processing Vol.26

Regular Paper

Identification and Elimination of the Overhead of Accelerate with a Super-resolution Application

Izumi Asakura[1,a)] Hidehiko Masuhara[1] Takuya Matsumoto[2] Kiminori Matsuzaki[3]

Received: April 7, 2017, Accepted: August 9, 2017

Abstract: This paper supplements the evaluation of libraries for parallel computing in Haskell by Matsumoto and Matsuzaki. The previous work implemented an image processing program by using two different libraries and compared expressiveness and parallel execution performance. In particular, it found that a parallel execution of the Accelerate program on a GPU is slower than the execution of the handwritten CUDA-C program by a factor of more than 70. We analyzed the execution of the Accelerate program and identified that the major sources of the overhead are in the process of the syntax tree before execution of a GPU kernel. Since the program realizes repeated execution by constructing a large syntax tree whose size is proportional to the number of repetitions, we rewrote this part by using a while-construct, which significantly reduces the overhead as it runs half as fast as the CUDA-C program. We also discuss language design alternatives for avoiding this type of overhead.

Keywords: GPGPU, Accelerate

1. Introduction

This work supplements the verification of usefulness of parallel libraries in Haskell studied by Matsumoto and Matsuzaki [4] (*previous work*). In the previous work, they compared two parallel libraries in Haskell —specifically, Repa [2] and Accelerate [1]—in terms of the performance and expressiveness with a target application that performs image super-resolution. In the paper, they reported that the execution time of the Accelerate program with 640 GPU cores was 1.15–1.29 times as long as that of the sequential Haskell program and sequential Java programs and was 80 times as long as that of the CUDA program with the same number of GPU cores.

Although this result could be due to the poor performance of Accelerate compared with CUDA-C and/or due to a bad description in the Accelerate program, it was not clear which of them was the main cause. Only the following facts were found by the investigation in the previous work: there were no problems in terms of branch divergence and the number of threads and 70% of the execution time was consumed for the process on the Haskell (CPU) side.

In this work, we analyze the execution time of the Accelerate program and report that there is significant overhead caused by the processing of syntax trees at the runtime. We also show that we can suppress the performance degradation up to about twice that of the CUDA program by changing the description of the program.

[1] School of Computing, Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
[2] Graduate School of Engineering, Kochi University of Technology, Kami, Kochi 782–8502, Japan
[3] School of Information, Kochi University of Technology, Kami, Kochi 782–8502, Japan
[a)] asakura@prg.is.titech.ac.jp

The problem addressed in this work is indeed restrictive. However, we consider that this work complements the previous work considerably since the Accelerate program has achieved reasonable performance for a practical application.

2. Image Super-resolution Naively Implemented with Accelerate

Firstly, we show the Haskell program with the Accelerate library developed and evaluated in the previous work [4] and review how it is processed.

Figure 1 shows the Haskell program developed with the Accelerate library. This program was developed by straight porting from the Haskell program with the Repa (REgular PArallel arrays) library. The following four changes were made for using the Accelerate library. (1) We used specific types wrapped by `Acc` and/or `Exp` defined in the Accelerate library for the arrays and variables processed on the GPU side (lines 8 and 11). (2) We used special operators for the conditional branches and their conditions processed on the GPU side (lines 19–22, 26, and 28). (3) We expanded a list comprehension by hand since list comprehensions were not allowed (line 27). (4) To apply function `step` many times on a GPU, we linked the processes by operator `>->` (line 5).

