The Journal of Supercomputing (2021) 77:14502–14524
https://doi.org/10.1007/s11227-021-03853-x

Compiler-directed scratchpad memory data transfer optimization for multithreaded applications on a heterogeneous many-core architecture

Xiaohan Tao[1] · Jianmin Pang[1] · Jinlong Xu[1] · Yu Zhu[1]

Accepted: 29 April 2021 / Published online: 15 May 2021

Abstract

The heterogeneous many-core architecture plays an important role in the fields of high-performance computing and scientific computing. It uses accelerator cores with on-chip memories to improve performance and reduce energy consumption. Scratchpad memory (SPM) is a kind of fast on-chip memory with lower energy consumption compared with a hardware cache. However, data transfer between SPM and off-chip memory can be managed only by a programmer or compiler. In this paper, we propose a compiler-directed multithreaded SPM data transfer model (MSDTM) to optimize the process of data transfer in a heterogeneous many-core architecture. We use compile-time analysis to classify data accesses, check dependences and determine the allocation of data transfer operations. We further present the data transfer performance model to derive the optimal granularity of data transfer and select the most profitable data transfer strategy. We implement the proposed MSDTM on the GCC complier and evaluate it on Sunway TaihuLight with selected test cases from benchmarks and scientific computing applications. The experimental result shows that the proposed MSDTM improves the application execution time by 5.49× and achieves an energy saving of 5.16× on average.

Keywords Heterogeneous many-core architecture · Scratchpad memory · Direct memory access · Network-on-chip · Multithreaded application

✉ Jianmin Pang
jianmin_pang@126.com

Xiaohan Tao
txh_0119@126.com

Jinlong Xu
longkaizh@126.com

Yu Zhu
18401653310@163.com

[1] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China

1 Introduction

The heterogeneous many-core architecture is widely used in the fields of high-performance computing and scientific computing [13, 32]. Because it requires a large number of cores [5], which can provide a high degree of parallelism and a high computing speed, reducing the energy consumption of many-core architectures has become a major challenge. To reduce the energy consumption, accelerator cores with local memories are provided in this architecture [13]. However, this means that two different kinds of memories are utilized in this architecture, i.e., off-chip memory and on-chip memory, which results in a more complex storage system, a profound challenge of which is determining how to handle the data transfer between off-chip memory and on-chip memory. For scientific computing applications, when using heterogeneous many-core processors to accelerate computations, the efficient use of storage systems is one of the most critical factors in improving performance and reducing energy.

Scratchpad memory (SPM) [3] is a kind of fast on-chip memory managed by software (a programmer or a compiler), while cache has to query the flag bit which is managed by hardware to check cache misses or hits. Compared with the hardware cache, SPM does not need to perform flag bit judgment and other tasks, and has the advantages of low power consumption and fast access. SPM is initially used in embedded systems to meet the real-time and time predictable requirements of embedded systems [17]. Besides, the Scratchpad memory is also extensively used in FPGAs and is also employed as application-specific caches [33, 34]. The current heterogeneous many-core processors, such as Adapteva Epiphany, Sunway TaihuLight, and IBM Cell, also use SPM to achieve better performance and lower energy consumption. The characteristic of this type of architecture is that each accelerator core has its own SPM that can be accessed at high speed but has limited space.

The SPM is connected to the off-chip memory through a bus [26]. The accelerator cores can only access the data of off-chip memory directly by global load/store instructions or direct memory access (DMA) [11, 29]. The accelerator cores can communicate with each other by a network-on-chip (NoC). However, programmers need to explicitly manage the data transfer between the SPM and the off-chip memory in the application, which hinders program development.

In single-threaded applications, we can use compiler-directed data buffering through DMA to optimize the data transfer between SPM and off-chip memory [4, 23]. This process requires precise analysis of the access patterns and careful management of the data size. With data buffering, global load/store operations to off-chip memory can be replaced with direct accesses to local buffers in SPM without redundant look-up operations. However, multithreaded applications have more synchronization than single-threaded applications. Keeping the coherence of multithreaded applications makes the optimization more complex.

Here, we propose a multithreaded SPM data transfer model (MSDTM) to optimize the data transfer between SPM and off-chip memory on heterogeneous many-core architecture. It first analyzes the application to classify data accesses

and determine the allocation of data transfer with the data transfer allocation (DTA) algorithm. Next, it uses the data transfer performance (DTP) model to derive the optimal granularity of data transfer and select the most profitable data transfer strategy. Then, the code is transformed by the MSDTM with loop distribution and strip-mining. We implement the proposed MSDTM on the GCC complier.

Optimizing data transfer operations by the MSDTM can effectively improve the performance of multithreaded applications and reduce the energy consumption. Since the MSDTM is used in the compilation process, it can also effectively reduce the programming difficulty.

The major contributions of this paper are as follows:

- We propose an algorithm to determine the allocation of data transfer for multithreaded applications with an analysis of data accesses and dependence checking.
- We formulate the data transfer strategy selection problem for multithreaded applications on an SPM-based heterogeneous many-core architecture and design a performance model to derive the optimal granularity of data transfer and select the most profitable strategy.
- We implement and evaluate our proposed model on Sunway TaihuLight with the kernel of scientific computing programs and applications from general benchmarks.

The remainder of the paper is organized as follows. In Sect. 2, we mention some related work, while in Sect. 3, we use a simple example to illustrate our motivation. Section 4 presents the MSDTM. We evaluate the proposed MSDTM in Sect. 5. Section 6 concludes this paper.

2 Related work

2.1 SPM-based heterogeneous many-core architectures

For energy consumption and scalability considerations, some heterogeneous many-core processors choose SPM as a fast on-chip memory. For example, the IBM Cell [6] processor includes a 64-bit PowerPC general-purpose processor core power process element and 8 coprocessor synergistic processor elements (SPEs). Each SPE contains 256 KB of local storage space for storing code and data executed on the SPE, the storage address is private, and threads on different SPEs can communicate only through the main memory. Adapteva Epiphany [15] is an SPM-based many-core architecture that is energy efficient and suitable for embedded systems. Each processor in Adapteva Epiphany consists of an RISC core, a DMA engine, a network interface, and a 32 KB SPM, connected using a 2D mesh grid. This architecture provides a shared address space that allows threads to complete communication by accessing nonlocal SPM. Sunway TaihuLight [12] has a heterogeneous many-core architecture. Each core group includes one management processing element

and 8×8 computing processing elements, and each computing processing element contains a 64 KB local data memory.

2.2 SPM data management

A number of works have optimized SPM data management on heterogeneous many-core architectures. [37] identifies continuous code blocks in memory, such as functions and basic blocks that are executed, and maps them to the SPM area. To improve the performance of embedded systems and minimize energy consumption, [8] performs static analysis on an MPSoC (multicore processor system-on-chip) shared distributed SPM to obtain the optimal memory allocation method. [24] optimizes the compiler on the basis of OpenMP and distributes the array data of the parallel part of applications executed on MPSoC to distributed SPMs. [36] formulates the SPM data allocation problem for multithreaded applications and proposes the NoC contention and latency aware compile-time framework to automatically determine the location of data variables, the replication degree of shared data, and on-chip placement. [20] maps the SPM management problem for data aggregates into the well-understood register allocation problem for scalars to automatically assign static data aggregates in a program to an SPM.

