Enabling Highly Efficient Batched Matrix Multiplications on SW26010 Many-core Processor

LIJUAN JIANG, Chinese Academy of Sciences
CHAO YANG, Peking University
WENJING MA, Chinese Academy of Sciences

We present a systematic methodology for optimizing batched matrix multiplications on SW26010 many-core processor of the Sunway TaihuLight supercomputer. Five surrogate algorithms and a machine learning–based algorithm selector are proposed to fully exploit the computing capability of SW26010 and cope with the sophisticated algorithm characteristics of batched matrix multiplications. Experiment results show that the algorithm selector is able to adaptively choose the appropriate algorithm for various matrix shapes and batch sizes with low overhead and high accuracy. In particular, the optimized batched matrix multiplications can substantially outperform the non-batched version and reach around 84.8% of the performance upper bound.

CCS Concepts: • Mathematics of computing → Mathematical software performance; • Computer systems organization → Multicore architectures;

Additional Key Words and Phrases: Batched matrix multiplication, batched GEMM, many-core architecture, SW26010 processor, Sunway TaihuLight

ACM Reference format:
Lijuan Jiang, Chao Yang, and Wenjing Ma. 2020. Enabling Highly Efficient Batched Matrix Multiplications on SW26010 Many-core Processor. ACM Trans. Archit. Code Optim. 17, 1, Article 3 (March 2020), 23 pages. https://doi.org/10.1145/3378176

1 INTRODUCTION

In the past few years, the batched matrix multiplications have drawn increasingly more attention in both the industry [1, 2] and the academy [8, 19, 30]. With the rapid development of high-performance computing, many-core-based architectures that rely on many lightweight computing cores and a deep memory hierarchy are becoming an important solution in designing modern supercomputers. To exploit the parallelism and near-processor memory of the powerful high-performance computing systems, it is common in application programs to bind many

This work was supported in part by National Key R&D Plan of China (grant no. 2016YFB0200603), Beijing Natural Science Foundation (grant no. JQ18001) and Beijing Academy of Artificial Intelligence (BAAI).

Authors’ addresses: L. Jiang, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China, and University of Chinese Academy of Sciences, Beijing 100049, China; email: lijuan2014@iscas.ac.cn; C. Yang (corresponding author), School of Mathematical Sciences & National Engineering Laboratory for Big Data Analysis and Applications, Peking University, Beijing 100871, China; Peng Cheng Laboratory, Shenzhen 518052, China; email: chao_yang@pku.edu.cn; W. Ma, Institute of Software & State Key Laboratory of Computer Science, Chinese Academy of Sciences, Beijing 100190, China; email: wenjing@iscas.ac.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.
1544-3566/2020/03-ART3
https://doi.org/10.1145/3378176
small operations into a cumulatively large computation. This trend gives rise to the applications of batched matrix multiplications, which can be frequently found in, e.g., quantum chemistry [13], astrophysics [39], metabolic networks [32], computational fluid dynamics [44], domain decomposition solvers [10], tensor computations [45], and deep learning [6, 11]. It has been proven that in these applications, the performance could be greatly improved by exploiting batched computations of small matrix multiplications [8, 19, 37]. As an important extension of the traditional Basic Linear Algebra Subprograms (BLAS) library [20], the new BLAS proposal has already suggested the batched matrix multiplications as an important complement [18]. Recently, vendor-provided libraries such as Intel MKL [1] and NVIDIA cuBLAS [2], and academic research programs such as MAGMA [9], have all added the support of this subroutine. Therefore, it is of great importance to study performance optimization methods for the batched matrix multiplications on state-of-the-art hardware platforms.

