Zheng F, Li HL, Lv H *et al.* Cooperative computing techniques for a deeply fused and heterogeneous many-core processor architecture. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 30(1): 145–162 Jan. 2015. DOI 10.1007/s11390-015-1510-9

Cooperative Computing Techniques for a Deeply Fused and Heterogeneous Many-Core Processor Architecture

Fang Zheng (郑 方), *Member, CCF*, Hong-Liang Li (李宏亮), *Member, CCF*, Hui Lv (吕 晖), *Member, CCF*
Feng Guo (过 锋), *Member, CCF*, Xiao-Hong Xu (许晓红), *Member, CCF*, and
Xiang-Hui Xie (谢向辉), *Senior Member, CCF*

State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214125, China
E-mail: {zheng.fang, li.hongliang, lv.hui, guo.feng, xu.xiaohong, xie.xianghui}@meac-skl.cn

Received November 13, 2013; revised October 7, 2014.

Abstract Due to advances in semiconductor techniques, many-core processors have been widely used in high performance computing. However, many applications still cannot be carried out efficiently due to the memory wall, which has become a bottleneck in many-core processors. In this paper, we present a novel heterogeneous many-core processor architecture named deeply fused many-core (DFMC) for high performance computing systems. DFMC integrates management processing elements (MPEs) and computing processing elements (CPEs), which are heterogeneous processor cores for different application features with a unified ISA (instruction set architecture), a unified execution model, and share-memory that supports cache coherence. The DFMC processor can alleviate the memory wall problem by combining a series of cooperative computing techniques of CPEs, such as multi-pattern data stream transfer, efficient register-level communication mechanism, and fast hardware synchronization technique. These techniques are able to improve on-chip data reuse and optimize memory access performance. This paper illustrates an implementation of a full system prototype based on FPGA with four MPEs and 256 CPEs. Our experimental results show that the effect of the cooperative computing techniques of CPEs is significant, with DGEMM (double-precision matrix multiplication) achieving an efficiency of 94%, FFT (fast Fourier transform) obtaining a performance of 207 GFLOPS and FDTD (finite-difference time-domain) obtaining a performance of 27 GFLOPS.

Keywords heterogeneous many-core processor, data stream transfer, register-level communication mechanism, hardware synchronization technique, processor prototype

1 Introduction

It is well known that the development of the semiconductor industry follows Moore's law. In the last decade, this has mainly been achieved via on-chip many-core architectures. This trend is more obvious in the high performance computing (HPC) domain. Compared with multi-core processors, many-core processors can provide higher computing ability, computing density, and ratio of computation to power consumption. As a result, they are considered as the most important component for future high performance computing systems[1-2].

Many-core processors can be classified as either homogenous or heterogeneous. Heterogeneous many-core processors can integrate different types of cores on a chip. Heterogeneous many-core processors provide a good balance among performance, energy efficiency, and computing density. Thus, they have received significant research interest from both academia and industry[3-6]. We propose a novel heterogeneous deeply fused many-core architecture (DFMC), which integrates management processing elements (MPEs) and computing processing elements (CPEs) on one chip. For the heterogeneous cores, DFMC provides a unified execution model as well as share-memory that supports cache coherence.

Regular Paper

The work was supported by the National High Technology Research and Development 863 Program of China under Grant No. 2014AA01A300 and the National Science and Technology Major Project of HeGaoJi under Grant No. 2013ZX0102-8001-001-001.

Moreover, due to the low ratio of memory bandwidth to computation and the small on-chip memory capacity, many-core processors have to address a more serious memory wall problem, which significantly constrains the processors' performance[1-2,7]. There has been much research on memory optimization technologies for many-core processors through memory access optimization and on-chip data reuse exploration[8-19].

On one hand, it is obvious that memory bandwidth is very important to boost many-core processors' performance. It is necessary to improve bandwidth utilization and hide memory access latency. For instance, the SIMT execution mechanism adopted in NVIDIA's GPGPU implementation[8] and multi-threaded techniques applied on Intel's MIC/Xeon Phi™ processors[10,20] are for those exact purposes. However, those processors are not able to support data distributed on different cores via collective memory access according to application features, such as broadcast, one-dimensional (1D) or two-dimensional (2D) block-cyclic data distribution, and so on. As a result, there is space for improving the memory access performance.

On the other hand, due to the constraints of on-chip resources, cache or on-chip memory dedicated to exploiting memory locality in many-core processors will be limited; thus, on-chip data reuse mechanisms are one of the most important research topics for many-core processor architectures. On-chip data reuse mechanisms can be classified into two types: on-chip share-memory and on-chip communication. For instance, GPGPU's shared memory[21-22] and Intel MIC's L2 Cache[23] are within the on-chip share-memory category, while the Intel SCC processor[13-15] adopts the on-chip communication approach, and the Tile64 processor[12,24] implements both on-chip share-memory and communication via its on-chip mesh interconnect. On-chip data reuse mechanisms have attracted lots of research interest[16-18,25-32].

To optimize memory access performance and improve on-chip data reuse, DFMC adopts a series of tightly coupled cooperative computing techniques to optimize on-chip data distribution, inter-core communication and synchronization.

The contributions of this paper are as follows:

- a series of cooperative computing techniques of CPEs, such as a multi-pattern data stream transfer, an efficient register-level communication mechanism, and a fast hardware synchronization technique, with basic performance evaluations;
- an implementation of full chip RTL and a prototype DFMC system based on FPGA, which includes four MPEs and 256 CPEs;
- a mapping of DGEMM, FFT, FDTD, BFS, and SpMV to the DFMC prototype system, and an analysis of the effect of each cooperative computing technique.

The results show that the cooperative computing techniques of CPEs can effectively improve the computational efficiency of typical applications with regular memory accesses.

The rest of this paper is organized as follows. Section 2 introduces the DFMC architecture in detail, and Section 3 describes the optimization efforts for DFMC's cooperative computing techniques. Section 4 illustrates the prototype implementation and Section 5 presents the experimental performance. Related work is presented in Section 6, and, finally, the paper is concluded in Section 7.

2 DFMC Architecture

2.1 Overview

Fig.1 shows the DFMC architecture. DFMC consists of management processing elements (MPEs), computing processing elements (CPEs) clusters, memory controllers (MCs), and the system interface (SI). All of these modules are interconnected via network on chip (NoC). DFMC is connected to an off-chip system via the system interface.

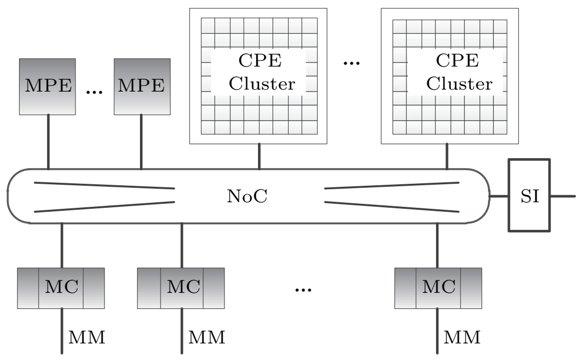


Fig.1. Architecture of DFMC.

The DFMC architecture is highly scalable. The number of MPEs, CPE clusters, and MCs can be adjusted in a flexible manner according to the practical implementation conditions. The number of CPEs in one CPE cluster is also flexible. The crossbar, mesh, or ring structures can be selected as the NoC topology according to the number of components connected via NoC.

An MPE is a fully functional core, which integrates two levels of cache structures and uses SIMD vector extension. MPEs can run in both the user mode and the system mode, and support superscalar, out-of-order issue, out-of-order execution, and speculative execution. Applications' serial sections can be efficiently performed on an MPE, and users can obtain standard services and operating environments. The proportions of MPEs' power and area are low in DFMC. Therefore, the goal of MPEs' design is to improve performance and general-purpose computing capability.

We consider some important design parameters of CPEs, such as issue width from one to three, out-of-order or in-order and branch prediction strategy. According to energy and area efficiencies, CPEs support dual-issue, out-of-order execution, and static branch prediction. They only run in the user mode, and support double-precision floating-point multiply-add and divide/square-root operations. They can largely simplify micro-architecture design under the premise of ensured computational efficiency.

DFMC's architecture has good adaptability for high performance computing. A large number of CPEs can efficiently process thread-level parallelism. And a small number of powerful MPEs can effectively explore instruction-level parallelism and process the serial sections of applications.

The key component of DFMC is the CPE cluster, whose structure is shown in Fig.2. Each CPE cluster consists of multiple CPEs, a CPE cluster network (CPE_NET) and a CPE controller (CPE_Ctrl). The CPE cluster has a series of optimization techniques to combat the memory wall problem.

CPE_NET uses an $N \times N$ mesh topology, credit-based flow control, and wormhole routing. Its design focuses on achieving low latency and lightweight communication to meet the demand of cooperative computing among CPEs. CPE_NET supports an efficient register-level inter-CPE communication, which will be described later in Subsection 3.2.

The main components of CPE_Ctrl include:

• CPE_NET NI: the CPE_NET network interface (NI) is used to process CPE_NET's requests.

• Stream engine: the stream engine is used to dispatch and manage the data stream transfer operations from the CPEs, which will be described in Subsection 3.1.

• Int_Ctrl: the interrupt controller is used to control the interrupt sent from the CPE cluster to the external system.

• Syn_Ctrl: the synchronization controller is used to handle inter-CPE fast hardware synchronization, which will be described in Subsection 3.3.

• CTLB: the CPE translation lookaside buffer is used for the translation and protection of CPE virtual addresses.

• Coherence process unit: the coherence process unit is used to process the coherence protocol between CPEs and MPEs.

