The Journal of Supercomputing (2019) 75:3810–3841
https://doi.org/10.1007/s11227-019-02749-1

NoT: a high-level no-threading parallel programming method for heterogeneous systems

Shusen Wu[1] · Xiaoshe Dong[1] · Xingjun Zhang[1] · Zhengdong Zhu[1]

Published online: 10 January 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract

Multithreading is the core of mainstream heterogeneous programming methods such as CUDA and OpenCL. However, multithreaded parallel programming requires programmers to handle low-level runtime details, making the programming process complex and error prone. This paper presents no-threading (NoT), a high-level no-threading programming method. It introduces the association structure, a new language construct, to provide a declarative runtime-free expression of different data parallelisms and avoid the use of multithreading. The NoT method designs C-like syntax for the association structure and implements a compiler and runtime system using OpenCL as an intermediate language. We demonstrate the effectiveness of our techniques with multiple benchmarks. The size of the NoT code is comparable to that of the serial code and is far less than that of the benchmark OpenCL code. The compiler generates efficient OpenCL code, yielding a performance competitive with or equivalent to that of the manually optimized benchmark OpenCL code on both a GPU platform and an MIC platform.

Keywords High-level parallel programming · Language construct · Association structure · Heterogeneous system · OpenCL

1 Introduction

Heterogeneous processors such as graphic processing unit (GPU), many integrated cores (MIC) have become important for building high-performance computer systems. With hundreds or thousands of simple cores, heterogeneous processors are particularly suited for computation-intensive or large-scale data parallel applications.

✉ Xiaoshe Dong
xsdong@xjtu.edu.cn

Shusen Wu
wuss153@stu.xjtu.edu.cn; xsdong@mail.xjtu.edu.cn

[1] School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, Shaanxi, China

Compute Unified Device Architecture (CUDA) [1] and Open Computing Language (OpenCL) [2] have emerged as mainstream languages for heterogeneous systems. However, programming with CUDA or OpenCL still involves enormous difficulties. Apart from the close-to-mental programming interface and low-level architecture abstraction, the main obstacle is the multithreaded parallel programming approach. Multithreaded programming allows a programmer to control the parallel execution directly and thus provides flexibility in parallelism expression. This can result in good performance but also means that programmers need to conduct task partition and thread mapping explicitly and manage runtime data movement, communication and synchronization. These tasks will lead to complex programming logic and make programming difficult and error prone. In addition, to fully utilize the massive parallelism of heterogeneous processors, large amounts of concurrent threads are required. The management of these threads challenges user programming skills. Moreover, as presented in [3, 4], the organization details of threads have a large effect on the utilization of computing units and memory access efficiency, resulting in diverse performances on different hardware. Applications require manual and particular optimization according to the hardware features to maintain satisfactory performance on different processors or systems. This increases the programming burden and weakens the portability and scalability of an application.

A straightforward solution is the compiling technique. Using source-to-source compilers [5–7], an existing application written in C or OpenMP could be translated to CUDA or OpenCL automatically, shielding heterogeneous programming details. However, without high-level semantic support, the carrying out of parallelism extraction during this translation will be difficult and inefficient. The directive-based OpenACC [8] provides only a higher-level programming interface than that of CUDA and OpenCL. Its programming model inherits CUDA and OpenCL and reserves the multithreading feature. To get rid of multithreading, higher-level semantics is required.

The research on general parallel programming languages and models provides some hints for addressing the multithreading problem. A high-level language construct such as the *Forall* statement is adopted in HPF [9] and the Chapel language [10]. C++ AMP [11] introduces the *parallel_for_each* structure. These methods provide an implicit data parallelism expression. The MapReduce model [12] invokes a large-scale data processing application. The parallel programming logic is simplified to a *map* process and a *reduce* process. Several programming languages [13, 14] employ the *map* structure for data parallelism expression. However, compared with the multithreaded approach, these structures support only specific data parallel patterns and lack flexibility in parallel programming. Their scope of parallelism expression is also limited.

In this paper, we present NoT, a high-level no-threading programming method. The core of the NoT method is a newly introduced language construct, i.e. the association structure. The association structure builds the connection between data structures and the computing process and indicates how the data are partitioned to form parallel computing instances at runtime. The NoT method achieves declarative data parallelism expression via the association structure and maintains the flexibility, similar to the multithreading approach. The NoT

programmer describes the data parallelism of the application via the association structure without paying attention to runtime implementation, thereby releasing the programmer from the heavy burden of low-level programming details. Meanwhile, the association structure records the high-level data parallel pattern. With the association structure as the guideline, the compiler and runtime system can automatically map the application onto different hardware environments, keeping the high-level application unified while ensuring satisfactory performance on different systems.

We designed C-like syntax for the association structure and built a source-to-source compiler and prototype runtime system using OpenCL as an intermediate language. The source-to-source compiler generates an executable OpenCL code from the NoT application. The runtime system encapsulates OpenCL APIs and takes charge of the automatic thread mapping and execution management. Multiple benchmarks rebuilt by the NoT method are compiled and tested on both a GPU and an MIC platform. Compared with the hand-written and optimized OpenCL code in the benchmark, the generated code yields a competitive or equivalent performance on both platforms, illustrating the effectiveness of the method and the efficiency of the compiler and the runtime system. In this paper, the NoT method uses OpenCL as an intermediate language for experimental verification, but it is not limited to OpenCL. The NoT method can be implemented on top of other programming methods with different source-to-source compilers. The NoT method is an open-source project at https://github.com/wusspsj/the-NoT-project.

The contributions of this paper include the following:

1. The introduction of an association structure. We present the association structure and the definition of semantic rules, which provides a declarative and runtime-free data parallelism organization, in contrast to multithreading.
2. The C-like syntax design of the association structure and language extensions. We employ the C-like syntax design for the NoT method to provide an easy-to-learn and easy-to-use user programming interface.
3. A compiler and runtime implementation that adopt OpenCL as an intermediate language. The compiler and runtime system conduct automatic threading mapping and data management, supporting the high-level cross-platform feature.
4. Multiple reconstructed benchmarks and a detailed experimental test comparing the generated code to hand-tuned OpenCL programs on both the GPU and MIC platforms. The generated code yields a performance competitive with or equivalent to that of the native OpenCL code on both platforms.

The rest of the paper is organized as follows. Section 2 presents the related works. Section 3 describes the overall design and semantic rules of the association structure. Section 4 presents the syntax and language extensions of the NoT method with an example. Section 5 introduces the implementation of the source-to-source compiler and the prototype runtime system. Section 6 discusses the experimental evaluation. Section 7 concludes the paper.

2 Related works

MPI [15] and OpenMP [16] are classic multi-process/multithreaded programming models and are the de facto industry standards for parallel programming. MPI cannot be directly used to program heterogeneous processors such as GPUs, which use simplified processing cores and reduced instruction sets. OpenMP provides heterogeneous support from version 4.0. CUDA [1] and OpenCL [2] are typical heterogeneous programming methods. CUDA provides a C-like kernel language and runtime programming interface that enable high-level language GPGPU programming. The programming model of OpenCL is similar to that of CUDA. As an open unified standard, it defines a unified programming interface and accesses to different processors via vendor runtime support. OpenACC [8] provides a directive-based heterogeneous programming method. Its programming model inherits CUDA and OpenCL, but the execution on different heterogeneous processors relies on the compiler. Trellis [17] provides a single set of directives derived from both OpenMP and OpenACC. Trellis emphasize the structured code feature, which is preserved in the NoT method through modular programming. Kokkos [18] provides high-level abstractions for fine-grain data parallelism and memory access patterns in a C++ library. Martineau et al. [19] evaluate Kokkos against CUDA and OpenCL. The conclusion shows that Kokkos is a promising option for performance portability, but it still requires up-front investment in code migration and exposes additional complexity to achieve good performance.

To simplify heterogeneous parallel programming, automatic compiling techniques that involve mapping existing applications to heterogeneous programming methods are studied. hiCUDA [6] provides a set of directives that guide the compiler to convert serial codes to CUDA programs automatically. Qilin [5] provides a C/C++ compatible programming interface and implements an adaptive mapping mechanism that adaptively maps computing tasks to both the CPU and GPU in a heterogeneous system. OpenMPC [20] extends OpenMP to achieve finer-grained OpenMP-to-CUDA conversion. An OpenMP-to-OpenCL compiler is implemented in [7]. A GPU runtime code generation framework that employs a scripting-based approach is introduced, along with the PyCUDA and PyOpenCL, in [21]. Automatic mapping or compiling techniques hide the underlying heterogeneous parallel programming method and extend the scope of existing applications. However, they are limited by the programming abstraction of existing applications, and the lack of sufficient parallel information lowers the mapping efficiency of the compiler, resulting in performance decay. The NoT method designs the association structure to preserve high-level parallel patterns and provide the basis for efficient compilation and runtime management. PetaBricks [22] introduces *choices* in high-level programming as the guideline for compilation and runtime optimization, which is similar to our approach. However, PetaBricks concentrate on performance and user-guided tuning, while the NoT method concerns programming productivity first.

High-level parallelism expressions have long been implemented via new syntax structures and statements. HPF [9] adds *Forall* statements to provide

vector-based data parallel expressions. The Chapel language [10] also adopts the *Forall* statement and introduces the domain concept to extend the scope of the Forall statement, increasing the flexibility of programming. C++AMP [11] provides the *parallel_for_each* structure. Compared with the association structure, these statements lack support for different data parallel patterns. Their scope of parallelism expression is also limited.

Domain-specific approaches such as OptiML [23] and Halide [24] provide languages of restricted expressiveness focused on a particular domain. Using domain-specific notation and constructs, the heterogeneous details are implicit. The NoT method adopts the idea and designs the association structure for high-level parallelism expression in general programming.