As a new generation flagship supercomputer in China, Sunway TaihuLight is built based on the Chinese home-grown heterogeneous many-core processor, SW26010. By aggregating over 40K SW26010 processors, the Sunway TaihuLight supercomputer can deliver an HPL performance of 93 Pflops/s in double precision. Ever since its release in 2016, a number of applications have been ported on Sunway TaihuLight, including atmospheric modeling [51], earthquake simulations [21], and phase field simulations [52], among many others. Previous studies on optimizing dense matrix multiplications for the SW26010 processor have been done in References [31, 36]. Although the dense matrix multiplications can sustain very high performance when the matrices are large enough, such operations with small matrices could barely take full advantage of the underlying hardware features. Instead of being compute-bound, the batched operations are usually memory-bound or have mixed features, thus requiring more subtle works to exploit the data movement capabilities of the processor. In addition, when matrices are small, parallelization and data reuse may play a critical role in improving the sustained performance. Another important thing worth noting here is that due to the large parameter space of the batched matrix multiplications, it is no longer practical to rely on a fixed kernel or several fixed kernels for many different cases. Instead, it becomes necessary to design an adaptive approach to deal with various application scenarios. In this work, we will present a systematic methodology to enable highly efficient batched matrix multiplications on the SW26010 processor of Sunway TaihuLight. The main contributions of this work can be summarized as follows:

- To fully excavate the computing capability of SW26010 and cope with the sophisticated algorithm characteristics of batched matrix multiplications, five surrogate algorithms are designed to put several essential factors, such as methods of parallelization, modes of memory access, and constraints of local device memory (LDM) capacity, into full considerations.
- With these surrogate algorithms at hand, we apply optimization techniques such as double buffering, register blocking, and instruction scheduling, and further design an algorithm selector based on a decision tree machine learning model to automatically choose the suitable algorithm for the batched matrix multiplications with various matrix shapes and batch sizes.
- A series of experiments is carried out to show that the decision tree–based algorithm selector is able to adaptively choose the appropriate surrogate algorithm with low overhead and high accuracy. Through optimizations, the batched matrix multiplications on the SW26010 can substantially outperform the non-batched version and reach around 84.8% of the performance upper bound. For matrix multiplications with large matrix dimensions, up to 93.1% of the peak performance can be sustained.
The remainder of the article is organized as follows. In Section 2, we will introduce some basic information on the hardware features of the SW26010 many-core processor and present some analysis on the various modes of memory access. A framework for batched matrix multiplications on SW26010 is then provided in Section 3, which includes five surrogate algorithms and an algorithm selector. Implementation and optimization details of the five surrogate algorithms are presented in Section 4, following which the algorithm selector is detailed in Section 5. Experimental results to examine and analyze the performance of the algorithm selector and the surrogate algorithms can be found in Section 6. Some related works are summarized in Section 7, and the article is concluded in Section 8.

2 SW26010 MANY-CORE ARCHITECTURE

In this section, we briefly describe the major hardware features of the SW26010 processor based on the Shen Wei (Sunway, abbreviated to SW) architecture [17, 22, 53]. An SW26010 processor is composed of different types of processing elements organized as four core groups (CGs) via a network on chip (NOC), sharing 32 GB memory space. As illustrated in Figure 1, each CG of the SW26010 processor mainly includes four components: a management processing element (MPE), 64 computing processing elements (CPEs), a protocol processing unit (PPU), and a memory controller (MC). 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 MPE and CPE of SW26010 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. The MPE is equipped with two double precision floating-point (FP64) pipelines, each of which can execute 8 FP64 operations per clock cycle. And each CPE has a single FP64 pipeline with the same execution rate, plus an extra pipeline that can facilitate integer operations and register communications. The theoretical peak performance of a CPE cluster is $8 \times 1.45 \times 64 = 742.4$ Gflops/s in double precision. As for data management, the MPE and CPE cluster from the same CG share a unified 8 GB main memory through the memory controller. The MPE has a 32 KB instruction cache, a 32 KB L1 data cache, and a 256 KB L2 cache. Each CPE, however, has a 16 KB instruction cache but no L2 or L3 cache. Instead, each CPE is equipped with a 64 KB scratchpad memory, also called the local device memory (LDM), which
Table 1. Sustainable Memory Bandwidth of the Three Typical DMA Modes Supported by SW26010

