A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming

Shigeyuki Sato and Hideya Iwasaki

Department of Computer Science
The University of Electro-Communications
sato@ipl.cs.uec.ac.jp, iwasaki@cs.uec.ac.jp

Abstract. Although today's graphics processing units (GPUs) have high performance and general-purpose computing on GPUs (GPGPU) is actively studied, developing GPGPU applications remains difficult for two reasons. First, both parallelization and optimization of GPGPU applications is necessary to achieve high performance. Second, the suitability of the target application for GPGPU must be determined, because whether an application performs well with GPGPU heavily depends on its inherent properties, which are not obvious from the source code. To overcome these difficulties, we developed a skeletal parallel programming framework for rapid GPGPU application developments. It enables programmers to easily write GPGPU applications and rapidly test them because it generates programs for both GPUs and CPUs from the same source code. It also provides an optimization mechanism based on fusion transformation. Its effectiveness was confirmed experimentally.

1 Introduction

It is more difficult to develop efficient parallel programs, because they are more complex than sequential ones due to interactions between processes. One approach to making parallel programming easier is *skeletal parallel programming* [1], in which parallel programs are built using *skeletons*, i.e., frequently used parallel computation patterns. Skeletons provide high-level abstraction and enable programmers to write parallel programs in a sequential manner.

Skeletal parallel programming has been studied from both theoretical and practical aspects. In the theoretical area, optimization based on *fusion* [2,3,4] has been studied [5,6,7]. In the practical area, skeleton libraries for distributed memory systems such as PC clusters have been developed [8,9,10,11]. However, not many practical applications rely on skeletal parallelism, which is a serious problem for skeletal parallel programming. To expand the area of its application, we applied skeletal parallelism to the programming for graphics processing units (GPUs).

The arithmetic performance and memory bandwidth of today's GPUs is ten times higher than that of today's CPUs, and the performance of GPUs is improving more rapidly than that of CPUs. This is why general-purpose computing on

GPUs (GPGPU) [12,13] is being actively studied in the field of high-performance computing and why many GPGPU applications have been developed.

Development of a GPGPU application is difficult and troublesome for two reasons. First, only parallel programs that are well optimized for GPU architectures can fully utilize the performance of GPUs. The performance of a GPGPU program that does not sufficiently exploit a GPU's capabilities is often worse than that of a simple sequential one running on a CPU. Second, programmers need to determine whether the target application is suitable for GPGPU. For example, an application may not be able to achieve the good performance due to data transfer from main memory to video memory and GPU start-up time.

As an approach to these difficulties of GPGPU programming, we propose applying high-level abstraction of skeletons to hide the use of GPUs. We have developed a skeletal parallel programming framework with a fusion optimizer that enables programmers to easily write GPGPU applications and test them rapidly. The proposed framework is designed so as to be embedded in the C language, i.e., programmers can use the framework without any language extensions to C. In addition, programmers can write efficient parallel programs for both GPUs and CPUs as the same source code. Thus, the suitability for GPGPU can be tested rapidly. Our main contributions can be summarized as follows.

– We show that skeletal parallel programming can be applied to a practical framework for rapid GPGPU application development. We also illustrate its effectiveness through specific examples. The proposed framework is a practical application of skeletal parallel programming.
– We present that the proposed framework enables programmers to rapidly check the suitability of target applications for GPGPU. From the same source code, the framework generates three kinds of programs, namely a GPGPU program, a portable C++ parallel program with OpenMP, and a portable sequential C program.
– We present an implementation of the optimizer based on fusion transformation of skeletons and show its effectiveness for GPGPU applications. In the best case, an optimized GPGPU program ran 2.44 times faster than the non-optimized version.

2 Preliminaries

2.1 BMF and Skeletal Parallelism

In this paper, we regard data parallel primitives in the Bird-Meertens Formalism (BMF) [14] as skeletons for BMF-based skeletal parallel programming [15,16]. Throughout this paper, we use the notation of Haskell for describing the specifications of skeletons and other primitive operations.

Three important skeletons in BMF are map, reduce and zipwith.

$$\mathsf{map}\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$
$$\mathsf{reduce}\ (\oplus)\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$
$$\mathsf{zipwith}\ f\ [x_1, x_2, \ldots, x_n]\ [y_1, y_2, \ldots, y_n] = [f\ x_1\ y_1, f\ x_2\ y_2, \ldots, f\ x_n\ y_n],$$

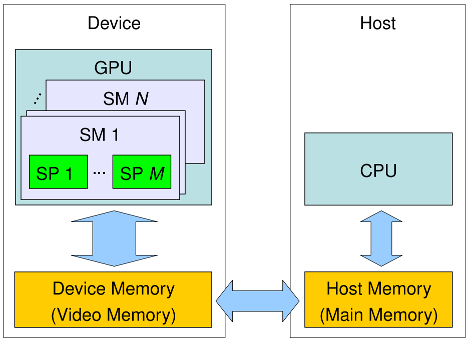


Fig. 1. CUDA hardware model

where $\oplus$ is an associative operator. We suppose that map, reduce, and zipwith are not given either empty or infinite lists.