2.3 Data transfer optimization

To optimize SPM data transfer, the direct blocking data buffer (DBDB) [6] is designed and implemented to optimize the use of local memory while providing a simple shared memory programming model for the Cell-BE architecture. [7] develops a model to automatically infer the optimal buffering scheme and size for static buffering, taking into account the DMA latency and transfer rates and the amount of computation in the application loop being targeted. [31] presents optimized buffering techniques and evaluates them for two multicore architectures: quad-core Opteron and the Cell-BE. [30] derives optimal and near-optimal values for the number of blocks that should be clustered in a single DMA command based on the computation time and size of the elementary data items as well as the DMA characteristics.

On heterogeneous many-core architectures, [38] presents an automatic approach to quickly derive a good solution for hardware resource partition and task granularity for task-based parallel applications, in order to exploit spatial and temporal sharing of the heterogeneous processing units. [28] presents a runtime system that automatically optimizes data management on SPM to achieve performance similar to that on the fast memory-only system with a much smaller capacity of fast memory.

To the best of our knowledge, ours is the first work to propose an SPM DTP model for multithreaded applications on SPM-based heterogeneous many-core architectures to reduce the overall application execution time, with evaluation on a real platform.

3 Motivating example

We use a simple motivating example to illustrate the efficiency of optimizing SPM data transfer for multithreaded applications. For illustration purposes, we use the system parameters of Sunway TaihuLight as stated in Tables 1 and 2 in this example [9]. In addition, the start-up overhead of DMA transfers is 300 cycles.

According to Tables 1 and 2, we show the relationship between the time spent on memory access by NoC and the cost of memory access by the DMA with different granularities in Fig. 1.

From the trend of the curve in Fig. 1, we can predict that when the granularity is increased, DMA will result in better transfer efficiency compared to using NoC for data transfer.

The execution time of a multithreaded program consists of the computation time and time spent due to memory access. To simplify our illustration, we assume that

Table 1 Cost of ld/st instructions and NoC

Strategy	Cost (cycles)
Global ld/st instruction	278
SPM ld/st instruction	4
Network-on-chip	10

Table 2 Performance of DMA with different granularities

Granularity	Speed of DMA (GB/s)	Cost of DMA (+300 cycles)
8 B	0.99	12
16 B	1.99	12
32 B	3.92	12
64 B	7.96	12
128 B	15.77	12
256 B	28.88	13
512 B	28.98	27
1024 B	27.97	55
2048 B	30.48	101

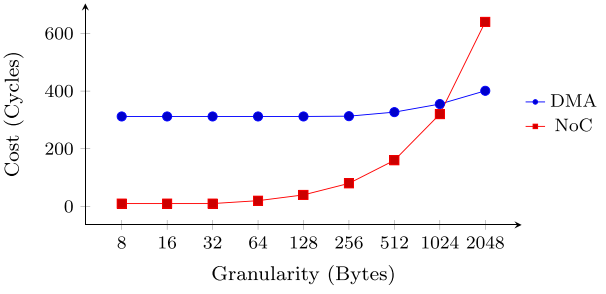

Fig. 1 The relationship between the time spent on memory access and the granularity of the data transfer

the computing performance of each thread is the same; thus, the time spent due to memory access is the only component that reduces the program execution time.

Since we execute multithreaded applications on heterogeneous many-core processors, the time spent due to memory access can be divided into (1) the latency of data access, depending on where the variable is located, i.e., SPM or off-chip memory (*AccessLat*), (2) the latency spent in data communication via the DMA operation (*DMALat*), and (3) the latency spent in data communication via NoC (*NoCLat*). The DMA latency and NoC latency can be divided further into the initialization time, transfer time and delay due to contention among memory requests. The efficiency of DMA transfers on 64 threads is lower than the efficiency on single threads, which is caused by the contention among memory requests. This observation means that the delay due to the contention among DMA requests is already included in the transfer time of DMA. Because NoC latency can be obtained by multiplying the total number of *Hops* and *HopLat*, we do not need to divide the NoC latency into the initialization time, transfer time and delay due to the contention among memory requests.

As shown in Fig. 2, we choose a multithreaded kernel of a scientific computing application that is executed on 64 threads.

Arrays *A*, *B* and *C* are global variables, which means that they need to be accessed from off-chip memory. The variables *id*, *j* and *coeff* are all private variables; thus, they need to be accessed only from the SPM of each core. Therefore, all we need to consider is how to optimize the time spent due to memory access for the *A*, *B* and *C* arrays. From line 6 and 7, we can see that there is a true dependence because the result of variable *B* in line 6 need to be used as a source operand in line 7. But what is different from common true dependence is that the result in line 6 is used by another thread. Here, we name this kind of true dependence as thread-carried true dependence.

In this example, variables *A*, *B* and *C* are allocated in off-chip memory by default. Unlike *A* and *C*, variable *B* is accessed twice in this piece of code. Without any optimization, we need to access the variables from the off-chip memory directly. Each thread issues $4 \times 64 = 256$ accesses to variables *A*, *B* and *C*. The access latency is $256 \times (\textit{off-chip AccessLat}) = 256 \times 278 = 71168$ cycles, which is also the total execution time due to memory access.

We use a data buffer to optimize the data transfer with DMA operations directly. After the transformation procedure, the immediate code is as presented in Fig. 3a.

Variable *A* is explicitly brought from off-chip memory to SPM via DMA. The off-chip memory access latency becomes the sum of the SPM memory access

```
1  extern double A[], B[], C[];
2  __thread_local int id, num, j;
3  __thread_local double coeff, temp[];
4  id = get_core_id();
5  for(j = 0; j < 64; j++) {
6    B[id][j] = ... + coeff * A[id][j];  //S1
7    C[id][j] = ... + coeff * B[(id+1)&7][j];  //S2
8  }
```

Fig. 2 Code without data transfer optimization

```
1  DMA_in (buf1, &A[id][0], 64*8);
2  for(j = 0; j < 64; j++) {
3    buf2[j] = ... + coeff * buf1[j];
4    DMA_out (&buf2[j], &B[id][j], 1*8);
5    DMA_in (&buf3[j], &B[(id+1)&7][j], 1*8);
6    buf4[j] = ... + coeff * buf3[j];
7  }
8  DMA_out (buf4, &C[id][0], 64*8);
```

The granularity of 8 B

```
1  DMA_in (buf1, &A[id][0], 64*8);
2  for(jj = 0; jj < 64; jj += 32) {
3    for(j = 0; j < min(jj+32, 64); j++) {
4      buf2[j] = ... + coeff * buf1[j];
5    }
6    DMA_out (&buf2[jj], &B[id][jj], 32*8);
7    DMA_in (&buf3[jj], &B[(id+1)&7][jj], 32*8);
8    for(j = 0; j < min(jj+32, 64); j++) {
9      buf4[j] = ... + coeff * buf3[j];
10   }
11 }
12 DMA_out (buf4, &C[id][0], 64*8);
```

The granularity of 256 B

Fig. 3 Code with data transfers by DMA with a granularity of 8 B and 256 B

latency and the DMA latency. Because of the thread-carried true dependence, we need to insert DMA operations between the source and the sink to maintain the dependence. The granularity of the two DMA operations is 8 B. The DMA cost can be obtained from Table 2. In addition to DMA operations, we also need to read/write data from the SPM via load/store instructions. The SPM access latency is $(Ld/stLat) \times 4 \times 64 = 4 \times 4 \times 64 = 1024$ cycles. Therefore, the total execution time due to memory access is $AccessLat + DMALat = 1024 + (327 \times 2 + 312 \times 2 \times 64) = 41614$ cycles. Compared to the default strategy, the data buffer has a $\mathbf{1.71}\times$ lower execution time.