• NoC NI: the NoC network interface in CPE_Ctrl.

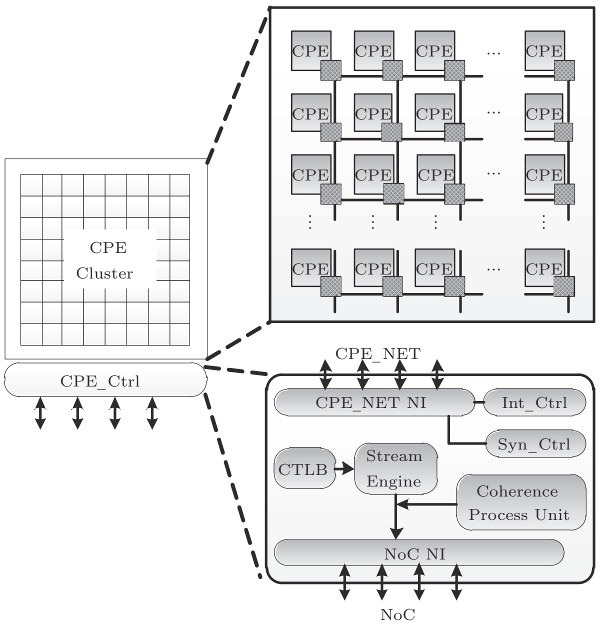


Fig.2. CPE cluster of the DFMC processor.

2.2 Deep Fusion of Heterogeneous Cores

A major feature of the DFMC architecture is the deep and seamless fusion of heterogeneous cores by applying the unified ISA (instruction set architecture) and the execution model, as well as the share-memory model, which supports cache coherence. It provides programmers with a consistent programming environment of MPEs and CPEs and minimizes main memory copy between heterogeneous cores.

• *ISA*. The basic ISAs of MPEs and CPEs are compatible; both are Alpha-like general RISC architectures with 32-bit fixed-length. There are 212 instructions in the basic ISA, which supports operand sizes of 8-bit, 16-bit, 32-bit, 64-bit, and IEEE 754 floating point in 32-bit (single precision) and 64-bit (double precision). The operations include memory access, arithmetic logic, control and floating point. In addition, CPEs add register-level communication, data stream transfer and hardware synchronization instructions, which are related to the structure of the CPE cluster.

• *Execution Model.* MPEs run standard Linux, with kernel version 2.6.28. CPEs run a customized lightweight operating system that just supports basic functions, such as processing/thread management and memory management. Both MPEs and CPEs can run independently. There are two hybrid execution models: the accelerating model and the service model, facilitating the cooperative performance of MPEs and CPEs. Under the accelerating model, the main program runs on an MPE, and the MPE offloads the computational kernel to CPEs; under the service model, the main programs run on CPEs, and the CPEs call for the MPEs to finish the file and provide MPI services.

• *Memory Space.* DFMC supports the sharing of main memory between MPEs and CPEs. Cache coherence is supported among MPEs. When accessing shared main memory (by load/store instructions or data stream transfer), CPEs can obtain the latest copy from the MPEs' cache. Thus, MPEs offloading computational kernels to CPEs are lightweight and effective because there is no need for the global memory copy. SPMs in the CPE cluster have their own address space with explicit data transfer instructions to move data to and from the main memory. There is no coherence between SPMs. The memory hierarchy design of DFMC is shown in Fig.3.

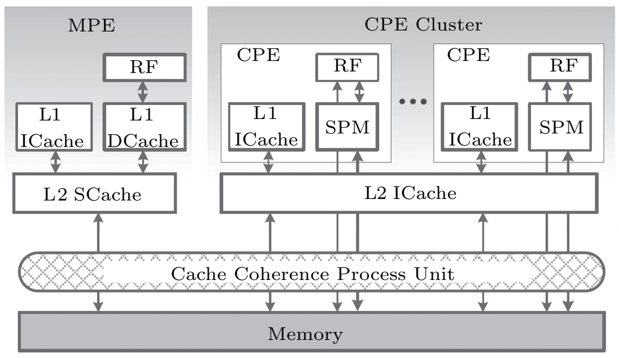


Fig.3. Memory hierarchy of the DFMC processor.

MPEs have a two-level on-chip memory hierarchy, including separated data and instruction L1 caches, and a shared data and instruction L2 cache. CPEs' instruction memory system provides a private L1 instruction cache for each CPE and a shared L2 instruction cache for the CPE cluster. The data memory system of CPEs is based on the organization of scratch pad memory (SPM), which is directly open to management via software. Compared with the cache, this approach simplifies the implementation, achieves the accurate use of space, reduces off-target traffic, and avoids the design complexity and performance degradation that would be caused by dealing with coherence among many CPEs. CPEs can access memory through load/store instructions, which support register, immediate, and displacement addressing modes and 1B/2B/4B/8B accessing granularity. CPEs also support multi-pattern data stream transfer between the SPM and the main memory to take full advantage of memory bandwidth. The CPE cluster translation lookaside buffer (CTLB) translates the virtual addressing to physical addressing by segmentation without paging. The virtual address is divided into a limited number of segments. CTLB keeps all of the segment information, such as physical base address, segment length, and access permission. The maximum number of total segments in the CPE cluster is equal to the number of CTLB entries. This mechanism can avoid the performance degradation of CTLB misses.

3 Cooperative Computing Techniques

There are three optimization techniques to support the cooperative computing of CPEs and alleviate the memory wall in DFMC: data stream transfer, register-level communication, and hardware synchronization. These techniques can adapt to many key applications of HPC.

3.1 Data Stream Transfer

Depending on the application features, the continuous or stride data can be moved between the SPM and the main memory with the multi-pattern data stream transfer in DFMC, which can utilize the SPM and the memory more accurately and efficiently. The data stream transfer, which is started by CPEs and independent of CPE pipeline execution, can pre-fetch data and effectively hide the memory access latency by double-buffering or multi-buffering. The stream engine in CPE_Ctrl can process several data stream transfers concurrently. To improve memory access performance and ensure QoS and fairness of the CPEs, there are three levels of scheduling in the engine: stream level, request flit level, and response flit level.

The instructions of CPEs that configure and start a data stream transfer are shown in Table 1. To improve efficiency, the stream engine performs transfers out-of-order. If a specified transfer group must be in order, a stream barrier can be used. The *stream_put/stream_get* instructions after a *stream_barrier* will not be executed until the *stream_put/stream_get* instructions before the *stream_barrier* are finished. When a data stream transfer is completed, the response generated by the stream engine can be written to the SPM. The CPE can just start the data stream transfer between the main memory and the SPM in its own cluster. The communication across the CPE cluster is routed through shared main memory. If there is more than one task running in a CPE cluster simultaneously, the corresponding relation between CPEs and tasks is recorded in the stream engine. An exception will be generated when CPEs that are assigned to different tasks take part in the same data stream.

Table 1. Instructions for Data Stream Transfer

Instruction	Description
stream_put ***ra***, ***rb***, ***rc***	Configure and start the data stream transfer from the SPM to the main memory
stream_get ***ra***, ***rb***, ***rc***	Configure and start the data stream transfer from the main memory to the SPM
stream_barrier	Data stream transfer barrier
stream_mask ***ra***	Set the masks of the stream transfer. ***ra*** is a 64-bit vector and each bit represents a CPE. A "1" bit represents that the corresponding CPE will be masked.

The *stream_put* and the *stream_get* instructions have three 64-bit source operand registers, which form a 192-bit command to describe the data stream transfer, such as SPM address, main memory address, transfer pattern, stride, transfer length, and so on.

For the cooperative computing of CPEs, there are multiple patterns of data streaming transfers. These transfer patterns can make the data arrangement in the CPE cluster multi-dimensional, which can effectively improve the data locality and save the main memory access bandwidth. A CPE cluster in DFMC supports seven patterns: the single CPE pattern, the broadcast pattern, the row pattern, the broadcast row pattern, the column pattern, the broadcast column pattern, and the array pattern. The broadcast pattern broadcasts data to all of the CPEs in a CPE cluster while accessing the main memory; the row pattern and the broadcast row pattern can circularly arrange the data blocks via row dimension; the column pattern and the broadcast column pattern can circularly arrange the data blocks via the column dimension; and the array pattern can circularly arrange the data blocks in a 2-dimensional (2D) array. The patterns are shown in Fig.4; CPE(R, C) indicates its location in the CPE cluster is row R and column C. In this paper, the patterns of data stream transfers are fixed corresponding to the CPE cluster's 8×8 structure.

3.2 Register-Level Communication Mechanism

We propose a data reuse mechanism based on register-level communication, implementing fine-grained low-latency data movement between CPEs at low hardware cost, and supporting multicast and broadcast communication functions. The user interface of register-level communication consists of MPI-like communication primitives, such as *send/recv*, *isend/irecv*, and *bcast*. The communication library of primitives is written by the embedded assembler, with customized instructions for register-level communication. Moreover, register-level communication does not ensure data coherence via hardware, according to the massage passing model.

