An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo

doi: 10.1007/s10766-013-0263-8

Noname manuscript No.
(will be inserted by the editor)

An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo

Kento Emoto · Kiminori Matsuzaki

Received: date / Accepted: date

Abstract Skeletal parallel programming is a promising approach to easy parallel programming in which users can build parallel programs simply by combining parts of a given set of ready-made parallel computation patterns called skeletons. There is a trade-off for this easiness in the form of an efficiency problem caused by the compositional style of the programming. One solution to this problem is fusion transformation that optimizes naively composed skeleton programs by eliminating redundant intermediate data structures. Several parallel skeleton libraries have automatic fusion mechanisms. However, there have been no automatic fusion mechanisms proposed for variable-length list (VLL) skeletons, even though such skeletons are useful for practical problems. The main difficulty is that previous fusion mechanisms are not applicable to VLL skeletons, and so the fusion cannot be completed. In this paper, we propose a novel fusion mechanism for VLL skeletons that can achieve both an easy programming interface and complete fusion. The proposed mechanism has been implemented in our skeleton library, SkeTo, by using the expression templates technique, experimental results have shown that it is very effective.

Keywords Algorithmic skeletons · Fusion equipped library

1 Introduction

As the use of parallel computers becomes more widely spread, parallel programming has become increasingly more important. However, in general, par-

K. Emoto
Kyushu Institute of Technology
680-4 Kawazu, Iizuka, Fukuoka 820-8502, Japan
E-mail: emoto@ai.kyutech.ac.jp

K. Matsuzaki
Kochi University of Technology
185 Miyanokuchi, Tosayamada, Kami, Kochi 782-8502, Japan
E-mail: matsuzaki.kiminori@kochi-tech.ac.jp

allel programming is much more difficult than sequential programming, because programmers have to consider extra factors such as complicated scheduling of tasks, data distribution, communication and synchronization between processors, etc. There is therefore a demand for easy parallel programming.

Skeletal parallel programming has been proposed and studied as a promising approach to easy parallel programming in which users build parallel programs by combining parts of a given set of ready-made parallel computation patterns called *skeletons* [6–8]. These include a map to apply a function to every element of a list and a reduce to take the summation of a list with a binary operator. For example, we can easily build a parallel program for computing the variance of list `x` with its average `ave` by using the skeletons with user-defined simple functions `plus`, `square`, and `sub` as

```
double var = reduce(plus, map(square, map(sub(ave), x)));
```

Although they enjoy easiness of programming, skeleton programs suffer from inefficiency caused by the production of intermediate data structures between successive skeletons due to the compositional style of the programming. For example, the skeleton program above has three local loops for two maps and the final reduce, and two intermediate lists are produced between successive loops, although we can compute the variance sequentially in a single loop.

Fusion transformation has been studied and used to optimize skeleton programs by removing redundant intermediate data structures, which dramatically improves the efficiency of naively composed skeleton programs. Indeed, optimization mechanisms based on fusion transformation have been implemented in several skeleton libraries and systems [5–7], including our own library, SkeTo[1], and fusion transformation has been shown to be actually effective. For example, although the skeleton program above appears to have three loops, it can be optimized into the following SPMD program with a single loop followed by the final global communication.

```
double r = 0.0;
for(int i = 0; i < x.local_size(); i++)
  r = plus(r, square(sub(ave)(x.local_get(i))));
global_reduce(plus, r);
```

Variable-length list (VLL) skeletons, such as concatmap to concatenate the results of applying a function to every element and filter to discard elements that do not satisfy a given predicate, are useful in practice [10]. These skeletons generate lists with different lengths from those of the input. For example, we can easily build a parallel program for the n-queen problem [10] by using these two skeletons:

[1] http://sketo.ipl-lab.org/

```
dist_list<board> x; x.push_back(emptyBoard);
for(int i = 0; i < n; i++)
  x = filter(invalidBoard, concatmap(putNewQueen, x));
long answer = x.length();
```

Starting from an empty board, the program repeatedly generates a list of new valid boards by putting one more queen in every board in the current list. In each iteration, it first generates all possible boards by using concatmap with `putNewQueen` to generate a list of new boards, each of which has a new queen in the top row. It then discards invalid boards by using filter with `invalidBoard` that returns `true` if the given board contains no collision of queens. Although this program is clear, it is inefficient due to the intermediate list generated between concatmap and filter. We hope that an automatic fusion mechanism can improve the efficiency.

Although they have great potential, no fusion mechanism has been proposed for VLL skeletons, mainly because previous fusion mechanisms for fixed-length list skeletons cannot be applied to VLL skeletons. Therefore, naively composed programs with VLL skeletons suffer from inefficiency caused by redundant intermediate data structures.

In this study, we designed and implemented a novel fusion mechanism for VLL skeletons that enables users to take advantage of both VLL skeletons and automatic fusion transformations to obtain efficient parallel programs for various problems in an easy way. We offer three main technical contributions:

- We propose a novel design for a *collector-based fusion mechanism* that enables both a simple programming interface and complete fusion results, which cannot be achieved by the previous mechanisms.
- The new fusion mechanism is implemented by using expression templates [11], so users need only a C++ compiler to use it.
- Our proposed mechanism can be used together with previous fusion mechanism in SkeTo, thus significantly broadening the application range of fusion optimization.