We can see that due to the existence of thread-carried dependences, two DMA operations need to be inserted in each loop iteration. However, the initialization time of each DMA operation is relatively long, while the amount of data transferred is small. This situation leads to no profit being gained from using DMA to optimize SPM data transfer. Therefore, as shown in Fig. 3b, we can use loop distribution and strip-mining to transform the code to change the granularity of the data transfer.

With the code transformation, the granularity of the DMA transfer in the loop becomes 256 B. The total execution time due to memory access is $AccessLat + DMALat = 1024 + (327 \times 2 + 313 \times 2 \times 2) = 2930$ cycles. The larger granularity yields us a $\mathbf{24.29}\times$ acceleration.

Next, we attempt to replace the DMA operation in the loop body with NoC for two different granularities in the data transfer process. The process of data transfer using NoC is shown in Fig. 4. Each box means an SPM for a thread. Since NoC can transfer data only by using XY routing in a row or a column, only one thread can send data at a time during the data transfer process.

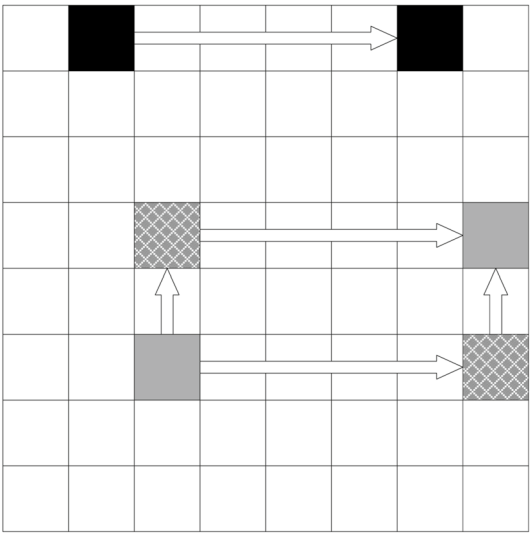

Fig. 4 Structure of data transfer by NoC

The code using NoC to optimize the date transfers is shown in Fig. 5. Because of the structure of NoC, all threads need 8 *Hops* to complete the data transfer in this piece of code. Therefore, when the granularity is 8 B, $NoCLat = 8 \times HopLat = 8 \times 10 = 80$ cycles. The total execution time due to memory access is $AccessLat + DMALat + NoCLat = 1024 + 327 \times 3 + 80 \times 64 = 7125$

```
1  DMA_in (buf1, &A[id][0], 64*8);
2  for(j = 0; j < 64; j++) {
3    buf2[j] = ... + coeff * buf1[j];
4    REG_PUTR (buf2[j], (id + 1)&7, 1*8);
5    REG_GETR (&buf3[j]);
6    buf4[j] = ... + coeff * buf3[j];
7  }
8  DMA_out (buf2, &B[id][0], 64*8);
9  DMA_out (buf4, &C[id][0], 64*8);
```

The granularity of 8 B

```
1  DMA_in (buf1, &A[id][0], 64*8);
2  for(jj = 0; jj < 64; jj += 32) {
3    for(j = 0; j < min(jj+32, 64); j++) {
4      buf2[j] = ... + coeff * buf1[j];
5    }
6    REG_PUTR (&buf2[jj], (id+1)&7, 32*8);
7    REG_GETR (&buf3[jj]);
8    for(j = 0; j < min(jj+32, 64); j++) {
9      buf4[j] = ... + coeff * buf3[j];
10   }
11 }
12 DMA_out (buf2, &B[id][0], 64*8);
13 DMA_out (buf4, &C[id][0], 64*8);
```

The granularity of 256 B

Fig. 5 Code with data transfers by NoC with a granularity of 8 B and 256 B

cycles. When the granularity is 256 B, $NoCLat = 8 \times 80 = 640$ cycles. The total execution time due to memory access is $AccessLat + DMALat + NoCLat = 1024 + 327 \times 3 + 640 \times 2 = 3285$ cycles. The acceleration is **21.67**×, while for a transfer granularity of 8 B, it is **9.99**×.

Figure 6 shows that using DMA and NoC to optimize data transfer can effectively optimize the SPM application and effectively improve the execution efficiency of the multithreaded application. However, when the granularity of the data transfer is different, the optimization effect of using DMA and NoC for data transfer also differs.

Therefore, we propose the MSDTM to achieve more efficient SPM data transfer optimization in multithreaded applications.

4 Multithreaded SPM data transfer model

In this section, we describe in detail the design and implementation of the MSDTM.

Figure 7 presents a high-level overview of the MSDTM framework. The input to the framework is a multithreaded application source code with marked kernel regions. Before we input the source code to the MSDTM, we need to port it to the heterogeneous many-core architecture. To simplify the description of the MSDTM, we focus only on the perfect loop nest wherein all content is in the innermost loop. We perform the loop transformation on the innermost loop.

As shown in Fig. 7, the MSDTM framework consists of three components: application analysis, the DTP model and a code transformation.

4.1 Application analysis

In this stage, we analyze the multithreaded application to obtain the per-thread kernel region memory access profile as the input to the DTP model.

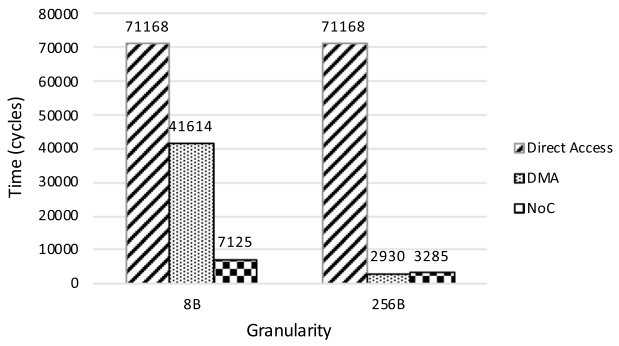

Fig. 6 The time spent due to memory access for the above three data transfer strategies under two different granularities

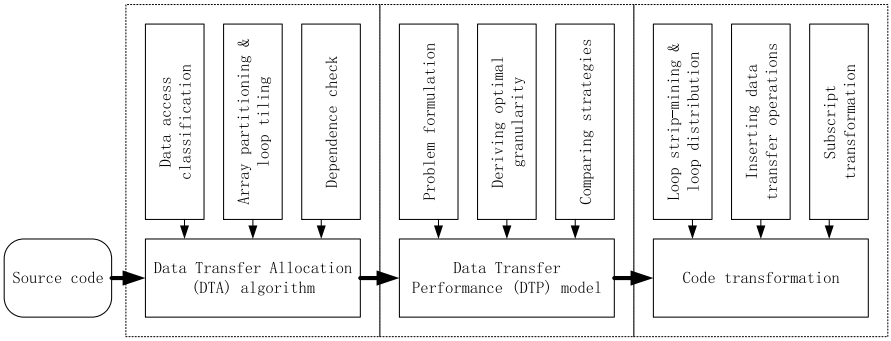

Fig. 7 Workflow of the proposed multithreaded SPM data transfer model (MSDTM)

4.1.1 Data access classification

We traverse the whole marked kernel region to obtain the memory access profile of the global variables. We identify the access types of the global variables: read-only, write-only and read-write. While we bring the data only from off-chip memory to SPM with read-only access, we also need to transfer the data from SPM to off-chip memory with write-only and read-write access.