Different from the traditional communication mechanisms for network-on-chip, register-level communication mechanisms provide direct data transfer between the general purpose register files of CPEs without passing through the local on-chip memory of CPEs, as shown in Fig.5. CPEs' pipelines and CPE_NET are tightly coupled in DFMC, and thus the data are directly injected into the CPE_NET from the source CPE's pipeline, and then carried to the destination CPE's pipeline. The topology of CPE_NET is a mesh that only needs adjacent communication and short inter-core wires, and thus we can fix the timing easily. The register-level communication function uses the producer-consumer protocol and implements lightweight blocking/non-blocking communication. As a sender, a CPE puts the data into a sending unit from a general-purpose register file, and then the pipeline will continue to run; the receiver CPE gets the valid data from a receiving buffer and takes them into its general-purpose register file. The sending/receiving hardware logic used for the register-level communication mechanism is simple and without virtual channels, which can reduce the area, power cost, and design complexity

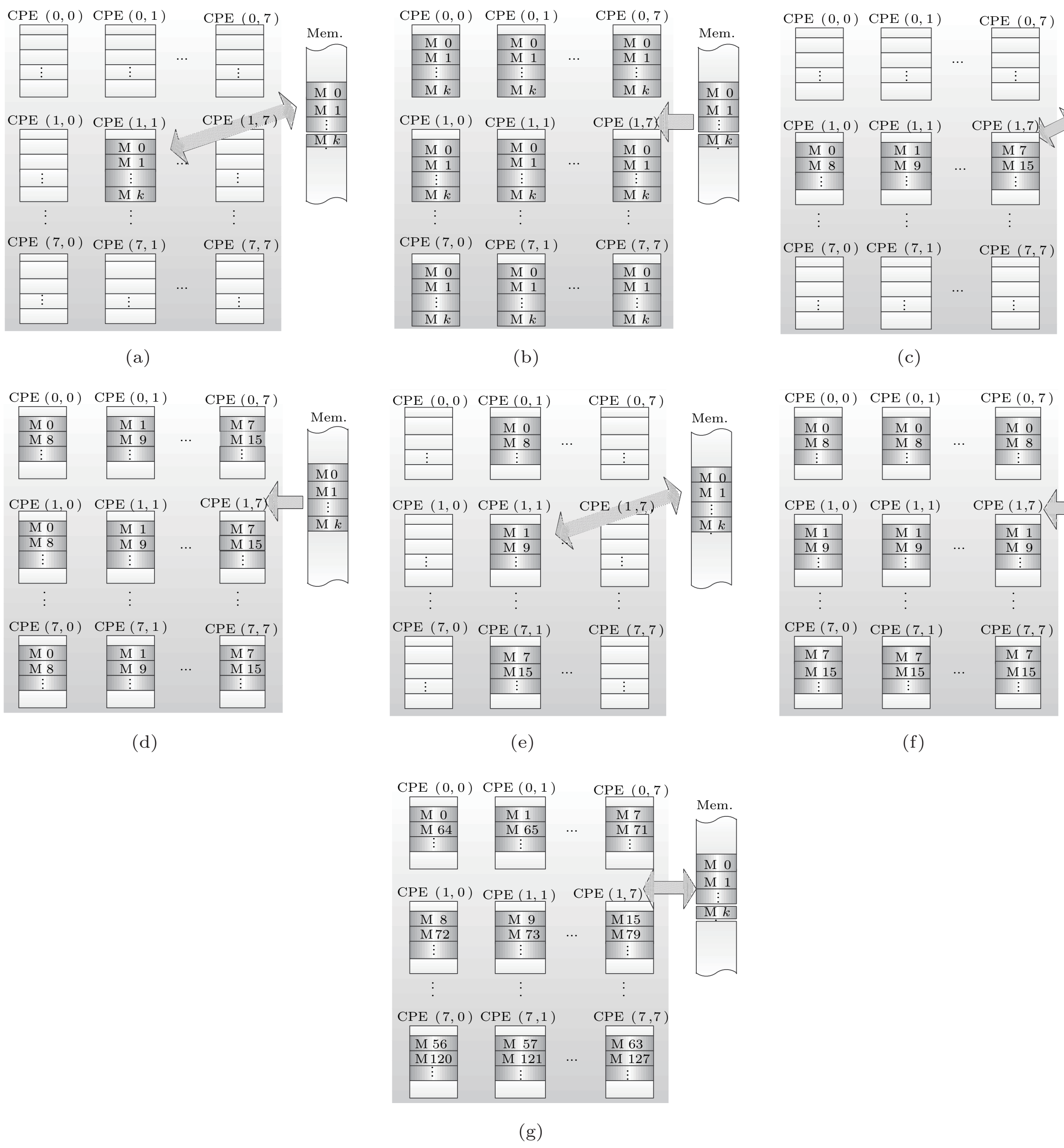


Fig.4. Patterns of data stream transfers. (a) Single CPE pattern. (b) Broadcast pattern. (c) Row pattern. (d) Broadcast row pattern. (e) Column pattern. (f) Broadcast column pattern. (g) Array pattern.

of both CPEs and CPE_NET. The producer-consumer protocol also avoids handshake or synchronization operations to establish a communication session, which can significantly shorten the communication delay.

The CPE_NET consists of 64 reduced routers to support register-level communication. The reduced router is similar to the classic on-chip network router, but parallel multicast and broadcast functions are added in order to improve the efficiency of collective communication.

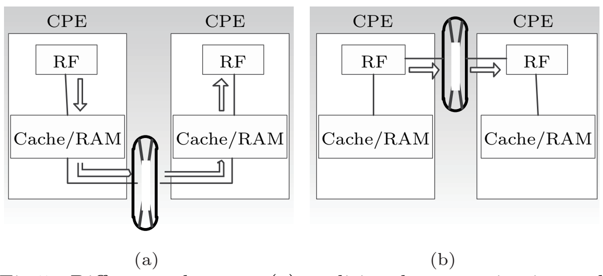


Fig.5. Differences between (a) traditional communication and (b) register-level communication.

There are four instructions used to control register-level communication; two of the instructions are used for sending data and the others are used for receiving data, as shown in Table 2. In this paper, the *send*/*bcast* instruction and the *receive* instruction must be used as a pair, or the next register-level communication will be blocked.

Table 2. Instructions for Register-Level Communication

Instruction	Description
send ***ra***, ***rb***/#***dest***	Send the data in ***ra*** to the CPE designated by ***rb*** or immediate ***dest***. ***rb*** is a 64-bit vector, and each bit represents a CPE. A "1" bit represents that the date will be sent to the corresponding CPE in the CPE cluster
bcast ***ra***, ***rb***	Broadcast the data in ***ra*** to the CPE designated by ***rb***
receive ***ra***, ***rb***	Receive the communication data blocked, and take the data into ***ra***. Then, put the ID of the sender into ***rb***
receive_test ***ra***, ***rb***	Receive the communication data non-blocked, if the data has not arrived. Sign a flag into ***rb***

3.3 Hardware Synchronization Technique

CPEs often work in a tightly coupled cooperative computing model, which leads to frequent synchronization and lock operations. Traditionally, the synchronization is implemented by software through atomic memory operations. The software synchronization issues many memory accesses, resulting in system performance degradation. To improve the synchronization performance, this paper implements an on-chip fast hardware synchronization technique without memory accesses, which can perform the most frequently used barrier operation in a CPE cluster. Other types of synchronization in a CPE cluster are performed by software. The comparison between the traditional software synchronization and the hardware synchronization is shown in Fig.6. When the synchronization between MPEs and CPEs or the synchronization across a CPE cluster is needed, we can still use the soft synchronization operations.

① and ④ are CPE execution phases before synchronization, ② and ⑤ are CPE synchronization phases, and ③ and ⑥ are CPE execution phases after synchronization. In phase ②, all CPEs that have reached the synchronization point always access memory until the last CPE arrives, wasting memory bandwidth. In phase ⑤, CPEs use a synchronization instruction to notify Syn_Ctrl in CPE_Ctrl. Syn_Ctrl sends finish signals to the CPEs after all of the CPEs that need synchronization have reached the synchronization point.

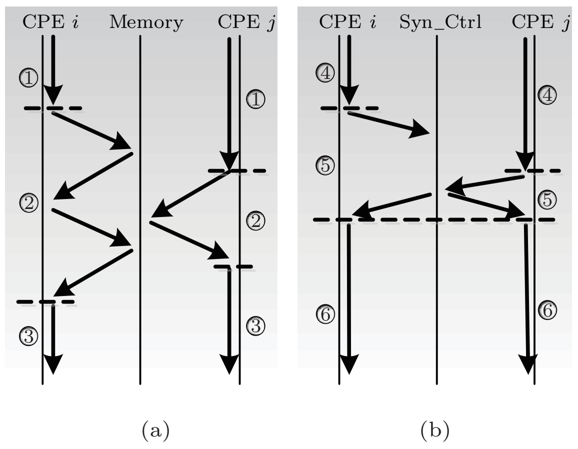


Fig.6. Comparison between (a) software synchronization and (b) hardware synchronization.

Syn_Ctrl completes synchronization by collecting hand-up state and then setting a completed flag. It can also process several groups of synchronization concurrently. Synchronization instructions in DFMC include a row synchronization instruction, a column synchronization instruction, and an array synchronization instruction. The row/column synchronization instructions have an 8-bit synchronization vector, and the array synchronization instruction has a 64-bit synchronization vector. Each bit of the vectors represents whether the corresponding CPEs need to synchronize with the current CPEs. Synchronization instructions are described in Table 3.

Table 3. Instructions for Hardware Synchronization

Instruction	Description
synr ***ra***/#***dest***	Row synchronization instruction. Register ***ra*** or immediate ***dest*** is an 8-bit vector, and a "1" bit represents that the corresponding CPE in current row participates in the synchronization
sync ***ra***/#***dest***	Column synchronization instruction. Register ***ra*** or immediate ***dest*** is an 8-bit vector, and a "1" bit represents that the corresponding CPE in current column participates in the synchronization
syn ***ra***	Array synchronization instruction. Register ***ra*** is a 64-bit vector, and a "1" bit represents that the corresponding computing-core in current CPE cluster participates in the synchronization

4 Implementation and Performance Evaluation

To validate DFMC, we implemented a full chip RTL design and built a prototype system with FPGA. The performance of cooperative computing techniques in the prototype system was evaluated. Furthermore, several typical applications were mapped to the DFMC architecture for performance analysis.