We can transform a program into an efficient one by merging successive skeletons into a single one, e.g., map f (map g as) $=$ map $(f \circ g)$ as. Such program transformation is called *fusion*, which is well-known in functional programming.

2.2 CUDA

CUDA is a general-purpose parallel computing architecture for GPUs. We briefly describe CUDA's features. Refer to the programming guide [17] for more details.

CUDA's hardware model is a distributed memory system that consists of host and device memory. These two kinds of memory are physically separated, as illustrated in Fig. 1. Host memory corresponds to main memory, while device memory corresponds to video memory. A GPU has several streaming processors (SMs), each of which consists of several scalar processor (SP) cores. Each SM supports multithreading.

Programmers use "C for CUDA"[1], an extended C language, to write GPGPU programs. Strictly speaking, CUDA is a subset of C++ with language extensions for using the device. These extensions include three additional function type qualifiers: `__global__`, `__device__`, and `__host__`. The `__global__` qualifier declares a function that is called from the host and executed in the device, the `__device__` qualifier declares one that is called from the device and executed in the device, and the `__host__` qualifier declares one that is called from the host and executed in the host. A function without one of these qualifiers is regarded to be qualified by `__host__`. A function qualified by both `__device__` and `__host__` is compiled for both the device and the host. `__global__` and `__device__` functions have several restrictions; e.g., they do not support the recursive call, the return type of each `__global__` function must be `void`, and a function pointer to a `__device__` function cannot be taken.

[1] In the rest of this paper, "C for CUDA" is simply called CUDA.

```
 1 #include <skeleton.h>
 2
 3 double sqr(int x) { return (double) x*x; }
 4 double add(double x, double y) { return x+y; }
 5
 6 double sqr_sum(int *buf, int n)
 7 {
 8   int *as[PTR_UNIT];     // declare wrapped array pointer
 9   double *tmp[PTR_UNIT]; // declare wrapped array pointer
10   double res;
11
12   skel_new(as);          // initialize as
13   skel_new(tmp);         // initialize tmp
14   skel_wrap(as, buf, n); // wrap array pointed to by buf
15
16   map(sqr, as, tmp);      // square each element of as
17   reduce(add, tmp, &res); // sum up all elements of tmp
18
19   skel_del(as);  // dispose of wrapped array
20   skel_del(tmp); // dispose of wrapped array
21
22   return res;
23 }
```

Fig. 2. Program that computes square sum of integer list using framework

Because CUDA had made GPGPU easier than before, CUDA became most popular in GPGPU programming. Nevertheless, GPGPU programming with CUDA remains difficult. For instance, when a matrix multiplication program that is simply and sequentially coded for a CPU is ported to CUDA for a GPU without much modification, the ported program is 200–2000 times slower than the original one as shown in an experiment[2]. This suggests that GPGPU programming with CUDA needs very hardware-conscious programming.

3 Overview of Proposed Framework

We briefly describe how to write a program using the proposed framework. Figure 2 shows an example program that computes a square sum of a list that is represented by an array using the framework.

First, the header file is included (line 1) to enable use of the framework APIs. Wrapped array pointers, which will be described in Sect. 4.1, are declared (lines 8–9) and initialized by `skel_new` (lines 12–13). Then, an array is wrapped by `skel_wrap` (line 14), whose third parameter is the number of wrapped elements in the array. Then, the skeletons operate on the lists (lines 16–17), where the last parameter given to each skeleton is the destination for storing the result.

[2] Refer to Sect. 7 for details on the experimental environment.

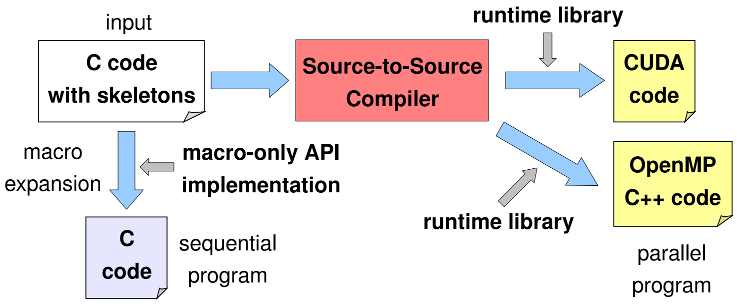


Fig. 3. Outline of proposed framework

Finally, `skel_del` disposes of the wrapped array that is no longer necessary (lines 19–20). With this framework, programmers can easily write GPGPU programs without any consideration of either hardware or parallelization.

The framework transforms a given program in which APIs of the framework are used. As shown in Fig. 3, it has three main components:

– a source-to-source compiler with a fusion optimizer for parallel programs,
– runtime libraries, and
– a macro-only API implementation for sequential programs.

The source-to-source compiler, which is the core of the framework, generates C code with skeletons into CUDA code for GPUs or C++ code with OpenMP for CPUs. Compiler driver scripts run a CUDA compiler or a C++ compiler with appropriate compile-time constants, and the generated code is compiled into executable code. The runtime libraries are used by the generated code. The macro-only API implementation is used for debugging and porting.