Moreover, we also classify the global variables as either regular or irregular [8]. Because of the predictable inefficiency, irregular access is ignored by the MSDTM. Furthermore, regular access can be classified as either contiguous or noncontiguous. We then aggregate the access of a variable to one single buffer and insert strided DMA operations for noncontiguous access.

4.1.2 Array partitioning and loop tiling

In most heterogeneous many-core architectures, the SPMs always have restricted space. However, in general, the kernels in an application may access large variables. Most arrays may not be accommodated in the SPM. Array partitioning and loop tiling can separate a large array into smaller ones to accommodate them in SPM [14, 21, 26]. Many mainstream compilers support the use of polyhedral model by programmers to perform automatic array partitioning and loop tiling [22]. Polyhedral model is an abstract representation of a loop program as a computation graph in which questions such as program equivalence or the possibility of parallel execution can be answered [10].

4.1.3 Dependence check

Before we perform the dependence check, we introduce a new kind of dependences, called input dependence [19], in which both the source and the sink use the same

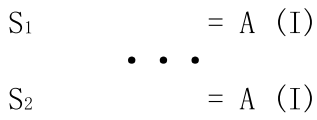

Fig. 8 Schematic diagram of input dependency

location. As Fig. 8 shows, the input dependence from S1 to S2 clearly indicates the opportunity to eliminate a load at the second reference.

We traverse all the memory accesses for dependence checking. The dependences are divided into thread-independent dependences and thread-carried dependences. Thread-independent dependences are used to check whether code transformations are legal, while thread-carried dependences are used to guide transfer operation insertions.

True dependence. The data from the source will be used by the sink; thus, transfer operations will be inserted before the sink to update the data.

Anti-dependence. Nothing needs to be done to achieve antidependence because the data used by the source are already brought to the SPM before the sink updates the data at the same location in the off-chip memory.

Output dependence. If output dependence is the only dependence that exists, only the last thread that updates the data at the same location in the off-chip memory needs to use DMA operations to bring the data from the SPM to the off-chip memory.

Input dependence. The threads with input dependence use the data from the same location in the off-chip memory; thus, inserting transfer operations before the sink may result in better efficiency.

We structure a thread-carried dependence graph (TDG) as a result of a dependence check. We take the code shown in Fig. 2 as an example. There are only two dependences in this piece of code. One is a thread-independent input dependence due to the read operations of scalar *coeff*, and the other is a thread-carried true dependence due to the write and read operations of variable *B*. So, in the TDG of the piece of code shown in Fig. 2, there is only one edge from the write of *B* (S_1) to the read of *B* (S_2). Because thread-carried dependence cannot be backward, no cycle of dependences will occur in the TDG.

4.1.4 Data transfer allocation (DTA) algorithm

To determine the allocation of data transfer operations in multithreaded applications, we propose the DTA algorithm (Algorithm 1). After data access analysis and a dependence check, the loop that needs to be transformed with its data access profile and the TDG are supplied to the DTP algorithm as the input. The main idea of Algorithm 1 is to insert data transfer operations according to TDG and the kind of dependences in TDG. In order to reduce the times of data transfer operations, we only insert one DMA_in or DMA_out operation corresponding to one input or output dependence at the beginning or the end of the loop. According to true

dependences in TDG, we insert a couple of data transfer operations to optimize the process of memory access.

Algorithm 1 Data Transfer Allocation (DTA) algorithm

Require: TDG - thread-carried dependence graph; L - the original loop; n - number of data references in the loop;
Ensure: L' - the loop with data transfer operations;
1: // Let $A = \{a_0, a_1, ...a_{n-1}\}$ be the data references in the loop;
2: // Let $G_i = (V_i, E_i)$ is the subgraph of TDG corresponding to a_i;
3: **for** each G_i **do**
4: $G_i := \emptyset$;
5: **end for**
6: **for** each $a_i \in A$ **do**
7: **for** each $u \in V$,$v \in V$,and u, v contains a_i **do**
8: **if** $\langle u, v\rangle \in E$ **then**
9: $V_i := V_i \cup \{u\} \cup \{v\}$;
10: $E_i := E_i \cup \{\langle u, v\rangle\}$;
11: **end if**
12: **end for**
13: **end for**
14: **for** each $G_i \neq \emptyset$ **do**
15: **if** $\langle u, v\rangle \in E_i$, and $\langle u, v\rangle$ is anti-dependence **then**
16: $G_i := G_i - \{\langle u, v\rangle\}$;
17: $G_n \leftarrow$ the SCC of G_n which contains v;
18: $G_i := G_i - G_n$;
19: $n++$;
20: **end if**
21: **end for**
22: **for** each $G_i \neq \emptyset$ **do**
23: **if** $\exists\langle u, v\rangle \in E_i$, and $\langle u, v\rangle$ is true dependence **then**
24: insert a couple of data transfer operations (copy_in & copy_out) between u and v based on Data Transfer Performance Model;
25: **else if** $\forall\langle u, v\rangle \in E_i$, and $\langle u, v\rangle$ is input dependence **then**
26: insert a DMA_in operation at the prologue of the loop;
27: **else if** $\forall\langle u, v\rangle \in E_i$, and $\langle u, v\rangle$ is output dependence **then**
28: insert a DMA_out operation at the epilogue of the loop;
29: **end if**
30: **end for**
31: **return** L';

We first divide the TDG into several subgraphs according to the data access profile (lines 6–13). Each subgraph contains all the dependences corresponding to the same data access. If any antidependence exists in the subgraph, we further divide the subgraph into two by the antidependence (lines 14–21). Now, we have several subgraphs without antidependences and a cycle of dependences. The dependences are then further classified. With true dependences, we insert a couple of data transfer operations (copy_in & copy_out) between the source and the sink of each dependence (lines 23, 24). Without true dependences, we insert DMA_in operations corresponding to the input dependences at the beginning of the loop, while we insert DMA_out operations corresponding to the output dependences at the end of the loop (lines 25–29). The strategies of the data transfer operations with true dependences are determined by the DTP model mentioned in Sect. 4.2. Since we need to

traverse G_i three times and a_i once, the worst-case time complexity of Algorithm 1 is $O(n)$.

We use the code in Fig. 2 as an example to illustrate the process of Algorithm 1. As we mentioned in Sect. 4.1.3, the TDG of the piece of code in Fig. 2 has only one edge from S_1 to S_2 which indicates there is only one thread-carried true dependence from S_1 to S_2. According to Algorithm 1, since there is no anti-dependence in TDG, we do not need any division of the graph. The only thing we need to do is to insert a couple of data transfer operations between the source (S_1) and the sink (S_2).

4.2 Data transfer performance (DTP) model

With the allocation of the data transfer, the memory access profile and hardware configuration of a specific heterogeneous many-core architecture are input into the DTP model. We first formulate the model for multithreaded applications using a specific hardware configuration. Next, we use the performance model to derive the optimal granularity and select the most profitable transfer strategy at that granularity.

4.2.1 Model formulation

The execution time of a multithreaded application is determined by the slowest thread; hence, we need to select the most profitable transfer strategy to minimize the execution time of the slowest thread. Furthermore, we assume that the execution time of computation is fixed; thus, reducing the execution time due to memory access is the only way to minimize the execution time of the slowest thread. Let T be the execution time due to memory access of the slowest thread in the multithreaded application. As mentioned above, the execution time due to memory access consists of data access latency (*AccessLat*), DMA transfer latency (*DMALat*) and NoC transfer latency (*NoCLat*).