With the rise in streaming processors, especially GPUs that fit stream programming models, the research on streaming programming languages and frameworks has seen great progress, e.g. StreamIt [25], BrookGPU [26], and the Sponge [27] compilation framework, which maps StreamIt to CUDA. StreamPI [28] and Lime [29, 30] integrate stream programming and object-oriented features. StreamMDE [31] presents a stream programming framework that schedules task and data parallelism in the message-driven execution paradigm. The stream programming model aims at stream applications and processors, which narrows the generality of parallelism expression. In addition, these methods are usually difficult to learn and use, which affects programming productivity.

MapReduce [12] provides a programming model for large-scale parallel processing. The Merge framework [32] provides a library-oriented high-level parallel programming language based on MapReduce and a corresponding compiler and runtime system. At the same time, the *map* structure has become an important method for data parallel expression such as Copperhead [13] and HiDP [14]. The *map* structure is similar to the previous *Forall* statement and *parallel_for_each* structure. The parallelism expressiveness of a single structure is limited. SkePU [33, 34], SkelCL [35] and Triolet [36] introduce *skeletons* including the *map* structures and additional structures such as *zip* and *scan*. The skeletons improve the flexibility in parallelism expression, but the evaluations show that the programming effort has not been reduced very much with skeletons. Lift [37, 38], NOVA [39] and Futhark [40] enable functional programming on heterogeneous systems with multiple parallel structures similar to *map* and *reduce*. However, functional languages are quite difficult to learn and use and could not benefit from legacy code written in C-like languages. The NoT method employs C-like syntax design and implements the association structure with identifiers. The language extensions of NoT method are easy to learn and easy to use. The NoT method enables C-compatible programming, which is familiar to heterogeneous computing programmers and minimize code migration.

3 Association structure

3.1 Association structure definition

When programming using either CUDA or OpenCL, it is necessary to find a kernel that is capable of processing different data in SIMD or SPMD mode and then perform

thread mapping to organize data parallel computing tasks into multiple computing instances for parallel execution. Figure 1 shows the parallelization process of vector addition. Mapping a task onto multiple threads is done to distinguish between various independent computing instances via the threads and divide the data range for each computing instance. The thread binds the data and the kernel.

If a high-level programming language is capable of describing how the data and kernel are bound to form independent computing instances and retain the parallelism pattern, it can guide the compiler and the runtime system to conduct automatic thread mapping, avoiding the use of multithreading in high-level programming.

Consider a mapping that represents a computing process:

$$f : D_1 \times \cdots \times D_n \to R_1 \times \cdots \times R_m \tag{1}$$

Each computing instance processes a data tuple $(d_1, d_2, \ldots, d_{n+m})$ as long as the following occurs:

$$\begin{cases} d_i \in D_i, & 1 \le i \le n; \\ d_i \in R_{i-n}, & n < i \le n+m. \end{cases} \tag{2}$$

For different independent data tuples, a corresponding number of computing instances can be generated for parallel computation.

Consider the data sets of a computing task:

$$\{DS_1, \ldots, DS_n, \ldots, DS_{n+m}\} \tag{3}$$

If the data sets can be partitioned into independent data tuples, data parallelism will occur. The number of independent data tuples determines the maximum number of computing instances, that is, the data parallelism degree of the computing task.

Consider a data unit d_i in a data tuple. The corresponding data set of d_i is DS_i. The relationship between d_i and DS_i can be described as follows:

$$\begin{cases} 1.\ DS_i \subset D_i \ or \ R_{i-n}, & d_i \in DS_i; \\ 2.\ DS_i \in D_i \ or \ R_{i-n}, & d_i \subsetneq DS_i; \\ 3.\ DS_i \in D_i \ or \ R_{i-n}, & d_i = DS_i. \end{cases} \tag{4}$$

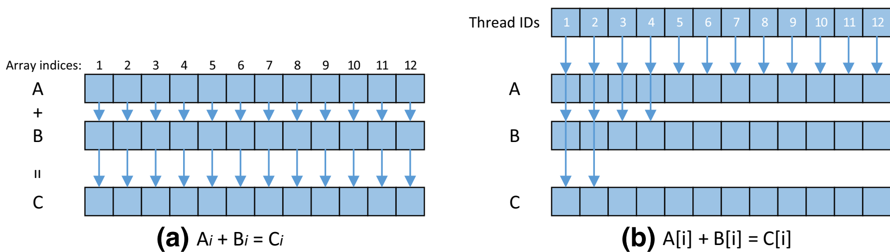

Fig. 1 Parallelization of vector addition

In Eq. 4-1, each element of DS_i can be a candidate for d_i. The range of d_i is the size of DS_i. In Eq. 4-2, DS_i can be partitioned into several subsets with no intersection. Each subset can be a candidate for d_i. The number of subsets determines the range of d_i. In Eq. 4-3, DS_i is the only candidate for d_i and cannot be partitioned. The range of d_i is 1.

If all the data units in the data tuple are orthogonal, the number of independent data tuples is the product of the range of each data unit. However, taking the algorithmic logic of a computing task into consideration, the partition of each data set can be non-orthogonal.

For example, as shown in Fig. 1, the following formula states the kernel of vector addition:

$$C_i = A_i + B_i \tag{5}$$

The element indices of different vectors in each computing instance must be the same. Therefore, the partition of the three vectors is non-orthogonal. The actual degree of parallelism is the length of a single vector rather than the product of the length of each vector.

We introduce partition identifiers to label the data. These identifiers are categorized into three kinds: the element identifier, the subset identifier and the full set identifier, according to the relationship between the data unit and the data set discussed above. The partition identifiers indicate how the data set is partitioned into data units of a tuple. An identifier can be applied to multiple data, retaining the inner relationship between the data. Regardless of the relationship between the data, the identifiers are always orthogonal. We define the association structure based on the identifiers.

An association structure is a 3-tuple, i.e. (*ID*, *DS*, *DP*), where *ID*, *DS*, *DP* are nonempty finite sets and

1. *ID* is the set of identifiers. It is the union of three sets *E*, *S*, and *F*. *E* is a collection of element identifiers, *S* is a collection of subset identifiers and *F* is a collection of full set identifiers, where

$$\begin{aligned} E &= \{e_i \mid 1 \le i \le k, k \in \mathbb{N}\}, \\ S &= \{s_i \mid 1 \le i \le l, l \in \mathbb{N}\}, \\ F &= \{f_i \mid 1 \le i \le m, m \in \mathbb{N}\}; \end{aligned} \tag{6}$$

2. *DS* is a collection of data sets, where

$$DS = \{DS_i \mid 1 \le i \le n, n \in \mathbb{N}^*\}; \tag{7}$$

3. *DP* is the set of data-identifier tuples, where

$$DP = \{(DS_i, p_i) \mid 1 \le i \le n, \quad p_i \in E \mid S \mid F\}. \tag{8}$$

To illustrate the effect of the identifiers, we define the semantic rules for the association structure. *para*() represents the parallelism degree of an identifier, *count*() indicates the size of a data set, *sp*() denotes the number of subsets after the data

set partition, and P indicates the total data parallel degree of a task. The semantic rules are defined as follows:

1. If $e_i \in E$ and $(DS_j, e_i) \in DP$, each element in DS_j can satisfy the need of a calculation process. $para(e_i) = count(DS_j)$. In a computing instance, the element indices of each data set identified by e_i should be the same;
2. If $s_i \in S$ and $(DS_j, s_i) \in DP$, DS_j can be split into multiple subsets at runtime and computed in parallel by multiple compute instances. $para(s_i) = sp(DS_j)$. Other data sets identified by s_i should be split into the same number of subsets. The subsets of different data sets processed by the same computing instance should correspond in sequence;
3. If $f_i \in F$ and $(DS_j, f_i) \in DP$, DS_j cannot be partitioned. $para(f_i) = 1$;
4. $P = \prod para(e_i) \times \prod para(s_i)$.

3.2 The effectiveness of association structure

Mattson et al. [41] summarize three important parallel patterns in the algorithm structure design space, the divide–conquer pattern, the geometric decomposition pattern and the pipeline pattern. The form of concurrency with the geometric decomposition pattern is also known as the coarse-grained data parallelism. The pipeline pattern is adopted in data flow programming [42] and stream programming. Kaeli et al. [43] further classify parallel programming strategies into two categories, the divide–conquer strategy and the scatter–gather strategy. The element identifier unrolls a data set and transforms a problem into subproblems. It describes the data partition feature of the divide–conquer strategy and is suitable for fine-grained data parallelism expression. It also identifies a collection of data elements needing the same processing, which is the core of the pipeline pattern. The subset identifier partitions a data set into subsets. It describes the key feature of the scatter–gather strategy and the geometric decomposition pattern and is capable of expressing coarse-grained parallelism. In summary, the association structure is effective in expressing parallelisms with different patterns and different granularities.

The parallelism degree and the data parallel pattern of a computing task can be expressed via the association structure. The compiler can distinguish between different parallel computing instances by analysing the data and association structure to conduct automatic thread mapping, thus avoiding the use of the multithreading in high-level programming. Meanwhile, the semantics of different identifiers provide the basis for runtime management. The semantic rules of the element identifier and the subset identifier indicate runtime data access pattern and define runtime multi-data access manner. The full set identifier marks the data objects that may cause runtime data race. The runtime system could carry out efficient data management according to the identifiers. The parallelism degree illustrates the parallel scale of a task. Based on the parallelism degree, the runtime system could adjust the execution and scheduling on a specific platform.

