Journal of Information Processing Vol.25 308–316 (Mar. 2017)

[DOI: 10.2197/ipsjjip.25.308]

Regular Paper

Evaluation of Libraries for Parallel Computing in Haskell — A Case Study with a Super-resolution Application

Takuya Matsumoto[1,a)] Kiminori Matsuzaki[2,b)]

Received: July 8, 2016, Accepted: October 25, 2016

Abstract: Haskell is a functional language featuring lazy evaluation and referential transparency. On one hand, Referential transparency is useful for parallel computing because the results do not depend on the evaluation order, but on the other hand, parallel computing requires an evaluation order that is different from that of lazy evaluation. There are some parallel programming libraries for Haskell, such as Repa (regular parallel arrays) and Accelerate. However, little research has been conducted on evaluation with real applications, and the usefulness of these libraries remains unclear. In this study, we evaluated the usefulness of parallel programming libraries for Haskell with an application that applies a super-resolution technique to fMRI images. We developed a CPU-based parallel program with Repa and GPU-based parallel program with Accelerate and compared their performance. We obtained reasonable speedups for the program with Repa, but not for the program with Accelerate. We also investigated Accelerate's performance issues with an implementation in C and CUDA and the log from the Accelerate program. In this paper, we report our findings through a case study, focusing on the advantages and difficulties in parallel program development with Haskell.

Keywords: parallel programming, Haskell, GPGPU

1. Introduction

Recently, hardware environments with accelerators such as GPUs as well as multicore CPUs are widely used, especially in high-performance computing. As these hardware environments are becoming more and more complex, the cost of developing efficient parallel programs that utilize these environments is also increasing. Under these circumstances, increasing the productivity of parallel programming is an important research topic, and several programming languages have been developed to achieve this aim such as X10 [4], Chapel [1], and XcalableMP [16]. Functional programming is often said to be highly productive due to the strong modularity of functional languages [8]. In the area of functional programming, several languages or libraries for parallel computing have been developed actively [3], [5], [9], [10], [11], [12], [14], [19], [21].

In this study, we use a functional programming language called Haskell [13]. Haskell has two important features: referential transparency and lazy evaluation. Referential transparency —*the evaluation results are independent of the evaluation order*— is very useful for parallel programming. Lazy evaluation —*only the required parts of the definition are evaluated*— is useful to write a program that manipulates infinite lists, which is hard to write in other languages. However, lazy evaluation is inherently sequential, and we require another evaluation strategy for actual parallel computation. Another important aspect of Haskell is that the user community has developed quite a few libraries. For instance, we have the following libraries for parallel computation: Control.Parallel [14] enables parallel computation by changing the evaluation order, Repa (regular parallel arrays) [9] helps parallel computation over multi-dimensional arrays, Accelerate [3] and Obsidian [21] enable parallel computing with GPUs, and Eden [11] provides parallel computation patterns as algorithmic skeletons.

How Haskell and its libraries raise the productivity of application development is an important question in programming research. The authors of these papers that proposed these libraries, of course, claimed their usefulness. In contrast, only two papers [17], [18] reported the experiences in actual application development and the performance of these libraries from an outsider's perspective. That is, the usefulness of parallel computing libraries in Haskell was not evaluated enough from the viewpoint of application development.

In this paper, we discuss the usefulness of parallel programming with Haskell and its libraries through a case study, in which we develop parallel programs of an existing application using two parallel programming libraries in Haskell: Repa and Accelerate. The target application (Section 3) is to apply a super-resolution technique to a set of four fMRI images [15]. The authors' group also used this application to evaluate GPU parallelization [20] and the optimization of sequential execution [7].

Important knowledge obtained from the case study is as follows.

- The Haskell program that was simply translated from the original Java program showed as good performance as the original.
- The Repa-based program achieved reasonable parallel

[1] Graduate School of Engineering, Kochi University of Technology, Kami, Kochi 782–8502, Japan
[2] School of Information, Kochi University of Technology, Kami, Kochi 782–8502, Japan
[a)] 205085u@gs.kochi-tech.ac.jp
[b)] matsuzaki.kiminori@kochi-tech.ac.jp

speedups on multicore CPUs (but the parallel speedups were smaller than linear speedups).
- The Accelerate-based program ran slower with GPUs than the Repa-based program did in sequential execution. Modifications such as inline expansion applied to some parts of programs made the Accelerate-based program run faster, but we consider such modifications would spoil the advantages of high-level functional programming with Accelerate.
- We guessed that the reason for lower performance was in Accelerate's automatic scheduling mechanism for the CUDA kernel code.
- The (latest) Accelerate library does not work in the latest CUDA environments, which makes it difficult to set up an environment for Accelerate.

The rest of the paper is organized as follows. Section 2 introduces the two Haskell parallel programming libraries used in the paper. Section 3 shows the basic algorithm of the target application that applies super-resolution to fMRI images and the original Java program. Section 4 explains how we developed parallel programs with the Repa and Accelerate libraries, and we discuss the porting cost. Section 5 evaluates the developed programs, including a hand-written one in C++ and OpenMP and another one in C and CUDA, in terms of the computing speed. Section 6 investigates the reason the Accelerate program was slower than sequential execution through several other experiments using debug tools. Finally, Section 7 reviews related work, and Section 8 concludes the paper.

