Towards highly efficient DGEMM on the emerging SW26010 many-core processor

Conference Paper · August 2017

DOI: 10.1109/ICPP.2017.51

9 authors, including:

Chao Yang
Peking University
77 PUBLICATIONS **647** CITATIONS
SEE PROFILE

Yulong Ao
Chinese Academy of Sciences
10 PUBLICATIONS **68** CITATIONS
SEE PROFILE

Peng Zhang
Chinese Academy of Sciences
6 PUBLICATIONS **22** CITATIONS
SEE PROFILE

Towards Highly Efficient DGEMM on the Emerging SW26010 Many-core Processor

Lijuan Jiang*†, Chao Yang*‡, Yulong Ao*†, Wanwang Yin§, Wenjing Ma*‡, Qiao Sun*, Fangfang Liu*†, Rongfen Lin§, Peng Zhang*†
*Institute of Software, Chinese Academy of Sciences, Beijing, China
†University of Chinese Academy of Sciences, Beijing, China
‡State Key Laboratory of Computer Science, Chinese Academy of Sciences, Beijing, China
§National Research Center of Parallel Computer Engineering and Technology, Beijing, China
Corresponding author: Chao Yang (yangchao@iscas.ac.cn).

***Abstract*—The matrix-matrix multiplication is an essential building block that can be found in various scientific and engineering applications. High-performance implementations of the matrix-matrix multiplication on state-of-the-art processors may be of great importance for both the vendors and the users. In this paper, we present a detailed methodology of implementing and optimizing the double-precision general format matrix-matrix multiplication (DGEMM) kernel on the emerging SW26010 processor, which is used to build the Sunway TaihuLight supercomputer. We propose a three-level blocking algorithm to orchestrate data on the memory hierarchy and expose parallelism on different hardware levels, and design a collective data sharing scheme by using the register communication mechanism to exchange data efficiently among different cores. On top of those, further optimizations are done based on a data-thread mapping method for efficient data distribution, a double buffering scheme for asynchronous DMA data transfer, and an instruction scheduling method for maximizing the pipeline usage. Experiment results show that the proposed DGEMM implementation can fully exploit the unique hardware features provided by SW26010 and can sustain up to 95% of the peak performance.**

***Keywords*-DGEMM, dense linear algebra, SW26010 processor, many-core architecture, Sunway TaihuLight**

I. Introduction

As a de facto programming interface, the Basic Linear Algebra Subprograms (BLAS) [1] contains a collection of basic and important vector and matrix operations that are often found in various linear algebra computations [2, 3]. Highly efficient implementations with hardware specific optimizations of BLAS for common single-core and multi-core processors are often provided by hardware vendors in commercial libraries such as the Intel MKL [4], AMD ACML [5], IBM ESSL [6], among others. Nowadays, heterogeneous many-core architectures have become an important trend to build extreme-scale supercomputers with substantially high computing throughput. Due to the complicated hardware features, it is often a challenging task to achieve high performance on such platforms. Designing high-performance BLAS implementations on state-of-the-art heterogeneous many-core processors such as the General Purpose Graphic Processing Units (GPGPUs), and the Intel Many Integrated Cores (MICs) has been receiving increasingly more attention from both the industry [7, 8] and the academy [9–12].

Among many standardized subroutines in BLAS, the matrix-matrix multiplication is a basic and important operation that often plays a critical role in many scientific and engineering applications [13–15]. Due to the compute-bound nature, matrix-matrix multiplications, under certain optimizations, are well suited for most of the current high-performance computers with hierarchical memory and parallel processing capability. These optimizations applied to matrix-matrix multiplications may serve as templates to guide performance optimizations or be used as building blocks in many application scenarios. For example, the DGEMM kernel in BLAS, which stands for the matrix-matrix multiplication in double precision with the general dense matrix format, is a performance-critical basis in the HPL package [2, 16, 17], which has been used as the standard to rank supercomputer in the TOP500 Lists [18] for over two decades. Therefore, it is of great importance to conduct thorough study on the implementation and optimization of these kernels. In particular, for heterogeneous many-core processors, a number of researches were done to optimize matrix-matrix multiplications on how to utilize data locality and hide memory access latency based on the underlying hardware [19–22].

Sunway TaihuLight, which took the top spot on the TOP500 List as of 2016 with a ground breaking floating-point peak performance of 125.4 Pflops/s and HPL performance of 93 Pflops/s in double precision, is a new generation supercomputer based on the Chinese home-grown Shen-Wei heterogeneous many-core processor, SW26010. The SW26010 processor has a number of unique hardware features, such as the introduction of the core group consisting of a single managing core and dozens of computing cores, the design of scratch pad memory on each computing core, the support of direct memory accessing (DMA) to transfer data between the main memory and the scratch pad memory, and the register communication mechanism to communicate among different computing cores. All these features may give rise to difficulties in the design of matrix-matrix multiplication kernels. In this paper, we take DGEMM as the ex-

ample to reveal the implementation and optimization details of matrix-matrix multiplications on the SW26010 processor of the Sunway Taihulight supercomputer, by penetrating into the micro-architecture of the heterogeneous many-core processor. In particular, we apply a three-level blocking algorithm to orchestrate data on the memory hierarchy and expose parallelism of DGEMM to fully utilize the highly parallel computing capability. Besides, architecture-aware methodologies are employed to exploit data locality and the effective memory bandwidth. Test results show that the optimized DGEMM routines can achieve a double-precision performance of 706.1 Gflops/s, corresponding to a computing efficiency of 95%.

The rest of the paper is organized as follows. Section II provides an overview of the SW26010 many-core architecture with emphasis on some important features we leverage in our work. We then elaborate in detail the implementation of DGEMM on SW26010 in Section III, following which the optimization techniques we employ on SW26010 will be given in Section IV. In Section V experimental results are provided to examine and analyze the performance of the DGEMM on SW26010. Some related work is discussed in section VI and the paper is concluded in section VII.

II. SW26010 Many-core Architecture

In this section, we briefly describe the major hardware features of the SW26010 processor based on the new generation Shen Wei (SW) architecture [23–25]. A SW26010 processor is comprised of different types of processing elements organized as four core groups (CGs) via a network on chip (NOC), sharing a local memory space of 32 GB. Each CG mainly includes four components: a management processing element (MPE), 64 computing processing elements (CPEs), a protocol processing unit (PPU), and a memory controller (MC), as illustrated in Figure 1.