Data access latency: Let $A = \{a_1, a_2, ..., a_n\}$ be the variables that need to access the SPM or off-chip memory. Let s_i represent the size of a_i ($1 \le i \le n$). Let α represent the load/store latency of the SPM, while β represent the load/store latency of the off-chip memory. Both α and β are defined by the hardware configuration. The data access latency is:

$$AccessLat = \begin{cases} s_i \times \alpha, & \text{if } a_i \text{ is on SPM} \\ s_i \times \beta, & \text{if } a_i \text{ is on off-chip memory} \end{cases} \tag{1}$$

For heterogeneous many-core architectures, α is much smaller than β. Therefore, data transfer operations can reduce the data access latency by transferring data from the off-chip memory to the SPM.

Data transfer granularity: As Table 2 shows, different data transfer granularities correspond to different DMA and NoC transfer speeds. We let g represent the granularity of data transfer operations. To obtain the granularity g in a loop, loop strip-mining and loop distribution are utilized during the code transformation. Thus, g is subject to the following constraint:

$$g \leq MIN(s_1, s_2, \ldots, s_n) \tag{2}$$

The cost per byte of DMA transfer can be defined relative to the granularity as:

$$v = f(g) \tag{3}$$

The cost per byte of NoC transfer can be defined relative to the granularity as:

$$u = h(g) \tag{4}$$

Normally, while v usually reduces by an inverse proportional function, u usually remains unchanged as g increases for most heterogeneous many-core architectures. To meet the sizes of variables, each data transfer process requires several data transfer operations. This number of operations (or times) can be computed as:

$$\text{times} = \lceil \frac{s_i}{g} \rceil \tag{5}$$

DMA transfer latency: Each DMA transfer operation can be divided into an initialization and a transfer process. Let I represent the initialization cost. At a transfer granularity of g, the latency of the DMA transfer per item is:

$$DMALat = [I + g \times f(g)] \times \text{times} \tag{6}$$

Both the initialization cost and the speed of the DMA transfer are determined by hardware parameters.

NoC transfer latency: The NoC transfer experiences contention in a link when several other transfers are simultaneously trying to utilize the same link. Because of the contention, all the threads need to transfer data via the NoC step by step. We let *Hops* represent the number of steps of the whole NoC transfer process. The latency of the NoC transfer per item is:

$$NoCLat = Hops \times g \times h(g) \times \text{times} \tag{7}$$

The variable *Hops* is determined by the analysis of the multithreaded application, while the speed of the NoC transfer is determined by hardware factors.

Execution time of memory access per data item: Let E be the execution time of memory access for variable a. For each variable, we have three transfer strategies to select from. Let E_{direct} represent the execution time of accessing data from off-chip memory directly. Let E_{DMA} represent the execution time of bringing the data from off-chip memory via DMA and accessing it from the SPM. Let E_{NoC} represent the execution time of obtaining data from other threads via NoC and accessing it from the SPM.

As mentioned in the discussion of the DTA algorithm, we consider data transfer with thread-carried true dependences. E can be computed as:

$$E = \begin{cases} E_{direct} \\ E_{DMA} \\ E_{NoC} \end{cases} = \begin{cases} AccessLat \times 2 \\ AccessLat \times 2 + DMALat \times 2 \\ AccessLat \times 2 + DMALat + NoCLat \end{cases}$$
$$= \begin{cases} s_i \times \beta \times 2 \\ s_i \times \alpha \times 2 + [I + g \times f(g)] \times \lceil \frac{s_i}{g} \rceil \times 2 \\ s_i \times \alpha \times 2 + [I + s_i \times f(s_i)] + Hops \times g \times h(g) \times \lceil \frac{s_i}{g} \rceil \end{cases} \tag{8}$$

For each DTA, each data item needs to be written back to memory and read by another thread. Furthermore, if we transfer the data via NoC between threads, we will need to write it back to the off-chip memory by DMA.

Total execution time of memory access: Since we propose MSDTM to derive the optimal granularity and select the most profitable data transfer strategy among direct access, DMA and NoC for each variable, the execution time due to memory accesses of the slowest thread T is the sum of E. The execution time of the slowest thread due to memory accesses can be computed as:

$$\begin{aligned} T &= \sum E \\ &= \sum E_{direct} + \sum E_{DMA} + \sum E_{NoC} \end{aligned} \tag{9}$$

4.2.2 Deriving the optimal granularity

After the problem formulation, we use the performance model to provide guidelines for deriving the optimal granularity and selecting a profitable transfer strategy. To minimize the total execution time of memory access, we need to minimize the execution time of memory access per data item. This process can be represented as:

$$\text{Minimize} : T \Leftrightarrow \text{Minimize} : E \tag{10}$$

In Equation 8, the parameters α, β, and I are defined by the hardware configuration, while s_i and $Hops$ are computed by an application analysis. These parameters will not change during the data transfer optimizations. Functions f and h are also defined by the hardware configuration. Therefore, during data transfer optimizations, minimizing *times* will lead to minimizing E_{DMA} and E_{NoC}. The derivation process is:

$$\begin{aligned} \text{Minimize} : E &\Leftarrow \text{Minimize} : E_{DMA} \ \& \ \text{Minimize} : E_{NoC} \\ &\Leftarrow \text{Minimize: times} \\ &\Leftrightarrow g = MIN(s_1, s_2, \ldots, s_n) \end{aligned} \tag{11}$$

The optimal granularity g for most heterogeneous many-core architectures is the minimal size of all the variables that need to be transferred in the loop body.

4.2.3 Comparison of strategies

With the optimal granularity g, the MSDTM can provide guidelines for selecting the most profitable strategy at each allocation of data transfer operations. The computation of the execution time of memory access per data item can be replaced with:

$$E = MIN(E_{direct}, E_{DMA}, E_{NoC}) \tag{12}$$

The relationship between E_{direct}, E_{DMA}, E_{NoC} and s_i is plotted in Fig. 9. Their points of intersections s' and s'' split the domain of s_i into three sub-domains. The execution time of memory access per data item E can be computed as:

$$E = \begin{cases} E_{direct}, & s_i \leq s' \\ E_{NoC}, & s' < s_i \leq s'' \\ E_{DMA}, & s_i > s'' \end{cases} \tag{13}$$

When the variable *Hops* which is obtained from application analysis changes, the intersections s' and s'' change as well.

4.3 Code transformation

When the application analysis is completed, the optimal granularity is derived, and the most profitable strategy is selected, the MSDTM transforms the code to optimize the data transfer operations.