2. Parallel Programming Library in Haskell Compared in This Study

2.1 Repa

Repa (regular parallel arrays) [9], [22] is a data-parallel programming library in Haskell and is especially useful for data-parallel programming in shared-memory environments. Usual Haskell programs manipulate lists, and the Repa library provides a mechanism for efficient manipulation of multi-dimensional arrays. More specifically, the Repa library enables generation of multi-dimensional arrays and parallel manipulation of those arrays.

When the target algorithm applies computation independently to every element of multi-dimensional arrays [*1], we can execute the computation in parallel with the `computeP` function provided by the Repa library. Here, we do not need to consider how to divide and distribute the data and tasks to threads.

Due to these features, we can obtain a shared-memory parallel program with the Repa library without large modifications from a normal (array-manipulating) sequential Haskell program.

2.2 Accelerate

Accelerate [2], [3] is an EDSL (embedded DSL) that enables GPU programming in Haskell and was proposed based on the earlier studies on GPU.Gen [10] and Repa [9]. The objective of Accelerate is to ease GPU programming in Haskell by concealing complex details of GPU execution.

The Accelerate library takes a Haskell program with some limitation and executes it on the GPU through CUDA. We can write Accelerate programs almost in the same manner as we write usual Haskell programs, except for the special types of elements of the multi-dimensional arrays on GPU devices and for the special operators [*2] for branches in the code for GPUs (kernel codes). In usual GPU programming, we need to explicitly write the code that transfers data between the host memory and the device memory, but with Accelerate we do not need to write it because the Accelerate library automatically does this. The kernel codes are compiled at runtime, and the compiled codes are cached to avoid the overhead of compilation.

3. Target Application

The target application in this study is to apply super-resolution [6] to fMRI images [15]. Super-resolution is an image processing technology that takes multiple input images of a certain resolution and reconstructs a higher-resolution image. The target application takes four fMRI images as input and reconstructs an image with twice the resolution in the horizontal and vertical directions.

The main reasons we selected this super-resolution application as our target application for the evaluation of parallel programming libraries are as follows.
- In this application, each pixel is updated independently with the values of neighboring pixels (i.e., this application is a stencil computation in a broad sense). Therefore, we can expect parallel speedups not only on multicore CPUs but also on GPUs.
- This application needs to deal with four static input fMRI images and a dynamically changing super-resolved image. This application is more complex than the benchmark applications evaluated so far [17], [18], and we consider that we can evaluate libraries in a more realistic setting.

3.1 Basic Algorithm of Super-Resolution

There are two approaches to super-resolution. One is to fill in the pixels by estimating the pixel values by pattern-matching or machine-learning techniques. The other is to take multiple observed images and reconstruct the original image so that statistical error is minimized. Here, we explain the latter, which is used in the target application.

When we take photos with some devices, the observed images are affected by several factors: motion of the objects, blur, (down-)sampling in the device, and noise (**Fig. 1**). Let y denote the observed image and x denote the image with the ideal resolution. With the geometric transformation for the object's motion M, transformation for blur B, transformation for down-sampling D, and noise n, we assume that the observed image y is given by the following equation:

$$y = DBMx + n \,. \tag{1}$$

[*1] Examples include the higher-order function `map` that applies a function independently to every element and stencil computations that take an index and compute the corresponding value from neighbor elements.

[*2] These operators have additional asterisks, for example, the Boolean sum operator is `||*`.

The goal of super-resolution is to compute the ideal high-resolution image x from the given multiple observed images y_i (**Fig. 2**). In general, we need to solve the inverse problem after estimating geometric transformations M_i and blur transformation B. In this study, based on the properties of the fMRI images (and the MR device), we assume for simplicity that geometric transformations M_i are given and blur B is the identity. In addition, we assume that the down-sampling reduces the resolution by half in the horizontal and vertical directions (we double the resolution in super-resolution).

However, this simple reconstruction, which minimizes the errors in the model above, generates an image with amplified noise due to overfitting. Therefore, in the target application, we iteratively update the value of each pixel so that the sum of the following error and penalty is minimized.

- (Error) The difference between the pixel value in the observed image and the pixel value in image $y_i' = DM_i x$ generated from the high-resolution image x.
- (Smoothness penalty) The difference between the pixel value and the average of the values of its four neighboring pixels in the high-resolution image x.

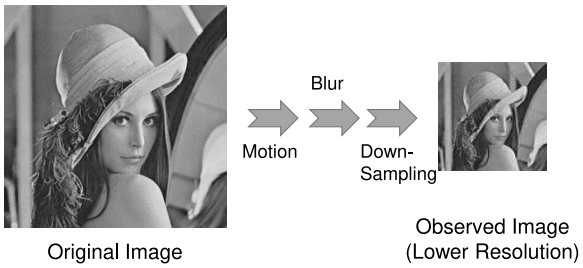

Fig. 1 Resolution decreases when image is observed.