4.1 Full Chip RTL

The RTL of DFMC is designed in-house; thus we can easily optimize the microarchitecture, extend the functionality, and balance the performance and the power-usage. Clock gate and fault tolerance technology are also used in this design. For the future test chip, we finished the physical design intended for fabrication in 40 nm technology.

The parameters of DFMC are compared with those of an Intel Xeon CPU and an NVIDIA GPU as shown in Table 4. These processors are different in architectures, but under the similar CMOS technology process. Because of the balance design of power and performance in CPEs, DFMC achieves the best peak performance and the ratio of computation to power consumption. However, the ratio of memory bandwidth to computation of DFMC is the worst. In this paper, DFMC combines a series of cooperative computing techniques to solve this problem.

4.2 Prototype System

The applications and tests run slowly in a software environment, thereby we implemented a full chip prototype system with FPGA for acceleration.

The FPGA prototype system adopts a modular structure, which consists of MPE cards, CPE cards, a PCIe card, an MC card, an NoC card, and so on. The prototype includes 256 CPEs, four MPEs and four MCs, as shown in Fig.7.

The FPGA prototype system uses a total of 352 Altera EP3C120, 21 Xilinx 5VLX330 and one Xilinx 5VLXT220. The frequency of the prototype system is 2.6 MHz. Table 5 lists the components and functions.

Although there are many cross-board signals, we balance all of the stages related to cross-board and ensure the FPGA prototype system is equal to the RTL design at the cycle level. Then, the foremost reason that the simulation is inaccurate is the main memory frequency. Compared with the target RTL design, the ratio of CPE frequency to MC frequency in the prototype is quite different, which results in simulation deviation. To ensure accuracy, the prototype system uses the performance calibration techniques. FPGA prototypes have many performance adjusters and counters, and we have an FPGA adjustment benchmark that includes more than one hundred short programs especially for memory systems. We define the deviation ratio as the ratio of a program's execution time on RTL to its execution time on FPGA. Then, we can adjust the latency, bandwidth, and scheduling in the FPGA prototype to find the minimum average deviation ratio for the benchmark. The performance counters can indicate which adjustment is more important. The test shows that the performance accuracy of the prototype system is up to 95% in the benchmark thanks to the calibration.

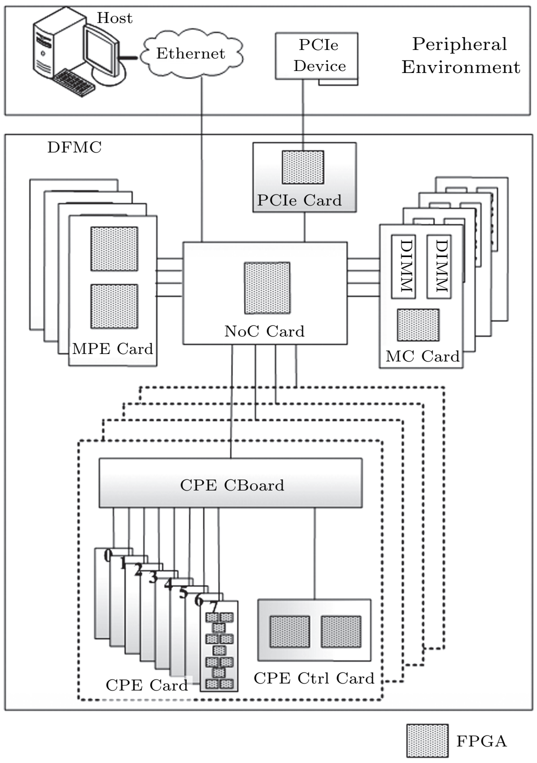
4.3 Software Layer

In this paper, the programs running on DFMC use the accelerated model. We designed a library-based programming approach to ease the task of utilizing DFMC. The library supports programming interfaces for thread management, data stream transfer, register level communication and synchronization. Programmers can use these interfaces to explicitly control the

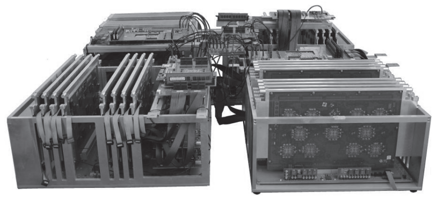
Table 4. Parameters of DFMC/Xeon/GPGPU

	DFMC (40 nm)	Intel® Xeon® 5680 (32 nm)	NVIDIA Fermi M2090 (40 nm)
Architecture	4 CPE clusters (256 CPEs) 4 MPEs, 4 MCs	6 cores	512 CUDA cores
NoC	Mesh	Ring topology	–
On-chip memory	32 KB in each CPE×256=8 MB	12 MB cache	1 024 KB share memory/L1 cache 768 KB L2 cache
Frequency	1 GHz	3.33 GHz	1.3 GHz
Computing ability	1000 GFLOPS DP	80 GFLOPS DP	665.6 GFLOPS DP
Memory bandwidth	102.4 GB/s DDR3	32 GB/s DDR3	177.6 GB/s GDDR5
Chip area	$\sim$ 400 mm^2@40 nm	240 mm^2 @32 nm	520 mm^2@40 nm
Power	$\sim$ 200 W	130 W	250 W

hardware and implement their applications on DFMC, and then the compiler translates source code to assembly code. It is not the intention of this paper to describe the way to design software layer in detail.



(a)



(b)

Fig.7. DFMC FPGA prototype system. (a) Logic diagram. (b) Physical photo.

Table 5. Component of FPGA Prototype System

Card/Board	FPGA	Function
CPE card	11×32 cards (EP3C120)	Eight CPEs in one card
MPE card	2×4 cards (5VLX330)	One MPE in one card
CPE_Ctrl card	2×4 cards (5VLX330)	One CPE_Ctrl in one card
MC card	1×4 cards (5VLX330)	One MC in one MC card
NoC card	1 (5VLX330)	DFMC NoC
PCIe card	1 (5VLXT220)	PCIe interface
CPE CBoard	–	Assemble CPE card and CPE_Ctrl card

A programming example on DFMC is shown in Fig.8. That is a kernel of matrix multiplication. The original program in Fig.8 is split into an MPE program and a CPE program. The MPE program, as the host program, invokes the CPE program to accelerate the kernel. In the CPE program, several steps are performed: arrays *aa*, *bb* and *cc* are first declared in SPM using the _SPM_ variable declaration; data is then fetched from the main memory to SPM through explicit data transfer operations; after hardware synchronization, the matrix multiplication is computed on all CPEs in parallel; the calculation is divided into eight rounds, and different CPEs do register-level broadcast in different rounds (e.g., row and column register-level broadcasts are performed by diagonal

```
//-----------------Main function in MPE-----------------------
Main(){
…
/* CPE_VEC is a 64 -bit vector corresponding to 64 CPEs in
#CCN CPE_Cluster */
CPE_Foo(CPE_VEC, CCN);
…
}
//------------------- Function in CPEs---------------------------
CPE_Foo{
__SPM__ double aa[N][N], bb[N][N], cc[N][N];
…
//data stream transfer:
//get data from source(main memory) to dest(SPM)
dt_get(source_addr_a, aa, size_a, pattern_a, tag_a);
dt_get(source_addr_b, bb, size_b, pattern_b, tag_b);
dt_get(source_addr_c, cc, size_c, pattern_c, tag_c);
// hardware synchronization: all CPEs in CPE cluster
syn_all();
//-------------Round 0------------------//
If(CPE_RowNum == CPE ColNum) // The diagonal CPEs
{
   for (ii=0; ii< N; ii++)
     for (kk=0; kk< N; kk++)
        for (jj=0; jj< N; jj++)
        {
          //register-level comm:
          //broadcast aa to CPE in the same row
          rlc_bcast (aa [ii][kk], row, size);
          // broadcast bb to CPE in same col
          rlc_bcast (bb[kk][jj], col, size);
          cc[ii][jj]=cc[ii][jj]−aa[ii][kk]*bb[kk][jj];}
else
{
   for (ii=0; ii<N; ii++)
     for (kk=0; kk<N; kk++)
        for (jj=0; jj<N; jj++)
        {
          //register-level comm:
          //receive data and save to aa_temp/bb_temp
          aa_temp=rlc_receive (row, size);
          bb_temp=rlc_receive (col, size);
          cc[ii][jj]=cc[ii][jj] −aa_temp*bb_temp;}
//-------------Rounds 1~7------------------//
…

syn_all();
//data stream transfer:
//put data from source(SPM) to dest(main memory)
dt_put(cc, dest_addr_c, size_c, pattern_c, tag _c);
}
```

Fig.8. Example of programming on DFMC.

CPEs in round 0); the results are finally put back to the main memory after computing. We use the library to abstract the details of new hardware instructions, such as *dt_get*/*dt_put*(data stream transfer), *rlc_bcast*/*rlc_receive*(register level communication), and *syn_all*(hardware synchronization). In addition, the register allocation and the spilling of register level communication are preceded by the compiler.

There is much manual work involved in mapping the algorithms to the architecture. However, openacc2.0 and openmp4.0 standards will be supported on DFMC in the future, which can alleviate the burden on the programmers.

