Hybrid Implementation and Optimization of OpenFOAM on the SW26010 Many-core Processor

Delong Meng*, Minhua Wen*, Jianwen Wei*, James Lin*†,
*Center for High Performance Computing, Shanghai Jiao Tong University, China
†Tokyo Institute of Technology, Japan
wenminhua@sjtu.edu.cn

Abstract

The Sunway TaihuLight supercomputer based on the Chinese-designed SW26010 many-core processors is the world's fastest system with a peak performance higher than 100 PFlops [1]. OpenFOAM is one of the most popular open source Computational Fluid Dynamics (CFD) software which is written in C++ and not fully compatible with compilers on SW26010. We propose a hybrid implementation and optimizations for OpenFOAM based on SW26010's MPE (management processing element)/CPE (computing processing element) cluster architecture. To overcome the compilation incompatibility problem, we adopt the mixed-language application design. We also apply several SW26010's feature-specific optimizations on the hotspot of OpenFOAM to deliver a high performance, such as the register communication, vectorization, and DMA optimization. The experiments on SW26010 using real datasets show that the single-CG (core group) code runs 8.03x faster than the optimized implementation on the MPE, and the performance of single-CG is as high as that of single-core of Intel(R) Xeon(R) CPU E5-2695 v3. The implementation and optimizations we present can also be referenced for other complex C++ programs to achieve high performance on SW26010.

Keywords: SW26010 Many-core Processor, OpenFOAM, Hybrid Implementation

1. Introduction

Computational Fluid Dynamics (CFD) is a cornerstone in the understanding of many scientific and technological areas such as meteorological phenomena, aerodynamics, and environmental hazards [2]. Computational science enabled by High-Performance Computing (HPC) makes it possible for scientists and researchers to simulate these phenomena in a virtual laboratory. CFD is extensively used throughout the whole process, from early concept phases to the detailed analysis of a final product [3].

Open Source Field Operation and Manipulation (OpenFOAM) [4] is is one of the most popular CFD applications written in C++. It consists of a large set of pre-processing utilities, partial differential equation (PDE) solvers, and post-processing tools. An attraction of OpenFOAM is in its modularity, which leads to an efficient and flexible design. It features a broad range of solvers employed in CFD, such as Laplace and Poisson equations, incompressible flow, multiphase flow, and user-defined models. Although it is a powerful framework for solving a variety of problems in the field of CFD, enormous computations on real datasets need to be finished in acceptable time.

The Sunway TaihuLight supercomputer is the world's fastest system with a peak performance higher than 100 PFlops. It is based on the Chinese-designed SW26010 many-core processors. The processor includes four core groups(CGs), each of which consists of one management processing element (MPE) and sixty-four computing processing elements (CPEs) arranged by an eight by eight grid. The basic compiler components on MPE support C/C++ programming language, while the compiler components on CPE only support C. The compilation incompatibility problem makes it difficult for C++ programs to exploit the computing power of the SW26010 processor.

As mentioned above, Massive computing resources are required for complex CFD simulations. The computing resource of the Sunway TaihuLight supercomputer is a good choice. However, it is based on the Chinese-designed processors. We need to implement and optimize solvers according to its unique architecture. And the methods we present can be referenced for other complex C++ programs to achieve high performance in a productive way on SW26010.

In this work, we specifically target three basic solvers and ten incompressible flow solvers in OpenFOAM. Then we study and optimize the most expensive components of the selected solvers. According to the profiling results and the performance analysis of the solvers, which is, in our case, the conjugate gradient solver (PCG). Therefore, we propose a hybrid implementation and optimizations for the linear solver based on SW26010's MPE /CPE cluster architecture.

This paper is organized as follows. Section 2 introduces the architecture of the SW26010 processor and related work, and Section 3 provides details about Navier-Stokes equations and the conjugate gradient algorithm. We describe mixed-language application design in Section 4.1, followed by a discussion of the implementation and optimizations on the MPE and the CPE cluster respectively in Section 4.2 and Section 4.3. Section 5 shows the experimental results, and Section 6 concludes.

Minhua Wen. wenminhua@sjtu.edu.cn

2. Background

2.1. The SW26010 Many-core Processor

The SW26010 processor includes four CGs connected via the network on chip (NoC). Each CG includes one MPE, one CPE cluster, one protocol processing unit and one memory controller (MC). Each CG has its own memory space, which is connected to the MPE and the CPE cluster through the MC. And each MPE has the access to the PCI-E interface. The interface can also support consistent cached data access. The processor is connected to other outside devices through a system interface.

The MPE is a complete 64-bit RISC core, which can run in both the user and system modes. The MPE completely supports the interrupt functions, memory management, and out-of-order execution. It also supports four-decode and seven-issue superscalar processing. In contrast, the CPE is also a 64-bit RISC core, but with limited functions. The CPE can only run in user mode and does not support interrupt functions. It supports two-decode and two-issue superscalar processing. The design goal of this element is to achieve the maximum aggregated computing power while minimizing the complexity of the micro-architecture. The CPE cluster is organized as an eight by eight mesh, with a mesh network to achieve low-latency register data communication among the eight by eight CPEs. Both the MPE and CPE support 256-bit vector instructions [5].