4 Design

4.1 Principles

C for Base Language. The framework was designed on the basis of the C language. Each API of the framework can be seen as a macro from the viewpoint of C programming, even though each API call and other parts of a program are transformed by our compiler for GPGPU. The framework also provides the macro-only implementation of each API to help users debug programs as on-CPU sequential C programs. Skeletons require no language extension to C. This is one of the great merits of our framework and skeletal parallel programming.

There are three reasons for selecting C as the base language. The first is CUDA's affinity for C: it is easy to translate C into CUDA because CUDA is an extended C language. The second and third reasons are the popularity and performance of C. In fact, many skeleton libraries [8,9,10,11,18] have been implemented in C/C++ for these two reasons.

Transparency. The framework is designed to have transparency, i.e., to hide the use of the GPU and distributed memory. This enables the framework to generate three kinds of programs: GPGPU programs, on-CPU parallel programs, and on-CPU sequential programs. Thus, transparency leads to portability. The transparency and portability of the framework owe much to the high-level abstraction of skeletons, an important advantage of skeletal parallel programming.

Pointer Contracts. The proposed framework imposes three *contracts*, i.e., promises that should be kept.

- A list passed to skeletons should be a wrapped array.
- Wrapped arrays should be accessed via only APIs of the framework.
- Every wrapped array pointer should have no alias.

If a skeleton received pointers into which the result of a computation was stored, memory copying between device and host would occur every time that skeleton was called. This would seriously degrade performance. This problem is caused by pointers that can freely access memory. To solve this problem, we introduce *wrapped arrays* to which access is restricted and *wrapped array pointers* that point to the head of a wrapped array, in contrast to the *raw pointers* and *raw arrays* natively supported in C.

Neither dereferencing nor pointer arithmetic against wrapped array pointers are permitted. They are only permitted to be passed to APIs. In addition, when part of a raw array is wrapped, the programmer must ensure that the area is not referred to by other pointers.

Because aliases make fusion optimization difficult, the framework forbids operations that may produce aliases of wrapped array pointers, e.g., assignment, indirect reference, passing to functions, and returning from functions.

4.2 APIs

The framework provides simple and natural APIs for C programmers. Table 1 shows the APIs with brief descriptions. Each skeleton is a procedure (a function with no return value) whose last parameter is the destination into which the result will be stored. The `mapls` and `maprs` APIs are introduced because C does not support partial application. The `generate` API is introduced because of its efficient construction of lists and synergy with fusion.

The APIs do not depend on list element types and have as polymorphic behaviors as macros.

Functions passed to skeletons are defined in C without special function type qualifiers even though skeletons are executed on GPUs. This enables programmers to transparently reuse functions.

No operations that change the length of a list are provided. Thus, the length of the resulting list of a skeleton call is automatically determined once the list length is set by `skel_wrap` or `generate`. Programmers need not be concerned about the list length because the framework appropriately propagates the length in the implementation of skeletons.

Table 1. API list (function identifier, wrapped array pointer, raw pointer, wrapped array, and raw array are abbreviated as FI, WAP, RP, WA, and RA, respectively)

API	Brief description
`map(`FI `f,` WAP `as,` WAP `bs)`	map
`reduce(`FI `op,` WAP `as,` RP `a)`	reduce
`zipwith(`FI `f,` WAP `as,` WAP `bs,` WAP `cs)`	zipwith
`mapls(`FI `f,` RP∪WAP `a,` WAP `bs,` WAP `cs)`	map $(\lambda x.f\ a\ x)\ bs$
`maprs(`FI `f,` RP∪WAP `a,` WAP `bs,` WAP `cs)`	map $(\lambda x.f\ x\ a)\ bs$
`generate(`FI `f, int n,` WAP `as)`	map $f\ [0,\ldots,n-1]$
`skel_new(`WAP `as)`	initializing WAP
`skel_del(`WAP `as)`	disposing of WA
`skel_wrap(`WAP `dst,` RP `src, int n)`	wrapping RA
`skel_unwrap(`WAP `as)`	unwrapping WA
`skel_dup_contents(`RP `dst,` WAP `src)`	copying WA to RA
`skel_get_element(`RP `dst,` WAP `src, int i)`	getter for WA
`skel_set_element(`WAP `dst, int i,` RP `src)`	setter for WA

The APIs do not include memory allocation operations. Instead, runtime libraries automatically allocate memory when a skeleton first accesses a wrapped array pointer. Hence, `skel_del` does nothing unless memory has been allocated.

For implementation reasons, the APIs have the following restrictions.

– A list element type must not include the pointer type.
– Each API call must be an expression statement.
– A function argument passed to a skeleton must be the function identifier.
– Functions passed to skeletons have the same restrictions as `__device__` functions in CUDA.
– Binary operators passed to `reduce` must be associative and commutative.

The first restriction comes from the fact that the framework does not support serialization. The second restriction is needed for the macro-only API implementation. For instance, a skeleton call in an expression causes a syntax error if the skeleton is implemented as a macro of a **for** loop. The third restriction helps both CUDA and C++ compilers to inline functions passed to skeletons. A function passed to a skeleton can be inlined only if it can be statically determined. The fourth restriction is needed because functions passed to skeletons are executed on GPUs. The last restriction is necessary to achieve efficient implementations of skeletons on GPUs. If commutative and associative operators are given, the reduction algorithm can be optimized for GPUs by using the Harris algorithm [19]. This restriction is not particularly severe because frequently used operators have commutativity.