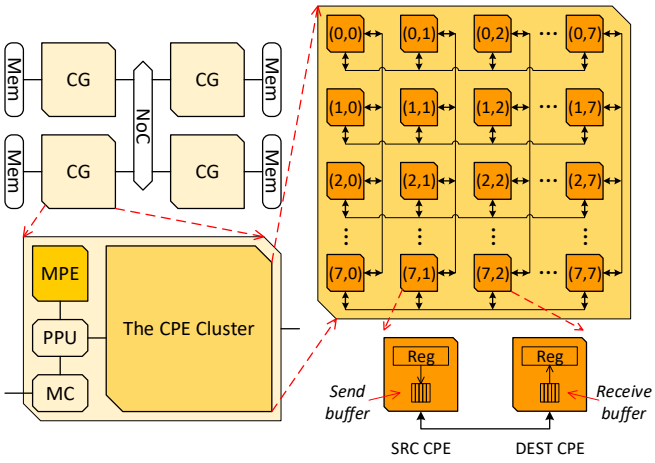

Figure 1. Architecture of the SW26010 many-core processor.

The 64 CPEs from each CG are organized as a CPE cluster on an 8×8 mesh network. Each CPE is configured with 32 256-bit registers. By default, a process runs on the MPE of a CG at the run time. The MPE process can start 64 threads, each of which runs on a CPE. Each thread has a row label and column label, corresponding to the row id and the column id of the CPE on the 8×8 mesh. For convenience, we use $\mathrm{thread}(i,j)$ $(0 \le i \le 7, 0 \le j \le 7)$ to represent the thread executing on the $\mathrm{CPE}(i,j)$ located at the i-th row and j-th column in the CPE cluster. The 64 threads work in the way of single-instruction multiple-thread (SIMT). They can communicate via a low-latency register communication mechanism, which runs in a producer/consumer mode, and includes two intercommunication operations: *row broadcast* and *column broadcast*. As shown in Figure 1, the register communication mechanism works as follows. The *SRC CPE* in the figure first loads a segment of 256-bit aligned data into the register, and then puts them into the CPE mesh network via the send buffer. Then the *DEST CPE* in the figure gets the broadcast data via the receive buffer from the CPE mesh network, and loads the data to its local register. Each CPE can exchange data with either one or more CPEs in the same row via *row broadcast*, or in the same column via *column broadcast*. The cost of the register communication is usually around several cycles.

The MPE and CPE are both designed based on a 64-bit RISC architecture, running at a clock rate of 1.45GHz. They both support out-of-order executions, 256-bit SIMD vectorizations, and fused multiply-add (FMA) instructions. An MPE has a dual pipeline executing 8 FP/cycle each. Every CPE has a floating point pipeline with the same clock rate, and another pipeline which can facilitate integer operations and register communication. Generally, the MPEs can perform management, communication and a small amount of computation tasks, while the CPEs are in charge of most of the computation tasks. The theoretical peak performance of a CPE cluster is 8flop/clock×1.45GHz×64=742.4Gflops/s.

Each MPE has a 32KB instruction cache and a 32KB L1 data cache, and 256KB L2 cache. Each CPE, on the other hand, has a 16KB instruction cache. The CPE does not have L2 or L3 cache; instead, each CPE is equipped with a 64KB scratch pad memory, also called the local device memory (LDM), which can be used as either a fast user-controlled cache or a software-emulated cache that achieves automatic data caching. The MPE and CPE cluster from the same CG share the unified 8GB main memory through the memory controller. To achieve high performance, a CPE can start an asynchronous direct memory access (DMA) instruction to transfer data between main memory and LDM efficiently. Through DMA, a CPE can access data from the main memory in successive or strided format, with a transaction unit of 128B. There are five DMA modes available, including `PE_MODE`, `BCAST_MODE`, `ROW_MODE`, `BROW_MODE`, and `RANK_MODE`, corresponding to different data distribution patterns. In particular, `PE_MODE` and `ROW_MODE` are used

in our work. The PE_MODE transfers data between the main memory and the LDM of a single CPE, while the ROW_MODE transfers data between main memory and the LDM of all the eight CPEs located in the same row of the CPE cluster. All DMA operations require the 128B data alignment. The ROW_MODE requires a synchronization of the 8 CPEs of the same row. In each ROW_MODE DMA transaction, 128B data are transferred to/from CPEs in a row, while each CPE gets/puts a successive 16B data.

III. Implementation of DGEMM on SW26010

The DGEMM is a standard level 3 BLAS operation that computes

$$C = \alpha AB + \beta C,$$

where α and β are two real numbers in double precision, and A, B, and C are $m \times k$, $k \times n$, and $m \times n$ general matrices with double precision entries, respectively. Here A and B can be in either transposed or non-transposed formats [1]. In this paper, we implement the case where A and B are non-transposed and the dimensions of matrices are the multiply of block factors. Besides, all matrices in our implementation are stored in a column-major format. With careful design of data layout through proper data distribution patterns of different DMA modes and data sharing scheme via register communication, we will elaborate the three-level layered blocking algorithm to take full advantage of the powerful cooperative computing capability provided by the CPE cluster.

A. Three-level Layered Blocking

Blocking algorithms [20–22, 26, 27] can effectively mitigate differences of bandwidth and memory access latencies in the memory hierarchy, and utilize data reuse on both multi-core and many-core architectures. To implement DGEMM on a CG of the SW26010 processor, we apply a three-level layered approach with data blocking based on the memory hierarchy, which can be abstracted to three levels: the main memory (off-chip memory), the LDM and the register files. Accordingly, we design a layered approach including three levels of matrix blocking: a CG-level blocking, a thread-level blocking and a register-level blocking. This multi-level blocking approach serves as a framework for us to not only effectively distribute and orchestrate data on different levels of memory hierarchy, but also effectively expose inter and intra-block parallelisms with enhanced data locality. Figure 2 illustrates how the three-level blocking strategy works.

The CG-level blocking process is illustrated in the first row of Figure 2. Here, matrices C, A and B are logically partitioned into CG-level blocks of $b_M \times b_N$, $b_M \times b_K$, and $b_K \times b_N$. These blocks are laid out as grids of $M \times N$, $M \times K$ and $K \times N$, where $M = m/b_M$, $N = n/b_N$ and $K = k/b_K$. And they are arranged to be updated by a straightforward algorithm in a N-K-M triple-nested loop, as is shown in Algorithm 1, where we use the notation $\delta X_{i,j}$ to represent the (i, j)-th CG-level block of matrix X. We take B as the reside matrix, whose blocks are held in the LDM until not needed.