| DMA modes | Bandwidth  | Granularity |
|-----------|------------|-------------|
|           | Less than 256 B | 256 B to 512 B | 512 B to 3 KB | Greater than 3 KB |
| PE_MODE   | contiguous  | 15–27 GB/s   | 22–27 GB/s   | 27–28 GB/s   | 27–28 GB/s   |
|           | strided    | 15–21 GB/s   | 23–24 GB/s   | 25–28 GB/s   | 28 GB/s      |
| ROW_MODE  | contiguous  | 23–28 GB/s   | 27–28 GB/s   | 27–28 GB/s   | 27–28 GB/s   |
|           | strided    | 25–26 GB/s   | 26–28 GB/s   | 26–28 GB/s   | 26–28 GB/s   |
| BROW_MODE | contiguous  | 3–7 GB/s     | 9–12 GB/s    | 15–25 GB/s   | 24–25 GB/s   |
|           | strided    | 15–18 GB/s   | 18 GB/s      | 18–21 GB/s   | 18–22 GB/s   |

can be used as either a fast user-controlled cache or a software-emulated cache that achieves automatic data caching.

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. On each CG, the 64 CPEs 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 as low as around several clock cycles. For convenience, we will use CPE(I, J) to represent the CPE located at the Ith row and Jth column of the CPE cluster.

To efficiently transfer data between the main memory and the LDM, the SW26010 processor supports several direct memory access (DMA) modes, including PE_MODE, ROW_MODE, BROW_MODE, BCAST_MODE, and RANK_MODE. In particular, the first three are used in this work. The data distribution strategies of the three DMA modes are summarized as follows:

- The PE_MODE transfers data between the main memory and the LDM of a single CPE.
- The ROW_MODE transfers data between the main memory and the LDM of all the eight CPEs located in the same row of the CPE cluster and requires a synchronization of the eight CPEs in the row. In each ROW_MODE DMA transaction, 128 B data are transferred to/from CPEs in a row, while each CPE gets/puts successive 16 B data.
- The BROW_MODE transfers data between the main memory and the LDM of all the CPEs in one or multiple rows of the CPE cluster and also requires row synchronization. An eight-bit mask defines rows that participate in the operation. In each BROW_MODE DMA transaction, 128 B data are transferred to the DMA engine of the CPE cluster and broadcast to every CPE located in the rows defined by the mask via the on-chip mesh network. By exploiting specific data distribution strategies, one can avoid repeated data transferring from/to the main memory in some application scenarios.

All aforementioned DMA modes of SW26010 require a 128 B data alignment and support both contiguous and strided data access. For each DMA mode, the continuity and the granularity of the data are the major influence factors on the performance. We summarize their effects in Table 1.
One can observe from the table that, in terms of the sustained memory bandwidth, both PE_MODE and ROW_MODE can reach 25–28 GB/s when the granularity is equal to or larger than 512 B. As the granularity is reduced below 512 B and gets smaller, the ROW_MODE is remarkably superior to the PE_MODE. For the PE_MODE, the strided DMA operation is not as good as the contiguous one when the granularity is less than 512 B. For the ROW_MODE, a relatively high performance of around 24–25 GB/s can be achieved only when conducting contiguous DMA operation with granularity of at least 3 KB. These observations will guide the design of implementation algorithms in the sequel.