5 Fusion Transformation

Two functions, which are not explicitly used by programmers, are used for intermediate representation of skeletons. They are introduced to implement the fusion transformation in a uniform way.

$$\mathsf{zipwith}_k\ f\ [x_1^1, \ldots, x_n^1] \cdots [x_1^k, \ldots, x_n^k] = [f\ x_1^1\ \cdots\ x_1^k, \ldots, f\ x_n^1\ \cdots\ x_n^k]$$
$$\mathsf{reduce}_k\ (\oplus)\ f\ [x_1^1, \ldots, x_n^1] \cdots [x_1^k, \ldots, x_n^k] = (f\ x_1^1\ \cdots\ x_1^k) \oplus \cdots \oplus (f\ x_n^1\ \cdots\ x_n^k)$$

$\mathsf{zipwith}_k$ returns a list zipping corresponding elements of given lists with a function f. reduce_k returns a value of folding with $\oplus$ the result of zipping corresponding elements of given lists with f. The following three equations hold.

$$\mathsf{map} = \mathsf{zipwith}_1 \qquad \mathsf{zipwith} = \mathsf{zipwith}_2 \qquad \mathsf{reduce}\ (\oplus) = \mathsf{reduce}_1\ (\oplus)\ id,$$

where id is the identity function.

The fusion rules for map, reduce, and zipwith are as follows.

$$\begin{aligned}
&\mathsf{map}\ f\ (\mathsf{zipwith}_k\ g\ as^1\ \cdots\ as^k) \longrightarrow \mathsf{zipwith}_k\ (f \circ g)\ as^1\ \cdots\ as^k\\
&\mathsf{reduce}\ (\oplus)\ (\mathsf{zipwith}_k\ f\ as^1\ \cdots\ as^k) \longrightarrow \mathsf{reduce}_k\ (\oplus)\ f\ as^1\ \cdots\ as^k\\
&\mathsf{zipwith}\ f\ (\mathsf{zipwith}_i\ g\ as^1\ \cdots\ as^i)\ (\mathsf{zipwith}_j\ h\ bs^1\ \cdots\ bs^j)\\
&\longrightarrow \mathsf{zipwith}_{i+j}\ \phi\ as^1\ \cdots\ as^i\ bs^1\ \cdots\ bs^j\\
&\qquad \textbf{where}\ \phi\ x^1\ \cdots\ x^i\ y^1\ \cdots\ y^j = f\ (g\ x^1\ \cdots\ x^i)\ (h\ y^1\ \cdots\ y^j)
\end{aligned}$$

Using these rules and the definitions of skeletons, we can express skeleton fusion results in terms of only $\mathsf{zipwith}_k$ and reduce_k. Therefore, implementations of $\mathsf{zipwith}_k$ and reduce_k should suffice for the framework.

From the perspective of efficiency, we implemented $\mathsf{zipwith}_k$ and reduce_k in imperative algorithms using loops for arrays.

$$\mathsf{step}_k\ i\ f\ [x_1^1, \ldots, x_n^1] \cdots [x_1^k, \ldots, x_n^k] = f\ x_i^1\ \cdots\ x_i^k$$

$\mathsf{step}_k\ i\ as_1\ \cdots\ as_k\ (i = 1, \ldots, n)$ is used at each iteration step that computes the i-th element of the result of $\mathsf{zipwith}_k\ as_1\ \cdots\ as_k$. Therefore, parallelization of $\mathsf{zipwith}_k$ can be implemented with loop splitting, and parallelization of reduce_k can be implemented with loop splitting and tree reduction.

The second parameter function of step_k, f, is composed of the functions passed to skeletons. Hence, the function can be constructed in a bottom-up manner from a tree structure of skeleton calls (a *skeleton tree*). Construction of $[0, \ldots, n-1]$ in `generate` can be avoided by using the value of the first parameter of step_k. Thus, if skeleton trees have been constructed, fusion is straightforward.

6 Implementation

6.1 Compiler

The source-to-source compiler was implemented using the COINS[3] compiler infrastructure. The `cfront` component of COINS translates C source code into high-level intermediate representation (HIR), which is a kind of abstract syntax tree, and the `hir2c` component translates HIR into C source code. Program transformation for skeletons was mainly implemented at the HIR level.

There are four steps in the compilation process.

[3] `http://coins-project.is.titech.ac.jp/international/`

1. The C source code with skeletons is translated into HIR by `cfront`.
2. The fusion optimizer constructs skeleton trees from the HIR and then performs HIR-to-HIR transformation.
3. The transformed HIR is translated into C code by `hir2c`. Then, `__device__`, `__host__`, and `inline` are appended to the prototype declarations of all functions passed to skeletons for CUDA code generation. For C++ code, `inline` are appended.
4. Code is generated from each skeleton tree and merged into the code generated in Step 3. Then, runtime libraries are included.