Algorithm 1 Framework of DGEMM blocking algorithm.

```
1: for j ← 0 to N − 1 do
2:     for l ← 0 to K − 1 do
3:         load block δB_{l,j} to the CPE cluster
4:         for i ← 0 to M − 1 do
5:             load block δA_{i,l} to the CPE cluster
6:             load block δC_{i,j} to the CPE cluster
7:             parallel block matrix multiplication:
8:                 δC_{i,j}+=α(δA_{i,l})(δB_{l,j})
9:             store block δC_{i,j} to main memory
10:        end for
11:    end for
12: end for
```

In each loop iteration, a CG-level block $\delta C_{i,j} (0 \le i \le M-1, 0 \le j \le N-1)$ and the corresponding $\delta A_{i,l} (0 \le l \le K-1)$ and $\delta B_{l,j}$ are loaded to the LDM of each CPE. To fit the CG-level blocks in LDM, CG-level blocks need to be further divided to thread-level blocks, and map to the LDM of 64 CPEs. The tiled CG-level blocks of matrices C, A and B are laid out as an 8×8 grid of thread-level blocks, whose dimensions are $p_M \times p_N$, $p_M \times p_K$ and $p_K \times p_N$, respectively. For the sake of load balance, the CG-level blocks are evenly divided with $p_M = b_M/8$, $p_K = b_K/8$ and $p_N = b_N/8$. The notation $\epsilon X_{u,v} (0 \le u \le 7, 0 \le v \le 7)$ are used to represent thread-level blocks of matrix X for convenience. Thread-level blocks $\epsilon C_{u,v}$, $\epsilon A_{u,v}$ and $\epsilon B_{u,v}$ are mapped to $\mathrm{thread}(u, v)$, with each thread loading its mapped local thread-level blocks of C, A and B to the LDM of the corresponding CPE. Matrix elements are transferred through DMA in PE_MODE between the LDM of CPEs and main memory.

In our implementation, $\delta C_{i,j}$ is updated by taking 8 multiplications of the column strip of $\delta A_{i,l}$ and the row strip of $\delta B_{l,j}$. Each strip is comprised of 8 thread-level blocks. Therefore, column strip and row strip multiplication can be divided to 64 thread-level tasks for block matrix multiplication, and can be assigned to the 64 threads for concurrent execution. Each thread executes the task whose results could be used to update its local thread-level C block. In each of the 8 steps, threads acquire the matrix elements of A and B for computation either from its own LDM, or from other threads by register communication, and compute thread-level block matrix multiplication concurrently, as illustrated in the second row of Figure 2. And the computation results of each step are accumulated to their local $\epsilon C_{u,v}$. We will detail the data exchange scheme via register communication in section III-B. Here, we take the first two compute steps by $\mathrm{thread}(2, 2)$ as an example. In the first step, $\mathrm{thread}(2, 2)$ gets $\epsilon A_{2,0}$ from $\mathrm{thread}(2, 0)$ and $\epsilon B_{0,2}$ from $\mathrm{thread}(0, 2)$, and computes the multiplication of $\epsilon A_{2,0}$ and $\epsilon B_{0,2}$. In the

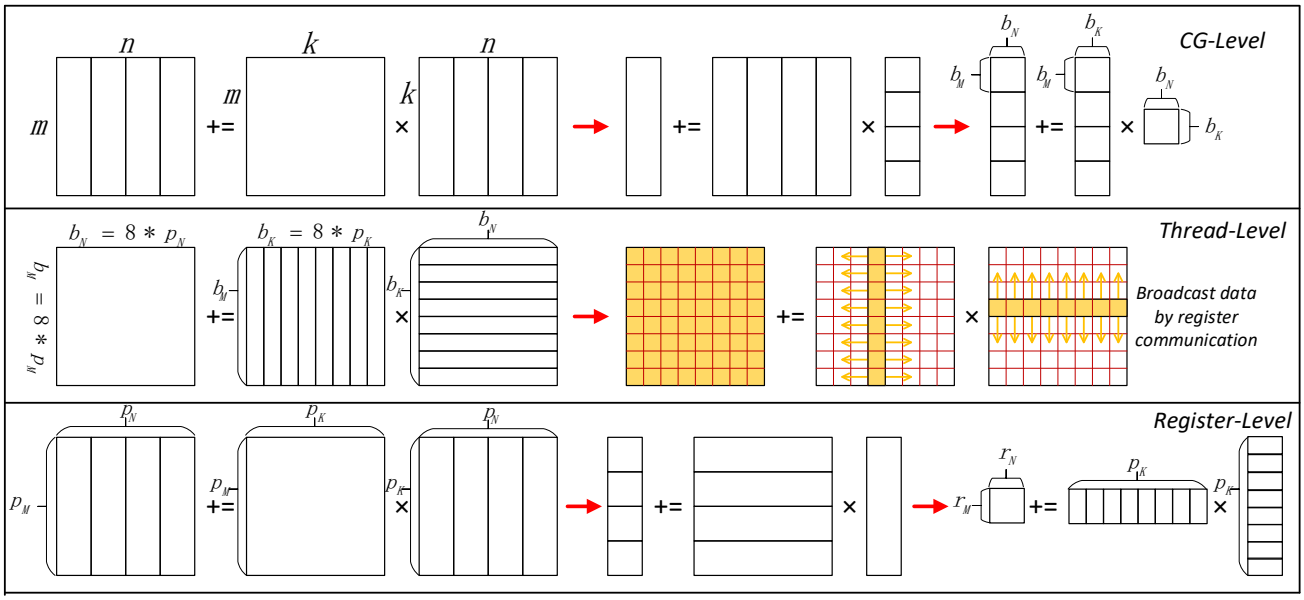

Figure 2. Three-level layered blocking. The first row illustrates the CG-level blocking process. The second row depicts the update of a CG-level block of C, which is transformed to concurrent update of 64 thread-level blocks by strip multiplications. The third row shows the register-level blocking strategy of the multiplication done on each thread-level block.

second step, $\mathrm{thread}(2,2)$ gets $\epsilon A_{2,1}$ from $\mathrm{thread}(2,1)$ and $\epsilon B_{1,2}$ from $\mathrm{thread}(1,2)$, and computes the multiplication of $\epsilon A_{2,1}$ and $\epsilon B_{1,2}$. The results of both steps are accumulated to its local thread-level block $\epsilon C_{2,2}$.