5 Performance Analysis

5.1 Performance of Cooperative Computing Techniques

5.1.1 *Data Stream Transfer*

Fig.9 shows the full-chip aggregate bandwidth of various data stream patterns. The horizontal axis represents the data size that each CPE obtains (byte); the vertical axis represents the aggregated data stream bandwidth (GB/s) of full chip with 256 CPEs. For the single CPE pattern (S), the row/column pattern (R/C), and the array pattern (A), we used four cases: *stream_get*, *stream_put*, *stream_get* with stride and *stream_put* with stride. For broadcast row/column pattern (BR/BC) and broadcast pattern (B), we used two cases: *stream_get* and *stream_get* with stride. Due to the limit of space in this paper, we just show the results of *stream_get* in all patterns in Fig.9.

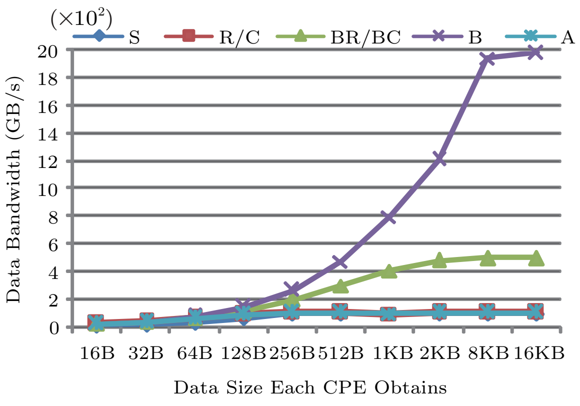


Fig.9. Bandwidth of data stream transfer.

The results show that in the case of the single CPE pattern, row/column patterns and array pattern, more than 90% of the peak bandwidth (physical link width × data rate) can be reached when the transfer length exceeds 256B. However, the bandwidth of the row/column pattern and array pattern is better than that of the single CPE pattern when the data size is less than 256B. In the case of the broadcast row/column pattern and broadcast pattern, the CPEs obtained bandwidth scales out of the main memory physical bandwidth because of the broadcast technique. Their maximum aggregate bandwidths are 593.6 GB/s and 2 165.8 GB/s, respectively.

5.1.2 *Register-Level Communication Mechanism*

We achieved a lightweight MPI-like communication primitive between CPEs by the register-level communication mechanism, and chose four different communication modes from typical algorithms, as shown in Fig.10. Mode *A* is from the butterfly transform of FFT, and the communication distance between CPEs is 2^n (n is 2 in our experiment). Mode *B* is from the famous Red-Black Colouring. Adjacent CPEs update alternately, and the communication mode is wavefront. Mode *C* is from the DGEMM algorithm based on row/column broadcast. The diagonal CPEs perform row and column broadcast. Other CPEs receive data. Mode *D* is a full broadcast from one CPE to all the other CPEs. CPE 0 is set to be the broadcasting CPE in this experiment.

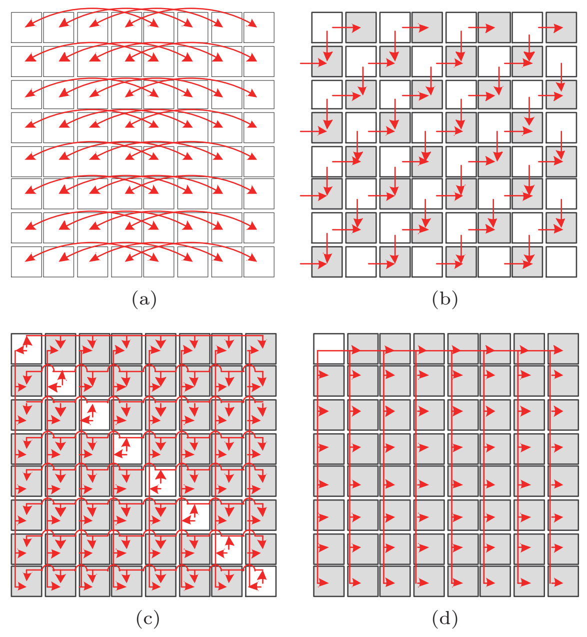


Fig.10. Register-level communication modes in test. (a) Mode *A*. (b) Mode *B*. (c) Mode *C*. (d) Mode *D*.

The aggregate bandwidths of four communication modes with different data sizes are shown in Fig.11. The horizontal axis represents the communication data sizes (byte) and the vertical axis represents the aggregate communication bandwidth of one CPE cluster (GB/s).

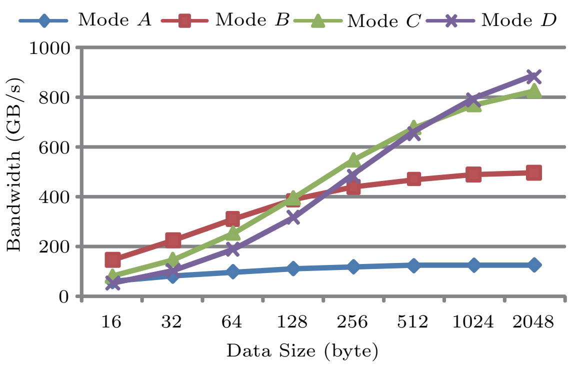


Fig.11. Bandwidth of the register-level communication test in one CPE cluster.

Because all of the communications of CPEs go through the middle of the network, there is a communication hotspot in the CPE cluster in mode A. The maximum aggregated bandwidth reaches only 126.9 GB/s. Mode B communications are approaching traffic with the shortest latency. The aggregated bandwidth is maximized when the data size is less than 128 B. The aggregated bandwidth reaches 502.2 GB/s when the data size is increased to 2 KB. For mode C, the maximum latency is that across a CPE cluster side and the aggregated bandwidth is up to 831.1 GB/s. For mode D, the maximum latency is that from diagonal CPEs and the aggregated bandwidth is up to 889.8 GB/s.

Fig.12 provides the maximum communication latency of the four communication modes with various data sizes. The horizontal axis represents the amount of communication data (byte); the vertical axis represents the maximum latency (cycle). This library is written by the embedded assembler and we use the inline optimization, and thus the latency shown in Fig.12 is in hardware.

In Fig.12, the communication latency of mode A is the maximum because there is a hotspot in the CPE cluster. For the other three cases, mode B, mode C and mode D, the maximum latency is less than 300 cycles. The register-level communication mechanism is simple and the latency is small, and thus the latencies of modes $A{\sim}D$ are only 17 cycles, 7 cycles, 22 cycles and 36 cycles, respectively, with the data size of 16 B. As a result, the register-level communication mechanism is also adapted to the small-data communication with small latency.

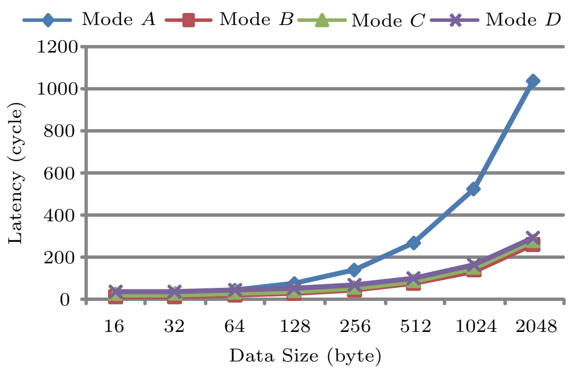


Fig.12. Latency of register-level communication test in one CPE cluster.

5.1.3 *Hardware Synchronization Technique*

Fig.13 gives a comparison between the hardware synchronization technique (Hard_Syn) and the traditional software synchronization technique with atomic operation (Spft_Syn). The horizontal axis represents the number of CPEs which participate in synchronization, from 2×2 to 8×8. The vertical axis represents the time required by synchronization. The hardware synchronization technique is implemented by the *Syn* instruction of DFMC, while the software synchronization technique is implemented with an atomic *fetch&add* operation.

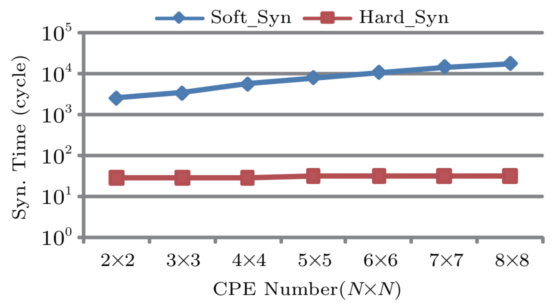


Fig.13. Performance of hardware synchronization technique.

The results show that the hardware synchronization needs only 29 to 33 cycles to finish synchronization, while the software synchronization needs 2 558 to 18 856 cycles. Thus, the hardware synchronization technique can achieve an 88X to 500X speedup, which improves the synchronization efficiency dramatically. In addition, the scalability of the hardware synchronization is also better, as the latency is independent on the number of participating CPEs. However, for the software synchronization, the synchronization cost of 8×8 is up to 7.4 times over 2×2.

5.2 Performance of Typical Applications Mapping

5.2.1 Typical Applications

To validate the design of DFMC, we chose five typical applications from 13 Dwarfs of Berkeley to map to the prototype[33]: dense linear algebra (DGEMM, double-precision matrix multiplication), spectral method (FFT, fast Fourier transform), structured grids (FDTD, finite-difference time-domain), sparse linear algebra (SpMV, sparse matrix vector product) and graph traversal (BFS, breadth-first search). DGEMM, FFT and FDTD are regular memory access applications, and SpMV and BFS are irregular memory access applications.

• *DGEMM.* It is the kernel of many linear algebra algorithms, which have important application prospects in the fields of science and engineering computing. DGEMM has good separability and regular memory access. Its input matrices are $N \times N$. Its computational complexity is $O(N^3)$, the amount of data access is $O(N^2)$, and the ratio of computation to data accesses is $O(N)$. On the general-purpose processor, DGEMM is computing bound. However, in the many-core processor, it will be bandwidth bound if the on-chip data reuse and the memory accessing do not work well.