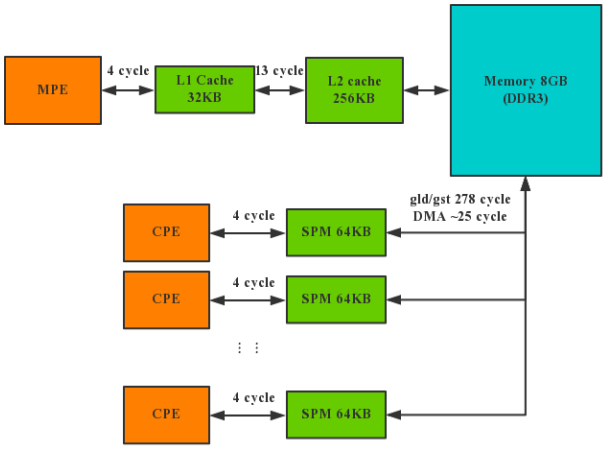

Fig. 1: Latency of memory access on the SW26010 processor

The Sunway TaihuLight supercomputer is based on the SW26010 processor, of which the high-level architecture is illustrated in Figure 1. Each MPE has a 32 KB L1 instruction cache and a 32 KB L1 data cache, with a 256 KB L2 cache for both instruction and data. The L1 cache on the MPE is four-way set associative, with 128-byte cache lines. The L2 cache on the MPE is eight-way set associative, with 128-byte cache lines. Each CPE has its own 64KB Scratch Pad Memory (SPM). SPM can be configured as either a fast buffer that supports precise user-control or a software-emulated cache that achieves automatic data caching. We can see the on-chip buffer size and the memory bandwidth are relatively limited for the CPE cluster when compared with existing GPU and MIC chips. Hence, reducing the overhead of data transfer on the CPE cluster is an emphasis in our optimization.

2.2. Related Work

In the past decade, researches in CFD computation catch more and more attention. Several studies have been conducted to port OpenFOAM on GPUs. Significant work has been done on GPUs, most of which is for linear algebra routines such as QR factorization on multi-GPU [6], Cholesky factorization [7], sparse direct solvers [8], Conjugate Gradient and multi-grid solvers [9]. While linear system solution leads the implementation wave onto new hardware, because it is a key kernel, there is new significant interest in migrating whole applications to the hybrid environment.

The addition of GPUs to high-end scientific computer systems anticipates new available performance levels. As a result, several works attempt to employ GPGPUs in CFD codes and much more attention is focused on sparse linear kernels, such as implementing the conjugate gradient on GPU [10]. These works include the Cufflink library [11], which extends the OpenFOAM linear solvers capabilities to perform on GPUs.

Pioneering CFD works have also addressed heterogeneous and hybrid systems. A parallel simulation of oil extraction has been designed and optimized for heterogeneous networks of computers [12]. Papadrakakis [13], attempts to balance the computation across heterogeneous hybrid CPU+GPU system. The balancing is performed at runtime by using task-based parallelism and migrations between the compute devices. On the other hand, a general framework is supported by StarPU [14] project, which provides and supports task-based programming and schedules the provided tasks on a heterogeneous platform, which combines CPUs and GPUs.

However, to the best of our knowledge, there has not been any implementation of OpenFOAM solvers on the SW26010 processor. The current implementation is limited to CPU/GPU. This work proposes a MPE/CPE model on SW26010 and model-based domain partitioning. It aims at more efficient exploitation and utilization of SW26010 for the CFD computation.

3. Algorithms

Modularity and flexibility are the main advantages of OpenFOAM code. One of the modules is the linear algebra module, which can be seen in almost all the PDE solver codes. One of the linear solvers, which can be used in all the selected solvers, is the conjugate gradient.

The Pressure Implicit with Splitting of Operators (PISO) algorithm, which is applied in time stepping loop, solves

the incompressible flow iteratively. It is commonly used in the solvers of OpenFOAM. The algorithm is essentially a predict-and-correct procedure for calculation of pressure on the collocated grid, where the flow quantities are defined in the control volume. The control-volume discretized Navier-Stokes equations for incompressible flow are as follows:

$$a_c u_c = -\sum_n a_n u_n + \frac{u^*}{\triangle t} - \sum_f S(p)_f \tag{1}$$

$$\sum_f S[(\frac{1}{a_c})_f \nabla p_f] = \sum_f S(\frac{H(u)}{a_c})_f \tag{2}$$

Here a_n is the matrix coefficient corresponding to the neighbors n and a_c is the central coefficient. The subscript f implies the value of the variable in the middle of the face and S is the outward-pointing face area vector. Equation 3 computes the face flux, F. The fluxes should satisfy the continuity equation.

$$F = S \times u_f = S \times [(\frac{H(u)}{a_c})_f - (\frac{1}{a_c})_f \nabla p_f] \tag{3}$$