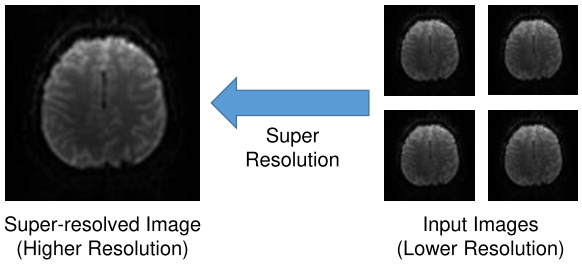

Fig. 2 Increase resolution by super-resolution.

3.2 Original Program in Java

We used a Java program developed by Miyazaki [15] as the original program for the target application. **Figure 3** shows the core part of the program.

Lines 6 and 7 in this program compute the adjusted position `lx` and `ly` for each low-resolution image. Lines 10–18 compute the average of pixel values corresponding to the high-resolution image and then the error `dEdp` from the pixel value in the low-resolution image. Lines 21–24 compute the smoothness penalty by comparing the pixel value with the average of four neighboring pixel values. Finally, line 26 updates the pixel value in the high-resolution image with the value of `dEdp`. A single step applies these computations to each pixel (for each `x` and `y`) in the high-resolution image, and we perform 200 steps to generate the output image.

Note that the input images are four 2D images with resolution 384×384. The MR device takes 36 slices of 64×64 images and we align those slices in 6×6 to obtain an input 2D image. The super-resolved image has a resolution of 768×768.

```
1  for (int x = 0; x < width; x++) {
2    for (int y = 0; y < height; y++) {
3      int dEdp = 0;
4      for (int l = 0; l < file_n; l++) {
5        int dx = param[l].x, dy = param[l].y;
6        int lx = (x + dx) / scale; if (lx < 0) continue; if (lx >= width/2)   continue;
7        int ly = (y + dy) / scale; if (ly < 0) continue; if (ly >= height/2) continue;
8        // Compute sum of corresponding region in high−resolution image
9        int hval = 0;
10       for (int hx = lx * scale − dx; hx < (lx + 1) * scale − dx; hx++) {
11         for (int hy = ly * scale − dy; hy < (ly + 1) * scale − dy; hy++) {
12           if (hx < 0 || hx >= width || hy < 0 || hy >= height) hval += 0;
13           else                                                 hval += org[hx + width * hy];
14         }
15       }
16       hval /= scale * scale;
17       int lval = img[l][lx + width/2 * ly];
18       dEdp += hval − lval;
19     }
20     // The generated image should be smooth
21     if (x > 0 && y > 0 && x < width − 1 && y < height − 1) {
22       int pn = (org[x−1 + width*y] + org[x+1 + width*y] + org[x + width*(y−1)] + org[x + width*(y+1)])/4;
23       dEdp += alpha * (org[x + width * y] − pn);
24     }
25     // updated the pixel value
26     nxt[x + width * y] = org[x + width * y] − (dEdp + (dEdp > 0 ? 2 : −2))/4;
27   }
28 }
```

Fig. 3 Core part of original super-resolution program implemented in Java. Variable `org` has last estimation of high-resolution image; `nxt` has updated estimation of high-resolution image. Four low-resolution images are given by `img`, and their positions are given by `param`.

4. Implementation of Parallel Programs in Haskell

4.1 Implementation Using Repa Library

Figure 4 shows the program implemented with the Repa library. In the implementation with the Repa library, we defined function `sr` for the process of super-resolution on a pixel and used the `computeP` function in the Repa library to apply function `sr` to all the pixels. The `computeP` function automatically applies the `sr` function in parallel with multiple cores.

Here, we explain the important points in Fig. 4.

`type OImg = Array U DIM2 Int` This defines the type of arrays for the output (super-resolved) images. `U` indicates that the boxing is not applied to the data.

`type IImg = Array U DIM3 Int` This defines the type of arrays for the input (low-resolution) images.

`repaSuperResolution :: IImg -> OImg -> OImg` This function computes super-resolution with the Repa library.

`step :: IImg -> OImg -> Int -> OImg` This function updates the overall image and is iterated for the specified times.

`sr :: DIM2 -> Int` This function takes a 2D index and returns the result value of the pixel at the specified index.

In function `sr`, lines 12, 16–23, and 27–30 come naturally from the corresponding parts of the Java program. The loops of `x` and `y` in the Java program are implemented by the function `computeP` in line 8, and the loop of `l` over images is implemented by the list comprehension in line 26. The iterations that update the whole image are described explicitly in the `step` function.

As we have seen above, we can implement the target application in Haskell using the Repa library at rather low cost.

4.2 Implementation Using Accelerate Library

Figure 5 shows the program implemented with the Accelerate library.

In the implementation with the Accelerate library, we need to define the arrays and variables used in the GPU processing to have specific types wrapped by `Acc` and `Exp` and to edit the program for some operations. The expression is partly limited in the function executed on GPUs. For instance, we cannot use normal Haskell lists, recursion, or iteration. Therefore, instead of the list comprehension used in the implementation with the Repa library, we expanded it and wrote down all the function applications (line 27). In the `sr` function that is executed on GPUs, we used dedicated operators for all the comparisons and replaced the branches with an expression similar to (Java-like) ternary operators. Although there were the differences listed above, the program with the Accelerate library was obtained without many modifications from that with the Repa library, and we consider that the development cost with the Accelerate library is rather small.