In the generated CUDA code, implementation of specific skeletons consists of two function templates, i.e., an entry function template and a `__global__` function template. First, the entry function template is called from a point of a skeleton call. In the entry function template, array length check, memory allocation, memory copy, and some preparations for CUDA are performed. In addition, depending on the array length, the execution of the skeleton body is switched to either GPUs or CPUs. Second, if the body is determined to be executed on GPUs, the `__global__` function template for the skeleton body, which includes parallel loops with some optimization techniques on CUDA, is called. The generated C++ code with OpenMP is similar, except that it does not use a `__global__` function template. These function templates are strongly typed. If compilation by our compiler and compilation of the generated code succeed, the use of skeletons is type-safe. Such generative approach reduces overhead and overcomes restrictions of `__device__` functions.

6.2 Fusion Optimizer

The fusion optimizer

1. finds *safely fusible* skeleton calls, whose fusion preserves the semantics of the program,
2. constructs a skeleton tree from each sequence of those calls, and
3. rewrites the HIR by using the result of the fusion.

Steps 1 and 2 are done by *fusion analyzer*, which is part of the fusion optimizer.

The algorithm of the fusion optimizer is based on a *greedy fusion strategy*, which fuses as many skeleton calls as possible regardless of recomputation. In fact, recomputation is not bad or sometimes good even though it seems to waste resources. In particular, recomputation is good for GPUs because arithmetic operations are much faster than memory accesses on GPUs. For instance, a recomputation of `generate` often performs better than a store/load of the results because it avoids construction of lists and memory accesses. Moreover, reducing skeleton calls is good for GPUs because GPUs take more time to start up. Therefore, we decided to recompute skeletons rather than to store/load the results in the fusion optimization process.

In addition to the above properties of GPUs, the greedy fusion strategy is used because light-weight functions rather than heavy-weight ones are often passed to

```
1 {
2   map(f, as, bs);
3   map(g, bs, ds);
4   zipwith(op, cs, ds, es);
5   map(h, bs, bs);
6   reduce(oq, es, &r);
7   skel_del(es);
8 }
```

(a) Before fusion

```
{

  ds = zipwith_1 (g ∘ f) as

  bs = zipwith_1 (h ∘ f) as
  r = reduce_2 oq op cs ds
  skel_del(es);
}
```

(b) After fusion (psuedo)

Fig. 4. Example of fusion optimization

skeletons and a skeleton call given a heavy-weight function is rarely fused with many other skeleton calls.

A target of the fusion analyzer is a basic block, which is a series of statements that does not include jumps or labels but can include function calls. The fusion optimizer performs a local optimization. Hence, the analysis is performed within each basic block.

In a basic block, the fusion analyzer (1) finds a skeleton call, s, (2) constructs U_s, where U_s is a set of all successive skeleton calls that use the result of s, and (3) checks whether the result of s is not used except for members of U_s. Then, (4) if the validation in (3) succeeds, a set of skeleton calls that are *safely fusible* with s is U_s; otherwise $\emptyset$. Finally, (5) s is fused[4] with every member of U_s. The fusion process proceeds to the next skeleton call of s.

Figure 4 shows an example of fusion optimization. When `map` (line 2) is s, $U_s = \{$`map` (line 3), `map` (line 5)$\}$. Because the result of `map` (line 2) is overwritten in line 5, i.e., that is not used any more, `map` (line 2) is fused with both `map` (line 3) and `map` (line 5). In this case, mapping `f` to `as` is computed twice on the basis of the greedy fusion strategy. Similarly, when `map` (line 3) is s, $U_s = \{$`zipwith` (line 4)$\}$. However, `map` (line 3) cannot be fused with `zipwith` (line 4) because the result of `map` (line 3), i.e., `ds`, is not deleted.

6.3 API Implementation

The wrapped array pointer was implemented by using a fixed-length array of pointers, each of which is of a pointer type to the list element type. The pointer array consists of a pointer to host memory, a pointer to device memory, and the length of the list in the case of CUDA code. The wrapped array pointer type behaves like a `struct` type that has parametric polymorphism. Although this solution seems to be ad hoc, both language extension to C and the use of void pointers were avoided.

A device pointer is extracted from each wrapped array pointer passed to skeletons within the implementation of each skeleton. Then, the array in the

[4] More precisely, the fusion analyzer only constructs skeleton trees.

device is directly accessed in the execution of the skeleton body. Therefore, the overhead of wrapped array pointers is small.

The polymorphism of the APIs was implemented with the void pointer type for the compiler. First, the prototype declarations of the APIs were defined using the void pointer type in the header file. Then, the API calls are rewritten to calls of function templates that implement the APIs by the compiler. After that, the void pointer type of the APIs is not used. These function templates have strongly typed polymorphism.

7 Experimental Results

It is difficult to determine the properties of a GPGPU application simply by analyzing the algorithms or reading the source code. Thus, it is of great use to generate programs for GPUs and CPUs from the same source code to compare their performance. To demonstrate the effectiveness of the proposed framework from this viewpoint, we tested four applications[5].