The algorithm consists of three stages. The first stage is the momentum predictor, which solves the momentum equation by using an initial or previous pressure field. This solution of the momentum equation gives the velocity field that is not divergence free but approximately satisfies the momentum equation. The second stage is the pressure solution that formulates the pressure equation and assembles the $H(u)$ term. Third is the explicit velocity correction stage. Equation 3 gives set of conservative fluxes consistent with the new pressure field. Therefore, the velocity field is corrected as a consequence of the pressure distribution. The velocity correction is performed in an explicit manner using Equation 4, which consists of two parts: $\frac{H(u)}{a_c}$ and $\frac{1}{a_n}\nabla p$. The first part is the transported influence of corrections of neighboring velocities. The second one is correction due to the change in the pressure gradient. The main velocity error comes from the error in the pressure term; however, it is necessary to correct the $H(u)$ term, formulate the new pressure equation, and repeat the procedure again [3].

$$u_c = \frac{H(u)}{a_c} - \frac{1}{a_n}\nabla p \tag{4}$$

The result of the discretization process is a system of algebraic equations. The matrices derived from the partial differential equations are large and sparse. The iterative methods avoid factorization, and instead perform matrix-vector operations and minimize the residual over the resulting vector space. These procedures are often referred to as Krylov subspace methods.

One of the best-known Krylov subspace methods is Conjugate Gradient [15], which is used to solve a system of linear equations given by the following form:

$$Ax = b$$

Here x is an unknown vector of size number of cells in the computational domain, b is a corresponding length vector, and A is a known square, symmetric, positive-definite matrix. A flow chart of the preconditioned conjugate gradient (PCG) algorithm is as Figure 2. The main building blocks of a PCG iteration, as formerly mentioned, are: a sparse matrix vector multiply kernel, an optional preconditioning, three global dot products, vector scalings and vector-vector additions. Another important conjugate gradient solver is the preconditioned bi-conjugate gradient (PBiCG) algorithm. They are used to solve the pressure and velocity fields respectively. We specifically targeted above two algorithms and designed the computational models on the MPE and the CPE cluster.

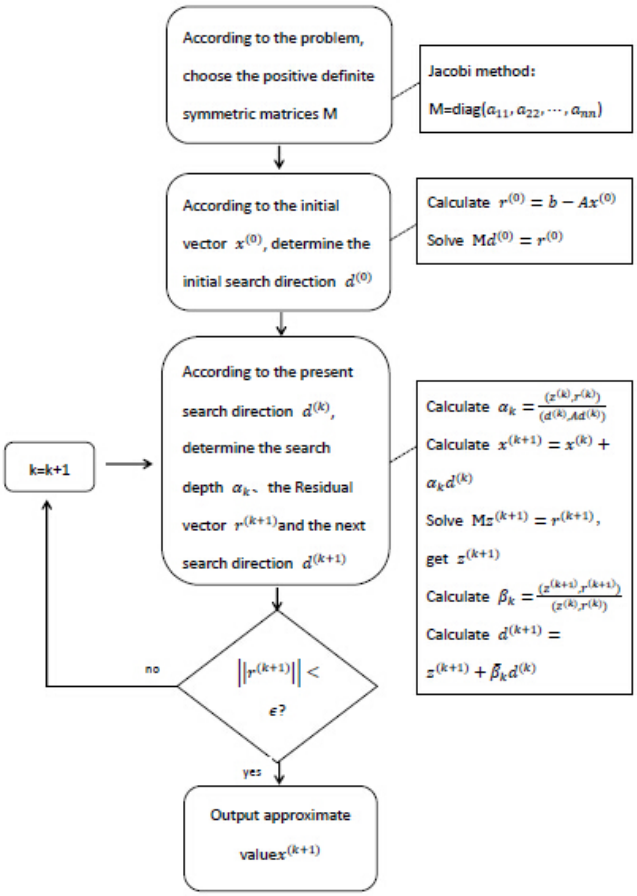

Fig. 2: PCG algorithm

4. OpenFOAM on the SW26010 Processor

4.1. Mixed-Language Application Design

The kernel code in OpenFOAM is written in C++. The object-oriented programming language is convenient to customize and extend its existing functionality. However, the SIMD library is only supported by the C compiler sw5cc on the MPE, which makes it impossible for the developers to

adjust vectorization on the C++ kernel code manually. And the CPE cluster only supports the C compiler sw5cc. The kernel code written in C is necessary for exploiting the computing power of the SW26010 processor. Also, sw5cc is more mature than the C++ compiler swg++-4.5.3 on the MPE. It can be further optimized with the compiler options.

However, considering the overall complexity of the program, it is quite difficult to implement the program simply with C language. We propose one mixed-language programming model for OpenFOAM. Since the architecture of the SW26010 processor is different, we modify the data storage format and re-implement the kernel code with C language.

The kernel code in OpenFOAM consists of several kinds of classes, such as lduMatrix and solverPerformance. We replace the object storage with arrays and implement the behaviors of the classes with only arithmetic operations. Without the templates and inheritance, the code is cleaner and more compatible with the compiler.

The rest part of the program is still based on C++. With the mixed-language application design, the performance of the solvers is improved, while the portability of the program can be ensured.

