IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 12, 2020, accepted August 19, 2020, date of publication August 25, 2020, date of current version September 9, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3019302

Runtime Adaptive Matrix Multiplication for the SW26010 Many-Core Processor

ZHENG WU, MINGFAN LI, MENGXIAN CHI, LE XU, AND HONG AN
School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China
Corresponding author: Zheng Wu (zhengwu@mail.ustc.edu.cn)

This work was supported by the National Key Research and Development Program of China under Grant 2018YFB0204102.

ABSTRACT The study of matrix multiplication on the emerging SW26010 processor is highly significant for many scientific and engineering applications. The state-of-the-art work from the swBLAS library, called *SWMM*, focuses mainly on the infrequent case involving special matrix dimensions and determines the execution action of matrix multiplication by one specified algorithm. To further adapt to various matrix shapes, in this article, we present a runtime adaptive matrix multiplication methodology, called *RTAMM*, which targets the features of the SW26010 architecture. The execution action of RTAMM is determined dynamically at runtime via several fundamental cost formulas and multiple sets of blocking factors, rather than determining the action at library generation time. With comprehensive trade-offs between the computation and data access, overall architecture-oriented optimization methods are introduced at three levels (macro, assistant, and micro) to fully exploit the computing capability of SW26010. The experiments show that RTAMM can achieve competitive peak performance compared with SWMM. Moreover, in tests on 6000 different matrix multiplication cases, RTAMM outperforms SWMM in 85.55% of the cases, and the improvements range from 5% to 308%, whereas RTAMM is slightly inferior to SWMM in only 1.28% of the cases. These results demonstrate that RTAMM has both great adaptability and considerable performance improvement.

INDEX TERMS Matrix multiplication, BLAS, dense linear algebra, many-core architecture, SW26010, Sunway TaihuLight.

I. INTRODUCTION

As an application program interface standard, BLAS (Basic Linear Algebra Subprograms) [1] contains many primary vector and matrix operations, which can be applied to different types of linear algebraic calculations [2]. As the core subprogram of BLAS, matrix multiplication is of great significance for many scientific and engineering applications [3]. The efficient implementation of BLAS [4]–[7] is constantly attracting the attention of researchers, and many-core processors have become a popular research platform.

The Sunway TaihuLight [8], which was developed by the National Research Center of China for Parallel Computer Engineering Technology, is the first supercomputer in the world with a peak performance exceeding 100 PFlops, and is composed mainly of 40k SW26010 heterogeneous many-core processors. The system can achieve 74% of the theoretical performance (93 PFlops) when running LINKPACK [9]. As the main contributor to the computational power of the Sunway TaihuLight, SW26010 has several special architectural features [10]–[12], such as an 8×8 CPE (computing processing element) cluster, software-controlled memory hierarchy, hardware-supported register communication, and CPE double-pipeline instruction execution, all of which have great potential for implementing matrix multiplication.

For matrix multiplication on SW26010, the state-of-the-art implementation derived from [13] in the swBLAS library, called *SWMM*, pursues the peak performance in the case where the matrix dimensions are sufficiently large and are the multiple of the optimal blocking factors. Although general matrix multiplication can straightforwardly rely on this special case at the expense of superfluous computation and data access overheads, the adaptability which receives more attention in the real world will be diminished. Another non-negligible consideration is that different matrix shapes have complicated characteristics, that is, various scales, ratios, and data alignments. Hence, if only one fixed execution action is relied upon, highly efficient implementation will not be feasible for different matrix multiplication cases.

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

To solve the above problems, in this article, we present a runtime adaptive matrix multiplication methodology called *RTAMM* for the architectural features of SW26010. For improved adaptability, the key novelty of this work is the coordination of several fundamental cost formulas and multiple sets of blocking factors, where each cost formula corresponds to one matrix multiplication algorithm. Moreover, referring to some parallel optimization technologies on SW26010 [11], [13], [14], we conduct more subtle research to balance the computation and data access for the high-performance implementation. The main contributions of our work can be summarized as follows:

- To cope with various matrix shapes, we propose RTAMM, a methodology for dynamically determining the most appropriate execution action during the operation of the program.
- Based on the possible loop orders of parameters (M, N, and K) and different parallel methods, we design several matrix multiplication algorithms for RTAMM. The execution cost of each algorithm is quantified using one fundamental cost formula to evaluate the execution efficiency. Moreover, we integrate many architecture-oriented optimization technologies, such as double buffering, register communication, and instruction scheduling, to ensure the highly efficient execution of RTAMM.
- An adaptive engine is generated to explore and estimate the potential execution actions of RTAMM. The engine consists of several fundamental cost formulas and a blocking factor pool that includes multiple sets of valuable blocking factors.
- The experiments comprehensively evaluate RTAMM and SWMM from two perspectives: (i) the peak performance and (ii) the adaptability. RTAMM achieves competitive peak performance in comparison with SWMM. In the adaptability comparison based on 6000 matrix multiplication cases with different configurations of M, N, and K, in 85.55% of the cases, RTAMM is superior to SWMM with improvements ranging from 5% to 308%. In contrast, SWMM is better than RTAMM in only 1.28% of the cases. Finally, we conduct an additional experiment to demonstrate that the dynamic decision method based on the adaptive engine is successful.

The remainder of this article is organized as follows. Section 2 introduces the background, including matrix multiplication and the SW26010 architecture. Section 3 presents the implementation details of RTAMM from different points of view. Section 4 validates the work experimentally, and Section 5 discusses the related works, leading to the conclusions and a discussion of future works in Section 6.

II. BACKGROUND

A. MATRIX MULTIPLICATION

There are three levels of subroutines in BLAS: vector-vector operations (Level 1 BLAS), matrix-vector operations (Level 2 BLAS), and matrix-matrix operations (Level 3 BLAS). Matrix multiplication, a matrix multiply-accumulate routine, is a Level 3 BLAS operation and defined as follows:

$$\mathbf{C} = \alpha\ op\,(\mathbf{A})\ op\,(\mathbf{B}) + \beta\ \mathbf{C},\quad op\,(\mathbf{X}) = \mathbf{X}\ or\ \mathbf{X}^T \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{M\times K}$ and $\mathbf{B} \in \mathbb{R}^{K\times \mathrm{N}}$ are input matrices with a transposed or non-transposed data format, and $\mathbf{C} \in \mathbb{R}^{M\times \mathrm{N}}$ is an output matrix. α and β are scalars that represent the operation coefficients. M, N, and K indicate the dimensions of the matrices.

As a critical component of many scientific applications [15], such as deep learning, signal processing and astrophysics, matrix multiplication is frequently applied to solve linear equations, least-square problems, singular and eigenvalue calculations [16]. Matrix multiplication is also used to evaluate the performance and efficiency of new processor architectures for scientific computing [17] and to investigate optimization methods on new architectures [18]. Because matrix multiplication is so widely used and compute-bound, it is significant to optimize its implementation. In this article, we implement the basic case where **A**, **B**, and **C** are double-precision, non-transposed, and row-major. This case is a performance basis in the HPL package [9], which has been used for ranking supercomputers in the TOP500 List for over two decades.

B. SW26010 PROCESSOR ARCHITECTURE

The SW26010 processor [8], [12] is a heterogeneous many-core architecture that uses distributed shared storage and on-chip computing array. As illustrated on the left side of Fig. 1, the processor consists of four CGs (core groups) with 260 processing cores. Each CG consists of an MPE (management processing element), an 8×8 CPE cluster, and a 4-way DDR3 MC (memory controller). Four CGs, each of which is connected directly to 8GB DDR3 main memory, are interconnected through the NoC (network on chip).

Both the MPE and the CPE have the frequency of 1.45 GHz and the vectorization size of 4, and support fused multiply-add instruction. The difference is that the MPE has two floating-point arithmetic pipelines, while the CPE has only one. Therefore, the theoretical peak performance of an MPE is 23.2 GFlops, while that of a CPE cluster is 742.4 GFlops.

The memory hierarchy of SW26010 is illustrated on the right side of Fig. 1. An MPE has a 32KB L1 instruction cache, a 32KB L1 data cache, and a 256KB L2 cache. Each CPE cluster consists of 64 equivalent CPEs and shares a 64 KB L2 instruction cache. A CPE has a 16 KB L1 instruction cache and a 64 KB user-controlled SPM (scratchpad memory), also called LDM (local data memory). The theoretical memory bandwidth of one chip is 136 GB/s, and each CG has 34.13 GB/s.

Two kinds of data transfer approaches are supported between the MPE and CPEs. One is global memory access, called gld/gst discrete access, which can read/write directly to the main memory. The gld/gst data access is user-friendly but has a high latency of up to 278 cycles. The other kind, known

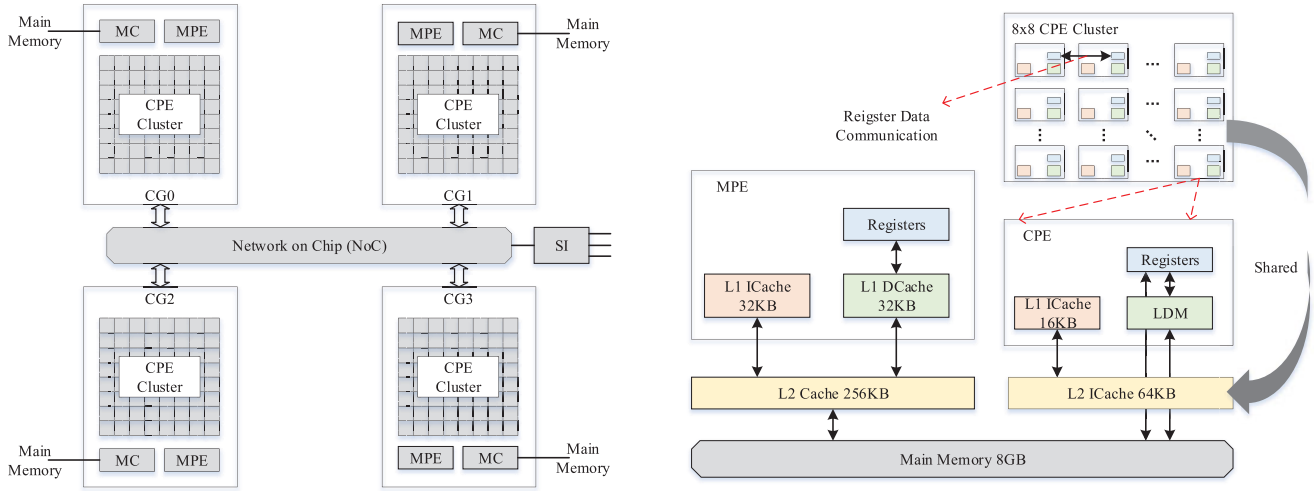


FIGURE 1. Architecture(left) and memory hierarchy(right) of the SW26010 processor.

as DMA (direct memory access) batched access, employs LDM as a bridge to access the main memory. The data access latency is relatively low, approximately 29 cycles. Due to the limited size of LDM and the complex nature of DMA operations, developers need to design parallel algorithms accurately.