A simple association structure $(ID_{va}, DS_{va}, DP_{va})$ can be defined to present the parallelism in vector addition shown in Fig. 1 and Eq. 5, where the $ID_{va} = \{e_{va}\}$,

$DS_{va} = \{A, B, C\}$, $DP_{va} = \{(A, e_{va}), (B, e_{va}), (C, e_{va})\}$. The three data objects of vector addition are signed by the same identifier e_{va}. The three vectors will be unrolled correspondingly, forming into a number of 3-tuples. A 3-tuple consists of three scalars, and each scalar comes from a vector, respectively. The parallelism degree of vector addition equals the number of 3-tuples, which is the length of each vector. Moreover, the parallel programming of the vector addition is simplified to serial programming of scalar addition, saving much programming effort.

4 Syntax design

This section shows the syntax design and main programming features of the NoT method. The syntax design of NoT has two main goals. The first is to provide user-friendly programming interfaces, reduce the difficulty of programming, and improve the flexibility of programming; the second is to provide sufficient information as a guideline for the compiler and runtime systems to accomplish code generation, thread mapping and data management automatically and efficiently. The NoT method employs a C-like syntax design for the association structure and other language extensions, providing a C-compatible programming interface which is easy to learn and easy to use. The NoT method provides three modularized components, namely the data, the association structure and the calculation kernel, to organize the computing tasks. A data association calculation (DAC) expression that combines the three components represents a computing task. The end of this section offers two examples to illustrate the usage of the NoT method and the simplicity of NoT programming.

4.1 Data structure

Compared to the data type and data content, the size and organization of the data and how the data are partitioned are issues of greater concern in parallel programming. A discretized data structure is more convenient for parallelism expression. The NoT method provides a unified vector data representation. The dimensions of the vector and the dimension values represent the size and organization of the data. Meanwhile, the multi-dimensional vector and the set representation can easily be transformed into each other, as shown in Fig. 2.

The informal data declaration is defined as follows:

$$DAC_data\ data_name[dim0] \ldots [dimN];$$

Legal data declarations start with the *DAC_data* identifier and must assign the data name. The dimension and dimension values of the data are optional. The *DAC_data*

Fig. 2 Conversion from high-dimensional vector to nested set

A[2][3]

	0	1	2
0	0	1	0
1	1	0	1

→ {{0,1,0},{1,0,1}}

Table 1 Data attribute references

Attributes	in *main*()	in calculation kernel
Data type	*.type*	*.type*
Dimension	*.dim*	Not allowed
Dimension value	*.range*[*i*]	*.ri*

Table 2 NoT data manipulation interfaces

Definition	Description
DAC_fill < *data list* > {…}	Mark the code segments of write operations to the data in the data list
DAC_get < *data list* > {…}	Mark the code segments of read operations to the data in the data list
DAC_shape(*name*, *type*[, *range*0, …])	Complement the data attributes of the data with *name*; dimension values within the brackets are optional

identifier distinguishes between data objects and normal parameters. The data are the operation objects of a calculation kernel, as defined in Sect. 4.2.

The data type does not appear in the data declaration. Instead, the data type, the dimensions of the data and the dimension values are initialized as built-in attributes of the data. The runtime system is responsible for creating built-in variables for each data attribute. Thus, these data attributes are parameterized, which can be set via the NoT APIs or referenced later in other functions. Table 1 shows the reference format of data attributes in the main function and the calculation kernel. The parameterization of data attributes can improve the flexibility of programming and the portability of an application. Parameterized data type provides a flexible way for parameter declaration and avoids the modification to high-level applications caused by data type changes. Parameterized data dimension and dimension values are convenient for the calculation of the parallelism degree. Being used as control variables, they release the user from defining such parameters and prevent potential overflow error.

Table 2 shows the NoT data manipulation interfaces. The *DAC_shape* provides the interface for data attribute supplementation or updating. The parameters of the *DAC_shape* interface are the data name, the data type and the dimension values of the data. Each dimension value can be any integer constant or variable, but it must be decidable at runtime. The number of dimension values implies the dimensions of the data. The data declaration has a higher priority for the specification of the data dimensions and dimension values. If the dimension values are specified in the data declaration, they can be omitted in *DAC_shape* parameter list. When a dimension value specified by *DAC_shape* is inconsistent with the data declaration, the compiler will report an error. If the number of dimension values specified in *DAC_shape* exceeds the data dimensions of the data declaration, the extra dimension values will be discarded.

The *DAC_fill* and *DAC_get* functions tag data objects and the following write and read operations to the data, providing a basis for runtime data consistency maintenance

and data management. Based on the data tag, the compiler can insert precise data-consistency operations before data read or write operations, and the runtime system can manage data movement automatically according to the specific runtime environment.

4.2 Calculation kernel

A calculation kernel is a function with no return value. The output data contain the result of the calculation. The calculation kernel is closed within its visible data range specified by its data list. By partitioning high-level data sets into different data ranges, multiple computing instances can be generated. Each computing instance operates within its own data range independently. The computing process inside a kernel is performed serially at runtime. Runtime data management, communication and synchronization operations do not concern the programming of the calculation kernel.

To distinguish it from ordinary functions, the calculation kernel is identified by the keyword *DAC_calc*. The definition of the calculation kernel is as follows:

```
DAC_calc calc_name(arguments) < data list >
{
    computing process;
}
```

The arguments and data list distinguish normal parameters and the computing data objects. The data list defines the data interface of the calculation process. The dimensions of each data parameter need to be specified in the data list. Since the data type becomes an attribute of the data and does not appear in the data declaration, the dimensions of each data parameter become the sole basis for data inspection. The data inspection converts from type checking to dimension checking. As long as the dimensions of the data conform to the data interface of the calculation kernel, the data can be processed by the calculation kernel without being limited by the data type.

The inside computing process is programmed using the C language. The majority of the computing process is compatible with legacy serial code, which reduces the programming effort. In addition, data attributes can be referenced. The intermediate variable can be defined by the *.type* reference. The values of each dimension of the data can also be obtained by calling *.ri*, where *i* is the serial number of dimensions, which starts from zero. The compiler can generate the corresponding runtime code according to the data attributes, avoiding manual modification when the data attributes change.

4.3 Association structure

The association structure describes how the data are partitioned into multiple independent sub-data that meet the requirements of the calculation kernel data interface. The informal definition of the association structure is as follows:

$$
\begin{aligned}
&DAC_shell\ shell_name() < data\ list > \\
&\{ \\
&\qquad association\ structures; \\
&\}
\end{aligned}
$$

The association structure is identified by the *DAC_shell* identifier. The data list defines the data interface of the association structure. The dimensions of each data parameter in the data list are also needed to be specified. The main body of the association structure specifies the input/output data and the partitioning method for each data parameter.

The input/output relationship between the data parameters is marked with the identifier <=>. The scope of the identifier is the single line separated by semicolons. On the left side of the identifier are the input data, while the output data are on the right side; either side can be blank. The principle for distinguishing between input and output data is whether there is a write operation applied to the data during the calculation. If so, it is output data. The input–output relationship provides a basis for automatic data copying at runtime.

According to the definition in Sect. 3.1, we designed the syntax for the partition identifiers, as shown in Table 3. These well-defined identifiers are easy to learn and easy to use.

Chapel [10] introduced concept of index variable. However, the index variables in NoT method are quite different from that of Chapel. Firstly, index variables of the NoT method only appear in the association structure. Secondly, they follow the naming conventions of C variables but do not need prior declaration. Lastly, they Follow the semantic rule for element identifiers. if an index variable signs a dimension of the data, the data will be expanded and partitioned into low-dimensional sub-data at runtime according to the dimension signed by the index variable. The same index variable can be used multiple times to sign different dimensions of different data. For different data signed by the same index variable, their sub-data in the same computing instance should have the same index in the dimension signed by the index variable. When the data parameter is a high-dimensional vector, its different dimensions can be signed by different index variables. According to the relationship between a vector and the set representation shown in Fig. 2, different index variables actually sign different nested sets. The data will be expanded recursively.

We define *sp* as the subset identifier. The data labelled by the *sp* identifier will be partitioned into several small-scale sub-data that maintain the dimensional

Table 3 Syntax for partition identifiers

Identifiers	Input data	Output data
Element	Index variables	Index variables
Subset	*sp*	*sp*
Full set	*bg*	*atomic*

characteristics of the data. The number of partitions is determined by the runtime system. The informal definition of the *sp* identifier is as follows:

$$sp < data\ list > [(post - processing)]$$

The scope of the *sp* identifier is the data list within the immediate pair of angle brackets. Each datum in the data list of *sp* identifiers is partitioned into the same amount of sub-data at runtime. In the same computing instance, the subsets of different data should share the same sequence number. Data within the scope of different *sp* identifiers are partitioned independently, and the subsets of different data can be freely combined. For example, the usage of the sp identifier could be as follows:

$$a) : sp < A, B <=> C > ; \qquad b) : sp < A, B ><=> sp < C > .$$

As shown in example (a), the scope of the *sp* identifier can span the input–output identifiers. Examples (a) and (b) are not equivalent. In example (a), all data are within the scope of the same *sp* identifier. At runtime, data A, B and C follow the same partition method. In example (b), data A, B and C are within the scope of the different *sp* identifiers, with the partition of data C being independent of data A and B at runtime.

At the end of the *sp* identifier definition is the optional call to the post-processing process. The post-processing process is a special kind of calculation kernel that conducts a reduction process. It is used only for the output data. In some cases, the partitioned output data contain only the intermediate results. Then, there is the need to merge the intermediate results with the post-processing process.