4.2. OpenFOAM on the MPE

4.2.1. Implementation on the MPE

Since the C/C++ compilers on the MPE are specially developed, we first propose a new compilation method for OpenFOAM. With the default compilation mode, ThirdParty and OpenFOAM share the same compiler. However, we need to get executable tools on the management machine when we compile ThirdParty. The new compilation method is to compile ThirdParty and OpenFOAM with GCC and swg++-4.5.3 respectively.

What's more, we find that the dynamic library doesn't work stable on the MPE. We change the linking mode of OpenFOAM and use the static library instead. However, some necessary libraries are determined according to the input conditions. They aren't added into the final executable file. So we explicitly link them with the object files of the solvers.

4.2.2. Optimization on the MPE

4.2.2.1. Vectorization

The MPE supports 256-bit vector instructions. Instead of automatic vectorization, we align part of the data and adjust vectorization. After aligning the data, we can use simd_load to load it into the vector registers and use simd_store to save it into the memory. The instruction maps the data to vector registers directly and operates on all loaded data in a single operation. It makes full use of the pipeline and significantly improves the floating point computing performance.

What's more, we can reduce the memory access by reusing vector registers. There are 32 vector registers in total on the SW26010 processor. In the precondition part of PBiCG, we need to read the data stored in the lduMatrix class repeatedly. To maximize the potential for register reuse, we adjust the execution order and finish all the related calculation before we update the data in the registers.

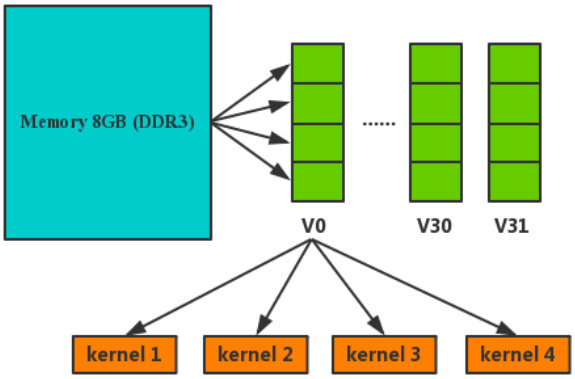

Fig. 3: Vector register reuse

4.2.2.2. Data Presorting

In the precondition part of PCG and PBiCG, we need to read data according to the column number of the sparse matrix. Since the column numbers are discontinuous, the memory access is indirect and irregular. While the data of the source field is not updated during the loop, we can allocate extra free space and resort the data according to the discontinuous column numbers. The input data is preprocessed before the kernel loop. As seen in Figure 4, it is stored in order according to the indexes of the non-zero elements in the sparse matrix.

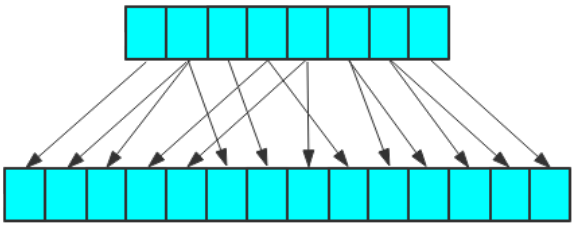

Fig. 4: Data presorting

In the original code, since the indexes are discontinuous, a cache line may contain only one effective element. After presorting the data, all the data in a cache line will be used in the present computation. We convert discrete data to continuous and improve data reuse in cache.

4.2.2.3. Algorithm Optimization

In the original precondition part of PCG and PBiCG in OpenFOAM, we need to calculate the multiplication of the

source field and the precondition vector in every time step until the tolerance for pressure-velocity system is reached. However, after generating the structure of mesh in simulation and the precondition vector, the multiplication can be finished in advance. Since the result is constant in the kernel, the dot product can be accessed directly from the memory. The computation part in Figure 5 is pre-processed before the kernel loop.

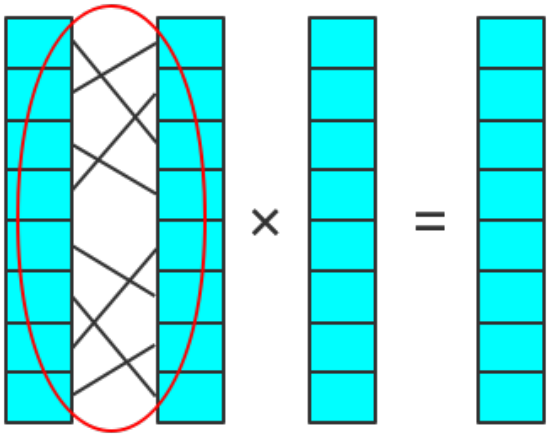

Fig. 5: Algorithm optimization

After optimizing the algorithm, the multiplication of the source field and the precondition vector doesn't need to be repeatedly calculated in the kernel. The cost of redundant computations is reduced. We also manually apply loop fusion in the kernel code to increase the computations in one loop. It can help overlap more computations and memory access on the MPE.