Between different CPEs on the same CG, low-latency register communication is supported to reduce the frequency of memory access. There are three basic principles: (i) each communication fixes the data size to 256 bits; (ii) each CPE exchanges data only with other CPEs in the same row or column; (iii) the communication is anonymous based on the FCFS (first-come-first-serve) principle. Each CPE is equipped with a sending buffer, a row receiving buffer, and a column receiving buffer, which can buffer 6, 4, and 4 register messages, respectively.

Each CPE has two execution pipelines, P0 and P1. The former supports floating-point and integer scalar/vector operations, while the latter supports data transfer, comparison, jump, and integer scalar operations. The two pipelines share an ID (instruction decoder) and an instruction queue. When the following two conditions are satisfied, two instructions can be issued to P0 and P1 simultaneously: (i) the two instructions belong to two separate pipelines; (ii) the two instructions are conflict-free with each other as well as with the unfinished instructions issued before.

III. IMPLEMENTATION AND OPTIMIZATION FOR RTAMM

The features of the SW26010 architecture make it more flexible and controllable. However, more efforts are required to develop parallel algorithms. We describe the coarse-grained RTAMM methodology on the basic architecture of SW26010, then expound the fine-grained implementation and optimization in detail. To facilitate the following discussion, we define the meanings of some basic symbols in Table 1.

For matrix multiplication, the traditional implementation includes two components [5], [19], [20]: (i) a blocking algorithm that utilizes the memory hierarchy to partition different levels of workloads; (ii) a computational kernel that fully utilizes the computational power of the hardware. As illustrated in Fig. 2, the *basic implementation* of RTAMM, based on traditional researches, is divided into three levels: *macro optimization*, *assistant optimization*, and *micro optimization*. In terms of different grained workloads, the basic implementation is composed of three parts, $RTAMM_{gb}$, $RTAMM_{cg}$, and $RTAMM_{th}$, which represent global matrix multiplication, blocked matrix multiplication mapped to one CG, and blocked matrix multiplication mapped to one CPE, respectively. Macro optimization aims to design a high-quality blocking strategy to map $RTAMM_{gb}$ to $RTAMM_{cg}$. Given various technologies used to alleviate the memory-bound nature of SW26010, such as double buffering, and register communication, assistant optimization focuses on improving the data access efficiency of $RTAMM_{cg}$. Finally, micro optimization addresses the highly efficient computation of $RTAMM_{th}$.

Rather than fixing the execution action at the library generation time, the action of RTAMM is dynamically determined at runtime via an *adaptive engine*, which comprises several *fundamental cost formulas* and a *blocking factor pool*. Synthesizing some essential factors, such as triple-nested loop orders of matrix multiplication parameters, and parallel methods, several algorithms are designed for the basic implementation. Each algorithm corresponds to one fundamental

TABLE 1. Descriptions of different symbols.

Symbol	Description
$\mathbf{mtxA}, \mathbf{mtxB}, \mathbf{mtxC}$	Represent the matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, respectively
M, N, K	Represent the parameters (matrix dimensions) of matrix multiplication
$X_{gb}, X_{cg}, X_{th}, X_{rg}$	Represent the global, CG, thread, and register-level X, respectively

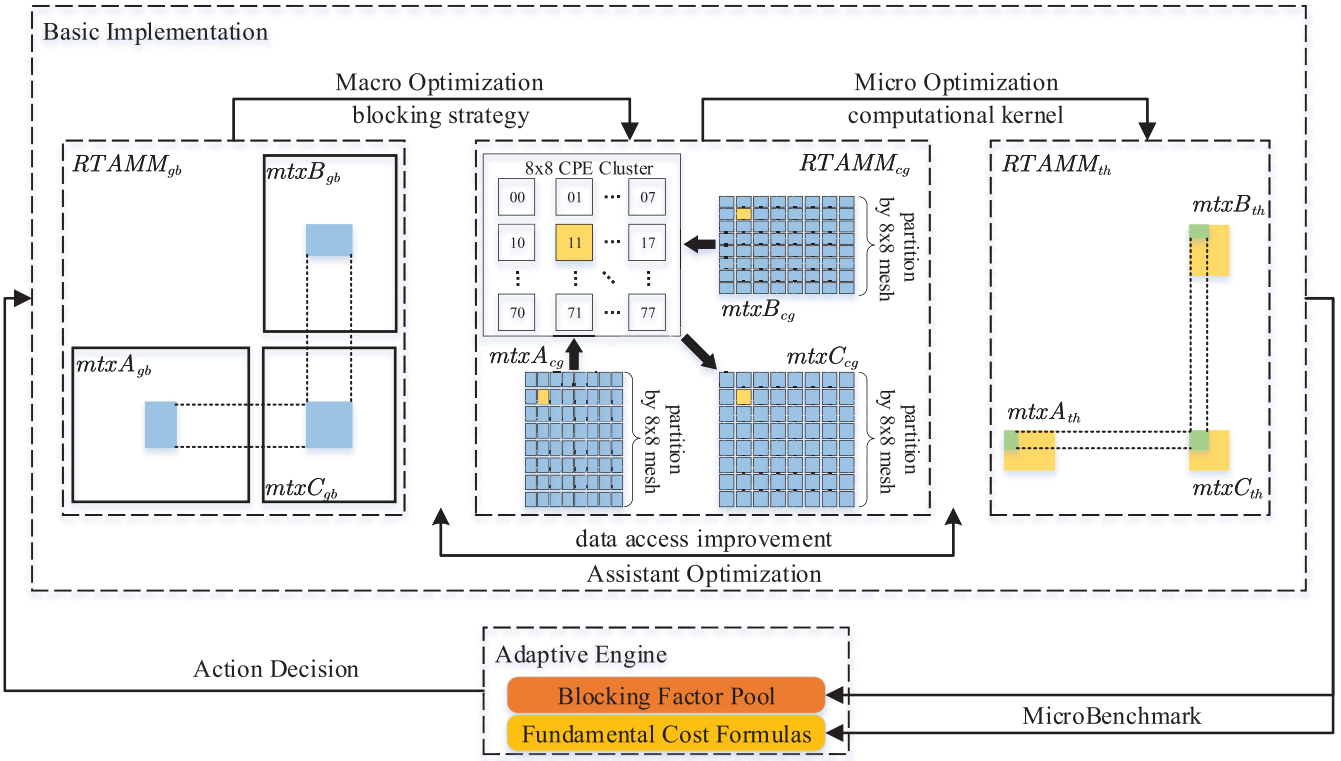


FIGURE 2. Guiding ideology of the RTAMM methodology for the SW26010 processor.

cost formula that evaluates the execution performance. Instead of analyzing a set of ideal blocking factors in theory, we gather multiple sets of potentially valuable blocking factors to construct the blocking factor pool. As demonstrated by experiments later on, the adaptive engine can effectively decide the execution action of RTAMM.

A. MACRO OPTIMIZATION

Macro optimization mainly addresses the blocking problem of matrix multiplication to map $RTAMM_{gb}$ to $RTAMM_{cg}$ considering varying computation and data access overheads and possible loop orders of matrix multiplication parameters.

1) BLOCKING STRATEGY BASED ON OVERHEAD FUNCTIONS

The memory hierarchy of modern processors can be simplified to three levels (from high to low): register, cache, and memory. The cache can be subdivided into L1 cache, L2 cache, and L3 cache [21]. The key to solving the blocking problem is to balance the computation and data access of two adjacent storage devices. Many works [22]–[24] have discussed this content from three perspectives: (i) maximizing the computation to data access ratio; (ii) utilizing the capacity of high-level storage to the maximum extent possible; (iii) maximizing the reutilization of data in low-level storage.

The memory hierarchy of the SW26010 processor is software-controlled with three levels (from high to low): register, LDM, and memory. Data can be transferred effectively via DMA between the LDM and the memory. There are many data-transfer modes, such as transferring data on one CPE or more CPEs, and transferring one continuous data block or multiple segmented data blocks. Different modes and data sizes result in different DMA bandwidths [10]. To adapt to the characteristics of the DMA, we propose a blocking strategy based on overhead functions. Two overhead functions are introduced, namely, OP_{cg}^{load} and OP_{cg}^{store}, which represent the time cost of loading one data from the memory to the LDM and the time cost of storing one data from the LDM to the memory, respectively. To further quantify the blocking strategy, we add one overhead function, OP_{cg}^{kernel}, which represents the time cost of one computation on the LDM.

$$\begin{aligned} Cost_{gb}^{MNK} &= n_{gb}MK \times OP_{cg}^{load}\left(M_{cg}, K_{cg}\right) + m_{gb}KN \\ &\quad \times OP_{cg}^{load}\left(K_{cg}, N_{cg}\right) + MN \times OP_{cg}^{load}\left(M_{cg}, N_{cg}\right) \\ &\quad + MN \times OP_{cg}^{store}\left(M_{cg}, N_{cg}\right) + 2MNK \\ &\quad \times OP_{cg}^{kernel}\left(M_{cg}, N_{cg}, K_{cg}\right) \end{aligned} \tag{2}$$

$$\begin{aligned} Cost_{gb}^{MKN} &= MK \times OP_{cg}^{load}\left(M_{cg}, K_{cg}\right) + m_{cg}KN \\ &\quad \times OP_{cg}^{load}\left(K_{cg}, N_{cg}\right) + k_{gb}MN \times OP_{cg}^{load}\left(M_{cg}, N_{cg}\right) \\ &\quad + k_{gb}MN \times OP_{cg}^{store}\left(M_{cg}, N_{cg}\right) + 2MNK \\ &\quad \times OP_{cg}^{kernel}\left(M_{cg}, N_{cg}, K_{cg}\right) \end{aligned} \tag{3}$$