In the previous work, the performance of this program was evaluated and compared with the original sequential Java program [5], a sequential Haskell program, and a program implemented in C and CUDA. The evaluation was conducted on a computer equipped with two CPUs, Intel Xeon E5-2620 v3 2.40 (3.20 GHz, 6 cores, HT off), 32 GB of memory, and GPU GM107-400-A2 (1,020 MHz, 640 CUDA cores, VRAM 2 GB) with the software GHC 7.6.3, Accelerate 0.13.0.3, CUDA 6.0, and GCC 4.8.4. The Haskell program with the Accelerate library ran in 18.4s. This execution time was longer than those of the

```
1  type OImg = Array DIM2 Int
2  type IImg = Array DIM3 Int
3
4  accSuperResolution :: Acc IImg -> Acc OImg -> Acc OImg
5  accSuperResolution iImg oImg  = foldl1 (>->) steps oImg
6    where  steps = [step iImg | _ <- [1..200]]
7
8  step :: Acc IImg -> Acc OImg -> Acc OImg
9  step iImg oImg  = generate (shape oImg) sr
10  where
11   sr :: Exp DIM2 -> Exp Int
12   sr ix = inOImg y x - (dEdp + (dEdp >* 0) ? (+2, -2)) `quot` 4
13    where
14     (Z :. y :. x) = unlift ix
15     srl (1, (dx,dy)) =
16       let
17         lx = (x + dx) `quot` 2;         ly = (y + dy) `quot` 2
18         hx = lx * scale - dx;          hy = ly * scale - dy
19         dso2i = ((hx    <* 0 ||* hx    >=* oW ||* hy    <* 0 ||* hy    >=* oH) ? (0, inOImg hy hx)) +
20                 ((hx+1 <* 0 ||* hx+1 >=* oW ||* hy    <* 0 ||* hy    >=* oH) ? (0, inOImg hy (hx+1))) +
21                 ((hx    <* 0 ||* hx    >=* oW ||* hy+1 <* 0 ||* hy+1 >=* oH) ? (0, inOImg (hy+1) x)) +
22                 ((hx+1 <* 0 ||* hx+1 >=* oW ||* hy+1 <* 0 ||* hy+1 >=* oH) ? (0, inOImg (hy+1) (hx+1)))
23         hval = dso2i `quot` 4
24         lval = iImg ! (lift (Z :. 1 :. ly :. lx))
25       in
26         (lx <* 0 ||* lx >=* iW ||* ly <* 0 ||* ly >=* iH) ? (0, hval-lval)
27     ds = srl (0, (1,1)) + srl (1, (0,1)) + srl (2, (1,0)) + srl (3, (0,0))
28     pn = (x >* 0 &&* y >* 0 &&* x <* oW - 1 &&* y <* oH - 1) ?
29           (4 * (inOImg y x - (inOImg y (x-1) + inOImg y (x+1) + inOImg (y-1) x + inOImg (y+1) x) `quot` 4), 0)
30     dEdp = ds + pn
31     inOImg y x = oImg ! (index2 y x)
```

Fig. 1 Haskell (Accelerate) program (Fig. 5 in previous work [4]).

sequential Java program (16.0s) and the sequential Haskell program (14.3s). It was 76 times longer than the execution time of the CUDA-C program on the same GPU.

3. Overhead Analysis

We examined the breakdown of the execution time of the Accelerate program developed in the previous work and considered the reason for the low performance. In this section, we outline the execution of Accelerate and explain the processes in the execution of the image super-resolution program. We then show the methods of measuring the execution time of each step and report the results.

3.1 Execution of Accelerate

Accelerate [1] is a data-parallel extension of Haskell and is implemented as an embedded domain-specific language. In Accelerate programs, we describe the sequential parts as functions in Haskell and the parallel parts with constructors provided for data-parallel computation (*parallel primitives*), such as generate, map, and fold.

The sequential parts of Accelerate programs are compiled by the Haskell compiler. In the runtime, parallel primitives generate a syntax tree for the parallel parts. When the `CUDA.run` function is applied to the syntax tree, the runtime system transforms the parallel primitives in the syntax tree to CUDA programs and calls the CUDA compiler to generate PTX codes [*1]. Then, the runtime system executes operations in the syntax tree in order and executes PTX codes through the foreign function interface of Haskell for parallel primitives.

[*1] Parallel thread execution codes, which are virtual machine codes executed by GPU cores. Here, PTX codes are assumed to include C++ host programs that call the virtual machine codes.

```
Whole program                                              (total)
  iImgHost := read images from file                        (file^H)
  superResolution                                          (sr^H)
    iImg := copy iImgHost to GPU                           (copy^H)
    ast0 := generate syntax tree                           (astdef^H)
                iterating step for 200 times
    CUDA.run(ast0, iImg)                                   (run^A)
      ast1 := convertAccWith(ast0)                         (convert^A)
      ast2 := compileAcc(ast1)                             (compile^A)
      executeAcc(ast2)                                     (exec^A)
        Repeat the following for 200 times                 (repeat^D)
          stepKern<<<...>>>(iImg,oImg)                     (kern^C)
    oImgHost :=  copy oImg to CPU                          (copy^H)
  write down oImgHost to file                              (file^H)
```

Fig. 2 Execution of Image Super-resolution Application.