4.3. OpenFOAM on the CPE cluster

4.3.1. Implementation on the CPE cluster

To further improve the performance of the solvers, we implement the master-slave cooperative algorithm of PCG method. OpenFOAM is developed based on C++ while the CPE cluster only supports the C compiler. To implement PCG on the CPE cluster, we need to use the C++ compiler to deal with the libraries on the MPE and the C compiler to deal with the kernel code on the CPE cluster. But the linkers provided on the platform can't support two characteristics in the meantime. Finally we use the loader directly and successfully link the code. But in the hybrid implementation, there are repeated memory accesses in the initialization of functions. We modify the library file and solve the problem.

Then we develop the computation method on the CPE cluster with the athread library, which is designed for the MPE/CPEs programming model on the SW26010 processor. We assign most of the computations to the CPE cluster and implement data transfer with DMA intrinsic. PCG computation method can be seen in the below figure.

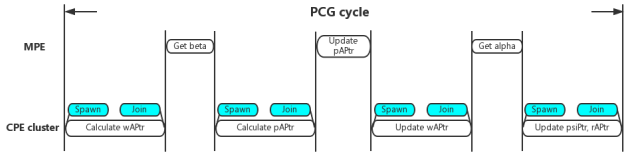

Fig. 6: PCG computation

In the Figure 6, wAPtr is the search depth, pAPtr is the search direction, rAPtr is the residual vector and psiPtr is the present position.

4.3.2. Optimization on the CPE cluster

4.3.2.1. Data Structure Transformation

The kernel of PCG is sparse matrix-vector multiplication (SpMV). Compared to dense linear algebra kernels, sparse kernels suffer from higher instruction and storage overheads per flop, as well as indirect and irregular memory access pattern [16]. To achieve higher performance, we first choose a more compact data structure.

The coefficient matrix of OpenFOAM is stored in the class lduMatrix. The diagonal elements, lower triangular elements, upper triangular elements are stored in individual vectors. The last two vectors share two index vectors, the upperAddr and the lowerAddr. The advantage of lduMatrix is most of the non-zero elements don't occupy extra storage space, especially when the non-zero elements are symmetrically distributed.

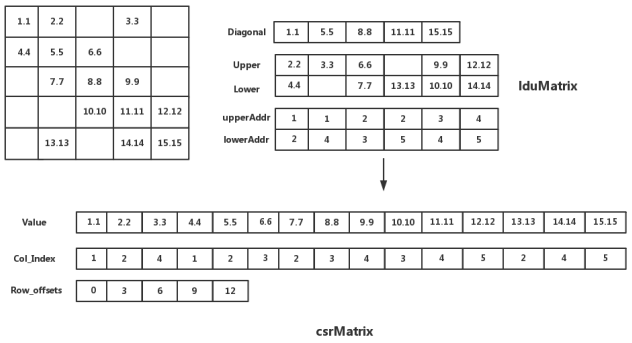

Fig. 7: Data structure transformation

Although the upperAddr is continuous, the lowerAddr is discrete. And the cost of memory access is quite high when calculating the search depth of discrete indexes. So we use csrMatrix to store the data instead. With the new data structure, we calculate the search depth from the first row to the last row in order. Then each CPE updates the data continuously stored in memory and transmits one data block. It is more suitable

for DMA intrinsic and improves the performance of memory access.

4.3.2.2. Register Communication

Before we calculate the search direction, the present position, and the residual vector, we need to calculate the search depth with the whole matrix. It is distributed among the CPEs. When we calculate the search depth on the MPE, there will be more redundant memory access. So we use register communication instead.

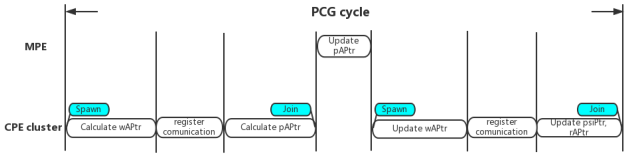

Fig. 8: Register communication

In the register communication, each CPE starts with a buffer containing one element. The data from all CPEs are combined and accumulated at the first CPE within the same row/column into one buffer of size 8. After the first CPE gets the sum, the data needs to be broadcast. Since all-to-one reduction and one-to-all broadcast needs a single core to send and receive data frequently, other cores need to wait until it finishes all the transmissions. We change the communication method and use a three-dimensional hypercube to improve the efficiency. Unlike a linear array, the hypercube broadcast would not suffer from congestion if node 0 started out by sending the message to node 4 in the first step, followed by nodes 0 and 4 sending messages to nodes 2 and 6, respectively. Finally nodes 0, 2, 4, and 6 sending messages to nodes 1, 3, 5, and 7, respectively.