N-Body (NB): This is an N-body simulation using the Euler method in two-dimensional space. Two lists (positions and velocities of bodies) are updated every step. An element in each list is a pair of `double` numbers. Its time complexity is $O(tn^2)$, where n is the number of bodies and t is the number of steps.

Numerical Integration (NI): This computes $\int_a^b x \log x \cos x \, \mathrm{d}x$ in $2n$ divisions (x is `double`) by using Simpson's rule. Its time complexity is $O(n)$.

Matrix Multiplication (MM): This computes AB, where A is an $m \times n$ matrix and B is an $n \times m$ matrix whose elements are `double`. A and B are represented as row and column vectors respectively. AB is computed with inner products of row and column vectors. Its time complexity is $O(m^2n)$.

Correlation Coefficient (CC): This computes the Pearson product-moment correlation coefficient from two sequences of samples whose lengths are n. Each sequence is represented as a list of `double`. Its time complexity is $O(n)$.

The experiments were performed on a PC with an Intel Core 2 Duo E8500 CPU (3.16 GHz, L2 cache 6 MB) and a NVIDIA GeForce GTX 280 GPU (via PCI-Express 2.0). The main memory was DDR2-800 4 GB. The video memory of the GPU was 1 GB. The operating system was Ubuntu 7.10 (32-bit). We used CUDA SDK 2.0 (driver version 177.67) and GNU C++ 4.2.1 (including OpenMP) for compiling on-CPU programs. Each binary was created in `-O3` optimization level.

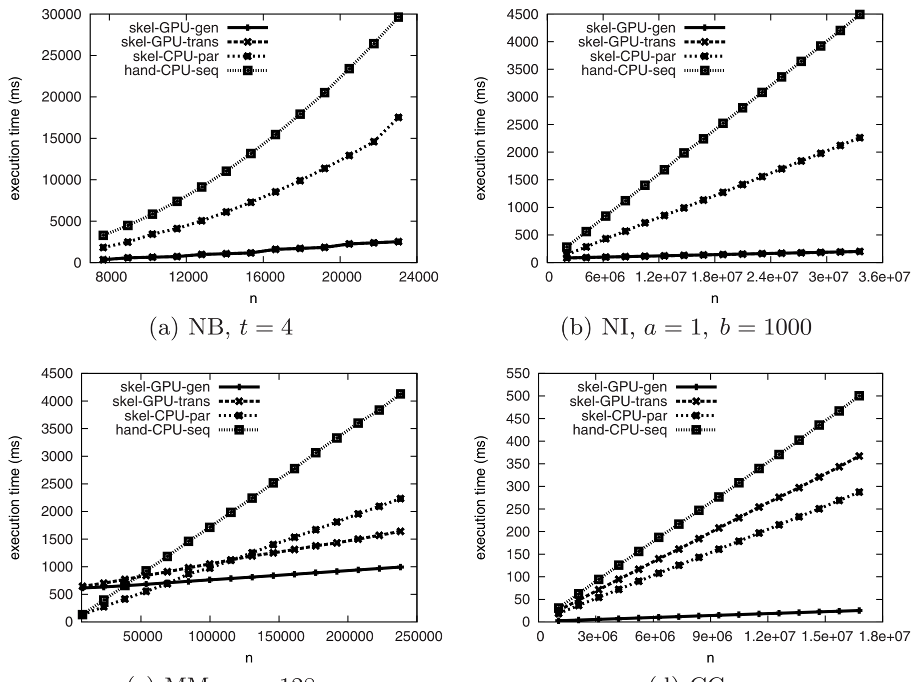
For each application, we used four programs.

skel-GPU-gen: An on-GPU skeleton program whose input data is generated on the GPU with `generate`.

skel-GPU-trans: An on-GPU skeleton program whose input data is generated sequentially on the CPU and transferred to device memory.

skel-CPU-par: An on-CPU skeleton program with OpenMP. It can be generated from the same source code used for skel-GPU-gen by our framework.

[5] Refer to Table 2 for the number of skeleton calls used in each application.



(a) NB, $t = 4$ (b) NI, $a = 1,\ b = 1000$

(c) MM, $m = 128$ (d) CC

Fig. 5. Execution time of four applications against input data size for skel-GPU-gen, skel-GPU-trans, skel-CPU-par, and hand-CPU-seq

hand-CPU-seq: A simple hand-coded sequential program in C++ that performs the same computation as skel-CPU-par after fusion optimization in a sequential manner on the CPU without using our framework.

Fusion optimization was applied to all skeleton programs. Because the macro-only API implementation does not support fusion optimization, we did not use the sequential on-CPU program of each skeleton program generated by the proposed framework. Instead, we used hand-CPU-seq in the experiments.

As shown in Fig. 5, NB and NI had the same tendency: skel-GPU-gen and skel-GPU-trans showed almost the same results and were always better than skel-CPU-par and hand-CPU-seq. From these results, we can see that NB and NI are suitable for GPGPU. For MM, skel-GPU-gen and skel-GPU-trans were better than skel-CPU-par and hand-CPU-seq when the amount of input data was large. This means that the suitability of MM for GPGPU depends on the amount of input data. For CC, skel-GPU-trans was always worse than skel-CPU-par: CC is not suitable for GPGPU due to the transfer of input data.