The runtime system caches the PTX codes on filesystems. When the `CUDA.run` function is applied to the same syntax tree, the runtime system skips the generation and compilation of the CUDA programs and executes the cached (compiled) PTX codes.

3.2 Execution of Image Super-resolution Application

To discuss possible overheads, we divide the execution of the super-resolution application into steps (**Fig. 2**). In the figure, we described the steps as pseudo code in which indentation means division of an execution step. The labels put on the rightmost of the lines are used in this work. The symbols in superscript above the labels show that the corresponding step is: (H) Haskell code, (A) code in the Accelerate library, (D) a sequential part of the Accelerate DSL program, and (C) (compiled CUDA code from) a parallel part of the Accelerate DSL program.

We explain the process of each step.

file^{H} reads and writes input/output data. We exclude the performance of this step from the target of the study as in the

Table 1 Experiment Environments.

CPU	Intel Xeon X5670 2.93 GHz (6 cores) × 2 (Hyperthreading enabled)
RAM	PC3-10600 54 GB
GPU	NVIDIA Tesla K20X × 3,732 MHz, 2688 CUDA cores
VRAM	6 GB (each)
OS	SUSE Linux Enterprise Server 11 SP3

previous work.

sr^{H} stands for the whole process of image super-resolution application executed on memory. In the actual program, it corresponds to Haskell function `superResolution` [*2].

astdef^{H} generates a syntax tree that executes the "single step of the image super-resolution" 200 times consecutively. In Fig. 1, it corresponds to `accSuperResolution`, and the single step of the image super-resolution corresponds to `step`. The syntax tree generated from function `step` corresponds to a single execution of a parallel primitive.

run^{A} (`CUDA.run`) generates and executes a CUDA program from the given syntax tree. It includes mainly three steps: convert^{A}, compile^{A}, and exec^{A}.

convert^{A} corresponds to function `convertAccWith` in the Accelerate runtime program. As the comment in the source code [*3] states "convert a closed array expression to de Bruijn form while also incorporating sharing observation and array fusion," it applies preprocessing to the given syntax tree.

compile^{A} corresponds to function `compileAcc` in the Accelerate runtime program. As the comment in the source code states "initiate code generation, compilation, and data transfer for an array expression," it generates CUDA code from the syntax tree and compiles it to PTX code. In the performance measurement, this step was not executed since the runtime system used the cached PTX code.

exec^{A} corresponds to function `execAcc` in the Accelerate runtime program. As the comment in the source code states "interleave compilation & execution state annotations into an open array computation AST," it executes the given syntax tree.

repeat^{D} calls a parallel primitive (generated from the image super-resolution application) 200 times.

kern^{C} corresponds to a call of parallel primitive generate, which calls the compiled PTX code.

3.3 Experiment Methods

We used Tsubame 2.5 (**Table 1**) in the Tokyo Institute of Technology for the measurement of execution time. It is equipped with 3 GPUs, but Accelerate used only one of them. Note that since we used different CPUs and GPUs from the previous work we cannot directly compare the execution time.

The software environment used was as follows. We used a different version of software due to the restrictions of the hardware environment.

Accelerate: GHC 7.8.3 + Accelerate-0.15.1.0 `-O2 -threaded -rtsopts` / `+RTS -H256M -s`

CUDA: NVIDIA CUDA SDK 7.0.27 `-O3` + GCC 4.9.3

[*2] This function is not in Fig. 1.

[*3] https://github.com/AccelerateHS/, reference on May 2017

Table 2 Execution time of each step (s).

	Prev	This	CUDA
total	10.151	0.913	0.809
– sr^{H}	9.864	0.602	0.289
– – copy^{H}	0.003	0.005	0.002
– – kern^{C}	0.139	0.139	0.041
– – astdef^{H} + run^{A} – kern^{C}	9.722	0.458	—
GC	5.17	0.15	—

Table 3 Duration ratio of each step (%).

	Prev	This
total	100.0	100.0
– run^{A}(`CUDA.run`)	72.3	36.5
– – convert^{A}(convertAccWith)	59.6	3.5
– – compile^{A}(compileAcc)	—*	0.0
– – exec^{A}(executeAcc)	2.2	31.5

(* We did not calculate an accurate duration ratio for compileAcc because many items corresponding to it appeared in the profiler's output. We estimate that it is about 10% from the other items.)