• *FFT.* It is one of the most important kernels for signal processing. It is used to convert signals from time domain to frequency domain. The computation complexity is $O(N\log N)$ and the space complexity is $O(N)$. FFT finishes the transmission within $\log N$ stages of the butterfly computation. During the FFT process, the stride size of data access is the power of 2. The bound is depending on the FFT size. If the work set cannot be stored on chip, memory access impacts on the performance significantly.

• *FDTD.* It is an important numerical analysis technique for modelling computational electrodynamics. That belongs to the grid-based modelling methods. The equations are solved in a leapfrog manner: the electric field vector components are solved at a given instant in time; then the magnetic field vector components are solved at the next instant in time; and the process is repeated over and over again until electromagnetic field behaviour is fully evolved.

• *BFS.* It is one of the most important graph algorithms, and it serves as a building block for many other algorithms. Benchmark suites targeting graph applications perennially include BFS as a primary element.

• *SpMV.* It is at the heart of many iterative solvers. SpMV is characterized by regular access patterns over non-zero elements and irregular access patterns over the vector, based on column index.

5.2.2 Performance

The five applications can be parallelized in the thread-level and mapped to CPEs. The MPEs were only used to manage the threads of CPEs in our experiments. We compared the performance of DFMC, Intel Xeon 5680, and NVIDIA GPU M2090. The MKL10.2.4 was used in Xeon for DGEMM and FFT. The CUBLAS 4.1 and the CUFFT4.1 were used in GPU for DGEMM and FFT. Another application is our own codes and CUDA was used in GPU, and we spent as much effort in optimizing for Xeon/GPU as for DFMC. The performances of FDTD, BFS, and SpMV in Xeon/GPU are the same as in other studies[7,34-37].

We used the "application performance" (amount of effective calculation/executive time) and the "computational efficiency" (application performance/peak performance) as the metrics for DGEMM, FFT, and FDTD, and the "speedup" (execution time on target architecture/execution time on DFMC) for BFS and SpMV.

• *DGEMM.* DGEMM was mapped in DFMC with all of the cooperative computing techniques. Multi-pattern data stream transfer not only supports data arrangement in CPE cluster according to the application requirement but also hides memory access latency by double buffering. DGEMM implements $\boldsymbol{C} = \boldsymbol{C} - \boldsymbol{A} \cdot \boldsymbol{B}$, where $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ are double-precision floating-point matrices. $\boldsymbol{A}$ and $\boldsymbol{C}$ adopt the row pattern, while $\boldsymbol{B}$ adopts the single CPE pattern. After that, the calculation is divided into eight rounds. There are eight CPEs doing row broadcast register-level communication, and eight CPEs doing column broadcast register-level communication in every round. The latency of register-level communication is completely hidden by computation.

In this paper, DFMC is compared with Intel® Xeon® and Fermi. The size of the selected matrix is 61 184. Fig.14 provides the results. Furthermore, in order to analyze the impact of cooperative computing techniques on performance, we performed the test of DFMC without register-level communication (DFMC no RLC), row pattern data stream transfer (DFMC no

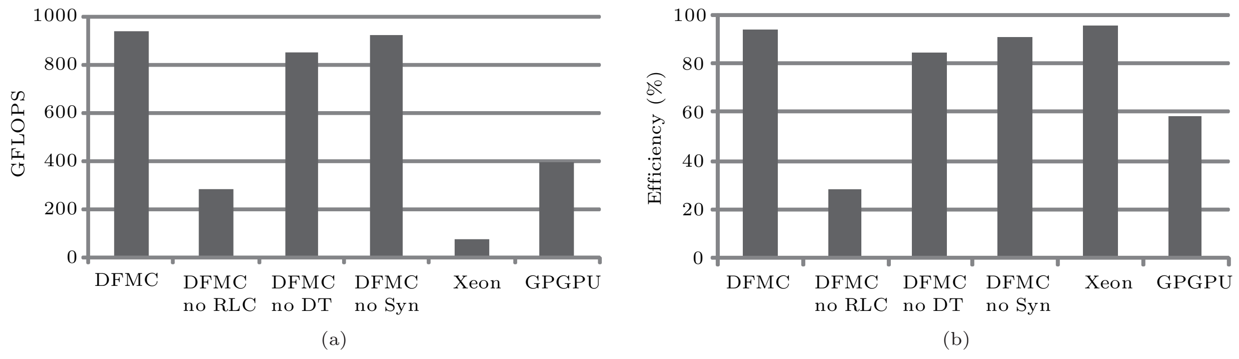


Fig.14. DGEMM in DFMC. (a) Application performance. (b) Computational efficiency.

DT) and hardware synchronization mechanism (DFMC no Syn).

The results show that the peak performance of DFMC reaches 940 GFLOPS, which is higher than those of Xeon multicore processors and GPGPU. And the efficiency is 94%, which is close to that of Intel® Xeon® multicore processors and exceeds that of GPGPU by 35%. In this algorithm, the most important optimization technique is register-level communication, which makes the whole CPE cluster work together. The size of working set on-chip is expanded by 64 times, then the amount of memory access is reduced to 12.5%, which shifts DFMC from the memory access bound to the computing bound. The computational efficiency is improved by 65.8% by register-level communication. In this algorithm, the efficiency is improved by 9.7% due to the row transfer pattern, which is fairer for each CPE. The efficiency is improved by 2.8% with hardware synchronization over software synchronization.

• *FFT*. FFT was mapped in DFMC with all of the cooperative computing techniques. Multi-pattern data stream transfer not only supports data arrangement in CPE cluster according to the application requirement but also hides memory access latency by double buffering. The data were transferred in the array pattern. After local computation, each CPE needs to communicate with other CPEs whose distance is the square of the loop round. Take CPE 0 for example, the target CPE of the 6-round communication is 1, 2, 4, 8, 16, and 32, exactly the CPEs in the same row or column.

The experiment was based on the double-precision FFT with the size of 32 768 and the results are compared with Xeon® and Fermi, as shown in Fig.15. Furthermore, we performed the experiments on DFMC without register-level communication (DFMC no RLC), array pattern data stream transfer (DFMC no DT), and hardware synchronization mechanism (DFMC no Syn).

The results show that the bound of FFT is the memory access. The peak performance is 207 GFLOPS, which is higher than that of Xeon® multicore processors and GPGPU. The efficiency is 20.7% and lower than that of Xeon® multicore processors by 5.8% and higher than that of GPGPU by 3.5%. In FFT, the most important technique is register-level communication, which can reduce the amount of memory access by approximately 50%. Without that, the computa-

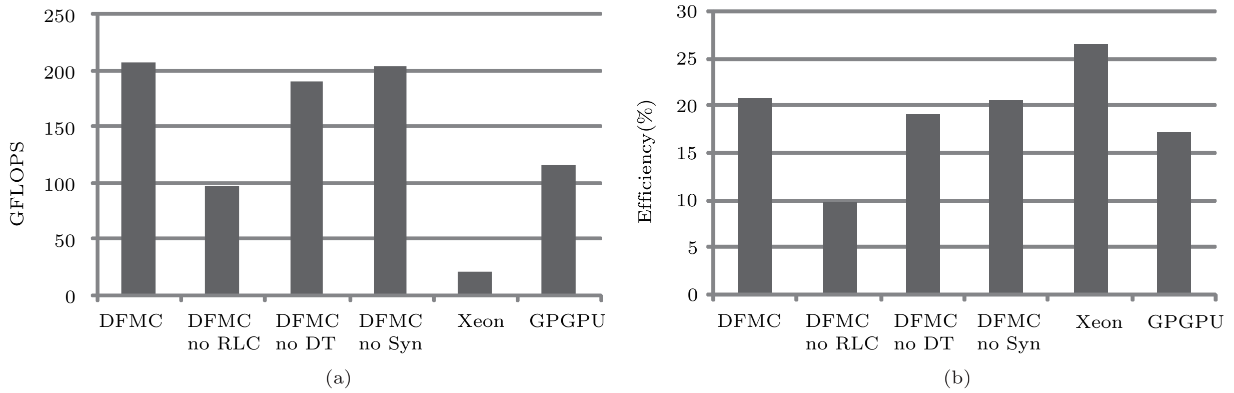
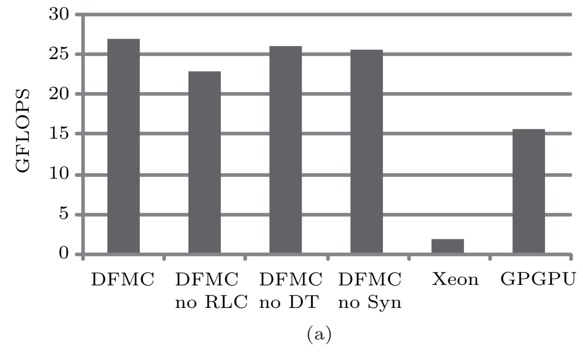


Fig.15. FFT in DFMC. (a) Application performance. (b) Computational efficiency.

tion efficiency is only 9.73%. In FFT, the array pattern improves the efficiency by approximately 1.6%, which is fairer for each CPE. There are a few synchronization operations in FFT and the impact of hardware synchronization on performance is small.

• *FDTD.* In DFMC, the grid of 3D-FDTD is divided into blocks, which are specified in two dimensions, and the data are transferred in the array pattern stream transfer. Each CPE obtained the border points data from other CPEs through adjacent mode register-level communication.