For all applications, skel-GPU-gen had the best performance except for MM on small input data. This shows that the framework is able to exploit the potential of the GPU. Depending on the application and amount of input data,

Table 2. Effects of fusion optimization on skel-GPU-gen. Number of skeleton calls was statically counted in source code, not counted at runtime

Application	NB	NI	MM	CC
Number of skeleton calls (before/after)	5/4	10/2	4/3	12/7
Maximum speed up (times)	1.00	1.48	1.71	2.44

Table 3. Overhead of skel-GPU-gen compared to hand-GPU-gen on large input data

Application	NB	NI	MM	CC
skel-GPU-gen (%)	27.3 s (100.00)	3.84 s (100.25)	994 ms (108.86)	25.3 ms (116.93)
hand-GPU-gen (%)	27.3 s (100.00)	3.83 s (100.00)	913 ms (100.00)	21.7 ms (100.00)

skel-GPU-trans may be slower than skel-CPU-par. This is due to the inherent properties of the application. It is quite difficult to determine the inherent properties of an application without running the programs. An important and distinguishing point of the proposed framework is that programmers can easily identify such properties by generating programs for both GPUs and CPUs from the same source code and comparing their performance.

Table 2 shows the effects of fusion optimization on skel-GPU-gen under the same condition as the benchmarks in Fig. 5. For each application, the maximum speed up was achieved at largest input data and the minimum was caused at nearly least input data. Overall, the fusion optimization had good effects on the performance in GPGPU.

Table 3 shows the overhead of skel-GPU-gen compared to hand-GPU-gen: a hand-coded parallel program in CUDA whose input data was generated on the GPU. The hand-GPU-gen programs of NB, NI, and CC were optimized so as to reuse functions passed to skeletons. The hand-GPU-gen program of MM was mainly implemented using the DGEMM subroutine of the CUBLAS library, which is an implementation of basic linear algebra subprograms on CUDA. The overhead was examined when the amount of input data was larger than or equal to the maximum in the benchmarks in Fig. 5. For NB and NI, which are suitable for GPGPU, there was very little overhead. For MM, although CUBLAS is a well optimized library, there was a little overhead. For CC, because hand-GPU-gen avoided recomputation efficiently and elaborately, there was the largest overhead of the four applications.

8 Related Work and Discussion

8.1 Skeletal Parallel Programming

Many skeletal parallel programming environments provide skeletons as libraries. Muesli [8], eSkel [9], Quaff [10], and SkeTo [11] are libraries implemented in C/C++ with MPI for distributed memory systems such as PC clusters. BlockLib

[18] is a library implemented in C equipped with C preprocessor macros for the Cell Broadband Engine processor. Our framework differs from these approaches in that the target is GPGPU.

Some implementations have optimization mechanisms for skeleton calls. The FAN skeleton framework [20] supports automatic rule-based program transformation; however, the transformation is ad hoc and requires many rules. Grelck and Scholz [21] presented three optimizers that merge with-loops, which are used for array skeletons, in a SaC [22] compiler. Their optimizer was focused on multi-dimensional different-bounds arrays. SkeTo supports an optimizer [23] that partially implements Hu et al.'s fusion [5,6] for BMF-based list skeletons. The SkeTo optimizer does not support `zipwith` fusion at all, which ours supports.

8.2 GPGPU Programming

Stream programming [24] has been proposed for efficiently exploiting stream processors. Brook for GPUs [25] supports stream programming for GPUs.

MapReduce [26,27] is a programming model that efficiently exploits large-scale PC clusters in the back-end of search engines. MapReduce systems for GPUs have been developed; Mars [28] is optimized for CUDA, and Merge [29] dispatches tasks to both GPUs and CPUs.

Stream programming is similar to BMF-based skeletal programming from the viewpoint that both compose operations of a specific data structure. However, in stream programming, the data structure is restricted to streams, while BMF can be extended to various data structures. MapReduce resembles BMF-based skeletal programming because both use higher-order functions. However, MapReduce does not treat the composition of higher-order functions. Therefore, BMF-based skeletal parallel programming, like our framework, has higher abstraction and wider generality than stream programming and MapReduce.

Lee et al. [30] proposed an embedded language and its online compiler for using GPUs in Haskell. Although they employed the idea of skeletons, their main challenge is to use GPUs with monads in Haskell. Their approach differs from ours in two respects: it had significant overhead and it did not support fusion optimization. Lee et al. [31] developed OpenMP optimized for CUDA, which is directive-based approach compared to our skeletal approach.

9 Conclusion

We have developed a skeletal parallel programming framework for GPGPU programming that has a fusion optimizer. The framework enables rapid GPGPU application development.

There are two directions for future work. One is to add other skeletons to enrich applications. More applications can be described using our framework if `scan` and `shift` are introduced. Thus, we will demonstrate expressiveness of skeletons. The other is to improve the fusion analyzer. In the current implementation, the fusion optimization is a local optimization. The fusion optimizer can perform a more powerful global optimization if the fusion analyzer gathers data flow among basic blocks. In addition, we want to enhance the fusion analyzer to check some part of contracts and restrictions of APIs at compile time.