The rest of this paper is organized as follows. Section 2 reviews our previous fusion mechanism for fixed-length list skeletons, and in Section 3 we review VLL skeletons, the target of our proposed fusion mechanism. Section 4 discusses several approaches to designing a fusion mechanism for VLL skeletons, and Section 5 presents and evaluates an implementation of the proposed mechanism. Finally, Section 6 discusses related work, and in Section 7 we conclude the paper.

2 Preliminaries

After introducing the notation we used for formal discussion, we briefly review the previous fusion mechanism for fixed-length list skeletons. It is worth noting

that the skeletons in this paper are based on data-parallelism and that we use single program, multiple data (SPMD) programs for their implementation.

The notation used in this paper is reminiscent of Haskell [2]. Function application is denoted by a space and the argument can be written without brackets, so $f\ a$ means $f(a)$ in ordinary notation. Functions are curried: they always take one argument and return a function or a value, and the function application associates to the left and binds more strongly than any other operator, so $f\ a\ b$ means $(f\ a)\ b$ and $f\ a \otimes b$ means $(f\ a) \otimes b$. Function composition is denoted by $\circ$, and $(f \circ g)\ x = f\ (g\ x)$ according to its definition. Binary operators can be used as functions by sectioning, as $a \oplus b = (a\oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$. A list is denoted by enclosing its elements with square brackets, e.g., $[a]$ represents a singleton list of element a, and $[a, b, c]$ a list of elements a, b, and c. The list concatenation operator is denoted by $+\!\!+$, so that $[a, b] +\!\!+ [c, d] = [a, b, c, d]$. An empty list is denoted by $[\,]$.

2.1 Fixed-Length List Skeletons and Their Fusion

Here, we briefly review a small subset of our fixed-length list (FLL) skeletons [7]. Their intuitive definitions are

$$\begin{aligned} \mathsf{map}\ f\ [a_1, \ldots, a_n] &= [f\ a_1, \ldots, f\ a_n] \\ \mathsf{reduce}\ (\oplus)\ [a_1, \ldots, a_n] &= a_1 \oplus \cdots \oplus a_n \end{aligned}$$

The skeleton map applies the given function f to every element of the given list to produce the new list. The skeleton reduce takes a summation of the given list by using the given associative binary operator $\oplus$.

For example, if we want to take a summation of a given list after doubling its even numbers, we can easily make the following parallel program for this by combining the skeletons. Here, map doubles even numbers by applying user function *evenDbl*, and reduce takes the summation of the results.

$$\begin{aligned} &\mathit{evenDblSum} = \mathsf{reduce}\ (+) \circ \mathsf{map}\ \mathit{evenDbl} \\ &\quad \textbf{where}\ \mathit{evenDbl}\ a = \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ a + a\ \textbf{else}\ a \end{aligned}$$

We designed our FLL skeletons based on a special recursive function called *homomorphism* so that they have good optimizability by fusion. The intuitive definition of homomorphism is given as

$$\mathsf{hom}\ (\oplus)\ f\ [a_1, \ldots, a_n] = f\ a_1 \oplus \cdots \oplus f\ a_n$$

It is easily seen that map and reduce are its special cases. Homomorphisms have good fusability [1], and thus our skeletons have good fusability too.

For example, we have the following fusion rules for the skeletons:

$$\begin{aligned} \mathsf{map}\ f \circ \mathsf{map}\ g &= \mathsf{map}\ (f \circ g) \\ \mathsf{reduce}\ (\oplus) \circ \mathsf{map}\ f &= \mathsf{hom}\ (\oplus)\ f \end{aligned}$$

In each rule, the left-hand side has two skeletons with an intermediate list between them and the right-hand side has only one skeleton (homomorphism)

and no intermediate list. The right-hand side is therefore expected to be faster than the left-hand side, and indeed, this has been shown to be true in experiments [7]. For example, we can write the example skeleton program *evenDblSum* to obtain a faster program $evenDblSum_{opt} = \mathsf{hom}\ (+)\ evenDbl$ by using the second rule.

2.2 Implementation of FLL Skeleton Fusion via Expression Templates

The fusion of FLL skeletons has been implemented in our skeleton library, SkeTo [7], by using expression templates (ETs) [11] with an index-based access method. Here, we briefly review the mechanism by using the following example user code implementing the example program *evenDblSum*. It uses `map` and `reduce` skeletons with the STL `plus` operator and user-defined function object `evenDbl` that implements *evenDbl*, in which `evenDbl` extends the base class to inform the library of its function type.

```
long evenDblSum(dist_list<long> z) {
  return reduce(plus<long>(), map(evenDbl, z));
}
struct evenDbl_t : function_base<long(long)> {
  long operator()(long a) const { return even(a)? a + a : a;}
} evenDbl;
```

In the ET-based library, production of the resulting list is postponed and a skeleton returns an *expression object* representing its computation so that it (the computation) can be fused into successive computations. Figure 1 shows the implementation of the skeleton `map`. It is defined to build an object of `MapObj` that has two fields, `f` and `x`, to represent the computation of $\mathsf{map}\ f\ x$. The object also has an index-based access method `local_get` that returns the result of applying `f` to the ith element of `x`, which is the ith element of the resulting list computed by this expression. This method is used to generate elements *on demand*, thus avoiding storing the intermediate results and leading to the fusion.