In this paper, DFMC is compared with Intel® Xeon® and Fermi. The size of the selected grid is 256×500×100. Fig.16 provides the results. We performed the test of DFMC without register-level communication (DFMC no RLC), array pattern data stream transfer (DFMC no DT), and hardware synchronization mechanism (DFMC no Syn).



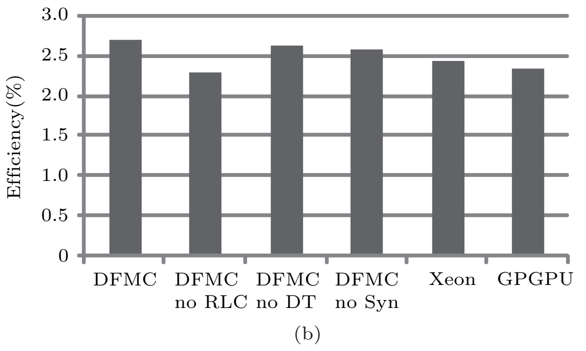


Fig.16. FDTD in DFMC. (a) Application performance. (b) Computational efficiency.

In FDTD, the ratio of computation to data accesses is $O(1)$, and the bound is the memory access. The results show the peak performance is 27 GFLOPS, and the efficiency is 2.7%, both are higher than those of Xeon® multicore processors and GPGPU. If without register-level communication, every CPE must load some redundant boundary data from the main memory. Then, the register-level communication can reduce the amount of redundant boundary data access by approximately 15%. The array pattern stream transfer improves the performance by approximately 2.8%. The performance is improved by 5% with hardware synchronization over software synchronization.

• *Applications with Irregular Data Access.* BFS and SpMV have a large working set and very little computation. Because of irregular data access, the cooperative computing techniques cannot be used in these applications. This experiment shows the limitation of DFMC and cooperative computing techniques. The performance of BFS and SpMV is measured by TEPS (traversed edges per second) and FLOPS. For briefness and clarity, we analyze BFS and SpMV together here with speedup (execution time on target architecture/execution time on DFMC) as the metric.

In DFMC, BFS scale is 23, and SpMV data size is 4GB with compressed row storage. Fig.17 provides the results. The most important factor of performance is the peak global memory bandwidth. Fermi bandwidth is approximately 1.4X of that of DFMC and 4.5X of that of Xeon. The application speedup results shown in Fig.17 almost report this trend.

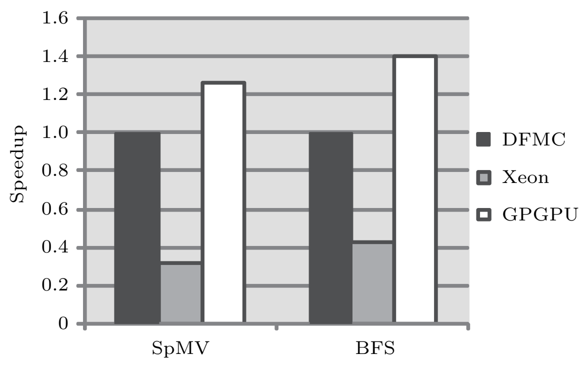


Fig.17. SpMV and BFS in DFMC.

Above all, the cooperative computing techniques can be used to improve the performance in the applications with regular data access, such as DGEMM, FFT, and FDTD. The register-level communication is the across-board best performer because it can reduce the amount of memory access by on-chip data reuse. The multi-pattern data stream transfer makes the data arrangement in the CPE cluster fit the application feature, and makes the memory access more fair for each CPE. Although the hardware synchronization is much more efficient than software counterpart, the best performance promotion is just 5% (FDTD) for less synchronization operations in these applications. It maybe plays a greater role in other fields.

One of the main drawbacks of the cooperative computing techniques is that they are hard to use in applications with irregular data access.

6 Related Work

Our proposed DFMC and cooperative computing techniques were inspired from many studies on processor architecture, on-chip memory management, on-chip communication and synchronization.

Amdahl's Law and Heterogeneous Many-Core Architecture. According to Amdahl's law, fully parallel applications are not available, and the whole application performance is deeply affected by the performance of sequential component[38]. Hill and Marty[39] found that the Amdahl's law is also effective in many-core era. By analyzing the kernel sections of key applications, heterogeneous many-core computing is the trend of future computing on chip[33]. Early in 2005, IBM released the Cell processor, the first heterogeneous processor in high performance computing, which includes one general purpose processing unit and eight synergistic processing elements[40-41]. In academic research, the heterogeneous architecture is also broadly adopted, such as the work in [42], which focuses on energy-efficient computing, and in [43], which studies the ISA of heterogeneous chip multiprocessors. Major companies have already started the research and development on heterogeneous many-core processors, for example, AMD's APU[44-45], NVIDIA's Echolen[46]. Those processors integrate powerful complex processing core and thread accelerating core within single chip, which can significantly improve the chip's performance while still keeping the abilities for accommodating different working sets.

On-Chip Memory Management. On-chip memory management is very important to improve application execution efficiency. GPGPU and Cell processor both use none cache structure[8,47-48], which is explicitly used by software. This on-chip software-managed memory can reduce the amount of external memory access by cache conflicts. NVIDIA GPGPU provides SIMT to fully hide the memory access by computing[8]. Cell processor uses asynchronous DMA and software-managed local store to overlap the memory access[47]. In comparison with Cell, our data stream transfer supports multi-pattern, which can better adapt to the application features and improve the performance of memory access.

On-Chip Communication and Synchronization. Keckler *et al.*[49] proposed fine-grain communication mechanism in multi-ALU processor, which supports register-to-register communication between clusters through writing directly into the register file of another cluster. Shared memory in GPGPU[8] and L2 cache in Intel MIC[10] are both shared by threads to improve data utilization on chip. Tile64[12] has five individual 2D mesh NoCs, which support the communication between tiles and distributed cache sharing. Intel SCC chip uses high performance mesh network and message passing buffer (MPB) to support data sharing and lightweight MPI communication among cores[13-15,50]. In comparison with [12, 49], the communication latency is similar, but the sender and the receiver are asynchronous in our register-level communication, and DFMC supports broadcast/multicast and blocking/non-blocking communication. We believe it is a better fit to achieve MPI-like primitives and easier to use.

Hardware implementations of synchronization have been around for a long time; most of them rely on a wired AND line connection for cores or processors. Abellan *et al.*[51] proposed the special network to allow for fast and efficient signaling of barrier arrival and departure. Watkins and Albonesi[52] designed specialized programmable logic (SPL) in heterogeneous CMP. Their work proposed modifications to the baseline SPL design to provide fine-grain inter-thread and barrier communication among cores for obtaining the additional performance benefit. The work in [53] uses NoC to construct distributed synchronization mechanism. In our work, however, synchronization network is a special hardware network and has a dynamic synchronization vector, providing fast and flexible barrier among cores.

7 Conclusions

In this paper, we presented a deeply fused many-core processor (DFMC) architecture for the high performance computing systems. The DFMC architecture integrates MPEs and CPEs, which are heterogeneous processor cores that cooperate seamlessly. Specifically, the DFMC processor can alleviate the memory wall problem through combing a series of CPEs cooperative computing techniques such as multi-pattern data stream transfer, efficient register-level communication mechanism, and fast hardware synchronization technique. With these techniques, we are able to improve the on-chip data reuse and optimize memory access performance.

In this paper, we also illustrated the implementation of a full system prototype with four MPEs and

256 CPEs using 374 FPGAs. Our experimental results based on the prototype system show that the maximum bandwidth of data stream transfer achieved is 2 165.8 GB/s, which is 21.2 times faster than the physical memory bandwidth, and the register-level communication bandwidth of a CPE cluster achieved is 889.82 GB/s in broadcast mode. Compared with traditional software synchronization, the speed-up ratio can achieve 88~500 times by using the hardware synchronization technique. We also mapped the DGEMM, FFT, FDTD, BFS and SpMV to the DFMC prototype. With cooperative computing techniques of CPEs, DGEMM can achieve the efficiency of 94% (940 GFLOPS), FFT can obtain the performance of 207 GFLOPS, and FDTD obtains the performance of 27 GFLOPS. Our analysis of the result shows that the cooperative computing techniques of CPEs can effectively improve the computational efficiency of typical applications with regular data access. Conversely, BFS and SpMV with irregular memory data access cannot use the cooperative computing techniques, and the performance is almost proportional to the physical global memory access bandwidth.

For future work, we expect to map and optimize more applications and algorithms on DFMC. In addition, we plan to tape out through MPW. Finally, we aim to carry out further studies on DFMC regarding the programming model, CPE pipeline, and power control.

References

[1] Manferdelli J L, Govindaraju N K, Crall C. Challenges and opportunities in many-core computing. *Proceedings of the IEEE*, 2008, 96(5): 808-815.

[2] Shalf J, Dosanjh S, Morrison J. Exascale computing technology challenges. In *Proc. the 9th Int. High Performance Computing for Computational Science–VECPAR*, June 2011, pp.1-25.

[3] Daga M, Aji A M, Feng W. On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing. In *Proc. Symposium on Application Accelerators in High-Performance Computing*, July 2011, pp.141-149.

[4] Chung E S, Milder P A, Hoe J C, Mai K. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proc. the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2010, pp.225-236.

[5] Lee V W, Grochowski E, Geva R. Performance benefits of heterogeneous computing in HPC workloads. In *Proc. the 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, May 2012, pp.16-26.

[6] Kumar R, Farkas K I, Jouppi N P *et al.* Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proc. the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO),* Dec. 2003, pp.81-92.