Execution of Program in Fig. 5

The program with the Accelerate library in Fig. 5 is executed as follows.

(1) Initialization of the Accelerate runtime.

(2) Compilation of the code executed on GPUs (the `step` function (lines 8–31 in Fig. 5)) into the kernel code in CUDA.

(3) Preprocessing by the Accelerate runtime. The authors believe that the calls of kernel code (lines 5–6 in Fig. 5) are expanded and scheduled to be executed on GPUs.

(4) Based on the scheduling above, the CUDA kernel code is

```
1  type OImg = Array U DIM2 Int
2  type IImg = Array U DIM3 Int
3
4  repaSuperResolution :: IImg -> OImg -> OImg
5  repaSuperResolution iImg oImg  = runIdentity $ step iImg oImg 0
6
7  step !iImg !oImg loops = return oImg
8  step !iImg !oImg k     = do oImg' <- computeP (fromFunction (Z :. oH :. oW) sr)
9                           step iImg oImg' (k+1)
10  where
11   sr :: DIM2 -> Int
12   sr (Z :. y :. x) = inOImg y x - (dEdp + if dEdp > 0 then 2 else -2) `quot` 4
13    where
14     srl (l, (dx, dy)) =
15       let
16         lx = (x + dx) `quot` 2;      ly = (y + dy) `quot` 2
17         hx = lx * scale - dx;        hy = ly * scale - dy
18         dso2i = (if hx    < 0 || hx    >= oW || hy    < 0 || hy    >= oH then 0 else inOimg hy hx) +
19                 (if hx+1 < 0 || hx+1 >= oW || hy    < 0 || hy    >= oH then 0 else inOimg hy (hx+1)) +
20                 (if hx    < 0 || hx    >= oW || hy+1 < 0 || hy+1 >= oH then 0 else inOimg (hy+1) hx) +
21                 (if hx+1 < 0 || hx+1 >= oW || hy+1 < 0 || hy+1 >= oH then 0 else inOimg (hy+1) (hx+1))
22         hval = dso2i `quot` 4
23         lval = iImg ! (Z :. l :. ly :. lx)
24       in
25         if lx < 0 || lx >= iW || ly < 0 || ly >= iH then 0 else hval-lval
26     ds = sum [srl param | param <- zip [0..3] [(1,1), (0,1), (1,0), (0,0)]]
27     pn = if x > 0 && y > 0 && x < oW - 1 && y < oH - 1
28       then 4 * (inOImg y x - (inOImg y (x-1) + inOImg y (x+1) + inOImg (y-1) x + inOImg (y+1) x) `quot` 4)
29       else 0
30     dEdp = ds + pn
31     inOImg y x = oImg ! (Z :. y :. x)
```

Fig. 4 Haskell code with Repa library.

```
type OImg = Array DIM2 Int
type IImg = Array DIM3 Int

accSuperResolution :: Acc IImg -> Acc OImg -> Acc OImg
accSuperResolution iImg oImg = foldl1 (>->) steps oImg
 where steps = [step iImg | _ <- [1..200]]

step :: Acc IImg -> Acc OImg -> Acc OImg
step iImg oImg = generate (shape oImg) sr
 where
  sr :: Exp DIM2 -> Exp Int
  sr ix = inOImg y x - (dEdp + (dEdp >* 0) ? (+2, -2)) `quot` 4
   where
    (Z :. y :. x) = unlift ix
    srl (l, (dx,dy)) =
     let
       lx = (x + dx) `quot` 2;       ly = (y + dy) `quot` 2
       hx = lx * scale - dx;         hy = ly * scale - dy
       dso2i = ((hx    <* 0 ||* hx    >=* oW ||* hy    <* 0 ||* hy    >=* oH) ? (0, inOImg hy hx)) +
               ((hx+1 <* 0 ||* hx+1 >=* oW ||* hy    <* 0 ||* hy    >=* oH) ? (0, inOImg hy (hx+1))) +
               ((hx    <* 0 ||* hx    >=* oW ||* hy+1 <* 0 ||* hy+1 >=* oH) ? (0, inOImg (hy+1) x)) +
               ((hx+1 <* 0 ||* hx+1 >=* oW ||* hy+1 <* 0 ||* hy+1 >=* oH) ? (0, inOImg (hy+1) (hx+1)))
       hval = dso2i `quot` 4
       lval = iImg ! (lift (Z :. l :. ly :. lx))
     in
       (lx <* 0 ||* lx >=* iW ||* ly <* 0 ||* ly >=* iH) ? (0, hval-lval)
    ds = srl (0, (1,1)) + srl (1, (0,1)) + srl (2, (1,0)) + srl (3, (0,0))
    pn = (x >* 0 &&* y >* 0 &&* x <* oW - 1 &&* y <* oH - 1) ?
          (4 * (inOImg y x - (inOImg y (x-1) + inOImg y (x+1) + inOImg (y-1) x + inOImg (y+1) x) `quot` 4), 0)
    dEdp = ds + pn
    inOImg y x = oImg ! (index2 y x)
```

Fig. 5 Haskell code with Accelerate library.

executed on GPUs (multiple times).