3 A FRAMEWORK FOR MATRIX MULTIPLICATIONS ON SW26010

In this article, we focus on the batched matrix multiplications, which can be seen as a collection of \( b \) independent dense matrix multiplications in the form of

\[
C^\ell \leftarrow a^\ell A^\ell B^\ell + \beta^\ell C^\ell \quad \ell = 1, 2, 3, \ldots, b,
\]

where \( a^\ell, \beta^\ell \) are scalars, \( A^\ell, B^\ell, \) and \( C^\ell \) are \( m \times k, k \times n, \) and \( m \times n \) dense matrices, respectively. For the special case with batch size \( b = 1 \), the batched operations degenerate to the classic matrix multiplication. There is a number of possible variations of the batched matrix multiplications, considering different aspects such as whether the matrices are in the transposed or non-transposed formats, whether the matrix shapes in the same batch are fixed or variable, and what data precision is supported. In this research, we focus on the most popular case in which the batched matrix multiplications are in double precision with matrices of fixed shapes and stored in non-transposed formats; this is a common choice in a number of studies [7, 26, 47].

To fully excavate the computing capability of SW26010 and cope with the sophisticated algorithm characteristics of batched matrix multiplications, several essential factors, such as methods of parallelization, modes of memory access, and constraints of LDM capacity, are worth thorough considerations. To this end, we derive five surrogate algorithms: PPP1, RBR8, RBR64, RPR64, and SRPR64. We remark that there do exist many other combinations of the DMA data transfer modes, but most of them can be eliminated based on analysis of the bandwidth behavior as depicted in Table 1. As will be demonstrated later, the five algorithms can achieve remarkable performance in a large coverage of matrix shapes. Therefore, we leave the search of other suitable algorithms as a possible future work. Based on the five surrogate algorithms, we further design an algorithm selector to adaptively choose the suitable algorithm for the batched matrix multiplications with various matrix shapes and batch sizes. The whole framework for computing the batched matrix multiplications on SW26010 is shown in Figure 2. When matrix sizes are very small, we use the PPP1 algorithm, in which each thread can perform one or multiple matrix multiplications independently. However, when the matrix sizes are sufficiently large, we use the SRPR64 algorithm with multilevel data blocking to orchestrate the whole memory hierarchy and exploit full mesh parallelism of SW26010. In between the two, there are three other surrogates. It is the job of the algorithm selector to properly select the suitable algorithm based on a low-overhead machine learning approach. To achieve accurate and efficient selection of the proper algorithm surrogate, a decision tree model is designed and trained on the datasets containing features such as blocking factors, LDM usage, and data access granularity.

In Table 2, some more details of the five surrogate algorithms are provided. These algorithms are mainly designed to adapt with various matrix shapes and batch sizes, considering how many threads are truly needed for each matrix multiplication in the batch, and which DMA mode is most appropriate for the data transfer. The selection of the DMA modes also need to cope with different data distribution strategies studied in Section 2 and are properly chosen to coordinate with the data layout of the surrogate algorithms while achieving high bandwidth efficiency. The names of the five surrogate algorithms originate from the initial letters of the selected DMA modes and
Fig. 2. Framework for computing the batched matrix multiplications on SW26010.

Table 2. Detail Configurations of the Five Algorithm Surrogates, Including the Number of Active Threads for Computing a Single Matrix Multiplication and the DMA Modes for Matrices A, B, and C

| Surrogate Algorithm | Num of threads | DMA mode for A            | DMA mode for B                | DMA mode for C                |
|---------------------|----------------|---------------------------|-------------------------------|-------------------------------|
| PPP1                | 1              | contiguous PE_MODE        | contiguous PE_MODE            | contiguous PE_MODE            |
| RBR8                | 8              | contiguous ROW_MODE       | contiguous BROW_MODE          | contiguous ROW_MODE          |
| RBR64               | 64             | contiguous ROW_MODE       | contiguous BROW_MODE          | contiguous ROW_MODE          |
| RPR64               | 64             | contiguous ROW_MODE       | strided PE_MODE               | contiguous ROW_MODE          |
| SRPR64              | 64             | strided ROW_MODE          | strided PE_MODE               | strided ROW_MODE             |

the number of active threads to compute a single matrix multiplication in the batch. In particular, S is used to distinguish where a strided DMA mode is used. When the matrices are very small, the PPP1 algorithm is applied, which simply uses contiguous PE_MODE for all three matrices. For larger matrices, other algorithm surrogates are used, with the number of active threads increased to 8 or 64 to cooperatively compute each matrix multiplication. In this case, with the guideline of the DMA characteristics provided in Table 1, ROW_MODE is the natural choice for matrix A and C. However, matrix B does not fit the data transfer pattern of ROW_MODE; we therefore use the contiguous BROW_MODE and the strided PE_MODE for different data granularity.