[7] Lee V W, Kim C, Chhugani J *et al.* Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *Proc. the 37th Annual International Symposium on Computer Architecture (ISCA)*, June 2010, pp. 451–460.

[8] Wittenbrink C M, Kilgariff E, Prabhu A. Fermi GF100 GPU architecture. *IEEE Micro*, 2011, 31(2): 50-59.

[9] Kapasi U J, Dally W J, Rixner S *et al.* The imagine stream processor. In *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors(ICCD)*, September 2002, pp. 282–288.

[10] Duran A, Klemm M. The Intel® many integrated core architecture. In *Proc. International Conference on High Performance Computing and Simulation (HPCS)*, July 2012, pp. 365-366.

[11] Alves M A Z, Freitas H C, Navaux P O A. Investigation of shared L2 cache on many-core processors. In *Proc. the 22nd International Conference on Architecture of Computing Systems (ARCS)*, March 2009, pp. 1-10.

[12] Wentzlaff D, Griffin P, Hoffmann H *et al.* On-chip interconnection architecture of the Tile Processor. *IEEE Micro*, 2007, 27(5): 15-31.

[13] Howard J, Dighe S, Hoskote Y *et al.* A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proc. IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb. 2010, pp.108-109.

[14] Hoskote Y, Vangal S, Singh A *et al.* A 5-GHz mesh interconnect for a Teraflops processor. *IEEE Micro*, 2007, 27(5): 51-61.

[15] Gries M, Hoffmann U, Konow M *et al.* SCC: A flexible architecture for many-core platform research. *Computing in Science and Engineering*, 2011, 13(6): 79-83.

[16] Balakrishnan A, Naeemi A. Interconnect network analysis of many-core chips. *IEEE Transactions on Electron Devices*, 2011, 58(9): 2831-2837.

[17] Taylor M B, Lee W, Amarasinghe S *et al.* Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In *Proc. the 9th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2003, pp.341-353.

[18] Kim J. Low-cost router microarchitecture for on-chip networks. In *Proc. the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO),* Dec. 2009, pp.255-266.

[19] Jung H, Ju M, Che H. A theoretical framework for design space exploration of manycore processors. In *Proc. the 19th Annual IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, July 2011, pp.117-125.

[20] Seiler L, Carmean D, Sprangle E *et al.* Larrabee: A many-core x86 architecture for visual computing. *IEEE Micro*, 2009, 29(1): 10-21.

[21] Chen P, Zhao H L, Tao C, Sang H S. Block-run-based connected component labelling algorithm for GPGPU using shared memory. *Electronics Letters*, 2011, 47(24): 1309-1311.

[22] Sawant N, Kulkarni D. Performance evaluation of feature extraction algorithm on GPGPU. In *Proc. International Conference on Communication Systems and Network Technologies (CSNT)*, June 2011, pp. 536-540.

[23] Heinecke A, Klemm M, Bungartz H J. From GPGPU to many-core: Nvidia Fermi and Intel many integrated core architecture. *Computing in Science & Engineering*, 2012, 14(2): 78-83.

[24] Bell S, Edwards B, Amann J *et al.* TILE64^{TM}-processor: A 64-core SoC with mesh interconnect. In *Proc. IEEE Int. Solid-State Circuits Conference (ISSCC)*, February 2008, pp.88-89, 598.

[25] Sewell K, Dreslinski R G, Manville T *et al.* Swizzle-switch networks for many-core systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2012, 2(2): 278-294.

[26] Kim J, Balfour J, Dally W. Flattened butterfly topology for on-chip networks. In *Proc. the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007, pp.172-182.

[27] Bakhoda A, Kim J, Aamodt T M. Throughput-effective on-chip networks for manycore accelerators. In *Proc. the 43rd Annual IEEE/ACM MICRO*, Dec. 2010, pp. 421-432.

[28] Fan D, Zhang H, Wang D *et al.* Godson-T: An efficient many-core processor exploring thread-level parallelism. *IEEE Micro*, 2012, 32(2): 38-47.

[29] Wang X, Gan G, Manzano J *et al.* A quantitative study of the on-chip network and memory hierarchy design for many-core processor. In *Proc. the 14th IEEE International Conference on Parallel and Distributed Systems*, Dec. 2008, pp. 689-696.

[30] Taylor M B, Psota J, Saraf A *et al.* Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. the 31st Annual International Symposium on Computer Architecture (ISCA)*, June 2004, pp. 2-13.

[31] Taylor M B, Kim J, Miller J *et al.* The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 2002, 22(2): 25-35..

[32] Bakhoda A, Kim J, Aamodt T M. Throughput-effective on-chip networks for manycore accelerators. In *Proc. the 43rd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2010, pp.421-432.

[33] Asanovic K, Bodik R, Catanzaro B C *et al.* The landscape of parallel computing research: A view from Berkeley. Technical Report, UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[34] Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. Conference on High Performance Computing Networking, Storage and Analysis*, November 2009, Article No.18.

[35] Choi J W, Singh A, Vuduc R. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proc. the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* January 2010, pp.115–126.

[36] Luo L, Wong M, Hwu W. An effective GPU implementation of breadth-first search. In *Proc. the 47th Design Automation Conference (DAC)*, June 2010, pp.52-55.

[37] Bo Z, Zheng-hui X, Wu R *et al.* Accelerating FDTD algorithm using GPU computing. In *Proc. IEEE International Conference on Microwave Technology & Computational Electromagnetics*, May 2011, pp.410-413.

[38] Hennessy J L, Patterson D A. Computer Architecture: A Quantitative Approach (5th edition). Morgan Kaufmann, 2011.

[39] Hill M, Marty M. Amdahl's law in the multicore era. *IEEE Computer*, 2008, 41(7): 33-38.

[40] Riley M W, Warnock J D, Wendel D F. Cell broadband engine processor: Design and implementation. *IBM Journal of Research and Development*, 2007, 51(5): 545-557.

[41] Kahle J A, Day M N, Hofstee H P *et al.* Introduction to the Cell multiprocessor. *IBM Journal Research and Development*, 2005, 49(4): 589-604.

[42] Woo D H, Lee H H S. Extending Amdahl's law for energy-efficient computing in the many-core era. *IEEE Computer*, 2008, 41(12): 24-31.

[43] Kumar R, Tullsen D M, Ranganathan P *et al.* Single-ISA heterogeneous multicore architectures for multithreaded workload performance. In *Proc. the 31st Annual International Symposium on Computer Architecture*, June 2004, pp. 64-75.

[44] Yang Y, Xiang P, Mantor M *et al.* CPU-assisted GPGPU on fused CPU-GPU architectures. In *Proc. the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, February 2012.

[45] Branover A, Foley D, Steinman M. AMD fusion APU: Llano. *IEEE Micro*, 2012, 32(2): 28-37.

[46] Keckler S W, Dally W J, Khailany B *et al.* GPUs and the future of parallel computing. *IEEE Micro*, 2011, 31(5): 7-17.

[47] Khunjush F, Dimopoulos N J. Extended characterization of DMA transfers on the Cell BE processor. In *Proc. IEEE International Symposium on Parallel and Distributed Processing*, April 2008.

[48] Gebhart M, Keckler S W, Khailany B *et al.* Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proc. the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2012, pp.96-106.

[49] Keckler S W, Dally W J, Maskit D *et al.* Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor. *ACM SIGARCH Computer Architecture News*, 1998, 26(3): 306-317.

[50] Korch M, Rauber T, Scholtes C. Memory-intensive applications on a many-core processor. In *Proc. the 13th IEEE International Conference on High Performance Computing and Communications (HPCC)*, September 2011, pp.126-134.

[51] Abellán J L, Fernández J, Acacio M E. Efficient hardware barrier synchronization in many-core CMPs. *IEEE Transactions on Parallel and Distributed Systems,* 2012, 23(8): 1453-1466.

[52] Watkins M A, Albonesi D H. ReMAP: A reconfigurable heterogeneous multicore architecture. In *Proc. the 43rd IEEE International Symposium on Microarchitecture,* Dec. 2010, pp. 497-508.

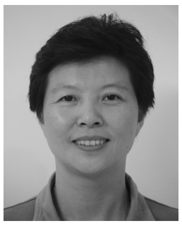
[53] Yu L, Liu Z, Fan D *et al.* Study on fine-grained synchronization in many-core architecture. In *Proc. the 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, May 2009, pp.524-529.

Fang Zheng received his M.S. degree in computer science from National Research Center of Parallel Computer Engineering and Technology, Beijing. Currently he is a Ph.D. candidate in computer science of State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi. His research interests include high performance computing and processors architecture.

Hong-Liang Li received his Ph.D. degree in computer science from National University of Defense Technology, Changsha. He is an associate professor of State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi. His research interests include computer architecture and processors architecture.

Hui Lv received his M.S. degree in computer science from Fudan University, Shanghai. Currently he is a Ph.D. candidate in computer science of State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi. His research interests include high performance computing and processors architecture.

Feng Guo received his Ph.D. degree in computer science from National Research Center of Parallel Computer Engineering and Technology, Beijing. Currently, he is an engineer of the State Key Laboratory of Mathematical Engineering and Advance Computing. His research interests include high performance computing and processors architecture.



Xiao-Hong Xu received her M.S. degree in computer science from National Research Center of Parallel Computer Engineering and Technology, Beijing. Currently, she is a senior engineer of the State Key Laboratory of Mathematical Engineering and Advance Computing. Her research interests include computer architecture and FPGA.

Xiang-Hui Xie received his Ph.D. degree in computer science from Institute of Computer Technology, Chinese Academy of Sciences, Beijing. He is a professor of State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi. His research interests include computer architecture and parallel computing.