A skeleton like `reduce` that does not produce a list receives an expression object and carries out the *fusion* in its computation. Figure 2 shows the implementation of the skeleton `reduce`. It is implemented as a single local loop followed by a global communication. In the local loop, it utilizes the method `local_get` to get the ith element of the given expression `x`. For example, in the program `evenDblSum`, the function `reduce` receives an object `MapObj(evenDbl, z)` built by `map(evenDbl, z)`, and thus the whole code becomes

```
long r = 0;
for(int i = 0; i < z.local_size(); i++)
  r = plus<long>()(r, evenDbl(z.local_get(i)));
global_reduce(plus<long>(), r);
```

```
template <typename F, typename X>
struct MapObj {
  const F& f; const X x;
  MapObj(const F& f, const X& x) : f(f), x(x) { }
  typename F::result_type local_get(int i) const { return f(x.local_get(i)); }
  int local_size() const { return x.local_size(); }
};
template <typename F, typename X>
MapObj<F, X> map(const F& f, const X& x) { return MapObj<F, X>(f, x); }
```

Fig. 1 ET implementation of skeleton map, which returns an expression object `MapObj`.

```
template <typename OP, typename X>
typename OP::result_type reduce(const OP& op, const X& x) {
  typename OP::argument_type r = identity_element<OP>::val;
  for(int i = 0; i < x.local_size(); i++) r = op(r, x.local_get(i));
  global_reduce(op, r);
  return r;
}
```

Fig. 2 ET-based SPMD implementation of skeleton reduce, which does fusion by using the index-based access method `local_get`.

This code does not produce the resulting list of *map evenDbl z*, rather it implements the fused computation $evenDblSum_{opt}$.

An important observation here is that the fused code uses the user-defined function object `evenDbl` *as is*. This point makes the fusion mechanism simple enough to be implemented by the index-based access method. Unfortunately, this does not hold true for variable-length list skeletons.

3 Variable-Length List Skeletons

In this section, we introduce variable-length list (VLL) skeletons [10] and briefly show some examples and their programming interface.

3.1 Definition and Example Use

Intuitive definitions of the VLL skeletons are given as

$$
\begin{aligned}
&\mathsf{concatmap}\ f\ [a_1, \ldots, a_n] = f\ a_1 +\!\!+ \cdots +\!\!+ f\ a_n \\
&\mathsf{filter}\ p\ [a_1, \ldots, a_n] \quad = [a_{i_1}, \ldots, a_{i_k}] \\
&\qquad \textbf{where}\ (\forall i, i \notin \{i_1, \ldots, i_k\} \Leftrightarrow p\ a_i = \mathsf{false}) \wedge (\forall j, i_j < i_{j+1}) \\
&\mathsf{append}\ x\ y \quad = x +\!\!+ y
\end{aligned}
$$

The skeleton concatmap, taking a function f to produce a list from the given argument, applies f to every element of the given list and concatenates the resulting lists. The skeleton filter, taking a predicate (a function returning a

```
dist_list<board> x; x.push_back(emptyBoard);
for(int i = 0; i < n; i++) x = filter(invalidBoard, concatmap(putNewQueen, x));
long answer = x.length();
```

Fig. 3 Examples use of VLL skeletons: n-queens problem.

```
dist_list<int> quicksort(const dist_list<int> &x) {
  if(x.get_global_size() < 2) return x;
  int pv = x.get(0);
  return append(quicksort(filter(less_than(pv), x))
                append(filter(equal(pv), x),
                       quicksort(filter(greater_than(pv), x))));
}
```

Fig. 4 Examples use of VLL skeletons: Quicksort.

Boolean value) and a list, removes its elements not satisfying the predicate. It is easily seen that filter p = concatmap $(\lambda a.\textbf{if}\ p\ a\ \textbf{then}\ [a]\ \textbf{else}\ [\,])$. The skeleton append simply concatenates the two given lists.

VLL skeletons are useful in practice [10], broadening the application range of skeletal parallel programming. For example, we can easily build a parallel program for the n-queen problem by using two skeletons, as shown in Fig. 3. Here, given a board, `putNewQueen` generates a list of new boards, each of which has a new queen in the top row, and `invalidBoard` returns `true` if the given board contains no collision of queens. In general, we can easily implement a parallel breadth-first search in a similar way.

Another important application of VLL skeletons is irregular divide-and-conquer algorithms, including the convex hull, the quicksort, and others. For example, the quicksort is implemented by using filter and append, as shown in Fig. 4.

3.2 Programming Interface of Naively Implemented VLL Skeletons

Here, we briefly review the programming interface of concatmap proposed in our previous work [10], in which the VLL skeletons are implemented naively without fusion.

The interface of the skeleton concatmap is

```
template<typename F, typename T, typename S>
dist_list<T> concatmap(const F&f, const dist_list<S> &l);
```

Here, the function object `f` (of type `F`) is expected to return an instance of `vector<T>`, as function f in concatmap f returns a list.

For example, if we want to duplicate every even number in a given list `x`, we can use `concatmap` with the user-defined function object `evenDup` (Fig. 5) of user function $evenDup\ a = \textbf{if}\ even\ a\ \textbf{then}\ [a, a]\ \textbf{else}\ [a]$ as follows.

```
struct evenDup_t {
  vector<long> operator()(long a) const {
    vector<long> v;
    if(even(a)) { v.push_back(a); v.push_back(a) } else { v.push_back(a); };
    return v;
  }
} evenDup;
```

Fig. 5 Vector-based implementation of *evenDup* $a =$ **if** *even* a **then** $[a, a]$ **else** $[a]$.

```
x = concatmap(evenDup, x);
```

The function object `evenDup` implements straightforwardly the function *evenDup* in the functional style: it simply returns a vector of one or two elements, as *evenDup* does. Since the functional style has been shown to be suitable for parallel programming [6–9] and our skeletons have been designed in the functional style, we conclude that this simple programming interface is effective.