Algorithm 1 $RTAMM_{gb}^{MNK}$ Algorithm With the Blocking Strategy

for $m = 0$ *to* $m_{gb} - 1$ **do**
  for $n = 0$ *to* $n_{gb} - 1$ **do**
    $Load_{DMA}$ $\mathbf{mtxC_{gb}}[m, n]$ *to* $\mathbf{mtxC_{cg}}$ $\quad MN \times OP_{cg}^{load}(M_{cg}, N_{cg})$
    for $k = 0$ *to* $k_{gb} - 1$ **do**
      $Load_{DMA}$ $\mathbf{mtxA_{gb}}[m, k]$ *to* $\mathbf{mtxA_{cg}}$ $\quad n_{gb}MK \times OP_{cg}^{load}(M_{cg}, K_{cg})$
      $Load_{DMA}$ $\mathbf{mtxB_{gb}}[k, n]$ *to* $\mathbf{mtxB_{cg}}$ $\quad m_{gb}KN \times OP_{cg}^{load}(K_{cg}, N_{cg})$
      $RTAMM_{cg}(\mathbf{mtxA_{cg}}, \mathbf{mtxB_{cg}}, \mathbf{mtxC_{cg}})$ $\quad 2MNK \times OP_{cg}^{kernel}(M_{cg}, N_{cg}, K_{cg})$
    end
    $Store_{DMA}$ $\mathbf{mtxC_{cg}}$ *to* $\mathbf{mtxC_{gb}}[m, n]$ $\quad MN \times OP_{cg}^{store}(M_{cg}, N_{cg})$
  end
end

$$
\begin{aligned}
&Cost_{gb}^{NKM} \\
&= n_{gb}MK \times OP_{cg}^{load}(M_{cg}, K_{cg}) + KN \\
&\quad \times OP_{cg}^{load}(K_{cg}, N_{cg}) + k_{gb}MN \times OP_{cg}^{load}(M_{cg}, N_{cg}) \\
&\quad + k_{gb}MN \times OP_{cg}^{store}(M_{cg}, N_{cg}) + 2MNK \\
&\quad \times OP_{cg}^{kernel}(M_{cg}, N_{cg}, K_{cg}) \qquad (4)
\end{aligned}
$$

In theory, there are six types of cases depending on the loop orders of the parameters M, N, and K. As shown in Algorithm 1, in the case of the $M \rightarrow N \rightarrow K$ triple-nested loop, denoted MNK, the m, n, and k loops along the M, N, and K matrix dimensions are blocked by sizes M_{cg}, N_{cg}, and K_{cg}, respectively. M, N, and K are $m_{gb}M_{cg}$, $n_{gb}N_{cg}$, and $k_{gb}K_{cg}$, respectively. The CG-level submatrix $\mathbf{mtxC_{cg}}$ resides in the LDM until one whole innermost loop is accomplished, then is written back to the memory. Each iteration loads one CG-level submatrix $\mathbf{mtxA_{cg}}$ and one CG-level submatrix $\mathbf{mtxB_{cg}}$ and updates the resident submatrix $\mathbf{mtxC_{cg}}$. In other words: (i) $\mathbf{mtxC_{gb}}$ is loaded and stored one time. $OP_{cg}^{load}(M_{cg}, N_{cg})$ and $OP_{cg}^{store}(M_{cg}, N_{cg})$ are the costs of loading and storing one element in $\mathbf{mtxC_{gb}}$, respectively; (ii) $\mathbf{mtxA_{gb}}$ is loaded n_{gb} times. $OP_{cg}^{load}(M_{cg}, K_{cg})$ is the cost of loading one element in $\mathbf{mtxA_{gb}}$; (iii) $\mathbf{mtxB_{gb}}$ is loaded m_{gb} times. $OP_{cg}^{load}(K_{cg}, N_{cg})$ is the cost of loading one element in $\mathbf{mtxB_{gb}}$; (iv) each element of $\mathbf{mtxC_{gb}}$ is updated via K multiply-add operations of a row of $\mathbf{mtxA_{gb}}$ and a column of $\mathbf{mtxB_{gb}}$ and one add operation with the prior $\mathbf{mtxC_{gb}}$ element, resulting in $2K$ arithmetic computing operations. Different blocking strategies will not change the total amount of computation for one matrix multiplication, so the total amount of computation is $2MNK$. Moreover, $OP_{cg}^{kernel}(M_{cg}, N_{cg}, K_{cg})$ is the cost of one arithmetic operation in the context of $RTAMM_{cg}$. Therefore, the cost of $RTAMM_{gb}^{MNK}$ is shown in formula (2).

The work in this article is implemented on one CG for the SW26010 processor because parallel algorithms across different CGs are usually at higher programming levels by users. Therefore, Algorithm 1 is executed serially without considering the parallel execution. Upon interchanging the two outermost loops of Algorithm 1, we can easily find $Cost_{gb}^{NMK} = Cost_{gb}^{MNK}$. The equation will not be affected by the implementation of $RTAMM_{cg}$, because different implementations change only the unit overhead OP_{cg}^{kernel} while leaving the overall computation and data access unchanged. Thus, the influence is the same for $Cost_{gb}^{NMK}$ and $Cost_{gb}^{MNK}$ regardless of how $RTAMM_{cg}$ is executed. Based on the above illustration, only two cases of algorithms, $M \rightarrow K \rightarrow N$ (MKN) and $N \rightarrow K \rightarrow M$ (NKM), are to be further discussed because they are equal to KMN and KNM, respectively. Similarly, the costs of $RTAMM_{gb}^{MKN}$ and $RTAMM_{gb}^{NKM}$ are shown in formulas (3) and (4), respectively. Moreover, we have $Cost_{gb}^{MKN} = Cost_{gb}^{KMN}$ and $Cost_{gb}^{NKM} = Cost_{gb}^{KNM}$.

For one matrix multiplication case, the process of selecting the blocking method is to compare the cost formulas (2) through (4).

B. ASSISTANT OPTIMIZATION

Assistant optimization aims mainly to reduce the data access overhead in $RTAMM_{gb}$. Based on asynchronous DMA operations and data sharing within the same CPE cluster, we eliminate unnecessary data access and overlap necessary data access and computation.

1) MIXED DOUBLE BUFFERING METHOD

$RTAMM_{gb}$ is composed of the data access by the DMA and the computation in $RTAMM_{cg}$. Although the appropriate DMA operation can quickly transfer data between the memory and the LDM, the overhead is still nonnegligible. To optimize the data access, we design a mixed double buffering method to overlap the computation and data access [25], [26].

$$
\begin{aligned}
Cost_{gb}^{MNK,M2B2} \approx{}& MN \times OP_{cg}^{load}(M_{cg}, N_{cg}) + MN \\
&\times OP_{cg}^{store}(M_{cg}, N_{cg}) + max\{LDST_{olp}, CMPT_{olp}\} \qquad (5)
\end{aligned}
$$

$$
\begin{aligned}
Cost_{gb}^{MNK,M3B2} \approx{}& MN \times OP_{cg}^{store}(M_{cg}, N_{cg}) \\
&+ max\{LDST_{overlap}, CMPT_{overlap}\} \qquad (6)
\end{aligned}
$$

$$
\begin{aligned}
Cost_{gb}^{MKN,M2B2} \approx{}& MK \times OP_{cg}^{load}(M_{cg}, K_{cg}) + k_{gb}MN \\
&\times OP_{cg}^{store}(M_{cg}, N_{cg}) + max\{LDST_{olp}, CMPT_{olp}\} \qquad (7)
\end{aligned}
$$

Algorithm 2 $RTAMM_{gb}^{MNK}$ Optimized Algorithm With the M2B2 Double Buffering

// super **begin**,**end**: begin and end of DMA operation
// sub **next**: the next loop related to the current loop
// **cmpt ldst**: the index of computation and data access
$LoadA_{DMA}^{begin}$ $\mathbf{mtxA_{gb}}$ $[0, 0]$ to $\mathbf{mtxA_{cg}}$ $[cmpt_A]$
$LoadB_{DMA}^{begin}$ $\mathbf{mtxB_{gb}}$ $[0, 0]$ to $\mathbf{mtxB_{cg}}$ $[cmpt_B]$
$LoadA_{DMA}^{end}$
$LoadB_{DMA}^{end}$
for $m = 0$ ***to*** $m_{gb} - 1$ **do**
  for $n = 0$ ***to*** $n_{gb} - 1$ **do**
    $LoadC_{DMA}^{begin}$ $\mathbf{mtxC_{gb}}$ $[m, n]$ to $\mathbf{mtxC_{cg}}$
    $LoadC_{DMA}^{end}$
    for $k = 0$ ***to*** $k_{gb} - 1$ **do**
      compute m_{next}, n_{next}, k_{next} of next dma operation about $\mathbf{mtxA_{cg}}$ and $\mathbf{mtxB_{cg}}$
      $LoadA_{DMA}^{begin}$ $\mathbf{mtxA_{gb}}$ $[m_{next}, k_{next}]$ to $\mathbf{mtxA_{cg}}$ $[ldst_A]$
      $LoadB_{DMA}^{begin}$ $\mathbf{mtxB_{gb}}$ $[k_{next}, n_{next}]$ to $\mathbf{mtxB_{cg}}$ $[ldst_B]$
      $RTAMM_{cg}$ $(\mathbf{mtxA_{cg}}\,[cmpt_A]\,, \mathbf{mtxB_{cg}}\,[cmpt_B]\,, \mathbf{mtxC_{cg}})$
      $LoadA_{DMA}^{end}$
      $LoadB_{DMA}^{end}$
      exchange the value of $ldst_A$ and $cmpt_A$
      exchange the value of $ldst_B$ and $cmpt_B$
    end
  end
  $StoreC_{DMA}^{begin}$ $\mathbf{mtxC_{cg}}$ to $\mathbf{mtxC_{gb}}$ $[m, n]$
  $StoreC_{DMA}^{end}$
end

$$Cost_{gb}^{MKN,M3B2} \approx k_{gb}MN \times OP_{cg}^{store}\left(M_{cg}, N_{cg}\right) + max\left\{LDST_{olp}, CMPT_{olp}\right\} \quad (8)$$

$$Cost_{gb}^{NKM,M2B2} \approx KN \times OP_{cg}^{load}\left(K_{cg}, N_{cg}\right) + k_{gb}MN \times OP_{cg}^{store}\left(M_{cg}, N_{cg}\right) + max\left\{LDST_{olp}, CMPT_{olp}\right\} \quad (9)$$

$$Cost_{gb}^{NKM,M3B2} \approx k_{gb}MN \times OP_{cg}^{store}\left(M_{cg}, N_{cg}\right) + max\left\{LDST_{olp}, CMPT_{olp}\right\} \quad (10)$$

The proposed method is based on the following considerations:

- Perform the computation and data access in parallel by interleaving their loop sequences.
- Because of the limited LDM, we balance the increase in the degree of double buffering and the increase in the computation task in $RTAMM_{cg}$.