The third row of Figure 2 depicts the process of register-level blocking for the thread-level multiplication. The register blocking is introduced to alleviate memory access latency differences between registers and LDM, while utilizing data reuse. In the innermost loop, thread-level blocks of A and B are partitioned to panels of $r_M \times p_K$ and $p_K \times r_N$, respectively, where r_M and r_N are determined by the registers assigned for C, A and B. Each time, a column of the A panel and a row of the B panel are loaded to the registers for computation. Accordingly, r_M and r_N registers are used to fetch matrix elements of A and B, respectively, and $r_M \times r_N$ registers to store accumulated matrix elements of C.

B. Collective Data Sharing

In the three-level blocking algorithm, the data mapping scheme is designed for maximizing on-chip data reuse. However, with this data mapping, each CPE may need to acquire thread-level blocks from other CPEs, and also send blocks to other CPEs in the computation. Therefore, in order to efficiently exchange data among different CPEs, we design a collective data sharing scheme based on the register communication.

In each step of the strip multiplication, the corresponding threads can be categorized into four types: threads owning valid A and B, threads only owning valid A, threads only owning valid B, and threads owning no valid matrix elements. We use *valid* to denote matrix elements that are needed by the current step. In our implementation, a CPE thread checks which type it belongs to by its coordinates in the mesh and the step number, and takes the corresponding action, which is described in the following paragraph.

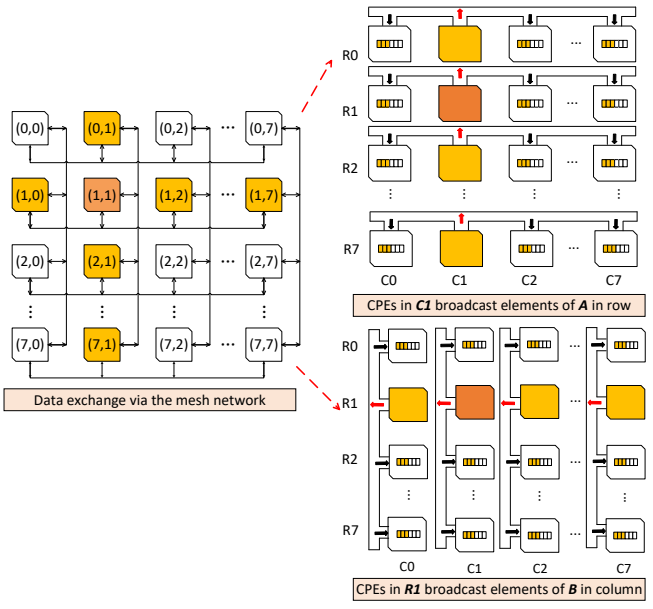

Figure 3. Collective data sharing scheme. Threads exchange matrix elements of A and B via register communication through the mesh network. The illustration gives an instance of the behaviors of each CPE in the second step of the strip multiplication.

Figure 3 shows one step in the data exchange procedure. When performing the i-th $(0 \leq i \leq 7)$ step of the eight strip multiplication, CPEs in the i-th column of the CPE cluster have the corresponding matrix elements of A, and CPEs in the i-th row of the CPE cluster have those of B. To distribute the data required by each CPE, $\mathrm{thread}(i,i)$ broadcasts its local elements of A to the CPEs in the same row, and broadcasts its local elements of B to the CPEs in the same column, as shown in Figure 3. It does not receive any data. Threads on the i-th column except $\mathrm{thread}(i,i)$ broadcast their local elements of A to the CPEs in the same row, and receive elements of B from $\mathrm{thread}(i,i)$. Threads on the i-th row except $\mathrm{thread}(i,i)$ broadcast their local elements of B to the CPEs in the same column and receive elements of A from $\mathrm{thread}(i,i)$. Threads in the other rows and other columns just receive elements of A and elements of B from the corresponding threads. Broadcast and receive operations are accomplished with the register communication mechanism.

In the collective data sharing scheme, four major register communication instructions are utilized, namely `vldr`, `lddec`, `getr`, and `getc`. Among them, `vldr` loads 256-bit aligned data before broadcasting it to the CPEs in the same row, `lddec` loads the same 64-bit data to all four 64-bit locations of the register simultaneously before broadcasting to the CPEs in the column, `getr`/`getc` receives data from the row/column mesh network before storing it to local registers. As illustrated in Figure 3, threads with valid A load four elements of A from its local LDM into one 256-bit register and broadcast them to threads in the same row, while other threads use $getr$ to receive the elements of A. Threads with valid B load one element of B, extend it to the 256-bit vector registers and broadcast to the threads in the same column, while other threads use $getc$ to receive the elements of B.

C. Block Size Determination

Changing the blocking size may have significant influence on the effectiveness of the memory bandwidth usage. Optimal choices of the block size can ensure more memory access amortized over computation between memory hierarchy [19, 21, 22, 27]. When choosing the block size for a blocking algorithm, memory access features of the machine as well as the capacity of the storage medium should be taken into consideration.

1) CG-level blocking: According to Algorithm 1, matrix C needs to be fetched and written K times, matrix A needs to be fetched N times, and matrix B needs to be fetched only once. Therefore, the total number of matrix elements transferred between the main memory and LDM of the CPE cluster is

$$2Kmn + Nmk + kn = mnk(2/b_K + 1/b_N) + kn.$$

The bandwidth reduction ratio is

$$S = \frac{2}{2/b_K + 1/b_N + 1/m} \approx \frac{2}{2/b_K + 1/b_N},$$

assuming that m is large enough as compared with block sizes. To maximize the performance, the required memory bandwidth B_r of DGEMM should be less than the theoretical bandwidth B_t:

$$B_r = FW\frac{1}{S} < B_t,$$

where F is the peak floating-point operations per second, W is the required word size (in byte) per floating-point operation. When $b_K = 2b_N$, the bandwidth reduction reaches its maximal mathematical value under the constraint of memory capacity. Therefore, we have the formula

$$b_N > \frac{FW}{B_t}.$$

For a CPE cluster on a single CG of the SW26010 processor, we have $F = 742.4$ Gflops/s, $W = 8$ bytes/flop (double precision), and $B_t = 34$ GB/s (by the DMA channel). To sustain high performance, $b_K \geq 350$ and $b_N \geq 175$ should be satisfied. And the larger b_K and b_N are, the better bandwidth reduction can be achieved.