4 Fusion Mechanism for Variable-Length List Skeletons

In this section, we discuss three approaches to designing the fusion mechanism of the VLL skeletons in order to find the one that achieves the best programmability and efficiency. Since filter is a special case of concatmap, and append simply concatenates two given lists, we focus on a fusion mechanism for concatmap.

4.1 Target Fusion Transformation

First, we clarify our target fusion transformation of concatmap by using the example program *evenDupSum*. Given a list, it first duplicates every even number in the list by using concatmap with user function *evenDup* and then creates a summation of the resulting list by using reduce.

$$\begin{aligned} \mathit{evenDupSum} = {} & \mathsf{reduce}\ (+) \circ \mathsf{concatmap}\ \mathit{evenDup} \\ & \textbf{where}\ \mathit{evenDup}\ a = \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ [a, a]\ \textbf{else}\ [a] \end{aligned}$$

It is easily seen that *evenDupSum* is equivalent to *evenDblSum* in Section 2.1.

The intermediate list that the *evenDupSum* program generates between reduce and concatmap seems inefficient, so we want to fuse these skeletons to obtain an efficient program. What should the resulting program of fusion be? Since *evenDupSum* is equivalent to *evenDblSum*, we expect the result of fusion to be the following $\mathit{evenDupSum}_{opt}$, which is the same as $\mathit{evenDblSum}_{opt}$, which does not produce any intermediate list:

$$\begin{aligned} \mathit{evenDupSum}_{opt} = {} & \mathsf{hom}\ (+)\ \mathit{evenDup}' \\ & \textbf{where}\ \mathit{evenDup}'\ a = \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ a + a\ \textbf{else}\ a \end{aligned}$$

The goal of our fusion mechanism is to obtain an efficient $evenDupSum_{opt}$ from the naive *evenDupSum*, but this fusion has a problem that did not appear in the previous fusion (Section 2.1).

The problem of this fusion is that in the fused program $evenDupSum_{opt}$, the user function *evenDup* is *not used as is*, which means that the fusion mechanism needs to create the new function $evenDup'$ from the definition of *evenDup* and +. However, in many programming languages it is difficult to obtain the body of a user function and create a new function from it, so the fusion mechanism has to use a user function as is. Therefore, we need a certain trick to define a user function to implement a fusion mechanism for VLL skeletons. This situation is quite different from that of the FLL skeletons, in which the fusion mechanism can use a user function as is. This is one of the reasons the previous fusion mechanism is not applicable to VLL skeletons.

In the following sections, we discuss three approaches to the fusion mechanism of VLL skeletons, focusing on how users define their functions and what code a fusion mechanism can produce, e.g., from the following main user code for *evenDupSum*:

```
long evenDupSum(dist_list<long> z) {
  return reduce(plus<long>(), concatmap(evenDup, z));
}
```

It is worth noting that in ET implementation, the skeleton `concatmap` produces an expression object holding the user function `evenDup` and the input `z`, and the skeleton `reduce` receives the object and uses it in the main loop.

4.2 Vector-based Approach

In this approach, a user function f used in `concatmap` f is implemented by a function object `f` that returns a concrete vector, which is a straightforward implementation of f that returns a list. For example, the user function *evenDup* is implemented as the function object `evenDup` (Fig. 5), where it returns a concrete vector of one or two elements.

This approach has the key advantage of good programmability: it provides a simple, functional programming style for defining a user function. In fact, this style is the same as the previous mechanism and is quite natural for use in programming with our skeletons that have functional style definitions.

However, this approach does have one big disadvantage: it suffers from *incomplete fusion*. Figure 6 shows a C++ SPMD program[2] equivalent to the fused program of *evenDupSum* that can be generated in this approach. In the main loop, the user-defined `evenDup` creates a small vector `v` at every iteration and the inner loop then runs on this vector to sum up its elements to the accumulator `r`. Since we cannot change the definition of `evenDup` at compile

[2] Since we use the expression technique to carry out fusions, the result of the fusion is not a C++ program but an assembly program. We show its discompiled version for readability.

```
long r = 0;
for(int i = 0; i < z.local_size(); i++) {
  vector<long> v = evenDup(z.local_get(i));
  for(int j = 0; j < v.size(); j++) r = plus<long>()(r, v[j]);
}
global_reduce(plus<long>(), r);
```

Fig. 6 C++ SPMD program equivalent to the fused version of *evenDupSum* in the vector-based approach.

time, this production of small vectors is unavoidable. Therefore, the fusion is incomplete. Actually, the code does not implement our goal $evenDupSum_{opt}$ but rather the incompletely fused program $evenDupSum''$:

$$\begin{aligned} evenDupSum'' = {} & \mathsf{hom}\ (+)\ evenDup'' \\ & \textbf{where}\ evenDup''\ a = \mathsf{reduce}\ (+)\ (evenDup\ a) \end{aligned}$$

At a glance, this program looks successfully fused because the composition of reduce and concatmap has been replaced with hom $(+)$ $evenDup''$. However, *the fusion is incomplete* in the sense that it creates intermediate data structures (lists) inside the new function $evenDup''$. This incompleteness creates a serious efficiency problem when a user function returns a big list.