First, loop distribution and strip-mining are required to make the size of the loop suitable for the optimal granularity. Loop distribution can be used to convert a sequential loop to multiple parallel loops, while strip-mining is a kind of optimizations to convert the available parallelism into a form more suitable for the hardware

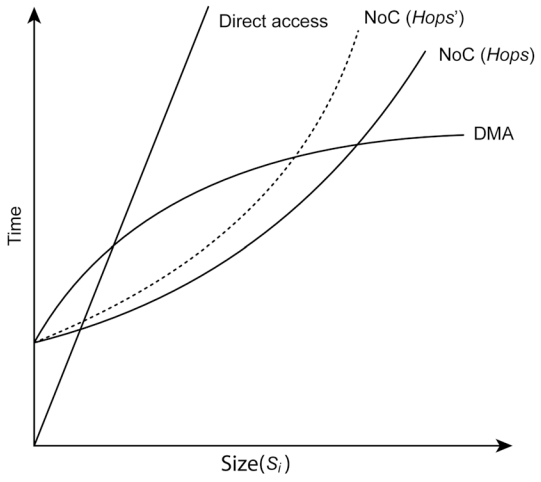

Fig. 9 The dependence of execution time of memory access per data item on the size of data

by grouping the iterations into sets, each of which is treated as a schedule unit [18]. Then, DMA or NoC transfer operations are inserted for each allocation of data transfer according to the DTP model. Finally, we transform the subscripts of variables that need to be optimized to access the SPM.

5 Experimental evaluation

This section presents the experimental evaluation of our proposed MSDTM on Sunway TaihuLight.

5.1 Sunway TaihuLight

In contrast to other existing heterogeneous supercomputers, which include both CPU processors and PCIe-connected many-core accelerators, the computing power of Sunway TaihuLight is provided by heterogeneous many-core SW26010 processors that include both the management processing elements (MPEs) and computing processing elements (CPEs) in one chip. The general architecture of the SW26010 processor [9] is shown in Fig. 10.

The processor includes four core groups (CGs). Each CG includes one MPE, one CPE cluster with 8×8 CPEs, and one memory controller. Each CG has its own memory space, which is connected to the MPE and the CPE cluster through the memory controller. The processor connects to other outside devices through a system interface.

In terms of the memory hierarchy, each MPE has a 32 KB L1 instruction cache and a 32 KB L1 data cache, with a 256 KB L2 cache for both instructions and data. Each CPE has its own 16 KB L1 instruction cache and a 64 KB user-controlled SPM.

As Table 1 shows, while the MPE has access to an 8 GB main memory, the CPE can directly access the main memory through gld/gst instructions. In addition, the CPE can implement batch data transfer between the SPM and main memory

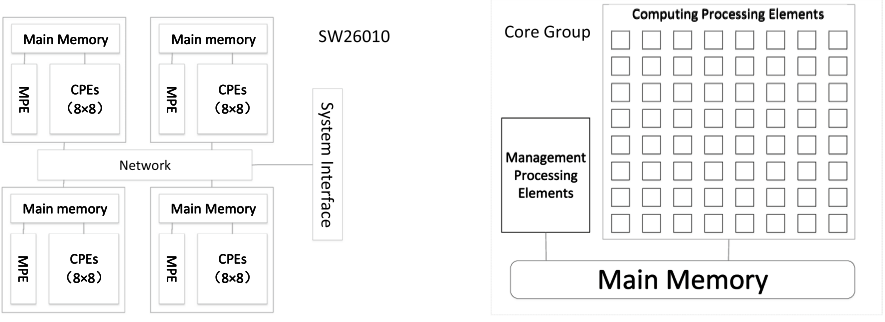

Fig. 10 General architecture of the SW26010 processer

via DMA commands. The efficiency of the DMA transfer is closely related to the amount of data transferred, the granularity of the DMA commands and the continuity of data in the memory. The ideal DMA transfer bandwidth of the processor is 134.4 GB/s. Register communication is used for data transfer as NoC between CPEs. Since the CPEs are physically arranged in an 8×8 array, register communication can transfer data using only XY routing. In XY routing, an access moves along the row-axis first and then along the column-axis. Through register communication, each CPE can perform row or column broadcasting and can send data to another specific CPE.

5.2 Experimental setup

The proposed MSDTM is implemented on the GCC compiler, which is firstly ported for Sunway TaihuLight. In the MSDTM implementation process, we reserve the switches for manual adjustment of the data transfer granularity and manual selection of the data transfer strategies. At the same time, we automatically obtain the optimal granularity and the most profitable strategy by the MSDTM.

To evaluate the performance of the proposed MSDTM, we select test cases from the NAS parallel benchmark suite (NPB) [2] and SPEC benchmarks [16], such as *EP*, *FT*, *IS*, *LU*, *MG*, and *SP* from the NPB and *lbm* [25] from the SPEC. In addition, we choose two representative application kernels, *Stencil* and *PhotoNs*. *Stencil* [1, 21, 27] computations are the foundation of many large applications in scientific computing, while *PhotoNs* is a cosmic N-body numerical simulation software developed by the National Observatory. Before the evaluation, we manually port the benchmarks and kernels for Sunway TaihuLight.

Besides, we select a simple but representative application kernel, 1*D-FFT* [35], to verify that the granularity obtained by the MSDTM is optimal and that the strategy is the most profitable one.

5.3 Experimental results

5.3.1 A case study with FFT

The 1*D-FFT* kernel is implemented based on butterfly computing with an input data size of 8192 bytes. We partition the data into 128 bytes to run the kernel on 64 threads and partition the data into 1024 bytes to run the kernel on 8 threads. In the 1*D-FFT* kernel, the size of the data in each thread limits the granularity of the data transfer. We manually set the granularity of the data transfer to 8, 16, 32, 64 and 128 bytes. In addition, we set the extra granularities to 256 bytes, 512 bytes and 1024 bytes for the 8-threaded version. We compare the execution time of the kernel at each granularity. For each granularity, we use the three transfer strategies mentioned above to optimize the data transfer.

Figure 11 shows the measured values for 8-threaded and 64-threaded kernels. We can observe that the execution time of the kernel decreases as the granularity of data transfer increases with DMA transfer or NoC transfer, while it remains basically