With the above considerations, the mixed double buffering method is proposed including M2B2 (double buffer two matrices) and M3B2 (double buffer three matrices). As shown in Algorithm 2, M2B2 targets more computation task in $RTAMM_{cg}$ by reducing the overlap between the computation and data access. We double buffer $\mathbf{mtxA_{cg}}$ and $\mathbf{mtxB_{cg}}$, and then prefetch the first $\mathbf{mtxA_{cg}}$ and $\mathbf{mtxB_{cg}}$; in addition, we guarantee that loading next $\mathbf{mtxA_{cg}}$ and $\mathbf{mtxB_{cg}}$ and computing current $RTAMM_{cg}$ are executed in parallel without data dependence. M3B2 aims to maximize the potential overlap between the computation and data access, which is similar to Algorithm 2. We double buffer $\mathbf{mtxA_{cg}}$, $\mathbf{mtxB_{cg}}$, and $\mathbf{mtxC_{cg}}$, and then prefetch the first $\mathbf{mtxA_{cg}}$, $\mathbf{mtxB_{cg}}$, and $\mathbf{mtxC_{cg}}$; similarly, we guarantee that loading next $\mathbf{mtxA_{cg}}$, $\mathbf{mtxB_{cg}}$, and $\mathbf{mtxC_{cg}}$ and computing current $RTAMM_{cg}$ are executed in parallel.

Although the first $LOAD_{DMA}$ and the last $RTAMM_{cg}$ cannot be overlapped, the influence is negligible relative to the entire matrix multiplication process. The approximate cost of $RTAMM_{gb}^{MNK}$ is extended, as shown in the formulas (5) and (6). Similarly, the costs of $RTAMM_{gb}^{MKN}$ and $RTAMM_{gb}^{NKM}$ are shown in the formulas (7), (8), (9), and (10).

In the above formulas, $CMPT_{olp}$ and $LDST_{olp}$ represent the computation overlapped and data access overlapped. Taking $Cost_{gb}^{MNK,M2B2}$ as an example, $2MNK \times OP_{cg}^{kernel}\left(M_{cg}, N_{cg}, K_{cg}\right)$ is equal to $CMPT_{olp}$, and $n_{gb}MK \times OP_{cg}^{load}\left(M_{cg}, K_{cg}\right) + m_{gb}KN \times OP_{cg}^{load}\left(K_{cg}, N_{cg}\right)$ is equal to $LDST_{olp}$.

2) BROADCAST-BROADCAST ON-CHIP COMMUNICATION

Intuitively, $\mathbf{mtxC_{cg}}$ can be partitioned by an 8×8 mesh and then mapped to one CG. Each CPE is responsible for one $\frac{\mathbf{mtxC_{cg}}}{64}$ submatrix, which transfers the data of one $\frac{\mathbf{mtxA_{cg}}}{8}$ submatrix, one $\frac{\mathbf{mtxB_{cg}}}{8}$ submatrix and one $\frac{\mathbf{mtxC_{cg}}}{64}$ submatrix

via the DMA. The $\mathbf{mtxA_{cg}}$ and $\mathbf{mtxB_{cg}}$ are loaded 8 times repeatedly, which makes it valuable to research on-chip data sharing. SW26010 does not support direct on-chip data sharing, but provides a register communication mechanism so that the 64 CPEs within the same CPE cluster can indirectly exchange data with each other. A CPE cluster is similar to a reduced distributed-memory parallel computer system, for which matrix multiplication optimization has been evaluated in many works [27], [28]. Referring to those works, we design a data sharing method, called *broadcast-broadcast on-chip communication*, to enhance the reutilization of data on the CPE cluster.

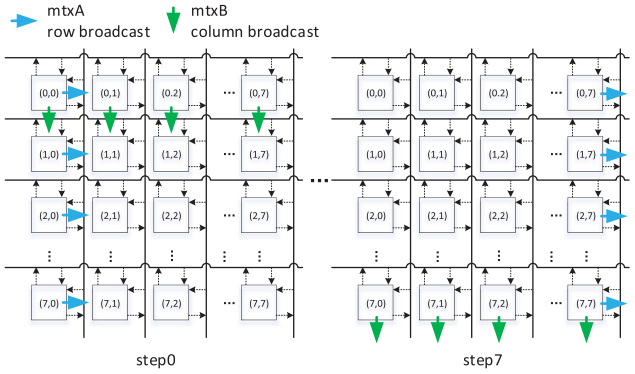


FIGURE 3. Broadcast-broadcast on-chip communication on an 8 × 8 CPE cluster.

As illustrated in Fig. 3, *CPE* (i, j) represents the CPE in the i-th row and j-th column ($i \in \{0, 1, \ldots, 7\}, j \in \{0, 1, \ldots, 7\}$). The $\mathbf{mtxA_{cg}}$, $\mathbf{mtxB_{cg}}$, and $\mathbf{mtxC_{cg}}$ are uniformly partitioned into thread-level submatrices by an 8 × 8 mesh, called $\mathbf{mtxA_{th}}$, $\mathbf{mtxB_{th}}$, and $\mathbf{mtxC_{th}}$, respectively. *CPE* (i, j) is responsible for the three submatrices $\mathbf{mtxA_{th}}\,(i, j)$, $\mathbf{mtxB_{th}}\,(i, j)$ and $\mathbf{mtxC_{th}}\,(i, j)$, to eliminate the unnecessary data access of $\mathbf{mtxA_{cg}}$ and $\mathbf{mtxB_{cg}}$. To regulate the computation $\mathbf{mtxC_{th}}\,(i, j) = \sum_{k=0}^{7} \mathbf{mtxA_{th}}\,(i, k) \times \mathbf{mtxB_{th}}\,(k, j)$, The CPEs of each row need to communicate with each other once regarding $\mathbf{mtxA_{th}}$, and the CPEs of each column need to communicate with each other once regrading $\mathbf{mtxB_{th}}$. The process is as follows:

- Step0: *CPE* $(i, 0)$ broadcasts the data of $\mathbf{mtxA_{th}}\,(i, 0)$ to other CPEs in the same row, which receive the row-broadcast data, by register communication. *CPE* $(0, j)$ broadcasts the data of $\mathbf{mtxB_{th}}\,(0, j)$ to other CPEs in the same column, which receive the column-broadcast data, by register communication. At the moment, all the CPEs have four statuses: (i) the row broadcast of $\mathbf{mtxA_{th}}$ and the column broadcast of $\mathbf{mtxB_{th}}$; (ii) the row broadcast of $\mathbf{mtxA_{th}}$ and the column reception of $\mathbf{mtxB_{th}}$; (iii) the row reception of $\mathbf{mtxA_{th}}$ and the column broadcast of $\mathbf{mtxB_{th}}$; (iv) the row reception of $\mathbf{mtxA_{th}}$ and the column reception of $\mathbf{mtxB_{th}}$. The CPEs perform $RTAMM_{th}$ by means of the local or remote $\mathbf{mtxA_{th}}$, the local or remote $\mathbf{mtxB_{th}}$, and the local $\mathbf{mtxC_{th}}$.
- Step1: *CPE* $(i, 1)$ broadcasts the data of $\mathbf{mtxA_{th}}\,(i, 1)$ to other CPEs in the same row, which receive the row-broadcast data, by register communication. *CPE* $(1, j)$ broadcasts the data of $\mathbf{mtxB_{th}}\,(1, j)$ to other CPEs in the same column, which receive the column-broadcast data, by register communication. Four statuses of CPEs perform the corresponding $SRTAMM_{th}$ separately.
- Step2 through Step7 are similar to Step1.

According to the broadcast-broadcast on-chip communication mechanism, we enhance the reutilization of data on the CPE cluster to eliminate unnecessary data access. $RTAMM_{cg}$ can be efficiently accomplished by 8 steps.

C. MICRO OPTIMIZATION

Micro optimization focuses mainly on the highly efficient computation of $RTAMM_{th}$. To acquire the high-performance computing kernel, we orchestrate the main instruction sequence to fully utilize all usable vector registers and reduce the idle time of the two pipelines (P0 and P1) on the CPE.

1) REGISTER BLOCKING

There are many limitations for register communication, such as the fixed data size of 256 bits and anonymous process. Therefore, we need to refine the broadcast-broadcast on-chip communication mechanism to guarantee the accuracy and efficiency of $RTAMM_{th}$.

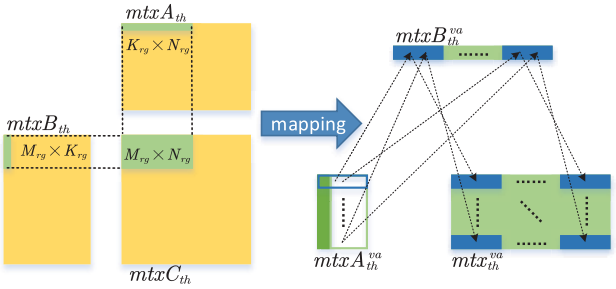


FIGURE 4. Register blocking based on the broadcast-broadcast on-chip communication.