For the data that cannot be partitioned during the calculation, we design different identifiers for the input and output data separately. For the input data, conflict read to the same data would occur between multiple parallel instances, which mainly affect the efficiency of the memory access. For the output data, multiple parallel instances will carry out simultaneous write operations on the same data. Write conflicts will cause the data consistency and correctness problems. Therefore, we design the *bg* identifier for the input data and the *atomic* identifier for the output to indicate the corresponding optimization at runtime. The *bg* and *atomic* identifiers are used as follows:

$$bg < data\ list > ; \qquad atomic < data\ list > .$$

These two identifiers imply the input/output relationship and therefore can appear only on the corresponding side of the input/output identifier or in a single line. Since the data labelled by these two identifiers are indivisible among parallel instances, the *bg* and *atomic* identifiers do not affect other partition identifiers.

4.4 DAC expression

The combination of the data, association structure and the calculation kernel constitutes the parallel computing tasks. That is the DAC expression. Its informal definition is as follows:

$$< data\ list > => shell_name(calc_name(arguments));$$

The connector => combines the three modules. The rules of combining these modules lie in the matching of data interfaces. The association structure builds a distribution pipeline from data to the calculation kernel. The computing data must satisfy the data interface requirements of the association structure, and the data tuple produced by the association structure must satisfy the data interface requirements of the calculation kernel. First, we define the following:

1. $data_list$, $data_list_{shell}$ and $data_list_{calc}$ as the data list of the DAC expression, the association structure and the calculation kernel. The $num(data_list_x)$ represents the number of data parameters in one of these data lists.
2. δ is a data item in $data_list$. δ_{shell} and δ_{calc} are the formal data parameters corresponding to δ in $data_list_{shell}$ and $data_list_{calc}$, respectively. The $dim(\delta_x)$ represents the dimensions of the data item or a formal parameter.
3. δ^*_{shell} is one of the sub-data of δ_{shell}, after being partitioned at runtime according to the partitioning identifiers, such that:

$$dim(\delta^*_{shell}) = \begin{cases} dim(\delta_{shell}) - x, & \delta_{shell} \text{ signed by index variables;} \\ dim(\delta_{shell}), & \delta_{shell} \text{ signed by other identifiers.} \end{cases} \quad (9)$$

where x is the number of dimensions signed by index variables in δ_{shell}.

Then, we define the combination rules as follows:

1. $num(data_list) = num(data_list_{shell}) = num(data_list_{calc})$;
2. $\forall \delta \in data_list, dim(\delta) = dim(\delta_{shell}), dim(\delta^*_{shell}) = dim(\delta_{calc})$.

Rule 1 indicates that the number of parameters in the data lists of the DAC expression, the association structure and the calculation kernel must be equal. Rule 2 ensures that the dimension requirements are satisfied. The modularity of NoT method and combination rules allow flexible task expression and enable reuse of existing modules. Different combinations of the modules can express different tasks. A new application can start with existing modules, reducing the programming effort.

4.5 NoT examples

Figure 3 illustrates the matrix multiplication implemented using the NoT method. Line 4 shows the flexible data declarations. Data a, b and c are the three matrices of matrix multiplication. Data attributes can be omitted in data declaration, as shown in data a and c, or partially omitted, as shown in data b. The omitted data attributes can be supplemented by the *DAC_shape* interface. Lines 5–7 use the *DAC_shape* interface to set the type attribute of all data to *int* and complement the dimension values of data a and b. Datum c is specified as a 10*10 two-dimensional integer vector. Lines 8–16 and lines 18–21 of the example use the *DAC_fill* and *DAC_get* interfaces, respectively, to mark data read and write operations. In the content of *DAC_fill*, the *.range* references to the data dimension values are used as the upper bound of the loop control.

```
1   int main(void)
2   {
3       int i,j;
4       DAC_data a[][],b[100][],c[][];
5       DAC_shape(a,int,10,100);
6       DAC_shape(b,int,100,10);
7       DAC_shape(c,int,10,10);
8       DAC_fill<a,b>
9       {
10          for(i=0;i<a.range[0];i++)
11              for(j=0;j<a.range[1];j++)
12              {
13                  a[i][j] = 1;
14                  b[j][i] = 2;
15              }
16      }
17      <a,b,c> => mtov(vm);
18      DAC_get<c>
19      {
20          printf("%d\n",c[5][5]);
21      }
22      return 0;
23  }//end main
24  //DAC_shell example
25  DAC_shell mtov()<a[][],b[][],c[][]>
26  {
27      <a[i][],b[][j]> <=> <c[i][j]>;
28  }
29  //DAC_calc example
30  DAC_calc vm()<a[],b[],c>
31  {
32      int i;
33      a.type num;
34      num = 0;
35      for(i=0;i<a.r0;i++)
36      {
37          num += a[i]*b[i];
38      }
39      c = num;
40  }
```

Fig. 3 Matrix multiplication in NoT

Lines 25–28 represent the association structure *mtov*. Line 27 shows the usage of index variables. Index variable *i* signs the first dimension of matrices *a* and *c*. Index variable *j* signs the second dimension of matrices *b* and *c*. At runtime, *a* is partitioned by rows, and *b* is partitioned by columns. The row vectors of *a* and the column vectors of *b* are freely combined since they are, respectively, signed by different index variables. Matrix *c* will be partitioned into scalars. These two independent index variables determine the data parallelism degree. The dimension values of the dimension signed by index variable *i* and *j* are 10 and 10, respectively. Then, the maximum number of independent data tuples is 10*10, i.e. 100. Therefore, up to 100 parallel computing instances can be generated at runtime.

Lines 30–40 illustrate the calculation kernel of matrix multiplication, namely vector multiplication. As shown in line 33, an intermediate variable can be defined by the data type of data *a* with the *.type* reference. The dimension value of the data can also be obtained by the *.r*0 reference in line 35.

Line 17 of the example illustrates the core of NoT programming: a DAC expression that associates data *a*, *b*, and *c* with the calculation kernel *vm* via the association structure *mtov*. Data *a*, *b*, and *c* satisfy the data interface requirements of the associated structure. The data tuple containing a row vector of *a*, a column vector of *b* and a scalar of *c* satisfies the data interface of the calculation kernel *vm*.

As shown in the example, the NoT method organizes the data, association structures, and calculation kernels around DAC expressions. The programming logic is simple and clear, enabling runtime-free no-threading programming. Compared with the serial C code shown in Fig. 4, the addition of the association structure only slightly increases the amount of code. The extra code overhead is small.

The second example is the implementation of the magnetic resonance imaging non-cartesian Q matrix calculation(MRI-Q) from the Parboil benchmark suite [44]. MRI-Q is a real-world application in MRI image reconstruction, which is a conversion from sampled radio responses to magnetic field gradients. Sample coordinates are in the space of magnetic field gradients or *k-space*. The Q matrix in MRI image reconstruction is a precomputable value based on the sampling trajectory, and the plan of how points in k-space will be sampled. Each element of

```
1   int main(void)                                   18  //matrixmul example
2   {                                                19  void matrixmul(int *A,int *B,int *C,int rowA,int colA,int colB)
3       int i,j;                                     20  {
4       int *a,*b,*c;                                21      int i,j,k,num;
5       a = (int *)malloc(sizeof(int)*10*100);       22      num = 0;
6       b = (int *)malloc(sizeof(int)*100*10);       23      for(i=0;i<rowA;i++)
7       c = (int *)malloc(sizeof(int)*10*10);        24      {
8       for(i=0;i<10;i++)                            25          for(j=0;j<colB;j++)
9           for(j=0;j<100;j++)                       26          {
10          {                                        27              for(k=0;k<colA;k++)
11              a[i*100+j] = 1;                      28                  num += A[i*colA+k]*B[k*colB+j];
12              b[j*10+i] = 2;                       29              C[i*colB+j] = num;
13          }                                        30          }
14      matrixmul(a,b,c,10,100,10);                  31      }
15      printf("%d\n",c[5][5]);                      32  }
16      return 0;
17  }//end main
```

Fig. 4 Matrix multiplication in C

the Q matrix is computed by a summation of contributions from all trajectory sample points. Each contribution involves a three-element vector dot product of the input and output 3D locations.

The implementation of MRI-Q includes two calculation kernels. Figure 5 shows the association structures and calculation kernels of MRI-Q. The first kernel calculates the magnitude of a complex vector, which is the Fourier transform of the spatial basis. The association structure *vtos* unrolls the vector to scalar using a index variable. Note that the *vtos* can also be used in general vector operations such as vector addition. The reuse of exiting structures can reduce the programming effort, showing the advantage of the modularity feature. The other kernel calculates the Q signals of the sample coordinates. The Q signals are also a complex vector. The association structure *mriq* unrolls the sample coordination vector and the Q vector to scalar using a index variable. The *k-space* and the precomputed magnitude vector are shared by all computing instances.