We measured the execution time of each step in the following way. It is ideal to measure the execution time of each step, but it is not easy due to the lazy evaluation of Haskell. Therefore, we measured the time for some steps at a time.

total, file^{H}, sr^{H}: We measured the execution time with the language-specific functions for obtaining the current time (function `System.IO.getCurrentTime` in Haskell and function `clock_gettime` in CUDA). We measured 4 times and computed the average of the 2nd–4th measurements. Due to the lazy evaluation, we did not measure the breakdown of sr^{H}.

kern^{C}, copy^{H}: We executed the whole program on the profiler nvprof (nvprof is attached to CUDA SDK) and measured the time for executing kernels and for data transfer between CPU and GPU memory. Note that we conducted the measurement with nvprof independently from the other measurements, and the overhead caused by nvprof did not affect the other execution times.

GC We measured the time for GC during the whole execution with the runtime option of GHC `-s`.

run^{A}, convert^{A}, compile^{A}, exec^{A}: We executed the whole program with GHC profiler and measured the *duration ratio* of the Haskell functions. Here, the duration ratio includes the duration of the function and that of the (other) functions called from the function. Note that the duration of functions may include the time for evaluating the parameters due to the lazy evaluation.

In fact, some other functions were executed, but the ratio of execution times for those functions were negligible as seen in the results.

3.4 Experiment Results

Tables 2 and **3** show the execution time of each step. The row (astdef^{H}+run^{A} – kern^{C}) in Table 2 shows the time for sr^{H} subtracted by the time for kern^{C} and copy^{H}. Note that the GC time is the time for GC to execute the program, and the time of each step includes part of the GC time. The columns named "Prev" and "CUDA" show the execution times of the programs developed in

Table 4 Number of iterations and execution time of sr^H (s).

	Prev	This
1	0.278	0.286
40	1.744	0.363
80	3.330	0.442
120	5.043	0.532
160	7.262	0.557
200	9.893	0.616
R^2 value	0.9869	0.9778

the previous work when executed in the new environment. The column named "This" shows the execution time of the program improved in this work. Although we introduce the improvement in the next section, we show the results in this table for comparison.

As we can see in Table 2, the program in the previous work spent most time executing $astdef^H$ and run^A, and the time executing $kern^C$ was relatively short. It also spent about half of the time executing GC. Although the $kern^C$ of the Accelerate program took 3.5 times more than that of the CUDA program, it was less than 2% of sr^H.

As we can see in Table 3, 80% of the execution time of run^A was for $convert^A$. As we described before, $convert^A$ is a preprocess for syntax trees.

Finally, **Table 4** shows the execution time of sr^H while changing the number of iterations of the image super-resolution process [*4]. The last row shows the R^2 value computed from the fitting linear line by the least squares method. From these values, we can confirm that the execution time is almost proportional to the number of iterations.

3.5 Consideration

From the experiment results, we can learn the following for the program in the previous work. As observed in the previous work, the overhead of the Accelerate program is not the execution of the PTX code. In particular, the main overhead is in the preprocess of syntax trees by the Accelerate runtime system. Since the whole execution time was proportional to the number of iterations, the overhead also increases as the number of iterations increases.

Here, we point out the fact that the syntax tree of the program in the previous work has a size proportional to the number of iterations. The syntax tree is defined in lines 5–6 in Fig. 1, and the the main part is as follows.

```
foldl1 (>->) [step iImg | _ <- [1..200]] oImg
```

The function `step` performs a single super-resolution process, i.e., a call of parallel primitive generate. The operator >-> takes two computations for an array and returns a syntax tree so that the result of the first computation is passed to the second computation. Therefore, the syntax tree has a size proportional to the number of iterations.

4. Cutting Overheads and Improved Performance

Based on the consideration in the previous section, we modify the Accelerate program so that the size of the syntax tree does not depend on the number of iterations. In this section, we show the modifications and the performance of the improved program.

[*4] We used a modified program in which the number of iterations was given as a parameter. The time for 200 iterations was not the same as that in Table 2, but the difference was less than 4%.

```
accSuperResolution iImg oImg =
    afst (awhile cond gen (lift (oImg, (unit 0))))
  where
    cond x =
      let (arr, i) =
        unlift x :: (Acc OImg, Acc (Scalar Int)) in
      (unit $ (the i <* lift loops))
    gen x =
      let (arr, i) =
        unlift x :: (Acc OImg, Acc (Scalar Int)) in
       lift (step iImg arr, unit (the i + 1))
```

Fig. 3 Program after modification.