4 IMPLEMENTATION AND OPTIMIZATION DETAILS

In all the five algorithm surrogates, the matrices are hierarchically partitioned and distributed on the CPE cluster with proper DMA modes. The parallelism of both independent matrix multiplications and block computations inside an individual matrix multiplication are exploited accordingly. After that, data exchange via the low-latency register communication is carried out to assort with the data layout to assist computation among the CPEs. To further hide the cost of the main memory access, a double buffering method is employed, which has been shown to be effective on GPUs and other many-core accelerators; more details of the double buffering approach can be found in, e.g., Reference [47].

The memory hierarchy of the SW26010 processor can be abstracted into three levels: the main memory, the LDM, and the register. Accordingly, we apply a series of layered approaches for
different matrix shapes to distribute the matrix data on the memory hierarchy. On the first two levels, i.e., the main memory level and the LDM level, we utilize gradually finer blocking schemes in the five surrogate algorithms with the growth of matrix dimensions to increase the efficiencies of data reuse and memory bandwidth. To assort with the blocking strategies, different DMA data transfer modes are chosen and different data sharing schemes are designed. Besides, methods such as register blocking and instruction scheduling are used in the assembly kernels for high floating-point computing efficiency. For each individual algorithm, the specific implementation and optimization techniques we use will be summarized in the following. For ease of description, on the main memory level, we use notation $A_j$ to represent the $j$th column panel of matrix $A$ when a one-dimensional blocking is applied and $A_{ij}$ to represent the $(i, j)$-th block of matrix $A$ when using a two-dimensional blocking, respectively. The blocking factors on the main memory level and the LDM level are denoted as $b_m, b_n, b_k,$ and $p_m, p_n, p_k,$ respectively.

4.1 PPP1
As mentioned earlier, the PPP1 algorithm is designed for matrix multiplications of very small sizes. In this case, there is no necessity to partition matrices on the main memory level and the LDM level; thus, we have $b_m = p_m = m, b_n = p_n = n, b_k = p_k = k,$ which means that no blocking is applied in this case. Due to the simplicity, the batched computations are assigned to the 64 threads within the same CG as evenly as possible. For the assigned computation tasks, each thread iteratively transfers a certain number of matrices from the main memory to its own LDM through the fast DMA of PE_MODE and then accesses the matrices from the LDM for computation. The number of matrix multiplications performed in an iteration is dynamically determined according to the matrix sizes, the LDM capacity, and the load balance. In addition, by applying the double buffering approach, the data transfer of matrix $A, B,$ and $C$ for every next iteration is done in parallel with the computation of the current iteration. The individual matrix multiplication relies on the highly optimized assembly kernels, which are also used by the other four algorithm surrogates. More details about the assembly kernels can be found in Section 4.5.