Table 4 shows the lines of code with different kernel implementations. The benchmark OpenCL implementation needs 42 lines, while the *DAC_calc* and the *DAC_shell* need only 26 lines in total. The novel functional programming approach Lift [38] does not seem to be appropriate for such application. The lines of Lift

```
1   DAC_shell vtos()<a[],b[],c[]>{
2       <a[i],b[i]> <=> c[i];
3   }

4   DAC_shell mriq()<kx[],ky[],kz[],pmag[],x[],y[],z[],qr[],qi[]>{
5       <x[i],y[i],z[i]> <=> <qr[i],qi[i]>;
6       bg<kx,ky,kz,pmag>;
7   }

8   DAC_calc ComputePhiMag()<pr,pi,pm>{
9       pm = pr*pr + pi*pi;
10  }

11  DAC_calc ComputeQ()<Kx[],Ky[],Kz[],PhiMag[],x,y,z,Qr,Qi>{
12      float expArg, cosArg, sinArg, phi;
13      int indexK;
14      cosArg = 0.0;
15      sinArg = 0.0;
16      for (indexK = 0; indexK < Kx.r0; indexK++) {
17              expArg = PIx2 * (Kx[indexK] * x +
18                          Ky[indexK] * y +
19                          Kz[indexK] * z);
20              phi = PhiMag[indexK];
21              cosArg += cos(expArg)*phi;
22              sinArg += sin(expArg)*phi;
23      }
24      Qr = cosArg;
25      Qi = sinArg;
26  }
```

Fig. 5 The association structures and calculation kernels of MRI-Q

Table 4 The comparison between different MRI-Q kernel implementations

	DAC_calc	*DAC_calc* & *DAC_shell*	OpenCL	Lift
Lines of code	19	26	42	43

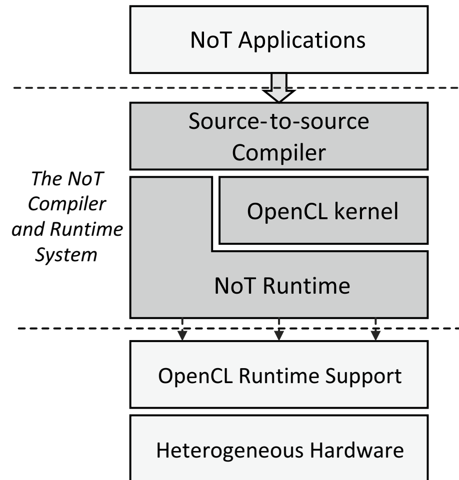

Fig. 6 NoT implementation architecture

kernel implementation reaches 43. The NoT method shows the advantage of simplicity on certain application.

5 Implementation

Section 4 illustrates the programming interface of the NoT method. In this section, we show the implementation of mapping high-level NoT applications to heterogeneous systems via OpenCL. Figure 6 shows the architecture of the implementation. The NoT-to-OpenCL compiler parses the NoT syntax and converts high-level applications into executable code that conforms to the OpenCL standard, passing high-level parallel information, including the parallel pattern and parallelism degree, over to the runtime system. The NoT runtime system encapsulates the OpenCL APIs and achieves automatic thread configuration and data management. Since the primary goal of the implementation is to verify the feasibility of the NoT programming method, the implementation in this paper targets single-device execution. The NoT runtime system selects the first computing device of the first OpenCL platform in the system by default. Multi-device support will be studied in future works.

5.1 Kernel generation

The source-to-source compiler translates NoT application to standard OpenCL code. As stated in the OpenCL specification [2], an OpenCL application is implemented

as both the host code and device kernel code. The host code submits the kernel to OpenCL devices, while the kernel code executes on the device in parallel to carry out a computing task. Even though the NoT runtime system encapsulates the OpenCL APIs and simplifies the programming logic of the host code, the OpenCL kernel generation is essential for the compiler.

The kernel generation starts with the DAC expressions. The compiler can parse out the name of the calculation kernel from the DAC expression. The body of the calculation kernel is the basis for generating the OpenCL kernel. However, to generate the OpenCL complement kernel code, several issues must be addressed.

First, the data interfaces of the NoT kernel and the OpenCL kernel are different. The NoT kernel addresses the partitioned sub-data, while the OpenCL kernel addresses the original data. The OpenCL kernel arguments require careful configuration. The data reference in the NoT kernel should also be converted from the sub-data to the original data. Second, the OpenCL kernel employs built-in thread indices and runtime index interfaces to organize parallel threads. The compiler needs to insert the index interfaces correctly and complete the data reference conversion with the thread indices. Third, the data attribute references in the NoT kernel must be dereferenced.

During the kernel generation, the compiler parses all the data in the DAC expression and resolves the data attributes. In addition to the normal parameters specified in the argument list of the NoT kernel, each data parameter involved in the calculation and its dimension values are set as the OpenCL kernel arguments. The dimension values of the data are necessary in data reference conversion and data attribute dereference. The compiler parses every data attribute reference and locates the corresponding data object, each *.type* reference is replaced with the specific data type and the *.ri* reference is replaced with the corresponding data dimension value argument listed in the kernel argument list.

The index space of OpenCL is called the NDRange. It is defined by three integer arrays: the global size array, the offset index array and the local size array. The offset array and the local size array can be initialized by default by the OpenCL runtime environment. The configuration of the global size array, which assigns the extent of the index space, is the job of runtime thread mapping. The number of elements in the global size array equals the dimensions of the NDRange. It determines the number of built-in indices. The NoT implementation selects the partition identifiers to set up the global size array. The compiler needs to know the number of selected partition identifiers to insert OpenCL APIs in the kernel to obtain the value of the built-in indices. The compiler parses the association structure in the same DAC expression. The NDRange is at most three dimensional. If the total numbers of index variables and *sp* identifiers are less than or equal to three, all of them are selected. Otherwise, only the top three identifiers can be selected. The compiler inserts the same number of index interface calls to the selected partition identifiers to obtain the thread indices.

For each data reference in the calculation kernel, the compiler can parse the local coordinate of the data reference with respect to the partitioned sub-data. Then, the compiler needs to conduct the conversion from the local coordinate in the sub-data to the global coordinate in the original intact data. The association structure is the

basis of the conversion. If the corresponding data in the association structure is not signed by the index variables or the *sp* identifiers, then the data will not be partitioned at runtime, and the coordinate requires no modification. If the corresponding data are signed by index variables, the local coordinate will lose the indices of the dimensions signed by the index variables. If the index variable is selected to set up the NDRange, the related thread index can be used to complete the coordinate. Otherwise, the compiler inserts an additional argument into the kernel argument list for the index variable and uses the argument to supplement the coordinate. When the data are signed by the *sp* identifier, the highest dimension of the data will be split at runtime to form multiple sub-data if the *sp* identifier is selected to set up the NDRange. The index of the highest dimension in the local coordinate should add an offset according to the thread index to conduct conversion to the global coordinate. If the *sp* identifier is not selected, then the data will not be partitioned during execution. The local coordinate is the global coordinate.

The OpenCL kernel standard does not support high-dimensional vector expression. Once the compiler obtains the global coordinate, the single-dimensional index of the data reference in the original data is calculated using the index in each dimension and the dimension values. The dimension values of each piece of data are presented in the kernel argument list.

During kernel generation, the compiler sets up a linked list for each NoT kernel to record the generated OpenCL kernels and corresponding data type attributes and association structure. The name of each generated OpenCL kernel consists of the NoT kernel name and a serial number, which represents the kernel generation order. When the same NoT kernel is invoked in a different DAC expression, the compiler checks the data list and association structure of the DAC expression. If the data type attributes and the association structure change, the *.type* dereference, NDRange setup and data reference conversion will be affected, resulting in different OpenCL kernels. For each DAC expression, the compiler searches the generated kernel list of the corresponding NoT kernel and checks whether the data type attributes and the association structure in the DAC expression match those in the generated kernel list. If the corresponding OpenCL kernel is already generated, then the compiler returns the kernel name. Otherwise, a new kernel is generated.

The OpenCL kernel generated from the NoT example in Fig. 3 is shown in Fig. 7.

5.2 Thread mapping

The NoT runtime system employs the OpenCL API beneath the NoT runtime functions to conduct the system check and execution environment configuration, shielding most of the details of the host code. In addition, the most important job of the NoT runtime is to handle the kernel execution.

The core of kernel execution is thread mapping. As stated earlier, the thread index space NDRange is defined by three arrays. The NoT runtime needs to assign the global size array according to the selected partition identifiers. The parallelism degree of each identifier is used to set the global size in each dimension, generating as many threads as possible to utilize the massively parallel processing units in

```
__kernel void vm1(__global int *a,int a_r0,int a_r1,__global int *b,/
int b_r0,int b_r1,__global int *c,int c_r0,int c_r1)
{
size_t tID0,tID1;
tID0 = get_global_id(0);
tID1 = get_global_id(1);
        int i;
        int num;
        num = 0;
        for(i=0;i<a_r1;i++)
        {
                num += a[tID0*a_r1+(i)]*b[(i)*b_r1+tID1];
        }
        c[tID0*c_r1+tID1] = num;
}
```

Fig. 7 OpenCL kernel generated from the NoT matrix multiplication example

the heterogeneous system. In the selection, the index variables are superior to the *sp* identifiers since they usually yield a higher parallelism degree. For each index variable, the parallelism degree of the identifier is the dimension value of the signed data dimension. For each *sp* identifier, the runtime initializes a segmentation parameter with a value of 64 by default. The segmentation parameter determines the parallelism degree of the identifier. The data are partitioned at the highest dimension. The number of sub-data is the same as for the segmentation parameter. When the segmentation parameter exceeds the highest dimension value of the data, a loop concession that divides the segmentation parameter by two each time is conducted to ensure that the segmentation parameter is within the extent of the highest dimension and is a power of two.

After setting the NDRange with the selected partition identifiers, the redundant partition identifiers need to be addressed. The compiler inserts a kernel argument for each redundant index variable during kernel generation. The NoT runtime is responsible for setting those arguments correctly. In the implementation, the OpenCL kernel launch interface is wrapped in a loop. The NoT runtime initializes a counter array for these arguments. Every state of the counter array triggers the launch of the kernel. The NoT runtime increases the counter to obtain the correct argument values in each kernel execution and completes the computing process via multiple kernel invocations. The redundant *sp* identifiers have no effect on the kernel generation and can be omitted at runtime.

5.3 Data management

Since the runtime system supports only single-device execution, the work of data management can be greatly simplified. In the OpenCL memory model, the kernel accesses the data via the memory object. If the memory of the computing device is separated from the main memory, the memory objects are created, along with device-side memory allocation and data movement from the main memory to the

device. If the device shares the memory with the host, the memory object can be set to point to the data in the main memory directly during creation.