2) Thread-level blocking: Under the limitation of LDM capacity, the number of matrix elements stored on each CPE should be less than 64KB/8B $= 8192$, which means

$$p_M p_N + p_N p_K + p_K p_M < 8192.$$

As mentioned earlier, $b_K = 8p_K$, $b_M = 8p_M$ and $b_N = 8p_N$. According to the CG-level blocking strategy, we have $p_K \geq 44$, $p_N \geq 22$ and increasing p_K and p_N will lead to better performance. Since the transaction size of DMA operations is 128B, p_K should be a multiple of 16. Therefore, we set $p_K = 96$, $p_N = 48$, $p_M = 16$.

3) Register-level blocking: We use r_M and r_N registers to fetch the matrix elements of A and B, respectively, and we use $r_M r_N$ registers to store the accumulated C results. Each CPE has 32 vector registers. Hence it is required that

$$r_M r_N + r_M + r_N < 32.$$

The bandwidth reduction between LDM and register is

$$\frac{2r_M r_N p_K}{r_M p_K + r_N p_K + 2r_M r_N} \approx \frac{2}{1/r_M + 1/r_N},$$

assuming that p_K is large enough as compared to r_M and r_N. Obviously, larger r_M and r_N ensures better bandwidth utilization. The maximum bandwidth reduction is achieved when $r_M = r_N$. Therefore, we assign the values of r_M and r_N to be $r_M = r_N = 4$.

IV. Optimization of DGEMM on SW26010

The three-level layered blocking algorithm serves as a good algorithmic framework for implementing DGEMM on the SW26010 processor. However, using this framework alone does not lead to satisfactory performance. A quick profiling shows that only less than 1/3 of the peak performance can be achieved without conducting further optimizations. The major obstacle to achieving high performance is still the cost of data movement. To further improve the performance, we employ three important optimization techniques, namely efficient data-thread mapping, double buffering, and instruction scheduling.

A. Data-thread Mapping

As discussed in section III-A, we simply use `PE_MODE` to distribute matrix blocks to the corresponding threads. However, different bandwidth efficiency could be achieved in different DMA modes. It is found that with careful design, `ROW_MODE` is also fit for data distribution of DGEMM in spite of its complicated data distribution strategy, which provides higher bandwidth. In order to compare the bandwidth achieved in `PE_MODE` and `ROW_MODE`, we designed and implemented a micro-benchmark. The micro-benchmark here only simulates the transfer of matrices in different sizes with an access pattern similar to our algorithm. Matrices with the size of mk are partitioned to be CG-level blocks with the size of $b_M b_K$. These CG-level blocks are loaded sequentially from the main memory to the LDM of the 64 CPEs in `ROW_MODE` and `PE_MODE`, respectively. And each of these blocks is transferred through DMA in the two modes with a thread-level blocking of $p_M p_K$, so as to be loaded to the LDM of each CPE. In particular, we set $b_M = 128$, $b_K = 768$, $p_M = 16$, and $p_K = 96$. As depicted in Figure 4, we find the sustained memory bandwidth of DMA in `ROW_MODE` is remarkably superior to `PE_MODE`. Therefore, based on further analysis on the data distribution strategy of both modes, we choose to transfer matrices in an efficient mixed mode, in which we use `ROW_MODE` for A and C, but `PE_MODE` for B (`ROW_MODE` is not applicable to B). This is because the loading of B does not fit the data transfer pattern of `ROW_MODE`.

Since the data distribution in different DMA modes could be different, the data-thread mapping of the mixed mode is different from the instinctive mapping strategy discussed in section III-A. Figure 5 gives abbreviative instances of the data distributions from the main memory to the LDM of the CPE cluster in both `ROW_MODE` and `PE_MODE`. Here we use *data unit* to represent a small block in the matrix to be loaded by a thread for convenience. A column strip of CG-level blocks consists of 64 data units. Particularly, data units may contain different numbers of matrix elements with respect to different matrices. Because `ROW_MODE` transfers data between the main memory and the LDM of CPEs in a row, and matrices are stored in column-major, column

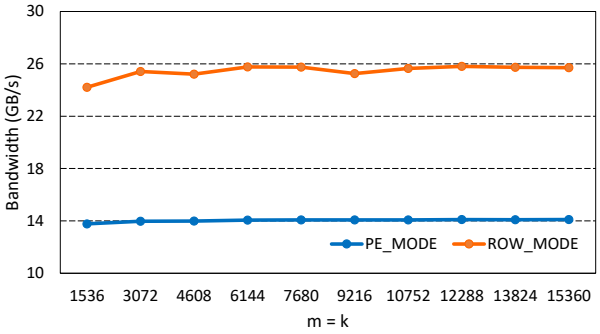

Figure 4. Sustained memory bandwidth of two DMA modes.

strips of CG-level blocks are mapped to the corresponding CPEs in a row. For matrices A and C in `ROW_MODE`, each CPE gets thread-level blocks that are comprised of 8 (with $b_M = 128, p_M = 16$) interleaved data units in the corresponding column strip. Specifically, the i-th $(0 \le i \le 7)$ row CPEs fetch the i-th column data units, which means $\mathrm{CPE}(j,i)(0 \le j \le 7)$ fetches data units (j,i), $(j+8,i)$, $(j+16,i)$, $\cdots$, $(j+56,i)$. As for A, each data unit contains $2p_K$ matrix elements, while for C, each contains $2p_N$ matrix elements. For B, though it is still transferred in `PE_MODE`, the mapping is different from section III-A for the sake of accuracy, since the mapping of A and C is changed. Now, column strips of the CG-level B blocks are mapped to CPEs in a row, with each CPE getting a thread-level block consisting of 8 continuous data units. Specifically, the i-th $(0 \le i \le 7)$ row CPEs fetch the i-th column data units, which means $\mathrm{CPE}(j,i)(0 \le j \le 7)$ fetches data units $(8j,i)$, $(8j+1,i)$, $\cdots$, $(8j+7,i)$. As a result, each data unit of matrix B contains $\frac{p_K}{8} p_N$ elements.