(5) Finally, the result is returned to where the Accelerate function was called from (this is not shown in Fig. 5).

During the execution of the program in Fig. 5, the preprocessing by the Accelerate runtime took much time. In Section 6, we investigate and consider the reason the Accelerate program was so slow.

Problem of Setting up Library

Although we could easily write the Accelerate program, we had to spend most of our efforts setting up the environment for Accelerate. The Accelerate library depends on other Haskell libraries such as cuda and c2hs, as well as the CUDA library for actual execution. In our experiments, we tried to install Haskell and Accelerate library version 0.15 (the latest) on an environment with CUDA library version 7.5 (the latest), but we could not execute the Accelerate program. Therefore, we tested other versions of the Accelerate and CUDA libraries and eventually succeeded in executing the program with CUDA library version 6.0 and Accelerate library version 0.13.

Downgrading the CUDA library may cause the problem that we cannot utilize new mechanisms of GPUs. In fact, when we executed the program, the CUDA library could not recognize the GPU device used and gave the warning "Unknown CUDA device" (but the program ran).

Another Way to Replace List Comprehension

In the Accelerate programs, the parts executed on GPU devices cannot include list comprehensions or high-order functions. This is a major disadvantage in functional programming. In this study, we wrote down all the elements since the number of elements is just four. When the number of elements is much greater, it causes a problem in terms of development/maintenance costs.

Table 1 Lines of source programs.

Implementation	Number of Lines
Java	50
Parallel Haskell with Repa	34
Parallel Haskell with Accelerate	36

Another way to resolve the problem is to use GPU-side arrays instead of the usual lists. Although programming with GPU-side arrays requires additional `pack` and `unpack` operations, it increases the maintenancability. The preliminary experiments showed that the program that replaced the four parameters with a GPU-side array ran slower by about 10%.

4.3 Discussion on Porting Cost

Table 1 shows the lines of programs implemented in this study.

We could implement the Repa and Accelerate parallel programs in fewer lines of code due to the characteristics of functional programming languages and loop processing with the map function. When we port applications from an imperative language like Java to Haskell, the differences of grammar or programming paradigm often become issues, but in this study it was rather straightforward. The porting between Accelerate and Repa is very easy, and this is an advantage when we test parallel processing on multicore CPUs and GPUs.

5. Performance Evaluation

5.1 Programs Evaluated and Environment

We conducted several experiments using a CPU-based parallel program with the Repa library and a GPU-based parallel program with the Accelerate library and compared them in terms of the ex-

Table 2 Hardware and OS environment.

CPU	Intel Xeon CPU E5-2620 v3 2.40–3.20 GHz 6 cores × 2 (hyper-threading off)
RAM	32 GB
GPU	GM107-400-A2 1020 MHz 640 CUDA cores
VRAM	2 GB
OS	Ubuntu 14.04 LTS

Table 3 Experiment results.

Implementation	Execution time (s)	Speedup
Java sequential	16.0	—
Haskell Repa (sequential)	14.3	—
Haskell Repa (parallel 2 cores)	9.87	1.44
Haskell Repa (parallel 4 cores)	5.69	2.51
Haskell Repa (parallel 8 cores)	2.87	4.98
Haskell Repa (parallel 12 cores)	2.40	5.96
Haskell Accelerate	18.4	0.78
C++ (sequential)	5.15	—
C++ (parallel 2 cores)	2.94	1.75
C++ (parallel 4 cores)	1.77	2.89
C++ (parallel 8 cores)	0.91	5.65
C++ (parallel 12 cores)	0.65	7.87
C and CUDA	0.21	24.4

ecution time. In addition to the original Java program, we used as opponents a sequential program in C++, its parallel version with OpenMP, and GPU-based parallel program in C and the CUDA library. Here, in the C++ programs and the CUDA program, the sizes of images were given as constants so that the compiler optimization worked [*3]. We did not apply optimization techniques on cache utilization such as the tiling technique.

Table 2 shows the hardware and OS environments used in the experiments. The systems and additional options for each language were as follows.

Repa GHC 7.6.3 + Repa 3.2.3.3

- Compilation: `-O2` (optimization), `-fllvm` (optimization)
- Runtime: `-qg` (invalidating parallel GC) [*4], `-H256M` (heap size)

Accelerate GHC 7.6.3 + Accelerate 0.13.0.3 + CUDA 6.0

- Compilation: `-O2` (optimization)
- Runtime: `-qg0` (parallel GC), `-H256M` (heap size)

Java Java 1.8

C++ GCC 4.8.4

- Compilation: `-O3` (optimization), `-std=c++0x` (language)

CUDA GCC 4.8.4 + CUDA 6.0

- Compilation: `-O3` (optimization)

5.2 Results and Discussion of Performance Evaluation

Table 3 and **Fig. 6** show the execution time excluding the data input/output. We conducted experiments using 1, 2, 4, 8, and 12 cores for the Repa program. In the case of execution using the single core, we used the `computeS` function instead of `computeP`. We call the execution using single core "Haskell Repa (sequential)".