For $RTAMM_{cg}$, each CPE needs to transfer a $\mathbf{mtxA_{th}}$ with size of $M_{th} \times K_{th}$, a $\mathbf{mtxB_{th}}$ with size of $K_{th} \times N_{th}$, and a $\mathbf{mtxC_{th}}$ with size of $M_{th} \times N_{th}$. M_{th}, N_{th}, and K_{th} are equal to $M_{cg}/8$, $N_{cg}/8$, and $K_{cg}/8$, respectively. Both $\mathbf{mtxA_{th}}$ and $\mathbf{mtxB_{th}}$ of each CPE must be broadcast once via register communication. The limitation of register communication, with one-time data size of 256 bits, makes it necessary to perform fine-grained blocking. As shown in Fig. 4, three thread-level matrix blocks are divided into multiple register-level matrix blocks, $\mathbf{mtxA_{rg}}$, $\mathbf{mtxB_{rg}}$, and $\mathbf{mtxC_{rg}}$, whose sizes are $M_{rg} \times K_{rg}$, $K_{rg} \times N_{rg}$, and $M_{rg} \times N_{rg}$, respectively. During the communication phase, there are four CPE statuses. We take the most complicated status with row broadcasting and column broadcasting as an example to explain register blocking in detail:

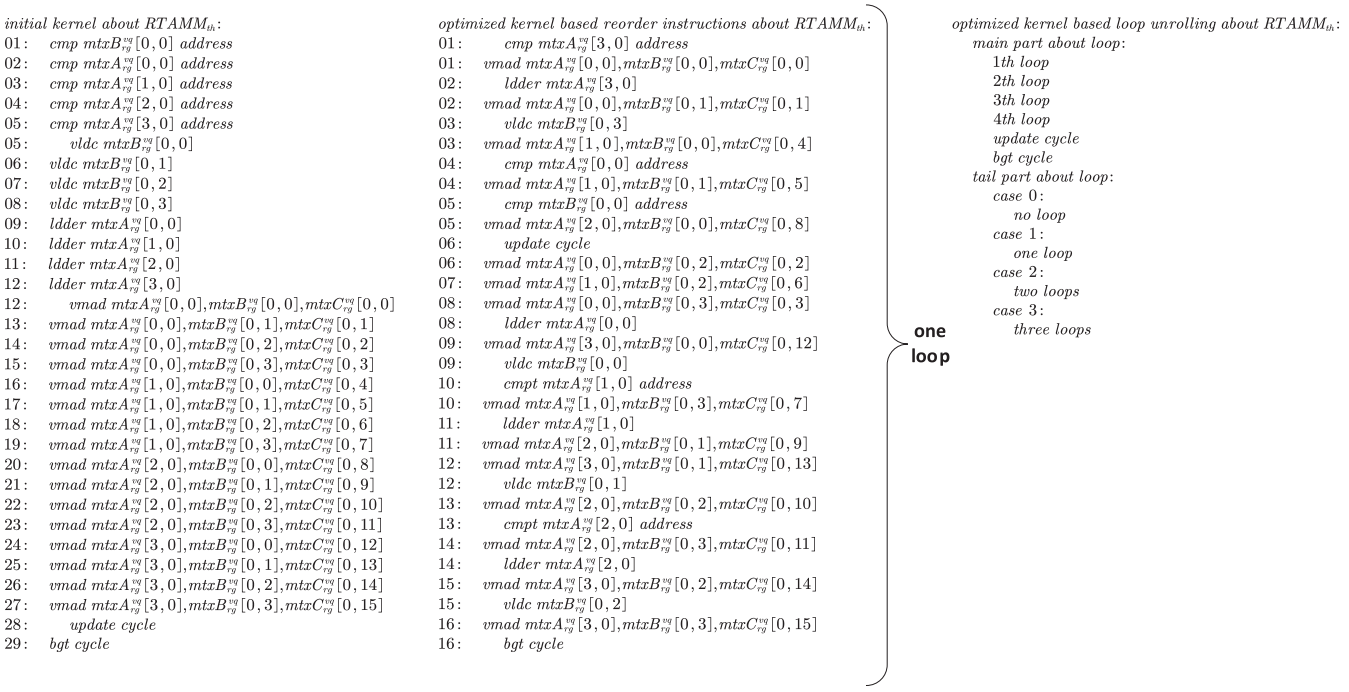


FIGURE 5. Instruction scheduling for the computational kernel.

- Load the 256-bit data segment in $\mathbf{mtxC_{rg}}$ by **vldd** instruction in turn, marked as the vector array $\mathbf{mtxC_{rg}^{va}}[0 : M_{rg}][0 : \frac{N_{rg}}{4}]$.
- Load each data in $\mathbf{mtxA_{rg}}$ to perform vector expansion in turn, marked as the vector array $\mathbf{mtxA_{rg}^{va}}[0 : M_{rg}][0 : K_{rg}]$, then perform the row broadcast, by **ldder** instruction.
- Load the 256-bit data segment in $\mathbf{mtxB_{rg}}$ in turn, marked as the vector array $\mathbf{mtxB_{rg}^{va}}[0 : K_{rg}][0 : \frac{N_{rg}}{4}]$, then perform the column broadcast, by **vldc** instruction.
- Perform the register-level matrix multiplication, $\mathbf{mtxC_{rg}^{va}}(i,j) + = \sum_{k=0}^{K_{rg}-1} \mathbf{mtxA_{rg}^{va}}(i,k) \times \mathbf{mtxB_{rg}^{va}}(k,j)$, by **vmad** instruction

For the highly efficient computation of register-level matrix multiplication, we make the following two guarantees: (i) there is no data dependence in the above process; (ii) each vector element is matched by an independent vector register. However, SW26010 has only 32 vector registers with a size of 256-bits, including the zero register and the SP (stack pointer) register. No more than 30 registers can be used with confidence because of other operations, such as the intermediate address calculation and loop judgment. With these considerations, we set $K_{rg} = 1$ to make guarantee (i) above. In addition, we have $M_{rg} + \frac{N_{rg}}{4} + M_{rg}\frac{N_{rg}}{4} < 30$ to ensure guarantee (ii) and synthesize the computation to data access ratio of $RTAMM_{th}$ as follows:

$$\frac{2M_{th}N_{th}K_{th}}{4M_{th}K_{th}\frac{N_{th}}{N_{rg}} + K_{th}N_{th}\frac{M_{th}}{M_{rg}} + 2M_{th}N_{th}} \approx \frac{2}{\frac{4}{N_{rg}} + \frac{1}{M_{rg}}} \tag{11}$$

To maximize the ratio, we acquire the minimum value of $\frac{4}{N_{rg}} + \frac{1}{M_{rg}}$ when $M_{rg} = \frac{N_{rg}}{4} = 4$.

2) INSTRUCTION SCHEDULING

To obtain higher peak performance, modern processors not only integrate more and more cores, but also widely apply superscalar technology [29]. Many studies [11], [13], [30], [31] have focused on instruction-level optimization methods based on superscalar technology. The same is true of the SW26010 processor, which has two pipelines (P0 and P1) on one CPE. P0 supports floating-point and integer operations of scalars/vectors, while P1 supports data transfer, comparison, jump, and integer scalar operations. As a result, the instruction arrangement and order are important to fully parallelize pipelines P0 and P1.

As an example, we take the CPE that requires the row broadcast of $mtxA_{rg}^{va}$ and the column broadcast of $mtxB_{rg}^{va}$ to explain the instruction scheduling method. For $RTAMM_{th}$, the main goal is to perform multiple register-level vector multiply-add operations. As shown on the left side of Fig. 5, the innermost loop, which is the core of $RTAMM_{th}$, includes 8 data access instructions, 16 vector multiply-add instructions, and some instructions for address calculation and loop judgment. Normally, the execution overhead is 29 cycles because of the terrible parallelization of P0 and P1, which forfeits almost half of the computational power. To improve the execution efficiency, we manually reorder the instructions to interleave the P0 instruction with the P1 instruction so that two instructions can be issued together in one cycle. As shown in the middle of Fig. 5, the execution overhead

is 16 cycles after reordering the instructions, reflecting an approximately 81.3% performance improvement. To further explore the potential of the instruction sequence, we unroll the innermost loop appropriately to reduce the number of instructions and the frequency of branch judgment. Considering the 16 KB L1 ICache on SW26010, we unroll the loop 4 times to guarantee that all instructions of $RTAMM_{th}$ can be stored in the L1 ICache without frequent instruction loading. As shown on the right side of Fig. 5, the innermost loop is divided into two parts. The main part is responsible for four consecutive register-level matrix multiplication, while the tail part for the remaining 0/1/2/3 ones. We reduce the number of instructions by approximately 4.7% and the frequency of branch judgment by approximately 75%.

D. ADAPTIVE ENGINE CONSTRUCTION

As illustrated in Fig. 2, the *adaptive engine* allows RTAMM to dynamically determine the execution action at runtime rather than fixing the action at library generation time. As demonstrated in subsequent experiments, the dynamic determination method greatly benefits the adaptability and performance of matrix multiplication. The adaptive engine is composed of two components: (i) several fundamental cost formulas; (ii) a blocking factor pool. After explaining the basic implementation of RTAMM, the formulas (5) through (10) are generated as the fundamental cost formulas to estimate the execution efficiency. Furthermore, we no longer analyze one set of theoretically optimal blocking factors, but instead gather multiple sets of potential blocking factors to establish a blocking factor pool and fully explore possible actions for one matrix multiplication.