4.2 RBR8, RBR64, RPR64
If the matrix sizes exceed the LDM capacity of a single CPE, then one should not use the PPP1 algorithm. Three other methods—namely, RBR8, RBR64, and RPR64—are available for intermediate-sized matrices. To reduce data movement between the main memory and the LDM, we apply one-dimensional blocking algorithms on the main memory level for all the three surrogate algorithms. There exist three types of one-dimensional blocking approaches—namely, stationary $A,$ stationary $B,$ and stationary $C$ algorithms [43]—corresponding to blocking along the $n$th dimension, $m$th dimension, and $k$th dimension, respectively. These three data-blocking methods, originally designed for distributed-memory parallel matrix multiplications, are suitable for different conditions considering communication overhead. For SW26010, in addition to data blocking on the main memory level, we also need to consider fine-grained parallelization and data exchange under the constraints of the LDM capacity, which could be substantially different from the distributed-memory computing systems. We choose to use the blocking method along the $n$th dimension and find that it works well with our current algorithm framework; the other two approaches could be similarly implemented if necessary. Hence, all the three algorithms use the same one-dimensional blocking approach along the $n$th dimension on the main memory level for a single matrix multiplication, indicating that $b_m = m$ and $b_k = k.$ The differences among them are the number of concurrent threads, the LDM blocking, and the method of inter-CPE data exchange, depending on the matrix sizes. Based on observations of bandwidth behaviors from Table 2, the ROW_MODE is used for the
Fig. 3. Computation process of the RBR8, RBR64, and RPR64 algorithms. All three algorithms use a one-dimensional algorithm blocking along the nth dimension on the main memory level. On the LDM level, the three algorithms use different numbers of threads and parallel update strategies. The RBR8 algorithm uses 8 threads located in the same row of the CPE cluster and shares no elements. The RBR64 and RPR64 algorithms both use 64 threads, but the RPR64 algorithm uses finer blocking in matrix B and shares the elements of B.

data transfer of A and C, but not for that of B. Instead, the BROW_MODE mode or the PE_MODE mode is used for the data transfer of B.

As illustrated in Figure 3, within an individual matrix multiplication of the three algorithm surrogates, we take matrix A as the resident matrix, which stays in the LDM until not needed. Matrices C and B are logically partitioned into blocks of $m \times b_n$ and $k \times b_n$, respectively. These blocks are laid out as $N$ column panels, where $N = n/b_n$. The blocks of C are updated by the multiplications of matrix A and the corresponding blocks of B on the CPE cluster, which is done in different ways in the three surrogate algorithms and will be detailed later. Block size $b_n$ can vary not only in accordance to the blocking strategies on the LDM level, but also under the constraints of the LDM capacity and the shapes of matrices. Again, the double buffering approach can be used inside each matrix multiplication as well as across different matrix multiplications in the same batch. The whole algorithms of the three surrogates are listed in detail in Algorithm 1.

4.2.1 RBR8. Two DMA modes, ROW_MODE and BROW_MODE, are utilized in the RBR8 algorithm. Data blocks of matrix C are updated by the eight CPEs in a row of the CPE cluster iteratively. Thus, eight independent matrix multiplications are performed by the eight rows of the CPE cluster concurrently. For the update of a single data block of C, data blocking on the LDM level is done for matrix A and the data block of C, with the row blocks mapped to the CPEs located in the same row of the CPE cluster. No further blocking is applied to the data block of matrix B. Thus, when RBR8 is employed, we have $p_m = m/8$, $p_n = b_n$, and $p_k = k$. For the update of the $\ell$th matrix C in the batch, the corresponding matrix A is initially distributed evenly to the CPEs in the $\ell \% 8$th row of the CPE cluster in ROW_MODE, with each CPE getting a sub-block of size $p_m \times k$. In each iteration for updating the data blocks of the $\ell$th matrix C, the CPEs fetch the elements of the same coordinates as matrix A from block $C_j$, where $0 \leq j < n/b_n$, through the fast DMA of ROW_MODE, with each CPE in the $\ell \% 8$th row getting a sub-block of $p_m \times b_n$. The corresponding block $B_j$ is broadcast to the CPEs located in $\ell \% 8$th row via the BROW_MODE DMA method. With this data layout, each CPE loads elements of the local sub-blocks of A and B to update the local sub-block of C by invoking the highly optimized assembly kernels.

4.2.2 RBR64. In the RBR64 algorithm, all 64 CPEs run in parallel to update the data blocks of matrix C. On the LDM level, matrix A and blocks of matrices B and C are evenly divided into
ALGORITHM 1: Framework of the RBR8, RBR64, and RPR64 algorithms with double buffering.