The original Java sequential program was not that efficient, but the Haskell Repa program (sequential) ran in almost the same time as the original Java program. We confirmed the parallel speedups for the parallel program with the Repa library.

[*3] In our previous study [7], we confirmed that the compiler optimization worked well when we defined the values (such as the sizes of images) to be constant.

[*4] We tested options for the parallel GC, and the program ran fastest with the option that invalidates parallel GC.

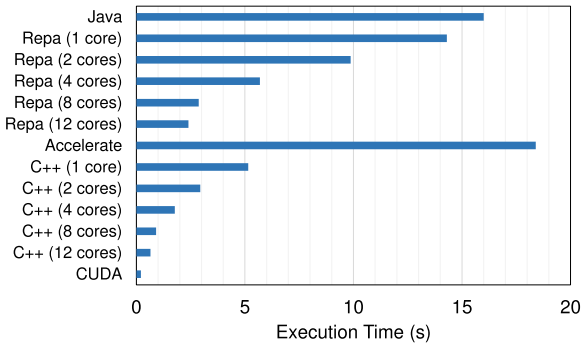

Fig. 6 Experiment results.

The parallel speedups in the 8-core case were 4.98 (for Repa) and 5.65 (for C++). We consider that we obtained enough parallel speedups using the Repa library, though the speedups were a bit smaller than expected. We consider the reasons we could not obtain the expected number of speedups were that we used sequential GC (since parallel GC had poorer absolute performance) and had less efficient cache utilization due to complex accesses to the memory.

The Accelerate program, in turn, took more time than the sequential implementation did. A possible reason was that the compiler optimization did not work well due to inappropriate descriptions of programs or algorithms in the Accelerate program. We will discuss this issue in more detail in Section 6.

6. Investigating Issues of Slower Accelerate Program

We considered the following issues to explain why the Accelerate program ran slower than the sequential Haskell program:

- branch divergence due to the many branches in the algorithm and the number of available threads,
- missing cache mechanism for the CUDA kernel, and
- overhead of the Accelerate library or runtime.

6.1 Branch Divergence and Number of Available Threads

The first issue regarding the branch divergence and the number of available threads comes from the mechanism of GPGPU computation. In GPGPU computation, the computation is executed in parallel by SIMD (single instruction multiple data) operations. When a program includes conditional branches, the GPUs need to execute both of the branches; thus, the overhead increases if the number and/or depth of branches increase. Since the target application includes many branches based on the coordinates and values of pixels, these branches would slow down. In terms of the number of available threads, it is important to use many threads to execute the SIMD operations efficiently.

The following two techniques are often used to resolve the branch divergence problem when many branches come from the boundary conditions. The first technique is to implement two processes: one for the boundary region and the other for the internal region without branches. The second technique is to remove the boundary conditions themselves by putting padding outside the region. Unfortunately, the current version of the Accelerate li-

Table 4 Results with NVIDIA Visual Profiler.

Implementation	Divergent branch	Number of threads
C and CUDA	19,967	1152×512
Accelerate	7,674	70×64

brary does not provide a function that applies a given function to part of the multi-dimensional arrays, and we cannot implement the program with either of these techniques.

To verify that the branch divergence and the number of available threads cause the slow-down problem, we compared the Accelerate program with the C and CUDA program using the NVIDIA Visual Profiler tool. **Table 4** shows the results from both programs.

The results in Table 4 show that the number of branches is larger in the C and CUDA program than in the Accelerate program. Because the C and CUDA program ran fast as we expected, we consider that the branch divergence is not the main reason for the slow down.

The number of available threads in the Accelerate program is much smaller than that in the C and CUDA program. The number of threads is not given by the user in Accelerate programs, and it depends on the implementation of the Accelerate library. If the number of threads is smaller than the number of GPU cores, the computation on the GPU side may slow down because we cannot fully utilize the parallelism. In fact, in terms of the execution time of the GPU, the Accelerate program took 12 times as much time as the C and CUDA program did. Therefore, the CUDA kernel code generated by the Accelerate library was less efficient than that written in C, and this is one of the reasons the Accelerate program ran slow.

6.2 Cache Mechanism for CUDA Kernel

The Accelerate library compiles functions (after filling the parameters) into CUDA kernels at runtime, and this compilation incurs unignorable overhead. Therefore, the Accelerate runtime caches the compiled kernel code and skips the compilation if the functions run in the same way as the cached ones. In the target application, the same function over the whole image is executed iteratively. When the function was compiled for each iteration, it caused a large overhead.

We executed the Accelerate program with the debug options of Accelerate runtime on and investigated how it ran. The three debug options we used were as follows.

`ddump-cc` This outputs the information of the CUDA kernel generated by the Accelerate library, number of threads, and resource.

`ddump-gc` This outputs the information of the garbage collection when the Accelerate program runs.

`ddump-exec` This outputs the information of the execution of the CUDA kernel.