The blocking factors for RTAMM are M_{cg}, N_{cg}, and K_{cg}, which are applied to map the global-level matrix multiplication to the CG-level one. Based on $K_{rg} = 1$ and $M_{rg} = \frac{N_{rg}}{4} = 4$ in Section 3.3, M_{cg}, N_{cg}, and K_{cg} should be the multiple of 32, 128, and 8, respectively. To satisfy the DDR3 interface which requires memory access in blocks of 128 bytes, the column sizes (N_{cg} and K_{cg}) of the CG-level matrix blocks need to be multiples of 128. On the other hand, the LDM size on one CPE is only 64 KB, and it is impossible to allocate all the LDM to matrix blocks because other data also occupy a certain amount of LDM. After performing tests on different allocated LDM sizes, we find that the reliable threshold is 61 KB ($499712 = 61 * 64 * 1024/8$). Therefore, corresponding to $RTAMM_{cg}^{MNK}$ by M2B2, $RTAMM_{cg}^{MKN}$ by M2B2, $RTAMM_{cg}^{NKM}$ by M2B2, and $RTAMM_{cg}$ by M3B2, M_{cg}, N_{cg}, and K_{cg} must satisfy the following inequalities:

$$\begin{aligned} 2M_{cg}K_{cg} + 2K_{cg}N_{cg} + M_{cg}N_{cg} &\leqslant 499712 \\ M_{cg}K_{cg} + 2K_{cg}N_{cg} + 2M_{cg}N_{cg} &\leqslant 499712 \\ 2M_{cg}K_{cg} + K_{cg}N_{cg} + 2M_{cg}N_{cg} &\leqslant 499712 \\ 2M_{cg}K_{cg} + 2K_{cg}N_{cg} + 2M_{cg}N_{cg} &\leqslant 499712 \end{aligned} \quad (12)$$

Based on the above considerations, after testing on different blocking factors of M_{cg}, N_{cg}, and K_{cg}, we select some potential blocking factors appropriately, as shown in Table 2. We do not search all of the useable blocking factors, because this article aims to demonstrate the feasibility of the RTAMM methodology rather than accomplishing the faultless implementation of matrix multiplication on SW26010. Instead, our future work will focus on the perfect implementation. Here, we design a *MicroBenchmark* to simulate the critical operations of RTAMM and measure their unit overheads OP_{cg}^{load}, OP_{cg}^{store}, and OP_{cg}^{kernel}. Upon testing the read and write DMA bandwidths of one matrix, we can obtain $OP_{cg}^{load}(M_{cg}, K_{cg})$, $OP_{cg}^{load}(K_{cg}, N_{cg})$, $OP_{cg}^{load}(M_{cg}, N_{cg})$, and $OP_{cg}^{store}(M_{cg}, N_{cg})$. Similarly, $OP_{cg}^{kernel}(M_{cg}, N_{cg}, K_{cg})$ can be obtained via the performance measurement for RTAMM without DMA operations. The results are shown in Table 2.

TABLE 2. Different blocking factors and unit overheads for RTAMM.

RTAMM	N_{cg}	K_{cg}	M_{cg}	$OP^{load}(M_{cg}, K_{cg})$	$OP^{load}(K_{cg}, N_{cg})$	$OP^{load}(M_{cg}, N_{cg})$	$OP^{store}(M_{cg}, N_{cg})$	$OP^{kernel}(M_{cg}, N_{cg}, K_{cg})$
MNK M2B2	256	256	480	0.2855	0.2855	0.2855	0.3112	0.1632
	256	384	288	0.3249	0.2858	0.2868	0.3095	0.157
	256	512	160	0.286	0.2882	0.2923	0.3073	0.1542
	384	256	320	0.2848	0.3189	0.3144	0.3367	0.1621
	384	384	160	0.3095	0.3226	0.3095	0.3312	0.1565
	512	256	224	0.2885	0.2851	0.2878	0.3077	0.1617
	640	256	128	0.2925	0.3097	0.2996	0.3218	0.1618
MKN M2B2	256	256	480	0.2855	0.2855	0.2855	0.3112	0.1632
	256	384	320	0.3144	0.2858	0.2848	0.3024	0.1568
	256	512	224	0.2878	0.2882	0.2885	0.3114	0.1537
	256	640	128	0.2996	0.2863	0.2925	0.3115	0.1535
	384	256	288	0.2868	0.3189	0.3249	0.347	0.1623
	384	384	160	0.3095	0.3226	0.3095	0.3312	0.1565
	512	256	160	0.2923	0.2851	0.286	0.3019	0.1621
NKM M2B2	256	256	416	0.2856	0.2855	0.2856	0.3143	0.1634
	256	384	288	0.3249	0.2858	0.2868	0.3095	0.157
	256	512	224	0.2878	0.2882	0.2885	0.3114	0.1537
	256	640	160	0.3116	0.2863	0.2923	0.3073	0.152
	256	768	128	0.2886	0.2879	0.2925	0.3115	0.1518
	384	256	288	0.2868	0.3189	0.3249	0.347	0.1623
	384	384	224	0.3185	0.3226	0.3185	0.349	0.1561
	384	512	160	0.286	0.3155	0.3095	0.3312	0.1532
	512	256	224	0.2885	0.2851	0.2878	0.3077	0.1617
	512	384	160	0.3095	0.2862	0.286	0.3019	0.1558
	640	256	160	0.2923	0.3097	0.3116	0.3413	0.1615
	768	256	128	0.2925	0.2849	0.2886	0.3018	0.1613
MNK MKN NKM M3B2	256	256	352	0.2868	0.2855	0.2868	0.3128	0.1636
	256	384	224	0.3185	0.2858	0.2885	0.3114	0.1583
	256	512	128	0.2905	0.2882	0.2925	0.3115	0.1546
	384	256	224	0.2885	0.3189	0.3185	0.349	0.1626
	384	384	128	0.3105	0.3226	0.3105	0.3274	0.1573
	512	256	128	0.2925	0.2851	0.2905	0.2924	0.1626

IV. EXPERIMENTAL RESULTS

For the fundamental math library, the adaptability of the implementation is more important than the peak performance because the former determines whether the library can adapt to changeable scenes in real applications. To verify the superiority of the proposed RTAMM, we evaluate the experimental results from three perspectives: (i) the peak performance; (ii) the adaptability; (iii) the effectiveness of the adaptive engine. All the experiments are built on one CG, because parallel algorithms across different CGs are usually at higher programming levels by users. Taking the state-of-the-art SWMM from the swBLAS library on SW26010 as the baseline, we assess our work in this article. During the experiments, we call directly the user API (application programming interface) of matrix multiplication in SWMM, *dgemm_*().

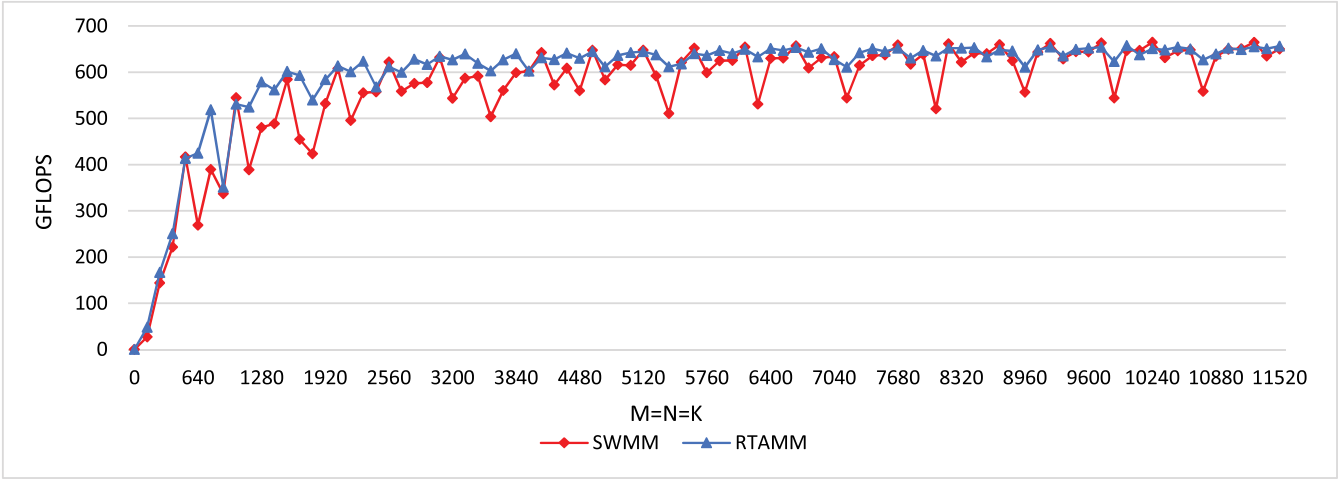


FIGURE 6. Peak performance comparison between RTAMM and SWMM. The X axis indicates different matrix multiplication configurations with *M*, *N*, and *K*. The Y axis indicates the matrix multiplication performance (GFlops)at runtime.

A. PEAK PERFORMANCE EVALUATION

The square matrix ($M = N = K$) is the most suitable choice for testing the peak performance of matrix multiplication. Considering that the DDR3 interface usually requires memory access in blocks of 128 bytes to obtain the optimal bandwidth, we set 90 matrix multiplications with different configurations of $M = N = K = 128X(X \in \{1, 2, \ldots, 90\})$.

Fig. 6 displays the RTAMM and SWMM performance trends as the matrix dimensions increase. When M is approximately 9216, RTAMM reaches a peak of 657.97 GFlops, while SWMM is 664.99 GFlops. In contrast to SWMM, RTAMM achieves competitive peak performance with a negligible performance gap ($\approx$ 1%). Moreover, RTAMM has a more stable performance than SWMM because of the slight fluctuation.

B. ADAPTABILITY EVALUATION

Here, we focus more on the adaptability of RTAMM than the peak performance. Accordingly, we measure and analyze 6000 matrix multiplication cases with different configurations of M, N, and K on RTAMM and SWMM.

The 6000 matrix multiplications can be divided into six categories (*SMA*, *MMA*, *BMA*, *SMNA*, *MMNA*, and *BMNA*) with each consisting of 1000 matrix multiplications. The notations above are abbreviations used to describe the attributes of the categories. *SM*, *MM*, and *BM* represent small-scale, medium-scale, and big-scale matrix multiplications, respectively, while *A* and *NA* are aligned and unaligned data. As shown in Fig. 7, the ratios of matrix dimensions range from 1 to 10, and *baseSize* describes the cardinal number of matrix dimensions, namely, 128, 640, or 1152. The best data alignment indicates that the data header address of each row is aligned by 128 bytes for all the matrices. The worst data alignment is the opposite, which indicates that the data header address of each row is unaligned. These matrix multiplications, excluding extreme cases such as unit dimensions, and enormous ratios, are adequate for comparing the adaptability between RTAMM and SWMM because the purpose of this article is not to implement perfect matrix multiplication on SW26010.