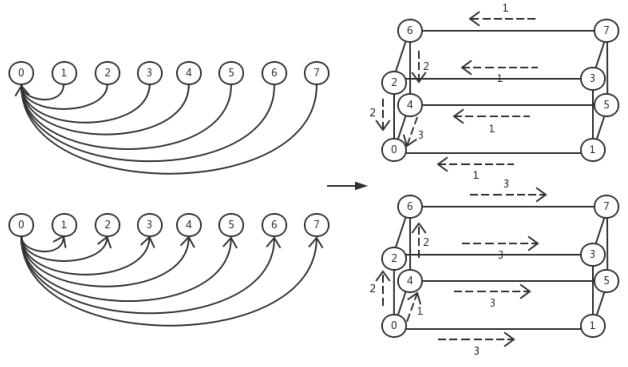

Fig. 9: Broadcast and hypercube

4.3.2.3. DMA Optimization

DMA (Direct Memory Access) is used to transmit data between SPM and memory. Athread library has get/put interfaces for DMA. However, when the CPEs launch athread_get and athread_put, DMA descriptors have to be repeatedly initialized. We replace the memory access functions with embedded assembly code and extract the invariants. With the embedded assembly code, we can initialize the DMA descriptors at the beginning of the function. In the loop, we can use one DMA descriptor to finish the data transmission of the same amount, operation, and mode. It reduces the redundant initialization of DMA descriptors and improves the efficiency of memory access.

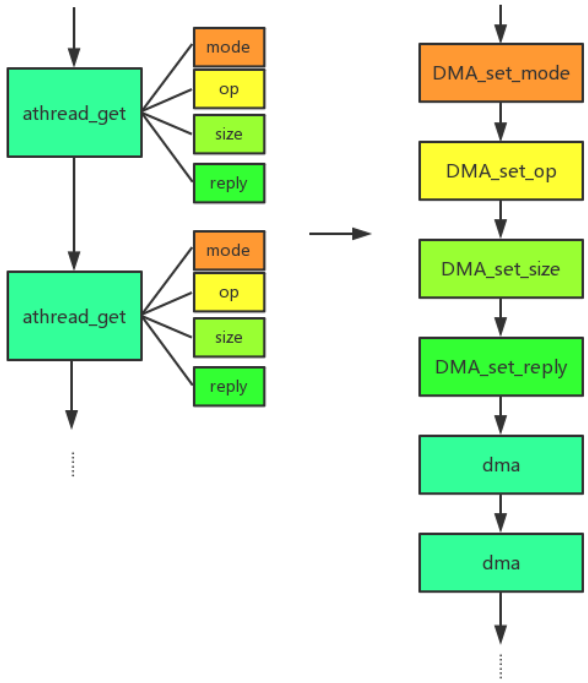

Fig. 10: DMA optimization

4.3.2.4. Other Optimization

1) Prefetching. In the PCG model on the CPE cluster, we need to preprocess the residual vector and calculate the search depth with register communication. Then we update the present search direction with that in the last loop. The data transfer has no data dependency with preprocessing and register communication. The search direction in the last loop is read before we launch register communication. By prefetching, register communication and memory access are overlapped during the loop. It improves the efficiency of the pipelines on the CPEs.

2) Double Buffering. Double buffering is used for storing data in two different buffers in two successive loops. Since there are two pipelines on the CPE, double buffering can be used for the overlap of the computation and memory access on the SW26010 processor. Before we update the present search

direction, we have to wait until the last search direction has been saved back into memory. Then two buffers are used to hold a block of data. After double buffering, while the data is transmitted between memory and SPM, the data in the last loop can be computed in the meantime. The computation and memory access are overlapped, which further improves the performance of memory access.

3) Data Reuse. Since each CPE only has 64KB SPM, we need to save part of the search direction temporarily back to memory. In the future computation, we need to load data repeatedly into SPM. The free space of SPM can only store one block of data. In the original computation, we store the data on the boundary in the free space. The rest data need to be transmitted frequently. To further make use of SPM, we store the data on the boundary in the space of the input vector. Then we can store the data of the last block in the free space. By reusing the data in SPM, we reduce the cost of memory access and data transfer.

5. Results

5.1. Hardware and Test Scenario

We conduct the experiment on the SW26010 TaihuLight supercomputer and the supercomputer π of Shanghai Jiao Tong University. We use one core group of the SW26010 processor and one core of Intel(R) Xeon(R) CPU E5-2695 v3 for the test. The hardware configuration is as Table 1.

TABLE 1: Test Scenario

	SW26010 Processor	Intel CPU E5-2695 v3
Frequency(MPE/CPU)	1.45GHz	2.3GHz
Frequency(CPE)	1.45GHz	/
Memory	8GB	8GB
Page size	8KB	4KB
Huge Page	8MB	2MB/1GB
L1 Icache	64KB, 2-way, 64b line	192KB
L1 Dcache	64KB, 2-way, 64b line	
L2 Cache	256KB	1536KB
L3 Cache	/	15360KB
Compiler	swg++-4.5.3/sw5cc	ICC

The baseline version is tuned by compiler options. Compiler options of the MPE: **-O3 -msimd -LNO:prefetch_ahead=5 -LNO:diverse_pf_ahead=7 -CG:pf_L1_ld:pf_L1_st:pf_L2_ld=0:pf_L2_st=0**. Compiler options of the CPEs: **−O3 −OPT:div_split=1 −msimd**.

Our library is implemented in C and C++ and plugged into OpenFOAM accordingly. Our version of OpenFOAM is OpenFOAM-2.1.0. We choose icoFOAM, which is an incompressible flow solver, to analyze the optimization result with different methods. The test case is the lid-driven cavity flow. The top boundary of the cube is a moving wall that moves in the x-direction, whereas the rest are static walls. The mesh is divided by 400 in the x-direction and 400 in the y-direction. It is generated using the OpenFOAM mesh utility blockMesh.