Acknowledgments. We wish to thank Masato Takeichi, Kenetsu Hanabusa, Zhenjiang Hu, Kiminori Matsuzaki, and other POP members in Tokyo for their fruitful discussions. This work was partially supported by Grant-in-Aid for Scientific Research (20500029) from the Japan Society of the Promotion of Science.

References

1. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1989)
2. Wadler, P.: Deforestation: Transforming programs to eliminate trees. In: Ganzinger, H. (ed.) ESOP 1988. LNCS, vol. 300, pp. 344–358. Springer, Heidelberg (1988)
3. Chin, W.: Safe Fusion of Functional Expressions. In: 7th ACM Conference on Lisp and Functional Programming, pp. 11–20. ACM Press, New York (1992)
4. Gill, A., Launchbury, J., Peyton Jones, S.L.: A Short Cut to Deforestation. In: Conference on Functional Programming Languages and Computer Architecture, pp. 223–232 (1993)
5. Hu, Z., Iwasaki, H., Takeichi, M.: An Accumulative Parallel Skeleton for All. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 83–97. Springer, Heidelberg (2002)
6. Iwasaki, H., Hu, Z.: A New Parallel Skeleton for General Accumulative Computations. International Journal of Parallel Programming 32, 398–414 (2004)
7. Emoto, K., Matsuzaki, K., Hu, Z., Takeichi, M.: Domain-Specific Optimization Strategy for Skeleton Programs. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 705–714. Springer, Heidelberg (2007)
8. Kuchen, H.: A Skeleton Library. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 85–124. Springer, Heidelberg (2002)
9. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible Skeletal Programming with eSkel. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 761–770. Springer, Heidelberg (2005)
10. Falcou, J., Sérot, J., Chateau, T., Lapreste, J.T.: QUAFF: efficient C++ design for parallel skeletons. Parallel Comput. 32(7-8), 604–615 (2006)
11. Matsuzaki, K., Emoto, K., Iwasaki, H., Hu, Z.: A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In: 1st International Conference on Scalable Information Systems, vol. 13 (2006)
12. Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C., Lefohn, A.: GPGPU: General-Purpose Computation on Graphics Hardware. In: ACM SIGGRAPH 2004 Course Notes (2004)
13. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A Survey of General-Purpose Computation on Graphics Hardware. Comput. Graph. Forum 26(1), 80–113 (2007)
14. Bird, R.: Lecture Notes on Theory of Lists. STOP Summer School on Constructive Algorithmics (1987)
15. Skillicorn, D.B.: The Bird-Meertens Formalism as a Parallel Model. In: Software for Parallel Computation. NATO ASI Series F, vol. 106, pp. 120–133 (1993)
16. Gorlatch, S.: Systematic Efficient Parallelization of Scan and Other List Homomorphisms. In: Fraigniaud, P., Mignotte, A., Robert, Y., Bougé, L. (eds.) Euro-Par 1996. LNCS, vol. 1124, pp. 401–408. Springer, Heidelberg (1996)

17. NVIDIA Corporation: NVIDIA CUDA^{TM} Programming Guide Version 2.2 (2009)
18. Ålind, M., Eriksson, M.V., Kessler, C.W.: BlockLib: A Skeleton Library for Cell Broadband Engine. In: 1st International Workshop on Multicore Software Engineering, pp. 7–14 (2008)
19. Harris, M.: Optimizing Parallel Reduction in CUDA. Technical report, NVIDIA Corporation (2007), `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`
20. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards Parallel Programming by Transformation: The FAN Skeleton Framework. Parallel Algorithms Appl. 16, 87–121 (2001)
21. Grelck, C., Scholz, S.: Merging compositions of array skeletons in SAC. Parallel Comput. 32(7-8), 507–522 (2006)
22. Scholz, S.B.: Single Assignment C: efficient support for high-level array operations in a functional setting. J. Funct. Program. 13(6), 1005–1059 (2003)
23. Matsuzaki, K., Kakehi, K., Iwasaki, H., Hu, Z., Akashi, Y.: A Fusion-Embedded Skeleton Library. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 644–653. Springer, Heidelberg (2004)
24. Kapasi, U., Dally, W.J., Rixner, S., Owens, J.D., Khailany, B.: The Imagine Stream Processor. In: 20th IEEE International Conference on Computer Design, pp. 282–288 (2002)
25. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream Computing on Graphics Hardware. ACM Trans. Graph. 23, 777–786 (2004)
26. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: 6th Symposium on Operating System Design and Implementation, pp. 137–150 (2004)
27. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51, 107–113 (2008)
28. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: A MapReduce Framework on Graphics Processors. In: 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 260–269 (2008)
29. Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.: Merge: A Programming Model for Heterogeneous Multi-Core Systems. In: 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 287–296 (2008)
30. Lee, S., Chakravarty, M.M.T., Grover, V., Keller, G.: GPU Kernels as Data-Parallel Array Computations in Haskell. In: Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (2009)
31. Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In: 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 101–110 (2009)