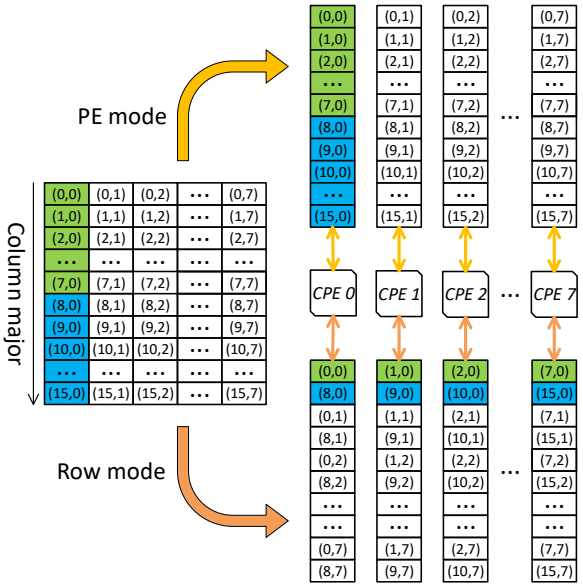

Figure 5. Data distribution of DMA in `ROW_MODE` and `PE_MODE`.

One thing to notice is that although the data-thread mapping strategy is different from the method used in the three-level blocking algorithm, there is no need to change the three-level blocking algorithm. We only need to adjust the register communication operations so that the matrix elements of A is broadcast among CPEs in the same column and the matrix elements of B is broadcast among CPEs in the same row, because we map each column strip to CPEs in a row.

B. Double Buffering

Although the DMA approach ensures fast data transmission between the main memory and the LDM, the transfer time of the CG-level blocks still cannot be ignored. To further hide the cost of the main memory access latency, we employ a double buffering method, which has been shown to be effective on GPU and other many-core accelerators [22].

Algorithm 2 DGEMM with double-buffering.

```
1: for j ← 0 to N − 1 do
2:     for l ← 0 to K − 1 do
3:         load block δB_{l,j} to CPEs
4:         load block δA_{0,l} to CPEs
5:         load block δC_{0,j} to CPEs
6:         sync
7:         load block δA_{1,l} to CPEs
8:         load block δC_{1,j} to CPEs
9:         parallel block matrix multiplication:
10:            δC_{0,j}+=α(δA_{0,l})(δB_{l,j})
11:        sync
12:        for i ← 2 to M − 1 do
13:            store block δC_{i−2,j} to main memory
14:            load block δA_{i,l} to CPEs
15:            load block δC_{i,j} to CPEs
16:            parallel block matrix multiplication:
17:                δC_{i−1,j}+=α(δA_{i−1,l})(δB_{l,j})
18:            sync
19:        end for
20:        store block δC_{M−2,j} to main memory
21:        parallel block matrix multiplication:
22:            δC_{M−1,j}+=α(δA_{M−1,l})(δB_{l,j})
23:        store block δC_{M−1,j} to main memory
24:    end for
25: end for
```

Algorithm 2 illustrates the refined DGEMM framework with double buffering. Memory access operations of prefetching thread-level C and A blocks and storing the updated thread-level C blocks can proceed in parallel with computation. To enable double buffering, extra memory space needs to be allocated in the LDM. Therefore the blocking sizes also need to be changed. Because if we still use $p_M = 16$, $p_K = 96$ and $p_N = 48$ as before, it would exceed the capacity of the LDM. When choosing the blocking sizes, the following rules should be followed.

- The space allocated cannot exceed the capacity of LDM.
- Since the transaction of DMA operations is 128B, and register-level blocking is 4×4, p_K should be a multiple of 16, and p_N should be a multiple of 4, as discussed earlier.

After testing on different configurations of p_M, p_N and p_K, we find that a superior performance can be achieved when $p_M = 16$, $p_N = 32$ and $p_K = 96$.

C. Instruction Scheduling

Due to the compute-bound nature of DGEMM, manually designing highly optimized assembly kernels is an important technique [22, 26]. To maximize the performance of DGEMM on SW26010, it is necessary to conduct instruction level optimization to take advantage of the dual instruction pipeline. In the SW26010 instruction set, there is an extra pipeline for register communication and integer operations. Therefore, double precision floating point instructions can be issued simultaneously with register communication instructions, and both double-precision floating-point instructions and register communication instructions can be issued simultaneously with integer operation instructions. With carefully arranged instructions in the assembly kernel, we successfully designed a scenario in which the cost of the LDM memory access is almost completely hidden.

Algorithm 3 Instruction scheduling for a thread-level block matrix multiplication in the inner most loop of DGEMM.

```
1: vmad rA[0], rB[0], rC[0], rC[0]; regA rA[3], ldmA[3][0];
2: vmad rA[0], rB[1], rC[1], rC[1]; regB rB[3], ldmB[0][3];
3: vmad rA[1], rB[0], rC[4], rC[4]; addl ldmA, PM, ldmA;
4: vmad rA[1], rB[1], rC[5], rC[5]; addl ldmB, 2, ldmB;
5: vmad rA[0], rB[2], rC[2], rC[2]; nop;
6: vmad rA[2], rB[0], rC[8], rC[8]; nop;
7: vmad rA[0], rB[3], rC[3], rC[3]; regA rA[0], ldmA[0][0];
8: vmad rA[3], rB[0], rC[12], rC[12]; nop;
9: vmad rA[1], rB[2], rC[6], rC[6]; regB rB[0], ldmB[0][0];
10: vmad rA[1], rB[3], rC[7], rC[7]; regA rA[1], ldmA[1][0];
11: vmad rA[2], rB[1], rC[9], rC[9]; nop;
12: vmad rA[3], rB[1], rC[13], rC[13]; regB rB[1], ldmB[0][1];
13: vmad rA[2], rB[2], rC[10], rC[10]; nop;
14: vmad rA[2], rB[3], rC[11], rC[11]; regA rA[2], ldmA[2][0];
15: vmad rA[3], rB[2], rC[14], rC[14]; regB rB[2], ldmB[0][2];
16: vmad rA[3], rB[3], rC[15], rC[15];
```

Since register communication instructions executed by different threads could be different, we use *regA* and *regB* to represent the communication instructions of exchanging matrix elements of A and B, respectively, and use *vmad* to represent the fused multiply-add vectorization. The RAW instruction latencies of *regA/regB* and *vmad* are 4 and 6 cycles, respectively. The pseudo-code of the innermost loop with the instruction scheduling strategy is depicted in Algorithm 3. In our instruction scheduling technique, *vmad* can be issued together with another instruction, which could be *regA/regB* or an integer operation instruction, such as the instruction pairs in Line 2 and Line 4. The two instructions