- Best Data Alignment

```
for(m=1; m<=10; m++)
  for(n=1; n<=10; n++)
    for(k=1; k<=10; k++)
      m*=baseSize; n*=baseSize; k*=baseSize;
```

- Worst Data Alignment

```
for(m=1; m<=10; m++)
  for(n=1; n<=10; n++)
    for(k=1; k<=10; k++)
      m*=baseSize; n*=baseSize; k*=baseSize;
      m+=1; n+=1; k+=1
```

FIGURE 7. Different matrix multiplication configurations.

As illustrated in Fig. 8, RTAMM achieves a considerable performance improvement in most cases. The improvements range from 5% to 308% compared with SWMM. The average performance improvements of SMAN, MMNA, and BMNA are 79.9%, 122.4%, and 131.1%, respectively, while those of SMA, MMA, and BMA are 36.2%, 18.4%, and 11.5%, respectively. For the average performance, SWMM has the least gap relative to RTAMM in the BMA cases, because the matrix dimensions in these cases are relatively large and close to the multiple of the optimal blocking factors. As a result, pursuing only the peak performance is restricted, and adaptability should be the main concern. Because SWMM is transparent for users, the fluctuations in Fig. 8 are difficult to be explained in detail. To evaluate the experimental results, we calculate the statistics of the number of cases for different degrees of performance gaps, with 5% as the threshold of the weak performance gap, 25% as the visible performance gap, and 50% as the strong performance gap. As shown in Table (3), when the performance gap is 5%, RTAMM outperforms SWMM in 85.55% of the cases, while SWMM

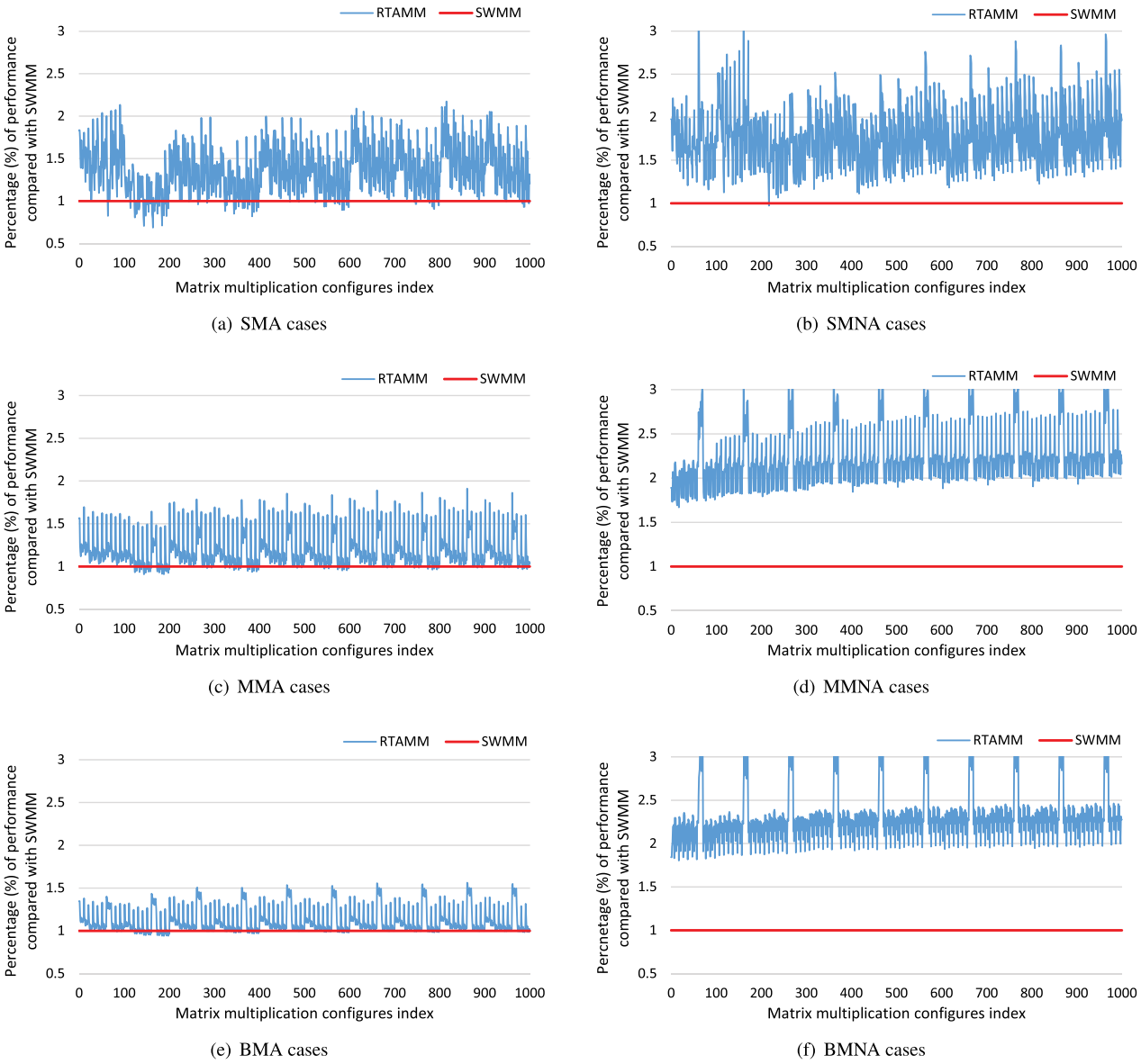


FIGURE 8. Adaptability comparison between RTAMM and SWMM. The X axis indicates the matrix multiplication configuration indexes from 1 to 1000. The Y axis indicates the performance percentage (%) of RTAMM compared with SWMM. The six subfigures are as follows: (a) small-scale matrix multiplication with aligned data; (b) small-scale matrix multiplication with unaligned data; (c) medium-scale matrix multiplication with aligned data; (d) medium-scale matrix multiplication with unaligned data; (e) big-scale matrix multiplication with aligned data; (f) big-scale matrix multiplication with unaligned data.

outperforms RTAMM in only 1.28%. As the performance gap increases to 25%, RTAMM still has better performance in 66.75% of the cases. At this threshold, the percentage of cases where SWMM is superior is only approximately 0.05%. Regarding the 50% performance gap, the number of cases where RTAMM is superior to SWMM is over half of the total, but no benefits can be gained from using SWMM. The results above confirm that RTAMM adapts better to various matrix multiplications.

In summary, Fig. 8 and Table 3 demonstrate that RTAMM has better adaptability and higher performance in most matrix multiplication cases.

C. EVALUATING THE EFFECTIVENESS OF THE ADAPTIVE ENGINE

To verify the effectiveness of the adaptive engine, we manually produce three versions of RTAMM: (i) the static RTAMM which fixes the execution action by an *MNK* nested loop, M2B2 double buffering, and one set of optimal blocking factors; (ii) the dynamic RTAMM which applies the adaptive engine; (iii) the ideal RTAMM which obtains the best performance via testing on all possible actions of the adaptive engine. The whole experiment is built on the 6000 different matrix multiplications described in Section 4.2.

TABLE 3. Quantity statistics of matrix multiplication based on different performance gaps.

Scenes	Performance Gap					
	Weak		Visible		Strong	
	>1.05	<0.95	>1.25	<0.75	>1.5	<0.5
SMA	852	56	616	3	299	0
SMNA	999	0	965	0	765	0
MMA	761	16	237	0	103	0
MMNA	1000	0	1000	0	1000	0
BMA	521	5	187	0	17	0
BMNA	1000	0	1000	0	1000	0
ALL	5133	77	4005	3	3184	0

TABLE 4. Statistics for the effectiveness of the adaptive engine in RTAMM.

Item	Value
Number of total scenes	6000
Max performance improvement (compared with the static RTAMM)	116.13%
Average performance improvement (compared with the static RTAMM)	18.78%
Decision accuracy (compared with the best RTAMM)	97.50%
Average decision overhead	<0.01%

As shown in Table 4, the dynamic RTAMM can achieve a performance improvement of 18.78% on average and 116.13% on maximum compared with the static RTAMM. For the decision accuracy, the dynamic RTAMM can reach 97.5% compared with the ideal RTAMM. Moreover, the decision overhead is so negligible that we can accomplish high-performance execution. In summary, the above statistics demonstrate that the adaptive engine is beneficial for the adaptability of RTAMM and has a negligible overhead.

V. RELATED WORKS

All BLAS operations can be implemented based on a high-performance matrix multiplication, which was proposed by Kågström *et al.* [32]. Many practical scientific applications are closely related to frequent dense linear operations. Because these operations have large computational requirements, the optimization of matrix multiplication is a hot research topic [4]–[7]. For example, Volkov and Demmel [4] proposed a quick implementation method on the G80 architecture, and used a blocking algorithm to exploit locality in shared memory. Nath *et al.* [33] developed the dense linear algebra library MAGMA for the heterogeneous architecture of many-core+GPU, which uses a method similar to double buffering to prefetch data into additional registers instead of shared memory. Lim *et al.* [24] optimized the performance of matrix multiplication on Intel KNL platform with C language based on blocking schemes, data prefetching, loop unrolling and Intel AVX-512. Compared with these works, we find two pivotal directions for optimizing matrix multiplication: the macro-level blocking method and the micro-level computational kernel. The former mainly exploits data locality to balance the computation and data access, while the latter focuses mainly on the efficiency of the bottom hardware execution unit by utilizing high-level languages such as C/C++ or low-level assembly code.

In addition to manual optimization, the automatic adjustment of matrix multiplication is also an important research field. Bilmes *et al.* [23] proposed an early prototype of the automatic matrix multiplication generation system known as PHIPAC. Subsequently, Whaley *et al.* [34] extended the idea of PHIPAC to all other dense matrix kernels of BLAS to form ATLAS. Jiang and Snir [35] developed a matrix multiplication automatic tuning system similar to ATLAS that generates multiple implementation versions based on a parameterized code generator and uses a dedicated search engine to search for the best version. Lim *et al.* [36] proposed a heuristic automatic tuning method to generate the computational kernel for Intel KNL and Intel Skylake-SP processors. The tuning parameters include the register-level/cache-level matrix block size, expected distance and depth of loop unrolling.