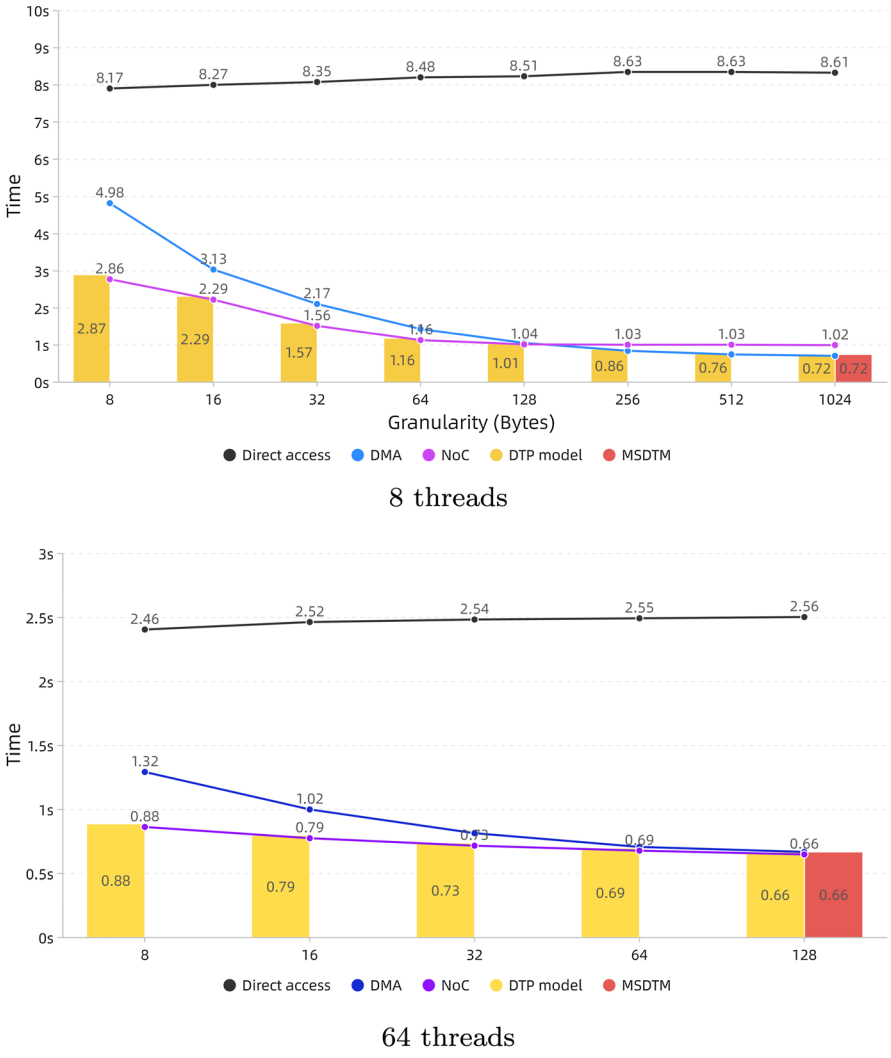

Fig. 11 Execution time of 1*D-FFT* on 8 threads and 64 threads for different strategies

unchanged with direct access. This result means that whether in an 8-threaded application or a 64-threaded application, the optimal execution efficiency of the application is obtained when the granularity of the data transfer is maximal. In other words, the optimal granularity of data transfer is the minimum of the sizes of all data in each thread. In addition, the execution time of the kernel due to the DTP model is equal to the minimum time spent on the three strategies. This observation proves that we can select the most profitable strategy at each granularity based on the DTP model.

Furthermore, the MSDTM selects not only the optimal granularity but also the most profitable data transfer strategy with the optimal granularity. For the 8-threaded

and 64-threaded 1*D-FFT* kernels, optimizing the data transfer with the MSDTM can yield speedups of **11**.**34**× and **3**.**72**× compared with version which use direct access.

5.3.2 Performance and energy evaluation

We evaluate the performance speedup of the proposed MSDTM compared to that of the original applications with direct memory accesses. In addition to the performance evaluation, we evaluate the energy reduction via a script supported by Sunway TaihuLight. The application with data transfer optimization by the MSDTM is executed on 8 threads, 16 threads and 64 threads.

Figure 12 shows the performance improvement and energy reduction of the test cases executed on 8 threads, 16 threads and 64 threads. We can observe that MSDTM performs well with respect to both performance improvement and energy reduction under all scenarios. However, as we can see from Fig. 12, the test cases we use perform the best when executed on 8 threads and the worst when executed on 64 threads. This is due to the DMA transfer bandwidth on Sunway TaihuLight, which results in the roofline curve of the DMA transfer's efficiency with the varying of memory transaction granularity, as shown in Table 2. In other words, the efficiency of DMA transfer is bounded by a threshold of the memory transaction granularity, and the performance of DMA transfer will not be improved when such a threshold is hit. This threshold is 128B when experimenting with 64 threads and the value increases when the number of used threads decreases. One can thus expect better

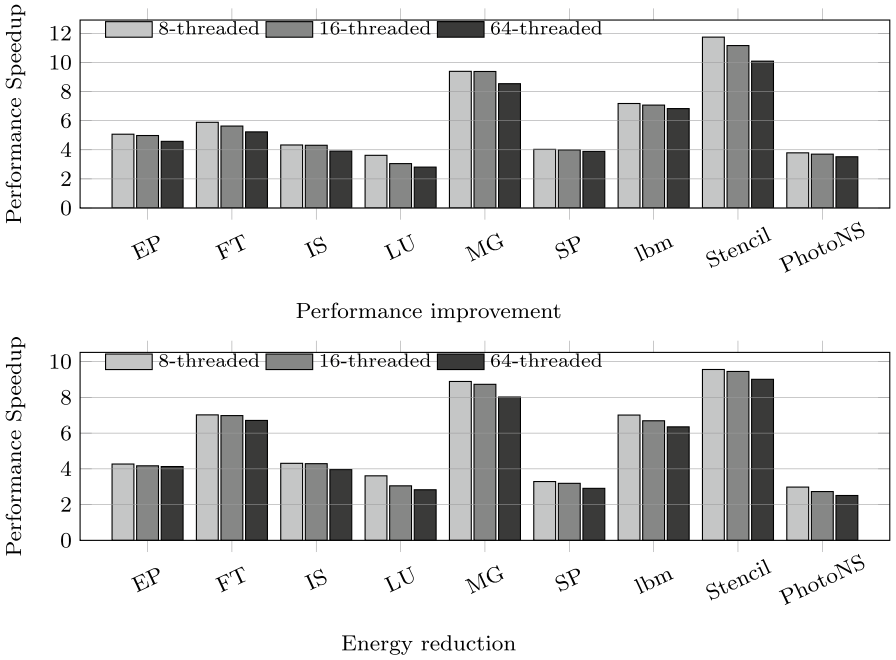

Fig. 12 Performance improvement and energy reduction on different threads

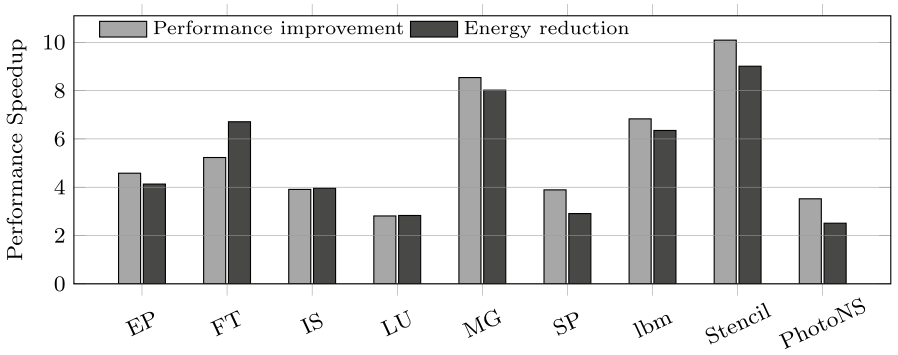

Fig. 13 Maximal performance improvement and energy reduction of the test cases

performance on 8 threads when using the threshold of 64 threads, i.e., 128B. The performance of the used test cases executed on 8 threads thus outperforms those of 16 threads and 64 threads, as shown in the figure.

As Fig. 13 shows, the MSDTM yields considerable acceleration of all the test cases. In particular, the acceleration ratio of *MG* is **8.54**×, while the acceleration ratio of *Stencil* is **10.09**×. The reason that the two test cases get better performance speedup is they have more thread-carried dependences than others, which lead to more DMA or NoC transfers, for example, the overlapping of loop tiling in *Stencil*. In general, the MSDTM provides an average acceleration ratio of **5.49**× on 64 threads and an energy reduction of **5.16**×.

Thus, we observe that the proposed MSDTM is effective in reducing the execution time and energy of the evaluated test cases.

6 Conclusions

In this work, we propose the MSDTM, a compile-time framework for optimizing multithreaded data transfer between SPM and the main memory on heterogeneous many-core architectures. This framework determines the allocation of data transfer operations via an application analysis and dependence checking. Next, the DTP model is used to obtain the optimal granularity of data transfer and select the most profitable strategy. In the experimental evaluation, the proposed MSDTM improves the application execution time by **5.49**× and achieves an energy savings of **5.16**×.