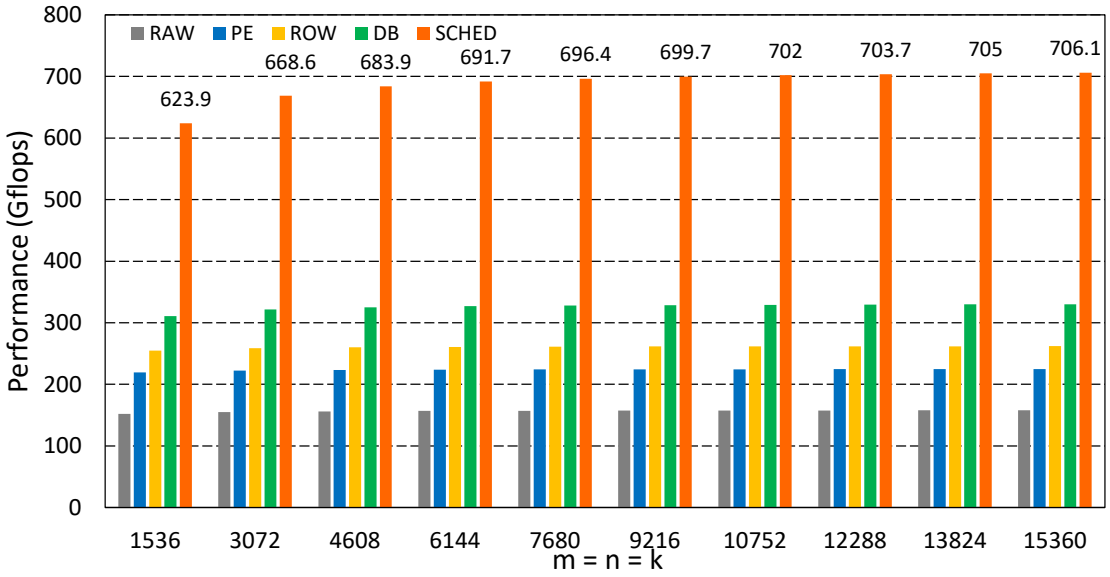

Figure 6. Performance comparison of different optimization methods.

in a pair will be executed in parallel. However, since DGEMM is compute-intensive, memory access instructions are lacking. And integer operation instructions could be issued with *vmad* in parallel ahead of schedule because of out-of-order instruction executions, which would disorder the whole software pipeline. Hence, some *nop* instructions are inserted, as shown in Line 6, to ensure instructions are issued in the right order. Profiling on this optimized version shows that the whole loop takes $101,858$ cycles in total, and *vmad* takes 97% of the cycles. It implies most memory access latency can be hidden in this way.

V. Experiment Results

To study how the performance is influenced by different optimization methods, we conduct the tests on five different implementations of the DGEMM algorithm, explained as follows.

- *RAW*: a straightforward implementation based on a simple N-M-K variant of the triple-nested loop, where C is partitioned to thread-level blocks and evenly assigned to the 64 threads to update, and matrix elements of A and B are fetched through DMA in `PE_MODE`.
- *PE*: an optimized implementation based on the three-level blocking algorithm and the data sharing scheme.
- *ROW*: on top of the *PE* version, with DMA `ROW_MODE` instead of `PE_MODE` applied to matrices A and C.
- *DB*: on top of the *ROW* version with double buffering.
- *SCHED*: on top of the *DB* version with instruction scheduling.

Figure 6 shows the performance of the five versions. In the test we fixed the study to the case when all matrices are square. It is seen from the figure that as the matrix size becomes larger the performance of all five DGEMM implementations increases monotonically until the maximum performance reaches when the matrix size is around 9216. With data sharing scheme utilizing register communication, the sustained performance of the *PE* version is 42.3% better than the RAW implementation. Using the `ROW_MODE` for data transfer, the *ROW* version achieved another 16.6% performance improvement. The *DB* code brings an extra 26% performance improvement over the *ROW* version by hiding the main memory access latency. Performance improvement of the *SCHED* code is prominent, with a 113.9% increase over *DB*, indicating that memory access latency of LDM is hidden in a nearly perfect way. The reason the instruction scheduling brings such prominent performance improvement lies in that LDM access pattern happens during the whole block matrix multiplication computation process, and appears to be the bottleneck. The maximum performance achieved by our optimized DGEMM implementation is 706.1Gflops/s, corresponding to 95% of the peak performance.

We then study the performance of the DGEMM when the matrices are not square. Figure 7 presents the sustained performance of the optimal DGEMM routine for different matrix shapes. The performance for matrices with small m is relatively low. This is because when applying the double buffering strategy in Algorithm 2, a block of A and C are prefetched before the main M-loop, causing an extra cost of data loading. When m is larger, the overhead of prefetching can be better amortized. On the other hand, the sizes of n and k have negligible influence on the sustained performance.

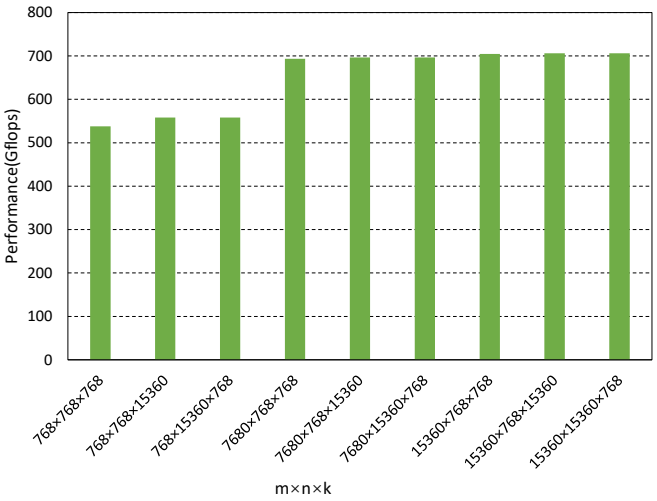

Figure 7. Performance comparison of different matrix shapes.

VI. Related work

Extensive researches have been done on implementing and optimizing GEMM on multi-core CPUs. For example, effective blocking algorithms on high performance systems with memory hierarchy were studied by Whaley et al. [28], Gunnels et al. [26] and Goto et al. [27]. And the ATLAS package based on the first work employed the auto-tuning optimization for GEMM on general-purpose CPUs [28]. Wang et al. [29] proposed AUGEM, a template-based optimization framework that can automatically generate fully optimized assembly codes of GEMM and several other dense linear algebra kernels. Although the architecture of multi-core CPUs is different from the many-core SW26010 processor, the algorithms studied in those work and the blocking methods they applied are of interest to us in our research.