For many years, almost all research on matrix multiplication focused on general-purpose processors, such as Intel Xeon/Xeon Phi and NVIDIA GPUs. In contrast, only a few studies [11], [13] have investigated the SW26010 processor which is a rising star in the modern many-core processor domain. Moreover, to achieve the ideal peak performance, these works discussed the performance optimization in only the special case where matrix dimensions are sufficiently large and the multiple of the optimal blocking factors. In this article, the proposed RTAMM methodology can provide the better implementation of matrix multiplication on SW26010. Because of the architectural differences between SW26010 and general-purpose processors (CPU and GPU), the research of matrix multiplication on SW26010 is of great significance for the development of many-core processors.

VI. CONCLUSION

In this article, we propose RTAMM to promote the adaptability and performance of various matrix multiplications for the SW26010 many-core processor. The basic idea is to quantize several fundamental cost formulas for different matrix multiplication algorithms, and then combine them with multiple sets of potentially valuable blocking factors to explore possible execution actions for one matrix multiplication case. Based on the above idea, RTAMM dynamically determines the execution action at runtime. With the state-of-the-art SWMM as the baseline, experiments demonstrate that RTAMM not only achieves competitive peak performance but also has better adaptability for various matrix multiplications.

In the future, we will further study the perfect implementation of matrix multiplication on SW26010 based on the dynamic adaptive methodology.

REFERENCES

[1] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, Mar. 1990.

[2] 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," *Sci. Program.*, vol. 5, no. 3, pp. 173–184, 1996.

[3] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. (2001). *Petsc*. [Online]. Available: http://www.mcs.anl.gov/petsc

[4] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2008, pp. 1–11.

[5] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMM kernels for the Fermi GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 11, pp. 2045–2057, Nov. 2012.

[6] V. Kelefouras, A. Kritikakou, I. Mporas, and V. Kolonias, "A high-performance matrix–matrix multiplication methodology for CPU and GPU architectures," *J. Supercomput.*, vol. 72, no. 3, pp. 804–844, Mar. 2016.

[7] V. Kelefouras, A. Kritikakou, and C. Goutis, "A Matrix–Matrix multiplication methodology for single/multi-core architectures using SIMD," *J. Supercomput.*, vol. 68, no. 3, pp. 1418–1440, Jun. 2014.

[8] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang, "The sunway TaihuLight supercomputer: System and applications," *Sci. China Inf. Sci.*, vol. 59, no. 7, Jul. 2016, Art. no. 072001.

[9] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present and future," *Concurrency Comput., Pract. Exper.*, vol. 15, no. 9, pp. 803–820, 2003.

[10] Z. Xu, J. Lin, and S. Matsuoka, "Benchmarking SW26010 many-core processor," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2017, pp. 743–752.

[11] J. Lin, Z. Xu, A. Nukada, N. Maruyama, and S. Matsuoka, "Optimizations of two compute-bound scientific kernels on the SW26010 many-core processor," in *Proc. 46th Int. Conf. Parallel Process. (ICPP)*, Aug. 2017, pp. 432–441.

[12] J. Lin, Z. Xu, L. Cai, A. Nukada, and S. Matsuoka, "Evaluating the SW26010 many-core processor with a micro-benchmark suite for performance optimizations," *Parallel Comput.*, vol. 77, pp. 128–143, Sep. 2018.

[13] L. Jiang, C. Yang, Y. Ao, W. Yin, W. Ma, Q. Sun, F. Liu, R. Lin, and P. Zhang, "Towards highly efficient DGEMM on the emerging SW26010 many-core processor," in *Proc. 46th Int. Conf. Parallel Process. (ICPP)*, Aug. 2017, pp. 422–431.

[14] Y. Liu, Q. Liao, J. Sun, M. Hu, L. Liu, and L. Zheng, "A heterogeneous parallel genetic algorithm based on SW26010 processors," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun.*, Aug. 2019, pp. 54–61.

[15] D. J. M. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. W. Leong, "A customizable matrix multiplication framework for the intel HARPv2 Xeon+FPGA platform: A deep learning case study," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 107–116.

[16] T. M. Smith, R. V. D. Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. V. Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 1049–1059.

[17] F. D. Igual, M. Ali, A. Friedmann, E. Stotzer, T. Wentz, and R. A. van de Geijn, "Unleashing the high-performance and low-power of multi-core DSPs for general-purpose HPC," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2012, pp. 1–11.

[18] F. Wang, H. Jiang, K. Zuo, X. Su, J. Xue, and C. Yang, "Design and implementation of a highly efficient DGEMM for 64-bit ARMv8 multi-core processors," in *Proc. 44th Int. Conf. Parallel Process.*, Sep. 2015, pp. 200–209.

[19] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 BLAS," *ACM Trans. Math. Softw.*, vol. 35, no. 1, pp. 1–14, Jul. 2008.

[20] X. Su, X. Liao, H. Jiang, C. Yang, and J. Xue, "SCP: Shared cache partitioning for high-performance GEMM," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, pp. 1–21, Jan. 2019.

[21] S. Mittal, "A survey of architectural techniques for improving cache power efficiency," *Sustain. Comput., Informat. Syst.*, vol. 4, no. 1, pp. 33–43, Mar. 2014.

[22] J. A. Gunnels, G. M. Henry, and R. A. Van De Geijn, "A family of high-performance matrix multiplication algorithms," in *Proc. Int. Conf. Comput. Sci.*, 2001, pp. 51–60.

[23] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI c coding methodology," in *Proc. 25th Anniversary Int. Conf. Supercomputing Anniversary*, 2014, pp. 253–260.

[24] R. Lim, Y. Lee, R. Kim, and J. Choi, "An implementation of matrix–matrix multiplication on the intel KNL processor with AVX-512," *Cluster Comput.*, vol. 21, no. 4, pp. 1785–1795, Dec. 2018.

[25] S. Cho, J. Hong, J. Choi, and H. Han, "Multithreaded double queuing for balanced CPU-GPU memory copying," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, Apr. 2019, pp. 1444–1450.

[26] J. Nie, C. Zhang, D. Zou, F. Xia, L. Lu, X. Wang, and F. Zhao, "Adaptive sparse matrix-vector multiplication on CPU-GPU heterogeneous architecture," in *Proc. 3rd High Perform. Comput. Cluster Technol. Conf.*, 2019, pp. 6–10.

[27] R. C. Agarwal, F. G. Gustavson, and M. Zubair, "A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication," *IBM J. Res. Develop.*, vol. 38, no. 6, pp. 673–681, Nov. 1994.

[28] G. C. Fox, S. W. Otto, and A. J. G. Hey, "Matrix algorithms on a hypercube I: Matrix multiplication," *Parallel Comput.*, vol. 4, no. 1, pp. 17–31, Feb. 1987.

[29] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals Superscalar Processors*. Long Grove, IL, USA: Waveland Press, 2013.

[30] Y. Zhang, B. Shu, Y. Yin, Y. Zhou, S. Li, and J. Wu, "Efficient processing of convolutional neural networks on sw26010," in *Proc. Int. Conf. Netw. Parallel Comput.*, 2019, pp. 316–321.

[31] M. Goshima, K. Nishino, Y. Nakashima, S. Mori, T. Kitamura, and S. Tomita, "A high-speed dynamic instruction scheduling scheme for supersealar processors," in *Proc. 34th ACM/IEEE Int. Symp. Microarchitecture.*, 1999, pp. 225–236.

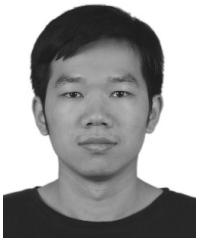
[32] B. Kágström, P. Ling, and C. van Loan, "GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Softw.*, vol. 24, no. 3, pp. 268–302, Sep. 1998.

[33] R. Nath, S. Tomov, and J. Dongarra, "An improved magma GEMM for Fermi graphics processing units," *Int. J. High Perform. Comput. Appl.*, vol. 24, no. 4, pp. 511–515, 2010.

[34] R. Clint Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Comput.*, vol. 27, nos. 1–2, pp. 3–35, Jan. 2001.

[35] C. Jiang and M. Snir, "Automatic tuning matrix multiplication performance on graphics hardware," in *Proc. 14th Int. Conf. Parallel Archit. Compilation Techn.*, 2005, pp. 185–194.

[36] R. Lim, Y. Lee, R. Kim, J. Choi, and M. Lee, "Auto-tuning GEMM kernels on the intel KNL and intel skylake-SP processors," *J. Supercomput.*, vol. 75, no. 12, pp. 7895–7908, Dec. 2019.



ZHENG WU received the B.S. degree in the Internet of Things engineering from the Hefei University of Technology, in 2015. He is currently pursuing the Ph.D. degree in computer science with the University of Science and Technology of China, Hefei, China.

His research interests include computer systems organization, parallel computing, high-performance computing, and artificial intelligence.

MINGFAN LI received the B.S. degree in computer science and technology from the University of Electronic Science and Technology of China, Chengdu, in 2017. He is currently pursuing the M.S. degree with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China.

His research interests include dataflow systems, parallel and distributed computing, and scheduling in heterogeneous environments.

MENGXIAN CHI received the B.E. degree from the University of Science and Technology of China, Hefei, China, in 2014, where he is currently pursuing the Ph.D. degree with the Advanced Computer System Architecture Laboratory, School of Computer Science and Technology.

His research interests include parallel computing, big data in scientific computing, and deep learning acceleration.

LE XU received the B.S. degree in computer science from Xidian University, in 2019. He is currently pursuing the M.S. degree in computer science with the University of Science and Technology of China, Hefei, China.

His research interests include computer architecture, parallel computing, and high-performance computing.

HONG AN received the Ph.D. degree in computer science from the University of Science and Technology of China (USTC), in 2000.

She is currently a Full Professor of computer science and technology and the Director of the Computer System Architecture Laboratory, USTC. She has published more than 150 research papers in international conferences and journals, such as ICS, SC, PPoPP, IPDPS, HPCA, ICPP, HPCC, IJPC, IJPP, and the IEEE TRANSACTIONS. Her major research interests include large-scale parallel computing chip and system structure, high-performance computing, parallel computing, and cognitive computing system for medical imaging.