The main problem of this approach is that the fused program produces many vectors—some of which are possibly quite big—inside the main loop, and we cannot avoid this as long as a user-defined function object returns a concrete vector. To avoid this incompleteness of the fusion, we need a user function that does not return a concrete vector.

4.3 Iterator-based Approach

In this approach, to avoid the production of vectors in the fused program, a user function is implemented to return an iterator (an object that yields elements one by one) instead of a concrete vector. Use of iterators to avoid intermediate data structures is natural in practical C++ programming.

Although this approach is advantageous in that we can achieve complete fusion, thus avoiding the problem faced by the vector-based approach, there are two disadvantages. The main disadvantage is the difficulty of user programming: implementing a user function to return an iterator is much more difficult than one to return a concrete vector. The other disadvantage is the risk of incomplete fusion. To avoid the difficulty of user programming, users may take the easy way: creating a vector and returning its iterator. Then, the fused program produces many vectors implicitly inside the calls of the user function, and the fusion is incomplete. We have limited space here, but interested readers can find the details with concrete codes in our technical report [4].

```
struct evenDup_t {
  template <typename Collector>
  void operator()(long a, Collector &c) const {
    if(even(a)) { c.push_back(a); c.push_back(a) } else { c.push_back(a); };
  }
} evenDup;
```

Fig. 7 Collector-based implementation of user function *evenDup*.

The main problem of this approach is the difficulty of the user programming, which stems from adopting the functional style such that a function object returns something. To achieve both good programmability and good efficiency, we need a slightly imperative style.

4.4 Collector-based Approach

In this approach, a user function is implemented to *receive* a *collector*, which is an object that receives elements one by one. It then puts elements into the collector. This is a dual approach of the iterator-based approach, adopting a slightly imperative style in the sense that the user function does not return anything. This approach can achieve both good programmability and good efficiency.

Figure 7 shows an implementation of the user function *evenDup*. The function object `evenDup` receives a collector `c` and puts elements into the collector by calling `c.push_back(a)`. Here, the type of collector `c` is generalized as the template type parameter `Collector`, so this user function can be used in various contexts. The code looks almost the same as the code in the vector-based approach (Fig. 5). The only difference is the place to which the elements are emitted: The former puts elements into the given collector while the latter puts elements into its created vector. Therefore, the programmability of this approach is as good as the vector approach, and much better than the iterator approach.

Figure 8 shows a C++ SPMD program equivalent of the fused program of *evenDupSum*. The fused program uses a collector defined as a new structure `ReduceCollector` in which the `push_back` method adds the given element `a` into its accumulator variable `r`. In each iteration of the main loop, the user function `evenDup` receives the collector `c` as well as the ith element `z.local_get(i)` of the input list `z` and puts one or two copies of the element into the collector. Since the collector immediately adds the given element into the accumulator, there is no production of intermediate vectors in this code. Therefore, this code successfully implements our goal program $evenDupSum_{opt}$.

This approach can fuse multiple concatmaps. To explain this, we use the following program with two concatmaps to compute a doubled summation of

```
struct ReduceCollector {
  long &r; ReduceCollector(long &r) : r(r) { }
  void push_back(int a) { r = plus<long>()(r, a); }
};
long r = 0;
ReduceCollector c(r);
for(int i = 0; i < z.local_size(); i++) evenDup(z.local_get(i), c);
global_reduce(plus<long>(), r);
```

Fig. 8 C++ SPMD program equivalent to the fused version of *evenDupSum* in the collector-based approach.

```
struct noOdd_t {
  template <typename Collector>
  void operator()(long a, Collector &c) const { if(even(a)) c.push_back(a); }
} noOdd;
```

Fig. 9 Collector-based implementation of user function *noOdd*.

only even numbers by using $noOdd\ a = \textbf{if}\ even\ a\ \textbf{then}\ [a]\ \textbf{else}\ [\,]$:

$$evenDupNoOddSum = \mathsf{reduce}\ (+) \circ \mathsf{concatmap}\ noOdd \circ \mathsf{concatmap}\ evenDup$$

Figure 9 shows a collector-based implementation of the user function *noOdd*. Our desired fused program is basically the following program:

$$evenDupNoOddSum_{opt} = \mathsf{hom}\ (+)\ (\lambda a.\textbf{if}\ even\ a\ \textbf{then}\ a + a\ \textbf{else}\ 0)$$

In order to fuse multiple concatmaps, we simply need to build new collectors from user functions. We use a new structure, `CombinedCollector` (Fig. 10) that has two fields to hold a user function `f` and to hold another collector `c`. The method `push_back` of `CombinedCollector` simply supplies the given element `a` and the collector `c` to the user function `f`.