In recent years, increasingly more important efforts have been made to optimize GEMM on many-core coprocessor such as GPUs and MICs, which have competitive computing throughput as SW26010. Volkov et al. [19] designed a blocking algorithm for GEMM to exploit data locality of the shared memory on a GPU and implemented micro-benchmarks to analyze the architecture features. Li et al. [30] implemented a GEMM auto-tuner that can automatically generate parametrized code variants and search for the optimal ones heuristically on GPUs. Nath et al. [20] proposed a register blocking method to mitigate the bandwidth differences between register files and shared memory. Nakasato [21] discussed the data layout and proposed the usage of texture cache on the Cypress GPU. Tan et al. [22] employed shared memory blocking and register blocking to exploit data locality of the memory hierarchy, and set up a performance model for the memory bandwidth to help choose optimal blocking sizes. However, the register blocking, double buffering and instruction scheduling we applied here are not the same because of the difference of the underlying architecture between SW26010 and GPU. Heinecke et al. [16] optimized DGEMM for hybrid CPU+MIC architecture, with jobs assigned to threads for work stealing. Gepner et al. [31] maximized reuse of the on-die cache for DGEMM optimization on MIC.

VII. Conclusion

In this paper we presented a detailed methodology to implement and optimize DGEMM on the SW26010 many-core processor. We proposed a three-level blocking algorithm with a collective data sharing scheme to expose parallelism and reduce data movement cost, and applied various optimizations such as thread-data mapping, double buffering, and instruction scheduling to further improve the performance. Experiments are done to show that the designed DGEMM kernel is highly efficient on SW26010. This work can be seen as an illustration on how to fully unleash the performance potential of the emerging SW many-core processor with careful data blocking and instruction scheduling. However, we would like to point out that writing assembly code by hand hinders productivity. In the future, we plan to generalize the methodology studied here and apply automatic code generation and automatic performance tuning so that the work can be smoothly extended to other dense matrix kernels and to future SW series processors.

Acknowledgment

We would like to thank Mr. Hao Liu, Dr. Qian Wang and Dr. Xianyi Zhang for their contributions at the early stage of this work. This work was supported in part by National Natural Science Foundation of China (91530323), Special Project on High-Performance Computing under the National Key R&D Program (2016YFB0200603), and Key Research Program of Frontier Sciences from CAS (QYZDB-SSW-SYS006).

References

[1] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[2] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003.

[3] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, "Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines," *Scientific Programming*, vol. 5, no. 3, pp. 173–184, 1996.

[4] https://software.intel.com/en-us/intel-mkl.

[5] http://developer.amd.com/tools-and-sdks/archive/compute/amd-core-math-library-acml/acml-downloads-resources/.

[6] http://www-03.ibm.com/systems/power/software/essl/.

[7] H. M. Klie, H. H. Sudan, R. Li, Y. Saad *et al.*, "Exploiting capabilities of many core platforms in reservoir simulation," in

SPE Reservoir Simulation Symposium. Society of Petroleum Engineers, 2011.

[8] Y. Wu, W. Jia, L. W. Wang, W. Gao, L. Wang, and X. Chi, *GPU Tuning for First-Principle Electronic Structure Simulations*. Springer Berlin Heidelberg, 2013.

[9] J. Dongarra, M. Gates, A. Haidar, Y. Jia, K. Kabir, P. Luszczek, and S. Tomov, "Portable HPC programming on intel many-integrated-core hardware with MAGMA port to Xeon Phi," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2013, pp. 571–581.

[10] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–8.

[11] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[12] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng, "Automatic library generation for BLAS3 on GPUs," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 255–265.

[13] J. Demšar, T. Curk, A. Erjavec, Č. Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič *et al.*, "Orange: data mining toolbox in Python," *Journal of Machine Learning Research*, vol. 14, no. 1, pp. 2349–2353, 2013.

[14] D. E. King, "Dlib-ml: A machine learning toolkit," *Journal of Machine Learning Research*, vol. 10, no. Jul, pp. 1755–1758, 2009.

[15] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc," *See http://www. mcs. anl. gov/petsc*, 2001.

[16] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey, "Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel Xeon Phi coprocessor," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 126–137.

[17] M. Bach, M. Kretz, V. Lindenstruth, and D. Rohr, "Optimized HPL for AMD GPU and multi-core CPU usage," *Computer Science-Research and Development*, vol. 26, no. 3-4, p. 153, 2011.

[18] https://www.top500.org/lists/2016/11/.

[19] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *2008 SC-International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2008, pp. 1–11.

[20] R. Nath, S. Tomov, and J. Dongarra, "An improved MAGMA GEMM for Fermi GPUs," 2010.

[21] N. Nakasato, "A fast GEMM implementation on the Cypress GPU," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 50–55, 2011.

[22] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of DGEMM on Fermi GPU," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 35.

[23] 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.

[24] F. Zheng, H.-L. Li, H. Lv, F. Guo, X.-H. Xu, and X.-H. Xie, "Cooperative computing techniques for a deeply fused and heterogeneous many-core processor architecture," *Journal of Computer Science and Technology*, vol. 30, no. 1, pp. 145–162, 2015.

[25] J. Dongarra, "Sunway TaihuLight supercomputer makes its appearance," *National Science Review*, vol. 3, no. 3, pp. 265–266, 2016.

[26] J. A. Gunnels, G. M. Henry, and R. A. van de Geijn, "A family of high-performance matrix multiplication algorithms," in *International Conference on Computational Science*. Springer, 2001, pp. 51–60.

[27] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, p. 12, 2008.

[28] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 1998, pp. 1–27.

[29] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 25.

[30] Y. Li, J. Dongarra, and S. Tomov, "A note on auto-tuning GEMM for GPUs," in *International Conference on Computational Science*. Springer, 2009, pp. 884–892.

[31] P. Gepner, V. Gamayunov, D. L. Fraser, E. Houdard, L. Sauge, D. Declat, and M. Dubois, "Evaluation of DGEMM implementation on Intel Xeon Phi coprocessor," *Journal of Computers*, vol. 9, no. 7, pp. 1566–1571, 2014.