4.1 Improved Accelerate Program

We modified the `accSuperResolution` function in the Accelerate program in the previous work as in **Fig. 3**.

In the modified code, we implemented the 200 times iteration of function `step` with the `awhile` syntax provided by the Accelerate DSL. The program `awhile cond body init` starts with `init` as the initial value of `v` and updates `v` with `body v` while `cond v` is true. Since `awhile` is a control structure provided in Accelerate, the size of syntax trees stays constant, and the iterations are performed in `CUDA.run`, which are the differences from the previous work.

To make the function take the output image (updated by `step`) and the number of iterations as a single input parameter, we described the construction and decomposition of a pair in the program in Fig. 3. Furthermore, the transforms between the pair in Haskell and the pair in Accelerate DSL make the definition complicated.

4.2 Performance

As shown in Table 2 in the previous section, the modified program ran 15 times faster than the program in the previous work and about 2 times slower than the CUDA program. The time for GC became about 1/34, which is due to the smaller size of the syntax tree. As shown in Table 3, the duration ratio of the preprocessing to syntax trees ($convert^A$), which was most of the execution time, became very small. Although the ratio of function $exec^A$ became larger, this was because the whole execution time was reduced.

5. Discussion

5.1 Overhead and Improvement in Other Applications

We discuss the overhead and its improvement in this work, which could be similar in other applications. In particular, we discuss the following two issues: (1) similar overhead was observed in applications that generate a large DSL syntax tree and (2) rewriting with `awhile` works well in those applications. Since generalizing these two issues is beyond the scope of this work, we only show some concrete examples.

Firstly, for (1), we expect that overhead is observed in wide-ranging applications because the size of syntax trees is proportional to the number of iterations in the applications that iterate GPU kernels (for example, a numerical solution to differential

equations). However, we cannot find a case that reports the overhead as far as we have surveyed.

Although three programs were used in the evaluation by Chakravarty et al. [1], none of them included a loop. Therefore, we consider that a large DSL syntax tree was not generated.

Parallel and Concurrent Programming in Haskell (*PCPH*) [3], (chapter 6) includes an example `fwaccel` that computes the shortest distance between two points in a graph with an iterative method. It generates a syntax tree whose size is proportional to the number of iterations. However, it reports briefly about the performance: the Accelerate program ran faster than the sequential program on a CPU. The fact that the overhead is related to the size of the syntax tree is not mentioned.

We found a discussion about implementation of iterative computation and its overhead in the Accelerate website [*5], but we cannot judge whether the issue in the discussion is similar to that in this work.

Since we found no similar overhead, we did not address issue (2). We rewrote the `fwaccel` program in PCPH above with `awhile` and confirmed that the execution time reduced to about 1/5. Therefore, we consider that similar overhead was in `fwaccel` and the improvement in this work worked well too.

5.2 Reasons for Using Expression-composition Operator

In the previous work, the expression-composition operator `>->` was used, and it generated a syntax tree of a size proportional to the number of iterations. Although Accelerate provides the `awhile` syntax used in the previous section, the authors in the previous work made a program that generated a large syntax tree for the following reasons.

Due to the hardware environment used in the previous work, limited combinations of the software were available. The Accelerate used in the previous work (version 0.13.0.3, published in May 11, 2013) did not provide the `awhile` syntax.

PCPH introduces an example with the expression-composition operator for the consecutive execution of parallel primitives but not the `awhile` syntax.

5.3 Possibility for Language Design Improvement

As we have seen in the previous sections, the problem is in the iterations described in the Accelerate DSL. We discuss the descriptions of those iterative computations with other possibilities.