Figure 11 shows the main loop of the fused program of *evenDupNoOddSum*, which simply supplies elements to the new collector built from the user functions. Figure 12 shows the computation flow of the new collector, in which `zi` corresponds to `z.local_get(i)` in the main loop. When `zi` is an even number (the solid line), by definition, `c2.push_back(zi)` calls `evenDup(zi, c1)` once, and the call of `evenDup` makes two calls of `c1.push_back(zi)`. Each call of `c1.push_back(zi)` invokes `noOdd(zi, c)` once, and this `noOdd` makes one call of `c.push_back(zi)`. Therefore, when `zi` is even, it is added to accumulator `r` twice. On the other hand, when `zi` is odd (the dashed line), the call of `evenDup` makes one call of `c1.push_back(zi)` and invokes `noOdd(zi, c)` once. Since `noOdd(zi, c)` does nothing when `zi` is odd, the accumulator `r` is kept unchanged in this case. Clearly, the main loop implements our desired fused program.

We now have a good design for a fusion mechanism that can achieve both an easy programming interface and complete fusion. Its concrete implementation with the expression templates technique [11] is explained in the next section.

```
template<typename NextCollector, typename F>
struct CombinedCollector {
  NextCollector c; const F f;
  CombinedCollector(const F&f, NextCollector &c) : f(f), c(c) {}
  void push_back(const long& a) { f(a, c); }
};
```

Fig. 10 Structure of combined collectors for fusing multiple concatmaps.

```
  long r = 0; ReduceCollector c(r);
  CombinedCollector<ReduceCollector, noOdd_t> c1(noOdd, c);
  CombinedCollector<CombinedCollector<ReduceCollector, noOdd_t>, evenDup_t>
    c2(evenDup, c1);
  for(int i = 0; i < z.local_size(); i++) c2.push_back(z.local_get(i));
```

Fig. 11 The main loop of the fused program of *evenDupNoOddSum*.

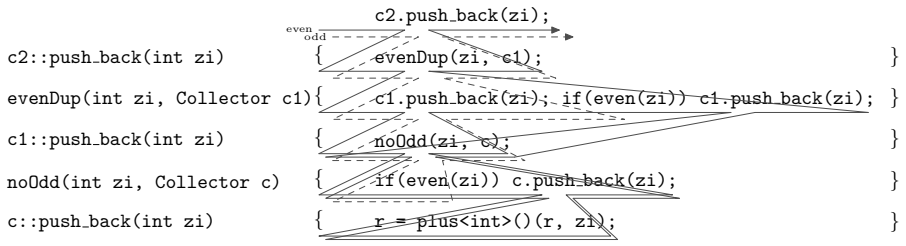

Fig. 12 The call chain of collectors inside the fused program of *evenDupNoOddSum*.

5 Implementation and Evaluation

We have implemented the fusion mechanism for VLL skeletons in our SkeTo library [7] by using expression templates [11]. We briefly describe this implementation and report some experimental results to show the effect of the fusion mechanism.

5.1 ET Implementation of the Fusion Mechanism for VLL Skeletons

Figure 13 shows the implementation of the fusion mechanism of the collector-based approach using the expression templates technique [11]. In the explanation below, we use as an example the following code straightforwardly implementing *evenDupNoOddSum*:

```
long eDNOSum = reduce(plus<long>(),
                      concatmap(noOdd, concatmap(evenDup, z)));
```

The skeleton function `concatmap` returns an expression object of `CMapObj` in order to postpone its computation and enable the fusion. The object has

two fields, a user function `f` and an expression object `x` that represents its target list. It also has several methods and type declarations, which will be explained later. For example, `concatmap`s in the example program create an object `CMapObj(noOdd, CMapObj(evenDup, z))`.

The skeleton function `reduce` receives a `CMapObj` object that represents its target list as well as an associative binary operator `op`. Before executing the main loop, it asks the object to find the initial list in the chain of `concatmap`s and build a combined collector from the initial collector `ic` of the generalized `ReduceCollector` that accumulates given elements to the accumulator `res` by `op`. For example, the initial list of the example above is `z`, and the combined collector is (equivalent to) the one explained at the end of Section 4.4. The extraction of the initial list and construction of the combined collector can be implemented by the simple recursive methods `getCollector` and `getInitialList` on expression objects. The main loop then supplies each element of the initial list to the combined collector.

The above mechanism implements our desired fusion for `concatmap`s.

It is easily seen that we can use the previous fusion mechanism for FLL skeletons in the main loop of the fused program because it uses the index-based access method `local_get`. This means that we can fuse FLL skeletons followed by a chain of VLL skeletons into one loop.

Finally, we should note that the resulting list of a chain of `concatmap` can be computed efficiently with fusion in a similar way: we simply use a `vector` as the initial collector instead of `ReduceCollector`.

5.2 Experiment Results

To evaluate the implemented fusion mechanism, we measured the execution time of skeleton programs with and without the fusion on a cluster consisting of 32 nodes, each of which has Intel(R) Xeon(R) E5645 and 12 GB memory and is connected to Gigabit Ethernet. We used one core per node. The programs were compiled with GCC 4.6.3.

Table 1 shows the measured execution time. An empty cell means that the program was not run due to the memory shortage. The size means the number of elements of the input or the board size n for an n-queens problem. The suffix "(lnr)" means that the input is the list $[0, 1, \ldots, (size - 1)]$, while "(rnd)" means a list of random numbers. Skeleton programs basically showed good scalability regardless of the fusion, unless their computation was too lightweight compared with synchronization-communication overheads.