By investigating the log from these experiments, we found the following facts.

(1) The Accelerate library compiled the Haskell code into the CUDA kernel the fewest times.

(2) The log recorded the memory management operations (by the Accelerate library) before the GPU device ran, and the cost of memory management was very large.

From fact (1), we confirmed that the caching mechanism of CUDA kernels worked well. We will discuss fact (2) in the following section.

6.3 Overhead of Accelerate Library

Fact (2) above, which we found by investigating the log of the Accelerate program, is a very important clue. The log with the debug option for Accelerate's garbage collection `ddump-gc` showed that 70% of the execution time was consumed for the process on the Haskell (CPU) side. In particular, look-up operations were executed on some arrays during the execution, and the number of operations was proportional to the number of iterations of the super-resolution process. We, however, could not determine exactly what this operation did.

To investigate the reason the program in Fig. 5 slowed down, we wrote the following three simpler Accelerate programs and measured the execution time.

AC-small1 This program increments the value of each pixel independently 200 times.

AC-small2 This program updates the value of each pixel based on the values of neighboring pixels 200 times.

AC-small3 This program takes four input arrays and computes the value for each pixel by reading the corresponding value in those arrays 200 times.

The execution times were 0.89 s for AC-small1, 2.58 s for AC-small2, and 1.50 s for AC-small3. These were much less than the execution time of 18.4 s for the program in Fig. 5. From these results, we consider that the computation itself in the target application can be executed efficiently with the Accelerate library.

The execution time for AC-small2 and AC-small3 differed greatly, even though the number of read/write/arithmetic operations was similar. From this fact, we expected that the (notational) complexity of the Accelerate programs affected the execution time especially for the preprocessing executed by the Accelerate runtime. We transformed the program in Fig. 5 where we applied the inline expansion by hand for the function `srl` in line 27. Then, the overall execution time shortened significantly to 11.6 s. This result suggested that in the preprocessing, the Accelerate runtime performed scheduling of CUDA kernel execution (and maybe functions as units).

7. Related Work

7.1 Other Haskell Libraries for Parallel Computing

7.1.1 Control.Parallel (Parallel Haskell)

One of the important features of Haskell is lazy evaluation. With the lazy evaluation strategy, subexpressions are evaluated in the order required, and the computation is inherently executed in a sequential manner. To enable parallel computing in Haskell, we need some mechanisms to allow evaluation in a different order from that of the lazy evaluation.

Control.Parallel [14] is a library that supports changing the evaluation order. Users can specify the parts to be executed in parallel, using basically two keywords: `par` and `seq`. The Control.Parallel library enables parallel computing in a similar way to the task parallelism of the fork-join model. Users can also spec-

ify some strategies to control the evaluation of the subexpressions with `par`.

7.1.2 Obsidian

Obsidian [21] is another Haskell library for GPU programming. The design of Obsidian is quite different from that of Accelerate: in Obsidian, users can specify low-level GPU manipulation. For example, users can distinguish between arrays in the host memory and in the device memory; they should specify explicit data transfer between the host and the device. This increases the cost of program development but has more potential to exploit the performance of GPUs.

7.1.3 Eden

The Eden library [11] provides semiexplicit channels between (distributed) processes in Haskell and enables parallel computing in a task-parallel manner. In particular, it provides several patterns of inter-process communication through channels as algorithmic skeletons. By using algorithmic skeletons, we can develop data-parallel applications rather easily on Eden.

7.2 Comparison of Parallel Programming Libraries

As far as the authors know, the following two studies compared Accelerate and Obsidian, which are GPU programming libraries developed in different approaches. Ouwehand compared the libraries in terms of expressiveness and performance with matrix multiplication as the target application [17]. Sadde did a case study with multiple applications including simple reduction [18]. Both studies reported that program development is easier with Accelerate and the performance is better with Obsidian. Sadde also stated that Accelerate is very useful when we port existing Haskell programs and Obsidian is very useful when we newly develop applications.

8. Conclusion

In this study, we evaluated the usability of two parallel Haskell libraries, Repa and Accelerate, through the implementation of a super-resolution application.

The Repa program ran (sequentially) on a single core as fast as the original Java program did. The relative speedups with respect to the number of cores was a factor of 4.98 with eight cores (88% of the factor of C++, 5.65). Although we have not obtained linear speedups, we consider that we obtained reasonable speedups since the speedups of the target application are bound by the memory speed. Since the development cost of Repa programs is small enough, we consider that Repa has high usability.

In contrast, there remains several issues for parallel programming with GPUs with Accelerate. The Accelerate program in Fig. 5 took more time than the sequential Haskell program did. The main problems for this slow down are the following. Firstly, due to the lower quality of the CUDA code compiled from the Accelerate program and fewer available threads, the execution time of the GPU was 12 times longer. Secondly, the Accelerate runtime took quite a long time for preprocessing before the GPU execution. We found that the time of preprocessing depends on how we write Accelerate programs from the fact that it was reduced by inline expansion by hand. In terms of the development cost, Accelerate has an advantage that programs can be easily ported from Repa programs. However, there remain some issues: the use of list comprehension is limited and no function is available to apply functions to part of arrays.