- In the previous work, a syntax tree with N processes was generated for N iterations. A problem occurred due to the overhead being proportional to the size of the syntax tree in Accelerate. Not only Accelerate but also embedded DSLs generating and composing DSL syntax trees in the iteration of the host language may lead to the same problem. If the number of iterations depends on the result of computation, we require another description.
- In this work, we described the iteration with the `awhile` syntax in the Accelerate DSL. We reduced the overhead of Accelerate with this description. However, if the original program used a recursive function for the iteration, we need extra effort to rewrite it to an imperative control structure.
  For instance, in the previous work, the authors developed a super-resolution application with a sequential program and then parallelized it. In the sequential program, the iteration was defined with the following recursive function [*6].
  ```
  steps oImg k   =
    if k = 0 then oImg
             else steps (step iImg oImg) (k-1)
  ```
  It is not straightforward to rewrite this program with the `awhile` syntax.
- Let us consider introducing a syntax for recursive function as an improvement of DSL. For example, if we introduce the `afix` syntax, we can define the `steps` function as follows (`acond` is DSL syntax for conditional branch. We omit the `lift` and `unlift` operations for input parameters.).
  ```
  afix $ \steps (oImg, k) ->
    acond (k = 0) oImg
                  steps (step iImg oImg, k-1)
  ```
  We expect a similar amount of description would be needed compared with that with `awhile`. However, in most cases of parallel programming, we first develop a sequential program and then selectively parallelize the loops in the program. A description similar to sequential programs would simplify parallelization by trial and error or understanding of parallelized programs.
- We may consider, as a further improvement, automatic transformation of a program that performs iteration by copying syntax trees to another tree that uses `awhile`. This kind of optimization is too specific to be implemented as a process in the Haskell compiler. It is also nontrivial to implement it as an optimization in Accelerate without introducing extra overhead because the optimization in Accelerate was the reason for the overhead. Therefore, we consider that a practical programming style should provide hints by users with `awhile` or `afix` proposed above.

6. Conclusion

In this work, we have investigated the overhead of the image super-resolution application with Accelerate, which was developed by Matsumoto and Matsuzaki, and shown that the reason for the overhead is the preprocess of syntax trees, which have a size proportional to the number of iterations. We have also shown that the rewriting of the target iteration with the `awhile` syntax in DSL eliminates most of the overhead. As a result, Accelerate runs with an overhead of a factor of about two compared with CUDA-C even for practical applications like image super-resolution.

Acknowledgments This work was supported by JST, CREST, and JSPS KAKENHI 26330078.

References

[1] Chakravarty, M.M., Keller, G., Lee, S., McDonell, T.L. and Grover, V.: Accelerating Haskell array codes with multicore GPUs, *Proc. 6th Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pp.3–14, New York, NY, USA, ACM (online), DOI: 10.1145/1926354.1926358 (2011).
[2] Keller, G., Chakravarty, M.M., Leshchinskiy, R., Peyton Jones, S. and

[*5] https://github.com/AccelerateHS/accelerate/issues/305

[*6] To be exact, pattern matching was used instead of conditional expression.

Lippmeier, B.: Regular, Shape-polymorphic, Parallel Arrays in Haskell, *Proc. 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pp.261–272, New York, NY, USA, ACM (online), DOI: 10.1145/1863543.1863582 (2010).
[3] Marlow, S.: *Parallel and Concurrent Programming in Haskell*, Oreilly and Associates Inc. (2013).
[4] Matsumoto, T. and Matsuzaki, K.: Evaluation of Libraries for Parallel Computing in Haskell: A Case Study with a Super-resolution Application, *Journal of Information Processing*, Vol.25, pp.308–316 (2017).
[5] Miyazaki, R.: *A Study on Super Resolution Techniques for fMRI Images*, Bachelor Thesis, School of Information, Kochi University of Technology (2015). (In Japanese)

Izumi Asakura is a doctor course student at Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. He received his B.S. and M.S. degrees from Tokyo Institute of Technology in 2014 and 2016, respectively. His research interests include design of programming languages, program optimization and program verification.

Hidehiko Masuhara is a Professor at Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. He received his B.S., M.S., and Ph.D. degrees from the University of Tokyo in 1992, 1994 and 1999 respectively. Before joining Tokyo Institute of Technology, he served as an Assistant Professor, Lecturer, and Associate Professor at Graduate School of Arts and Sciences, the University of Tokyo. His research interests include design and implementation of programming languages and software development environments.

Takuya Matsumoto is a master-course student of Graduate School of Engineering, Kochi University of Technology in Japan. He received his B.E degree from Kochi University of Technology in 2016. His research interest is in parallel programming and large-scale graph processing.

Kiminori Matsuzaki is an Associate Professor of Kochi University of Technology in Japan. He received his B.E., M.S. and Ph.D. from The University of Tokyo in 2001, 2003 and 2007, respectively. He was an Assistant Professor (2005–2009) in The University of Tokyo, before joining Kochi University of Technology as an Associate Professor in 2009. His research interest is in parallel programming, algorithm derivation, and game programming. He is also a member of ACM, JSSST, IEEE.