The measured execution time of fused `evenDupSum` compared with that of the non-fused version shows the basic impact of the proposed fusion mechanism. The fusion improved the efficiency dramatically, achieving a $5\times$ to $20\times$ speedup. For the linear list $[0, 1, \ldots, , (size - 1)]$, the fused program achieved an absolute speed slightly faster than the fused version of `evenDblSum` generated by the previous fusion mechanism, and `evenDupSumHand`, which is the following hand-written single sequential loop:

```
template <typename F, typename X>
struct CMapObj {
  const F f; const X x; CMapObj(const F &f, const X &x) : f(f), x(x) { }
  typedef typename X::InitialType InitialType;
  const InitialType &getInitialList() const { return x.getInitialList(); }
  template <typename NextCollector>
  /* omit the type */ getCollector(NextCollector &c) const {
    return x.getCollector(CombinedCollector<NextCollector>(f, c));
  }
};
template <typename F, typename X>
CMapObj<F, X> concatmap(const F& f, const X& x){ return CMapObj<F, X>(f, x); }

template <typename OP, typename A>
struct ReduceCollector {
  const OP& op; A &r; ReduceCollector(const OP&op, A &r) : op(op), r(r) { }
  void push_back(const A& a) { r = op(r, a); }
};
template <typename OP, typename F, typename X>
typename OP::result_type
reduce(const OP &op, const CMapObj<F, X> &cmobj) {
  const typename X::InitialType &l = cmobj.getInitialList();
  typename OP::result_type res = get_identity<OP>();
  ReduceCollector<OP, typename OP::result_type> ic(op, res);
  /* omit the type */ c = cmobj.getCollector(ic);
  for(int i = 0; i < l.local_size(); i++) c.push_back(l.local_get(i));
  global_reduce(op, res);
  return res;
}
```

Fig. 13 Expression templates implementation of the collector-based fusion mechanism

Table 1 Measured execution time (seconds) of skeleton programs

program	fusion	size	#processes 1	2	4	8	16	32
evenDupSum	w/o	400M (lnr)	9.64	4.59	2.30	1.17	0.63	0.34
	w/	400M (lnr)	0.50	0.27	0.15	0.11	0.09	0.08
	w/	2G (lnr)			0.67	0.35	0.20	0.15
evenDblSum	w/	400M (lnr)	0.59	0.32	0.18	0.12	0.10	0.09
	w/	2G (lnr)			0.80	0.40	0.28	0.16
evenDblSumHand	w/	400M (lnr)	0.59	—	—	—	—	—
evenDupSum	w/o	400M (rnd)	11.92	5.74	2.85	1.44	0.75	0.41
	w/	400M (rnd)	2.30	1.17	0.61	0.33	0.20	0.13
	w/	2G (rnd)			2.91	1.47	0.77	0.42
evenDblSum	w/	400M (rnd)	0.59	0.32	0.16	0.09	0.07	0.04
	w/	2G (rnd)			0.78	0.42	0.23	0.14
evenDblSumHand	w/	400M (rnd)	0.59	—	—	—	—	—
evenDupFibSum	w/	400M (rnd)	132.30	66.10	33.18	16.52	8.28	4.20
evenDblFibSum	w/	400M (rnd)	134.44	67.19	33.55	16.80	8.42	4.25
nqueen	w/o	14				44.34	22.05	12.22
	w/	14	123.22	61.85	36.51	18.77	9.63	5.55

```
for(i=0; i < n; i++) r += (x[i]&1) ? x[i] : x[i] + x[i];
```

In contrast, for the list of random numbers, the fused `evenDupSum` was slower than the others. This performance difference was caused by the compiler's choice of instructions for branching: The compiled assembly code (Fig. 14) of `evenDupSum` uses a conditional branch instruction `jne` for the branch on the judgment `if(even(a))`, while the assembly code (Fig. 15) of `evenDblSum` and `evenDupSumHand` uses a conditional move instruction `cmovne`. Therefore, `evenDupSum` is faster than the others for such a regular input on which the branch prediction works well and slower for an irregular input such as the random list. It seems that the complicated ET implementation of the VLL fusion mechanism disturbs the compiler's analysis and optimization because it is expected that code with a conditional move instruction is faster than code with a conditional branch instruction, and the compiler must try the optimization to replace the latter with the former. We could tune the ET implementation in order to not disturb the compiler's work, but this is beyond the scope of this paper. In any case, the results show that the proposed fusion mechanism produces efficient code comparable to hand-written code.

Figure 16 shows two simple programs, `evenDupFibHand` and `evenDblFibSum`, that map heavy function *fibWeight* that computes and adds the tenth Fibonacci number to the given element before the computation of `evenDupSum` and `evenDblSum`, respectively. Their measured execution times show that the performance decline caused by the compiler's different choice of instructions and the synchronization-communication overheads is ignorable for non-lightweight computation.

Comparison of the measured times of `nqueen` with and without fusion shows the effect of the fusion on practical programs: it achieves a more than 2× speedup for the practical program.

Interested readers can find additional experiment results about mixed use of FLL and VLL skeletons in our technical report [4]. The proposed fusion mechanism works well with our previous one.