With Accelerate, users can develop a program combining abstract computing patterns provided as higher-order functions. Concrete execution with GPUs, such as data assignment and communication between the host and devices, are handled in Accelerate. We found from the experiments in this study that it may take a long time for the preprocessing by the Accelerate runtime (more than that for the GPU processing). In the target application, the input (low-resolution) images do not change, and we can reuse the buffers for high-resolution images by the so-called double-buffering technique. However, we cannot express these programmers' knowledge in the current Accelerate programs.

Our future work is to evaluate the performance improvement with another GPU-programming library, Obsidian. With Obsidian, we can describe rather low-level GPU manipulations that Accelerate conceals, but in turn we need to compare them with direct CUDA programming in terms of the development and/or maintenance costs.

Acknowledgments This work was partially supported by Ministry of Internal Affairs and Communications, Strategic Information and Communications R&D Promotion Programme (SCOPE) and JSPS KAKENHI No. JP25330088.

References

[1] Balaji, P. (ed.): *Programming Models for Parallel Computing*, chapter 6, The MIT Press (2015).
[2] Chakravarty, M.M.T., Clifton-Everest, R., Keller, G., Lee, S., Lever, B., McDonell, T.L., Newtown, R. and Seefried, S.: Accelerate: An embedded language for accelerated array processing, available from ⟨https://hackage.haskell.org/package/accelerate⟩ (2016).
[3] 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, ACM (2011).
[4] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing, *Proc. 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (*OOPSLA '05*), pp.519–538, ACM (2005).
[5] Fluet, M., Rainey, M., Reppy, J. and Shaw, A.: Implicitly-threaded parallelism in Manticore, *Journal of Functional Programming*, Vol.20, No.5–6, pp.537–576 (2010).
[6] Greenspan, H.: Super-resolution in medical imaging, *The Computer Journal*, Vol.52, No.1, pp.43–63 (2009).
[7] Hatanaka, R. and Matsuzaki, K.: Evaluation of Program Optimization — A Case Study with Super-Resolution Application, Poster (Category 3), *18th Programming and Programming Language Workshop* (*PPL2016*) (2016). (in Japanese)
[8] Hughes, J.: Why Functional Programming Matters, *Computer Journal*, Vol.32, No.2, pp.98–107 (1989).
[9] Keller, G., Chakravarty, M.M., Leshchinskiy, R., Peyton Jones, S. and Lippmeier, B.: Regular, Shape-polymorphic, Parallel Arrays in Haskell, *Proc. 15th ACM SIGPLAN International Conference on Functional Programming* (*ICFP '10*), pp.261–272, ACM (2010).
[10] Lee, S., Chakravarty, M.M.T., Grover, V. and Keller, G.: GPU Kernels as Data-Parallel Array Computations in Haskell, *Proc. Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods* (*EPHAM 2009*) (2009).
[11] Loogen, R., Ortega-mallén, Y. and na marí, R.P.: Parallel Functional Programming in Eden, *Journal of Functional Programming*, Vol.15, No.3, pp.431–475 (2005).
[12] Loulergue, F., Gava, F. and Billiet, D.: Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction, *Proc. International Conference on Computational Science* (*ICCS 2005*), Springer, pp.1046–1054 (2005).
[13] Marlow, S.: Haskell 2010 Language Report, available from

⟨https://www.haskell.org/onlinereport/haskell2010/⟩ (2010).
[14] Marlow, S.: *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*, Oreilly & Associates Inc. (2013).
[15] Miyazaki, R.: A Study on Super Resolution Techniques for fMRI Images, Bachelor Thesis, School of Information, Kochi University of Technology (2015). (in Japanese)
[16] Nakao, M., Lee, J., Boku, T. and Sato, M.: Productivity and Performance of Global-View Programming with XcalableMP PGAS Language, *The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (*CCGrid 2012*), pp.402–409 (2012).
[17] Ouwehand, C.: GPU Programming in Functional Languages, available from ⟨http://www.cse.chalmers.se/˜joels/writing/GPUFL.pdf⟩ (2013).
[18] Sadde, A.: Functional GPU Programming: A Comparison between Obsidian and Accelerate, Poster and oral presentation at the Student Research Competition, *20th ACM SIGPLAN International Conference on Functional Programming* (*ICFP '15*) (2015).
[19] Scholz, S.-B.: Single Assignment C — Efficient Support for High-level Array Operations in a Functional Setting, *Journal of Functional Programming*, Vol.13, No.6, pp.1005–1059 (2003).
[20] Soler Ferrer, J.L. and Matsuzaki, K.: MRI Image Processing with OpenCL, *Proc. 32nd JSSST Conference* (2015).
[21] Svensson, J., Sheeran, M. and Claessen, K.: Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors, *Proc. 20th International Conference on Implementation and Application of Functional Languages* (*IFL '08*), pp.156–173, Springer-Verlag (2011).
[22] The DPH Team: Repa: High performance, regular, shape polymorphic parallel arrays, available from ⟨https://hackage.haskell.org/package/repa⟩ (2016).

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.

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.