5.2. SW26010 Processor Performance

5.2.1. Optimization Results on the MPE

Different optimizations are applied on MPE and the performance results are as Figure 11. We set the best performance tuned by compiler options as the baseline. And 2.34x speedup is achieved on MPE finally.

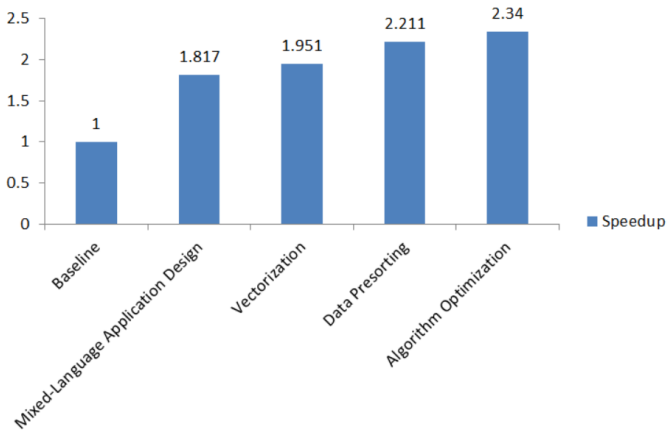

Fig. 11: Test results on the MPE

As described in Section 3.1, a mixed-language programming model is proposed based on C and C++ for OpenFOAM. In the baseline implementation, vectors involved in the conjugate gradient solvers are called by the array form of new[]-expressions to allocate all storage required for an array. With this allocation, the address of the array cannot be guaranteed to be aligned. Additional cost would be generated when unaligned data is loaded. Therefore, we need to call malloc function to allocate extra space. Then we select the address in align from extra space to store the first element of the array. This optimization ensures that the data structures fit the required layout to maximize the utilization of the vector unit. And also the C compiler sw5cc is more mature than the C++ compiler swg++-4.5.3 on the MPE.

For the hotspot of the program on the MPE, we first manually apply vectorization to improve the floating point computing power. And the cost of dynamic memory allocation on the SW26010 processor is relatively high. Hence, we resort the data according to the discontinuous column numbers. By converting discrete data to continuous, we improve the efficiency of cache. Finally we optimize the algorithm and reduce many unnecessary computations. The performances of the computation and memory access are both improved, explaining why we achieve 2.34x speedup based on the baseline implementation.

The PCG is a general algorithm to solve linear algebra equations and can be used in most OpenFOAM solvers, so our optimizations should be valid for other solvers. To further confirm it, we test the performance of three basic solvers and ten incompressible solvers. Similar to icoFOAM, we achieve more than 2x speedup on all the above solvers.

5.2.2. Optimization Results on the CPE Cluster

The optimization result of PCG on the CPE cluster is as Figure 12. We compare the performance of our optimized implementation and the baseline implementation with the Athread library. Finally we achieve 3.7x speedup based on the Athread version.

Since we have utilized SIMD and FMA operations in the CPE cluster, DMA bandwidth limits the performance of the solvers. Hence, we apply different methods to optimize memory access in the hotspot of the program on the CPE cluster. For the data structure, we use csrMatrix to store the data instead of lduMatrix. It is more suitable for DMA intrinsic and improves the memory bandwidth. In the baseline implementation, the search depth vector distributed among the CPEs requires data synchronization on the MPE. The latency of DMA is about 25 cycles, while the latency of register communication is only 10 cycles. To reduce the redundant latency, we use registers to communicate with other CPEs within the same row or column.

Then we replace athread_get and athread_put function with the embedded assembly code. It is a more efficient method to transmit the data between SPM and the memory. Besides, we overlap register communication, computation and memory access with prefetching and double buffering. Finally, data reuse helps us make full use of the data in SPM and reduce the redundant DMA operations. The above optimization methods significantly reduce the overhead of data transfer and provide a 3.7x performance increase.

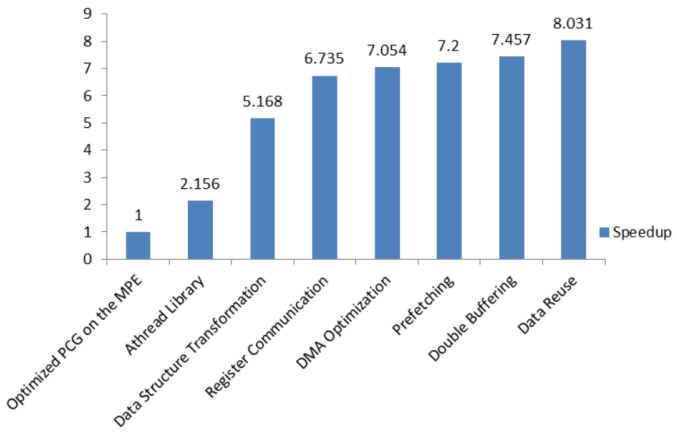

Fig. 12: Test results on the CPE cluster

5.2.3. SW26010 Processor vs. Intel(R) Xeon(R) CPU E5-2695 v3

We make a comparison of the performance of the MPE, the CPE cluster and Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz. After the optimization on the CPE cluster, We achieve 8.03x speedup based on the optimized implementation on the MPE. And the performance of the CPE cluster is 1.18x faster than the single core of Intel(R) Xeon(R) CPU E5-2695 v3. While the performance of the CPE cluster is faster than the single core of Intel(R) Xeon(R) CPU E5-2695 v3, the efficiency is not satisfying. We find several reasons for the efficiency limit.