For most heterogeneous systems with a separated memory space, the latest version of the data needs to be copied from the main memory into the device memory ahead of kernel launch. After kernel execution, the computed results need to be copied back. Automatic data management usually employs redundant data copy to ensure the correctness of the execution. However, frequent data movement may cause serious performance decay due to a limited bandwidth. To reduce the number of redundant data copies and the IO overhead, we designed a timestamp-based data management mechanism.

1. Initialize the timestamp for each data and memory object upon data initialization and device-side memory object creation.
2. After the host-side write operation is completed, check the timestamp of the data and the corresponding memory object, and then update the timestamp of the data, ensuring that it is up to date.
3. Check each input data and its corresponding memory object immediately prior to kernel launch. If the memory object does not exist, create one. Check the timestamp of the input data and the corresponding memory object. If the timestamp of the memory object is not up to date, copy the data in and update the timestamp of the memory object to match that of the input data.
4. After kernel execution, check the timestamp of each output data and its corresponding memory object, and update the timestamp of the memory object to ensure that it is up to date.
5. Check the timestamp of the data and the corresponding memory object right before executing the host-side read operation. If the timestamp of the data is not up to date, copy the data from the device to the host and update the timestamp of the data to match that of the memory object on the device.

The timestamp guarantees that data movement occurs only when necessary. Both the host and the device can maximize the reuse of the data and reduce the number of data copies in automatic data management. For a shared memory system, the timestamp mechanism is bypassed by the NoT runtime system.

6 Experimental evaluation

To verify the feasibility of the NoT programming method and demonstrate the efficiency of the compiler and runtime system, this paper selects several typical applications from the Parboil benchmarks [44] and reconstructs them using the NoT method. The Parboil benchmarks were developed by the impact group of Illinois University. For each application, the Parboil benchmarks provide implementations of a variety of programming methods, such as C/C++ and OpenCL, along with data sets of different sizes, making it well suited for comparison tests. The Parboil benchmarks are now part of the SPEC ACCEL benchmark [45].

6.1 Case studies and performance evaluation

The reconstructed applications were compiled and executed on both a GPU and an MIC platform. The test environment configuration is shown in Table 5. An experimental test of the reconstructed code was conducted using the data sets provided by the benchmarks, taking the benchmark OpenCL implementation for comparison. Since the Parboil benchmarks provide only OpenCL implementation for the GPU, the OpenCL code must be manually ported to the MIC platform. Applications built via the NoT method do not need such modification. With the NoT runtime support, the NoT application can be executed smoothly on different platforms after compilation is conducted.

SGEMM Dense matrix multiplication is an important and basic application in numerical linear algebra with a well-understood and easy-to-parallelize computing process. The computing process of SGEMM is mapped into i*j vector multiplication via two index variables in the NoT implementation, as shown in the example of Sect. 4.5.

The benchmark OpenCL code of SGEMM employs the same method to organize threads. Each thread is responsible for calculating an element of the output matrix. The difference is that thread block size is optimized in the benchmark code to take advantage of data locality.

The Parboil benchmarks provide two data sets: *small* and *medium*. The input matrix sizes of the *small* data set are 128*96 and 96*160. The input matrix sizes of the *medium* data set are 1024*992 and 992*1056. The overall execution time is shown in Fig. 8. *OCL_GPU* in the figure represents the execution time of the benchmark OpenCL code on the GPU platform. *OCL_MIC* represents the execution time of the benchmark OpenCL code ported onto the MIC platform. *NoT_GPU* and *NoT_MIC* represent the execution times of the OpenCL code generated automatically by the compiler from the NoT application. The generated OpenCL code can be executed directly on the two platforms without manual modification, demonstrating a good cross-platform feature. As shown in Fig. 8, the performance of the automatically generated code is very close to that of the benchmark code on both platforms.

MRI-Q The NoT implementation of MRI-Q is shown in Sect. 4.5. Figure 9 shows the performance of the generated code compared to the benchmark

Table 5 Experimental environments

	GPU platform	MIC platform
OS	CentOS 6.9	RHEL 6.3
Kernel	2.6.32-696.el6.x86_64	2.6.32-279.el6.x86_64
CPU	Intel Xeon E5620	Intel Xeon E5-2670
GPU/MIC	NVIDIA Tesla C2050	Intel Xeon Phi 7110P
OpenCL package	CUDA 6.5	Intel OpenCL runtime 14.2
OpenCL version	OpenCL 1.1	OpenCL 1.2
Host compiler	GCC 4.4.7	GCC 4.4.6

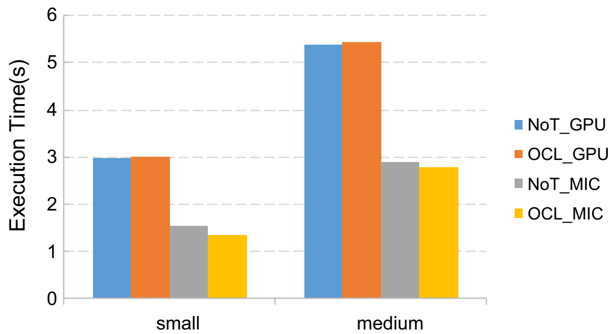

Fig. 8 SGEMM test results

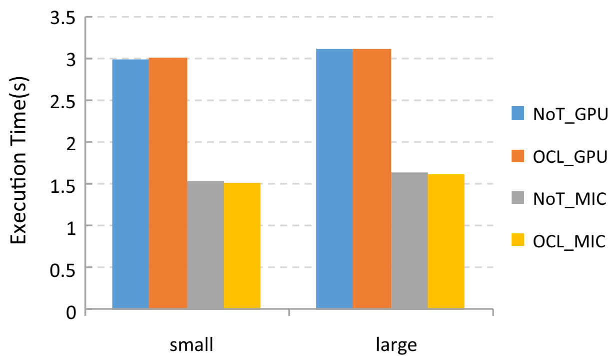

Fig. 9 MRI-Q test results

OpenCL code. The *small* data set contains 32K pixels, and the *large* data set contains 262K pixels. The performance of the generated code is very close to that of the benchmark code on both platforms.

Table 6 further shows the detailed kernel execution time of the generated code and the benchmark code. The speedup on the GPU platform is 0.94x to 1.10x, and the speedup on the MIC platform is 1.10× to 1.21×. According to the performance evaluation of Lift [38], the speedup of optimized Lift kernel is around 1.0× to 1.1. The performance of the NoT method is comparable to that of Lift.

STENCIL The Stencil benchmark implements an iterative Jacobi solver of the heat equation on a 3D structured grid. The number of nodes needed to perform the Jacobi iteration in the 3D grid determines the degree of parallelism. Since the Jacobi iteration on each node relies on all its neighbours, the coordinate of the node is essential. In the NoT implementation, three coordinate arrays are set to obtain the coordinate of each node. These arrays are signed by three index variables to unfold the computing process on the 3D grid to a single node.

According to the definition of the association structure, the same data cannot be input and output at the same time. Therefore, two grids are used in the

Table 6 The execution time of benchmark HISTO OpenCL code on MIC platform

Data set	NoT_GPU	OCL_GPU	Speedup	NoT_MIC	OCL_MIC	Speedup
Small	0.0331	0.0364	1.1020	0.0221	0.0268	1. 2110
Large	0.1392	0.1310	0.9409	0.0595	0.0658	1. 1042

reconstructed code during alternate iterations. The input grid is signed by the *bg* identifier and is accessible to all computing instances.

The grid size of the *small* data set is 128*128*32. The grid size of the *default* data set is 512*512*64. The iteration parameter in both data sets is 100. The benchmark code adopts the basic optimization when setting the thread block size . As shown in Fig. 10, the performance of the automatically generated code on the GPU platform is very similar to that of the benchmark code and is 76% 83% of that of the benchmark code on the MIC platform

The results show a performance decay of the NoT implementation on the MIC platform. We investigate the execution details of the application with the help of the Parboil benchmark timing functions. Tables 7 and 8 present detailed information regarding the execution on the GPU and MIC platforms. The *IO* item includes the time needed to read the input data from the data set and then write the result back to the output file. The *Copy* item is the data movement time between the host and the device. The *Kernel* item is the execution time of the device-side kernel. The *Ocl* item is the time overhead of various OpenCL API calls.

The illustrations of the *IO* item and *Copy* item on both platforms are very similar, showing that the overhead of the automatic data movement is small. The *kernel* item shows that the kernel execution time of *NoT_MIC* is longer than that of *OCL_MIC*. The *Ocl* item shows the NoT runtime introduces obvious overhead on the MIC platform, which is the main reason for the performance decay. Compared to hand-written and optimized benchmark code, the generated code lacks specific optimization and the runtime system involves redundant OpenCL API calls. That shows the shortcomings of the NoT method in automatic kernel optimization and the runtime system implementation. However, the overhead of the NoT runtime is tiny

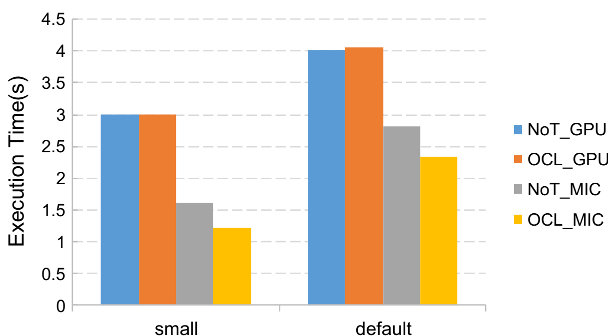

Fig. 10 STENCIL test results

Table 7 Detailed execution time of STENCIL with small data set

	IO	Copy	Kernel	Ocl	Total
NoT_GPU	0.0213	0.0024	0.0104	2.9596	2.9937
OCL_GPU	0.0223	0.0022	0.0096	2.9588	2.9928
NoT_MIC	0.0326	0.0051	0.0669	1.5081	1.6128
OCL_MIC	0.0310	0.0049	0.0350	1.1496	1.2205

Table 8 Detailed execution time of STENCIL with default data set

	IO	Copy	Kernel	Ocl	Total
NoT_GPU	0.6671	0.0461	0.3316	2.9641	4.0090
OCL_GPU	0.6734	0.0457	0.3711	2.9568	4.0469
NoT_MIC	0.6283	0.1130	0.5499	1.5332	2.8246
OCL_MIC	0.6274	0.1028	0.3273	1.2810	2.3385

on the GPU platform and the optimization adopted by the benchmark has little effect on the GPU platform but yields performance improvement on the MIC platform. That shows the differences of these two platforms in OpenCL runtime implementation and in hardware architecture.