The future works of this paper include further optimizations for SPM data transfer operations, such as overlapping the process of data transfer with kernel computation and combining the granularity of data transfer with the size of loop tiling to achieve higher efficiency.

Acknowledgements This work is supported by National High-tech R&D Program of China (863 Program) (No. 2014AA01A301) and National Key Research and Development Project "High Performance Computing" Key Project (No. 2016YFB0200503).

References

1. Ao Y, Yang C, Wang X, Xue W, Fu H, Liu F, Gan L, Xu P, Ma W (2017) 26 pflops stencil computations for atmospheric modeling on sunway taihulight. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, pp 535–544
2. Bailey D, Barszcz E, Barton J, Browning D, Carter R, Dagum L, Fatoohi R, Frederickson P, Lasinski T, Schreiber R, Simon H, Venkatakrishnan V, Weeratunga S (1991) The NAS parallel benchmarks. Int J Supercomput Appl 5(3):63–73. https://doi.org/10.1177/109434209100500306
3. Banakar R, Steinke S, Lee BS, Balakrishnan M, Marwedel P (2002) Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In: Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627). IEEE, pp 73–78
4. Bandyopadhyay S (2006) Automated memory allocation of actor code and data buffer in heterochronous dataflow models to scratchpad memory. Master's thesis, EECS Department, University of California, Berkeley
5. Borkar S (2007) Thousand core chips: a technology perspective. In: Proceedings of the 44th Annual Design Automation Conference, pp 746–749
6. Chen T, Raghavan R, Dale JN, Iwata E (2007) Cell broadband engine architecture and its first implementation: a performance view. IBM J Res Dev 51(5):559–572
7. Chen T, Sura Z, O'Brien K, O'Brien JK (2006) Optimizing the use of static buffers for DMA on a cell chip. In: International Workshop on Languages and Compilers for Parallel Computing. Springer, pp 314–329
8. Cho D, Pasricha S, Issenin I, Dutt N, Paek Y, Ko S (2008) Compiler driven data layout optimization for regular/irregular array access patterns. In: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp 41–50
9. Dongarra J (2016) Report on the sunway taihulight system. Technical report, UT-EECS-16-742. http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf
10. Feautrier P, Lengauer C (2011) Polyhedron model. Springer, Boston, pp 1581–1592
11. Francesco P, Marchal P, Atienza D, Benini L, Catthoor F, Mendias JM (2004) An integrated hardware/software approach for run-time scratchpad management. In: Proceedings of the 41st Annual Design Automation Conference, pp 238–243
12. Fu H, Liao J, Yang J, Wang L, Song Z, Huang X, Yang C, Xue W, Liu F, Qiao F et al (2016) The sunway taihulight supercomputer: system and applications. Sci China Inf Sci 59(7):072001
13. Gao Y, Zhang P (2016) A survey of homogeneous and heterogeneous system architectures in high performance computing. In: 2016 IEEE International Conference on Smart Cloud (SmartCloud). IEEE, pp 170–175
14. Grosser T, Cohen A, Kelly PH, Ramanujam J, Sadayappan P, Verdoolaege S (2013) Split tiling for gpus: automatic parallelization using trapezoidal tiles. In: Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, pp 24–31
15. Gwennap L (2011) Adapteva: more flops, less watts. Microprocess Rep 6(13):11–02
16. Henning JL (2006) Spec cpu2006 benchmark descriptions. ACM SIGARCH Comput Archit News 34(4):1–17
17. Janapsatya A, Parameswaran S, Ignjatovic A (2004) Hardware/software managed scratchpad memory for embedded system. In: IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004. IEEE, pp 370–377

18. Kelly W, Pugh W (1995) A unifying framework for iteration reordering transformations. In: Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing, vol 1, pp 153–162. https://doi.org/10.1109/ICAPP.1995.472180
19. Kennedy K, Allen JR (2001) Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc, Burlington
20. Li L, Feng H, Xue J (2009) Compiler-directed scratchpad memory management via graph coloring. ACM Trans Archit Code Optim 6(3):1–17
21. Li P, Brunet E, Namyst R (2013) High performance code generation for stencil computation on heterogeneous multi-device architectures. In: 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing. IEEE, pp 1512–1518
22. Lim AW, Liao SW, Lam MS (2001) Blocking and array contraction across arbitrarily nested loops using affine partitioning. In: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and practices of Parallel Programming, pp 103–112
23. Liu T, Lin H, Chen T, O'Brien JK, Shao L (2009) Dbdb: optimizing dma transfer for the cell be architecture. In: Proceedings of the 23rd International Conference on Supercomputing, pp 36–45
24. Marongiu A, Benini L (2010) An openmp compiler for efficient use of distributed scratchpad memory in mpsocs. IEEE Trans Comput 61(2):222–236
25. Pananilath I, Acharya A, Vasista V, Bondhugula U (2015) An optimizing code generator for a class of lattice-Boltzmann computations. ACM Trans Archit Code Optim 12(2):1–23
26. Panda PR, Dutt ND, Nicolau A (2000) On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. ACM Trans Des Autom Electron Syst 5(3):682–704
27. Rahman SMF, Yi Q, Qasem A (2011) Understanding stencil code performance on multicore architectures. In: Proceedings of the 8th ACM International Conference on Computing Frontiers, pp 1–10
28. Ren J, Luo J, Wu K, Zhang M, Li D (2019) Sentinel: Runtime data management on heterogeneous main memorysystems for deep learning
29. Riesbeck CK, Martin C (1986) Direct memory access parsing. Experience, memory and reasoning, pp 209–226
30. Saidi S, Tendulkar P, Lepley T, Maler O (2012) Optimizing explicit data transfers for data parallel applications on the cell architecture. ACM Trans Archit Code Optim 8(4):1–20
31. Sancho JC, Kerbyson DJ (2008) Analysis of double buffering on two different multicore architectures: Quad-core opteron and the cell-be. In: 2008 IEEE International Symposium on Parallel and Distributed Processing. IEEE, pp 1–12
32. Sandrieser M, Benkner S, Pllana S (2011) Explicit platform descriptions for heterogeneous many-core architectures. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. IEEE, pp 1292–1299
33. Shao Z, Li R, Hu D, Liao X, Jin H (2019) Improving performance of graph processing on fpga-dram platform by two-level vertex caching. In: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '19, pp 320–329. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3289602.3293900
34. Shao Z, Liu C, Li R, Liao X, Jin H (2020) Processing grid-format real-world graphs on dram-based fpga accelerators with application-specific caching mechanisms. ACM Trans. Reconfig. Technol. Syst. 13(3):4. https://doi.org/10.1145/3391920
35. Van Loan C (1992) Computational frameworks for the fast Fourier transform, vol 10. Siam, Philadelphia
36. Venkataramani V, Chan MC, Mitra T (2019) Scratchpad-memory management for multi-threaded applications on many-core architectures. ACM Trans Embed Comput Syst 18(1):1–28
37. Verma M, Marwedel P (2006) Overlay techniques for scratchpad memories in low power embedded processors. IEEE Trans Very Large Scale Integr Syst 14(8):802–815
38. Zhang P, Fang J, Yang C, Huang C, Tang T, Wang Z (2020) Optimizing streaming parallelism on heterogeneous many-core architectures. IEEE Trans Parallel Distrib Syst 31:1878–1896