TABLE 2: SW26010 Processor vs. Intel CPU E5-2695 v3

Test case	PCG code	Runtime(s)	Speed-up
PCG icoFOAM Matrix size (400*400*1)	the MPE	1044	1
	the CPE cluster	130	8.03
	Intel(R)Xeon(R) CPU E5-2695 v3 @ 2.30GHz	154	6.80

First, we can see the size of cache and SPM of the SW26010 processor is much smaller than the Intel CPU. So we need to load data repeatedly into SPM. It limits the performance of memory access.

Second, the latency of DMA is relatively high compared with the memory access on the Intel CPU. Due to the low arithmetic intensity of OpenFOAM, the performance of the program is limited by memory access.

Third, the automatic optimizations of the SW26010 processor applied by the compiler is less efficient than the Intel CPU. The ICC compiler contains some highly optimized math library.

6. Conclusions

In this paper we propose a hybrid implementation method for OpenFOAM and optimize the hotspot according to the architecture of the SW26010 processor. We discuss the optimizations performed on the MPE and the CPE cluster accordingly. Then we share our experiences throughout the implementation and optimization strategies.

We run tests with single CG of the SW26010 processor and achieve significant performance improvement, i.e. 2.34x speedup on the MPE compared with the best performance tuned by compiler options. And the single-CG code runs 8.03x faster than the optimized implementation on the MPE. We also compare the performance of OpenFOAM between the SW26010 processor and Intel E5-2695 v3 CPU, and the performance of the CPE cluster is better than that on a single core of Intel(R) Xeon(R) CPU E5-2695 v3. In addition, PCG, the hotspot we optimize, is a general method to solve sparse linear equations. Hence, our sparse linear solver is not only limited to one single application/test case; but also can be applied to others.

Furthermore, the implementation and results we present demonstrate how complex codes and algorithms can be efficiently implemented on such diverse architectures as hybrid MPE-CPEs systems. We can hide hardware-specific programming models into libraries and make them general purpose.

OpenFOAM is now ready to effectively exploit the new supercomputing system based on the SW26010 processor. We intend to continue maintaining and developing the code and updating to exploit new hardware features. And we will keep developing the MPI module to utilize the computing power of multi CGs.

7. Acknowledgements

This research is supported by China National High-Tech R&D Plan (863 plan) 2014AA01A302 and National Research Center Of Parallel Computing Engineering & Technology. Thanks Yichao Wang of SJTU for his sincere help. We also thank Xin Liu and Hong Qian of National Research Center Of Parallel Computing Engineering & Technology for their valuable suggestions. James also greatly acknowledges support from Japan JSPS RONPAKU (Thesis PhD) Program.

References

[1] H. Meuer, "Top500 supercomputer sites," June 2016. [Online]. Available: https://www.top500.org/lists/2016/06/

[2] J. D. Anderson and J. Wendt, *Computational fluid dynamics*. Springer, 1995, vol. 206.

[3] A. A. AlOnazi, "Design and optimization of openfoam-based cfd applications for modern hybrid and heterogeneous hpc platforms," Ph.D. dissertation, 2014.

[4] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, "A tensorial approach to computational continuum mechanics using object-oriented techniques," *Computers in physics*, vol. 12, no. 6, pp. 620–631, 1998.

[5] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao *et al.*, "The sunway taihulight supercomputer: system and applications," *Science China Information Sciences*, vol. 59, no. 7, p. 072001, 2016.

[6] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "Qr factorization on a multicore node enhanced with multiple gpu accelerators," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 932–943.

[7] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, "A scalable high performant cholesky factorization for multicore with gpu accelerators lapack working note# 223."

[8] L. Buatois, G. Caumon, and B. Levy, "Concurrent number cruncher: a gpu implementation of a general sparse linear solver," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 24, no. 3, pp. 205–223, 2009.

[9] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," in *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3. ACM, 2003, pp. 917–924.

[10] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser, "A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2010, pp. 583–592.

[11] D. Combest and J. Day, "Cufflink: a library for linking numerical methods based on cuda c/c++ with openfoam," *URL: http://cufflink-library. googlecode. com (: 02.12. 2012)*, 2011.

[12] A. Lastovetsky, B. Chetverushkin, N. Churbanova, and M. Trapeznikova, "Parallel simulation of oil extraction on heterogeneous networks of computers."

[13] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis, "A new era in scientific computing: Domain decomposition methods in hybrid cpu–gpu architectures," *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 13, pp. 1490–1508, 2011.

[14] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[15] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," 1994.

[16] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix–vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.