SpMV The sparse matrix-dense vector multiplication (SpMV) benchmark of the Parboil benchmarks adopts the jagged diagonal storage (JDS) format to store the sparse matrix. The computing process involves doubly nested loops. The outer loop calculates each element of the output vector. For each output element, the number of nonzero elements in the corresponding row of the sparse matrix is obtained according to the position of the output element. The inner loop conducts the vector multiplication. The NoT implementation unfolds the outer loop via the index variable. Each output vector element and the number of corresponding row elements uniquely determine a computing instance. The other index arrays of the JDS format matrix are signed by the *bg* identifier, which is shared among all the computing instances.

After being converted to JDS format, the sparse matrix provided by the Parboil benchmarks has 1138 columns and 18 rows in the *small* data set, 11,948 columns and 49 rows in the *medium* data set, and 146,689 columns and 49 rows in the *large* data set. The kernel is designed to repeatedly execute 50 times during the test to determine the overall execution time. The benchmark code focuses on the optimization of irregular access to dense vectors, using constant memory in the device to improve the access efficiency and adopting prefetching techniques to hide the latency. As shown in Fig. 11, the performance of the generated code on the GPU and the MIC platforms is 99% to 100% and 80% to 87% that of the benchmark code, respectively.

The results in Fig. 11 also show the performance decay on the MIC platform. Table 9 presents the execution details of the NoT code on the MIC platform. The *Copy* item shows a speedup of the NoT implementation, demonstrating the efficiency of the NoT runtime data management. The *Kernel* item shows that the optimization measures adopted by the benchmark do achieve good performance. Meanwhile, the

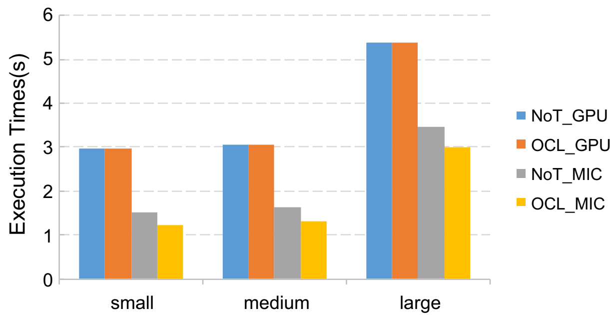

Fig. 11 SpMV test results

Table 9 Detailed execution time of NoT SpMV code on MIC platform

Data set	Code Ver	IO	Copy	Kernel	Ocl	Total
Small	NoT	0.0050	0.0050	0.0225	1.4732	1.5056
	OCL	0.0052	0.0044	0.0175	1.1967	1.2239
Medium	NoT	0.0754	0.0049	0.0247	1.5223	1.6273
	OCL	0.0803	0.0049	0.0172	1.2046	1.3071
Large	NoT	1.7070	0.0184	0.2157	1.5132	3.4544
	OCL	1.7121	0.0124	0.0675	1.2079	2.9999

overhead caused by the NoT runtime is still the main reason for the decrease in the execution efficiency on the MIC platform.

BFS Breadth-first search is a commonly used algorithm in graph computing. The NoT implementation employs two search queues. One is initialized with the source node as the current search queue and the other one caching the search results. The application iterates between two queues until the current search queue is empty. The parallelism degree of the breadth-first search depends on the length of the current search queue. The current search queue is signed by the index variable, while the other one is signed by the *atomic* identifier.

The benchmark provides fine-tuned OpenCL code. Each thread block creates multiple local queues in its private local memory. Each thread in the thread block updates the corresponding local queue via a hash algorithm to reduce the write conflict. Since the parallelism degree of the algorithm dynamically changes, only one thread block is created when the search queue is small. The update of the search queue is then performed in the local memory, which can speed up the processing.

The *NY* data set corresponds to an irregular map obtained from the map abstract of New York City. The *SF* data set involves a near-regular map converted from a scale-free map. The 1*M* data set contains one million nodes. The test results are shown in Fig. 12. The performance of the reconstructed code reaches 94–99% that of the benchmark code on the GPU platform and 88–98% on the MIC platform.

HISTO The histo benchmark is a straightforward histogram operation. By accumulating the occurrences of each output value in the input data set (996*1040), an output matrix (256*4096) is obtained. Because each value in the input data set may

be related to any output values, the parallelization of the histogram operation can be conducted only by partitioning the input data set into subsets. Therefore, in the NoT implementation, the input data set is signed by the *sp* identifier, while the output matrix is signed by the *atomic* identifier. The problem is divided into multiple smaller problems that are processed in parallel.

The benchmark OpenCL code is first optimized from the algorithm level according to the characteristics of the input data set. The input data set roughly follows a Gaussian distribution centred on the output histogram. Optimization in the benchmark code focuses on improving the throughput of data near the centre. The benchmark code builds four kernels to complete the calculation and employs local memory to optimize data access during the kernel implementation.

According to the thread mapping method in Sect. 5.2, the segmentation parameter determines the number of runtime threads in the NoT implementation. The default value of the segmentation parameter is 64. To study the effect of different segmentation parameter values, we obtained the performance curve by setting the segmentation parameters from 4 to 8192. The number of runtime threads is also increased from 4 to 8192. The NDRange configuration of the benchmark OpenCL code remains unchanged in Fig. 13. It can be found that the segmentation parameters, i.e. the number of threads, have a great influence on the performance of the application regardless of the underlying platform.

Because the output data set is much larger than the input one, the probability of atomic operation conflicts is relatively low. As the number of threads increases, the execution time on both platforms shows a downward, yet nonlinear trend. However, the performance curves under the GPU and MIC platforms are different. On the GPU platform, the performance is insensitive to the changes in the number of threads when it is less than or equal to 32. After the number of threads exceeds 32, significant performance improvement begins. The performance gradually approaches that of the optimized benchmark code. On the MIC platform, the execution time initially fluctuates. When the number of threads is 32, the performance is the worst. Then, the execution time decreases as the number of threads increases. The test results on these platforms demonstrate that the number of threads does affect the performance of the application and that the differences between platforms have an influence on the choice of the

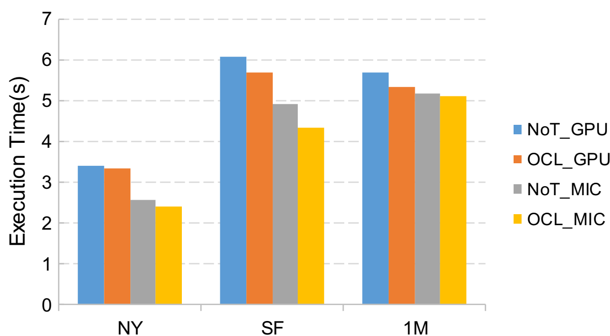

Fig. 12 BFS test results

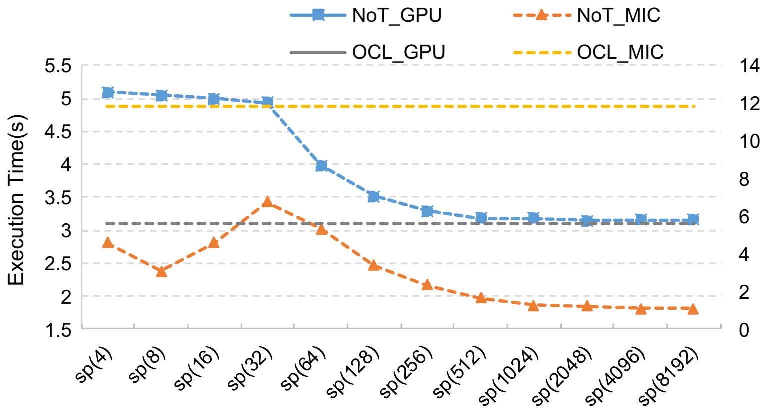

Fig. 13 HISTO test results

number of threads. The adaptive selection method of the segmentation parameters on different platforms deserves further study.

The benchmark OpenCL code performs differently on the GPU platform and on the MIC platform. On the GPU platform, the optimized benchmark code yields satisfactory performance, while the reconstructed code can only gradually approach this performance when the number of threads is large. After porting to the MIC platform, the benchmark code suffers significant performance decay. Table 10 lists the detailed execution times of the benchmark OpenCL code on the MIC platform. It can be seen that the major reason for the performance decay is the overhead of the OpenCL API calls. Combined with the detailed execution data of previous benchmarks, it can be concluded that the Intel OpenCL runtime implementation is sensitive to the number of OpenCL API calls. A few more OpenCL API calls may cause significant performance decay on the MIC platform.