6 Related Work

Skeletal parallel programming was first proposed by Cole [3], and a number of systems (libraries) have been proposed since then. Among them, OSL [6] and SaC [5]—as well as our own library, SkeTo [7]—are the ones equipped with fusion mechanisms to optimize skeleton programs. OSL is a skeleton library based on the BSP model implemented using MPI and C++, and its fusion mechanism is implemented using the expression templates technique [11]. Its set of its fusion rules is almost the same as our previous fusion mechanism [7]. SaC is an array programming language mainly suited for areas such as numerically intensive applications and signal processing. It has the with-loop fusion mechanism combining high-level program specifications with runtime efficiency

```
 .L567:                              # # this is then-block
    addq $1, %rax                    # i++
    addq %rsi, %rdx                  # r += %rsi
    cmpl %eax, %edi                  #
    movq %rdx, (%r8)                 # # write r back to the cache
    jle .L566                        # if(i >= n) goto .L566
 .L569:                              #
    movq (%r10,%rax,8), %rsi         # %rsi = x[i]
    testb $1, %sil                   #
    jne .L567                        # if(%rsi & 1) goto .L567
    leaq (%rdx,%rsi,2), %rdx         # r += %rsi + %rsi # else-block
    addq $1, %rax                    # i++
    cmpl %eax, %edi                  #
    movq %rdx, (%r8)                 # # write r back to the cache
    jg .L569                         # if(i < n) goto .L569
 .L566:                              #
```

Fig. 14 Compiled assembly code of fused `evenDupSum`

```
 .L569:                              #
    movq (%r9,%rax,8), %rsi          # %rsi = x[i]
    leaq (%rsi,%rsi), %rdx           # %rdx = %rsi + %rsi
    testb $1, %sil                   #
    cmovne %rsi, %rdx                # if(%rsi & 1) %rdx = %rsi
    addq $1, %rax                    # i++
    addq %rdx, %rbx                  # r += %rdx
    cmpl %eax, %edi                  #
    jg .L569                         # if(i < n) goto .L569
 .L566:                              #
```

Fig. 15 Compiled assembly code of fused `evenDblSum` and `evenDupSumHand`

```
long fib(int a) { return (a < 2) ? 1 : fib(a-1) + fib(a-2); }
struct fibWeight_t : sketo::functions::base<long(long)>{
  long operator()(const long &a) const {
    return a + fib(N);
  }
} fibWeight;

long evenDupFibSum(dist_list<long> z) {
  return reduce(plus<long>(), concatmap(evenDup, map(fibWeight, z)));
}

long evenDblFibSum(dist_list<long> z) {
  return reduce(plus<long>(), map(evenDbl, map(fibWeight, z)));
}
```

Fig. 16 Simple programs with heavy computation.

similar to that of hand-optimized low-level specifications. Unfortunately, none of them provide VLL skeletons with a fusion mechanism.

7 Conclusion

We proposed a novel fusion mechanism for variable-length list (VLL) skeletons with a collector-based approach for defining user functions that achieves both good programmability and good performance. We implemented the proposed mechanism in our skeleton library, SkeTo, by using expression templates, and its impact on efficiency has been shown via experiment results. In addition, it can be used in conjunction with our previous fusion mechanism, so a wide variety of skeleton programs can take advantage of our fusion optimizations.

A VLL skeleton may cause an imbalance of distributed data, and in such cases we might need to rebalance data between successive VLL skeletons to achieve good parallelism. However, to avoid performance decline caused by intermediate data structures, we want to fuse successive VLL skeletons, which removes the chance of rebalancing. Therefore, we need to be careful with fusing VLL skeletons in order to achieve the best performance. Automatic control of fusion in such cases and an accompanying cost model are a part of our future work.

Acknowledgements This work was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Young Scientists (B) 24700025, and PaPDAS project supported by ANR (ANR-2010-INTB-0205-02) and JST (10102704). The authors would like to thank Liu Yu and Shigeyuki Sato for fruitful discussions in the early stages of this work.

References

1. Bird, R.: An introduction to the theory of lists. In: Proceedings of NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design, pp. 5–42 (1987)
2. Bird, R.: Introduction to Functional Programming using Haskell. Prentice-Hall (1998)
3. Cole, M.: Algorithmic Skeletons : A Structured Approach to the Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing. Pitman Publishing (1989)
4. Emoto, K., Matsuzaki, K.: An automatic fusion mechanism for variable-length list skeletons in sketo. Tech. Rep. METR2013–04, University of Tokyo (2013). Available on web: `http://www.keisu.t.u-tokyo.ac.jp/research/techrep/`
5. Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SaC. Parallel Computing **32**(7-8), 507–522 (2006)
6. Javed, N., Loulergue, F.: OSL: Optimized bulk synchronous parallel skeletons on distributed arrays. In: Advanced Parallel Processing Technologies, *Lecture Notes in Computer Science*, vol. 5737, pp. 436–451. Springer Berlin Heidelberg (2009)
7. Matsuzaki, K., Emoto, K.: Implementing fusion-equipped parallel skeletons by expression templates. In: Implementation and Application of Functional Languages, *Lecture Notes in Computer Science*, vol. 6041, pp. 72–89. Springer Berlin Heidelberg (2011)
8. Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer-Verlag (2002)

9. Skillicorn, D.B.: The Bird-Meertens formalism as a parallel model. In: J. Kowalik, L. Grandinetti (eds.) Software for Parallel Computation, *NATO ASI Series*, vol. 106, pp. 120–133. Springer Berlin Heidelberg (1993)
10. Tanno, H., Iwasaki, H.: Parallel skeletons for variable-length lists in sketo skeleton library. In: Euro-Par 2009 Parallel Processing, *Lecture Notes in Computer Science*, vol. 5704, pp. 666–677. Springer (2009)
11. Veldhuizen, T.: Expression templates. C++ Report **7**, 26–31 (1995)