6.2 Programming productivity

The experimental test demonstrates the effectiveness of the NoT method and the efficiency of the compiler and runtime system. In addition, the NoT method can greatly simplify heterogeneous programming. Section 4.5 illustrates the advantages of the NoT method in improving programming productivity with two examples. This simplicity is intuitively reflected by the code size of the reconstructed benchmarks. As shown in Fig. 14, the code size of the applications reconstructed using the NoT method is comparable to that of the benchmark C/C++ code and far less than that of the benchmark OpenCL implementation. Benefitting from the simplified programming logic,

Table 10 The execution time of benchmark HISTO OpenCL code on MIC platform

IO	Copy	Kernel	Ocl	Total
0.0253	0.0749	1.3620	10.3606	11.8229

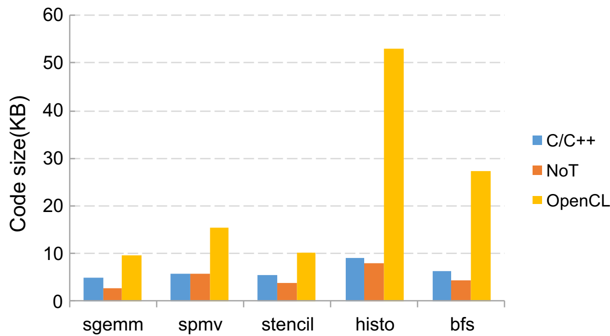

Fig. 14 The code sizes of different implementation

easy-to-use and easy-to-learn syntax design, and runtime system encapsulation of the OpenCL programming interface, the NoT method effectively reduces the workload of heterogeneous parallel programming.

On the other hand, the NoT method implements automatic thread mapping and execution management via the compiler and the NoT runtime, hiding the differences in the underlying hardware architectures and providing a cross-platform feature. The upper application can be executed on different platforms without specific modification. The compiler and the runtime system guarantee the execution efficiency. This saves the programming effort in repeatable application development or migration for different architectures and extends the scope and life cycle of the application.

7 Conclusions

This paper presents NoT, a high-level no-threading programming method for heterogeneous systems based on the association structure. Centred on the association structure, the NoT method expresses the intrinsic data parallelism of an application via the data, association structure and calculation kernel. NoT simplifies heterogeneous parallel programming logic, hides the underlying multithreading details and enables runtime-free machine-independent user programming. The NoT method adopts C-like grammar to design easy-to-use syntax for the association structure, simplifying programming while preserving the high-level parallel information as the guideline for the compiler and the runtime to map high-level applications to different architectures automatically. The compiler and runtime system guarantee the scalability and portability of the application, avoiding specific modification to the upper application and providing unified and cross-platform programming features. This paper employs OpenCL as an intermediate language to implement the source-to-source compiler and the NoT runtime. The source-to-source compiler translates the high-level NoT application to OpenCL. The runtime system encapsulates OpenCL APIs and implements automatic thread mapping and execution management. In the experimental evaluation, multiple benchmarks are reconstructed and tested on different heterogeneous platforms. The code size of the reconstructed benchmarks is

much less than that of the benchmark OpenCL code, showing that the NoT method can effectively reduce the workload and the difficulty of heterogeneous parallel programming. The test results show that the performance of the reconstructed code is similar to that of the hand-written and manually optimized benchmark code. This demonstrates the effectiveness of the NoT method and the efficiency of the compiler and runtime system.

Acknowledgements This work was supported by the National Key Research and Development Program of China (2017YFB0202002) and the National Natural Science Foundation of China (Grant No. 61572394).

References

1. The CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. Accessed 10 May 2018
2. The OpenCL standard. https://www.khronos.org/opencl/. Accessed 10 May 2018
3. Ryoo S, Rodrigues CI, Baghsorkhi SS, Stone SS, Kirk DB, Hwu WW(2008) Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'08, pp 73–82
4. Alberto M, Christophe D, Michael OB (2014) Automatic optimization of thread-coarsening for graphics processors. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT'14, pp 455–466
5. Luk CK, Hong S, Kim H (2009) Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pp 45–55
6. Han TD, Abdelrahman TS (2011) hiCUDA: high-level GPGPU programming. IEEE Trans Parallel Distrib Syst 22(1):78–90
7. Wang Z, Grewe D, O'boyle MFP (2015) Automatic and portable mapping of data parallel programs to OpenCL for GPU-based heterogeneous systems. ACM Trans Archit Code Optim 11(4):1–26
8. The OpenACC Homepage. https://www.openacc.org/. Accessed 10 May 2018
9. High Performance Fortran Forum. http://hpff.rice.edu/. Accessed 10 May 2018
10. Chamberlain BL, Callahan D, Zima HP (2007) Parallel programmability and the Chapel language. Int J High Perform Comput Appl 21(3):291–312
11. C++ Accelerated Massive Parallelism. https://msdn.microsoft.com/en-us/library/hh265137.aspx. Accessed 10 May 2018
12. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
13. Catanzaro B, Garland M, Keutzer K (2011) Copperhead: compiling an embedded data parallel language. ACM SIGPLAN Not 46(8):47–56
14. Zhang Y, Mueller F (2013) Hidp: a hierarchical data parallel language. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO'13, pp 1–11
15. High-Performance Portable MPI. http://www.mpich.org/. Accessed 10 May 2018
16. The OpenMP API specification. http://www.openmp.org/specifications/. Accessed 10 May 2018
17. Szafaryn LG, Gamblin T, Supinski BRD, Skadron K (2013) Trellis: portability across architectures with a high-level framework. J Parallel Distrib Comput 73(10):1400–1413
18. Carter EH, Trott CR, Sunderland D (2014) Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J Parallel Distrib Comput 74(12):3202–3216
19. Martineau M, Mcintosh-Smith S, Boulton M, Gaudin W (2016) An evaluation of emerging many-core parallel programming models. In: Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'16, pp 1–10
20. Lee S, Eigenmann R (2010) OpenMPC: extended OpenMP programming and tuning for GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10, pp 1–11

21. Klöckner A, Pinto N, Lee Y, Catanzaro B, Ivanov P, Fasih A (2012) PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation. Parallel Comput 38(3):157–174
22. Phothilimthana PM, Ansel J, Ragan-Kelley J, Amarasinghe S (2013) Portable performance on heterogeneous architectures. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13), pp 431–444
23. Chafi H, Sujeeth AK, Brown KJ, Lee HJ, Atreya AR, Olukotun K (2011) A domain-specific approach to heterogeneous parallelism. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11), pp 35–46
24. Pu J, Bell S, Yang X, Setter J, Richardson S, Ragan-Kelley J, Horowitz M (2017) Programming heterogeneous systems from an image processing DSL. ACM Trans Archit Code Optim 14(3), Article 26
25. Thies W, Karczmarek M, Amarasinghe S (2002) StreamIt: a language for streaming applications. In: Horspool RN (ed) Compiler construction, CC 2002, pp 179–196, vol 2304. Lecture Notes in Computer Science. Springer, Heidelberg
26. Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P (2004) Brook for GPUs: stream computing on graphics hardware. ACM Trans Graph 23(3):777–786
27. Hormati AH, Samadi M, Woh M, Mudge T, Mahlke S (2011) Sponge: portable stream programming on graphics engines. ACM SIGPLAN Not 46(3):381–392
28. Hong J, Hong K, Burgstaller B, Blieberger J (2012) StreamPI: a stream-parallel programming extension for object-oriented programming languages. J Supercomput 61(1):118–140
29. Auerbach J, Bacon DF, Cheng P, Rabbah R (2010) Lime: a Java-compatible and synthesizable language for heterogeneous architectures. ACM SIGPLAN Not 45(10):89–108
30. Dubach C, Cheng P, Rabbah R, Bacon DF, Fink SJ (2012) Compiling a high-level language for GPUs: (via language support for architectures and compilers). In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12), pp 1–12
31. Su Y, Shi F, Talpur S, Wei J, Tan H (2014) Exploiting controlled-grained parallelism in message-driven stream programs. J Supercomput 70(1):488–509
32. Linderman MD, Collins JD, Wang H, Meng TH (2008) Merge: a programming model for heterogeneous multi-core systems. ACM SIGPLAN Not 43(3):287–296
33. Enmyren J, Kessler CW (2010) SkePU: a multi-backend skeleton programming library for multi-GPU systems. In: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications (HLPP'10), pp 5–14
34. Ernstsson A, Li L, Kessler C (2018) SkePU 2: flexible and type-safe skeleton programming for heterogeneous parallel systems. Int J Parallel Program 46(1):62–80
35. Steuwer M, Kegel P, Gorlatch S (2011) SkelCL: a portable skeleton library for high-level GPU programming. In: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pp 1176–1182
36. Rodrigues C, Jablin T, Dakkak A, Hwu WM (2014) Triolet: a programming system that unifies algorithmic skeleton interfaces for high-performance cluster computing. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14), pp 247–258
37. Steuwer M, Fensch C, Lindley S, Dubach C (2015) Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, pp 205–217
38. Steuwer M, Remmelg T, Dubach C (2017) LIFT: A functional data-parallel IR for high-performance GPU code generation. In: Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization, pp 74–85
39. Collins A, Grewe D, Grover V, Lee S, Susnea A (2014) NOVA: a functional language for data parallelism. In: Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14), pp 8–13
40. Henriksen T, Serup NGW, Elsman M, Henglein F, Oancea CE (2014) Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'17, pp 556–571
41. Mattson T, Sanders B, Massingill B (2004) Patterns for parallel programming. Addison-Wesley Professional, Boston

42. Johnston WM, Hanna P Jr, Millar RJ (2004) Advances in dataflow programming languages. ACM Comput Surv 36(1):1–34
43. Kaeli DR, Mistry P, Schaa D, Zhang DP (2015) Heterogeneous computing with OpenCL 2.0. Morgan Kaufmann, San Francisco
44. Stratton JA, Rodrigues C, Sung IJ, Obeid N, Chang LW, Anssari N, Liu GD, Hwu WW (2012) Parboil: a revised benchmark suite for scientific and commercial throughput computing. http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf. Accessed 10 May 2018
45. The SPEC ACCEL benchmark. http://www.spec.org/accel/. Accessed 10 May 2018