Input: input matrices: $A, B, C$, scalars: $\alpha, \beta$, batch size: $b$
Output: updated matrices: $C$

1 //For RBR8 $initial\_value = \text{row}_id$ and $chunk\_size = 8$, otherwise $initial\_value = 0$ and $chunk\_size = 1$.
2 for $\ell \leftarrow initial\_value$ to $b - 1$ step $chunk\_size$ do
3     load block $B_0^\ell, C_0^\ell$ to the LDM of CPEs;
4     store block $C_{N-2}^{\ell-chunk\_size}$ to the main memory;
5     $C_{N-1}^{\ell-chunk\_size} = \beta C_{N-1}^{\ell-chunk\_size}$;
6     parallel block matrix multiplication:
7     $C_{N-1}^{\ell-chunk\_size} += \alpha (A_{N-1}^{\ell-chunk\_size})(B_{N-1}^{\ell-chunk\_size})$;
8     sync;
9     load matrix $A_0^\ell$ to CPEs;
10    sync;
11    load block $B_1^\ell, C_1^\ell$ to the LDM of CPEs;
12    store block $C_{N-1}^{\ell-chunk\_size}$ to the main memory;
13    $C_0^\ell = \beta C_0^\ell$;
14    parallel block matrix multiplication:
15    $C_0^\ell += \alpha (A_0^\ell)(B_0^\ell)$;
16    sync;
17    for $j \leftarrow 2$ to $N - 1$ do
18        store block $C_{j-2}^\ell$ to the main memory;
19        load block $C_{j-1}^\ell, B_j^\ell$ to the LDM of CPEs;
20        $C_{j-1}^\ell = \beta C_{j-1}^\ell$;
21        parallel block matrix multiplication:
22        $C_{j-1}^\ell += \alpha (A_{j-1}^\ell)(B_{j-1}^\ell)$;
23        sync;
24    end
25 end

eight column strips. Then the column strips of $A$ and $C$ are further divided to row blocks, which
means that $p_m = m/8$, $p_n = n/8$, and $p_k = k/8$. The DMA modes we use in the RBR64 algorithm
are similar to those in RBR8. Since ROW\_MODE transfers data between the main memory and the
LDM of the CPEs in the same row, and matrices are stored in column major, each CPE located
in the same row can load elements of a column strip of the matrices. The data distributions
of matrix $B$, transferred under BROW\_MODE, need to fit the data thread mapping of matrices $A$ and $C$,
transferred under ROW\_MODE, for the sake of correctness. For the update of an individual matrix
$C$, each column strip of matrix $A$ is initially mapped to the corresponding row of the CPE cluster
in ROW\_MODE. CPE($I, J$) fetches a sub-block of $p_m \times p_k$ from the $I$th column strip. In each step for
updating the blocks of matrix $C$, column strips of block $C_j$ are distributed to the CPE cluster in the
same way as that of matrix $A$ under ROW\_MODE, with each CPE getting a sub-block of $p_m \times p_n$. Each
column strip of block $B_j$ is broadcast to the CPEs located in its mapped row under BROW\_MODE.
CPE($I, J$) gets the $I$th column strip, which consists of $k \times p_n$ elements of block $B_j$. The update of
data block $C_j$ is achieved through the update of 64 local sub-blocks distributed in the CPE cluster.
Each thread updates its local sub-block of $C$ and needs eight steps to read the sub-block of $A$ and
then multiply the sub-blocks of $A$ and $B$ by invoking the corresponding assembly kernels. In the $I$th step, threads in the $I$th row of the CPE cluster load elements of matrix $A$ and broadcast to the CPEs in the same column, while other threads receive the elements of $A$ from the corresponding threads. All the threads load elements of matrix $B$ from its local LDM and compute the sub-block multiplication. The computation results of each step are accumulated to their local sub-block of matrix $C$.

4.2.3 RPR64. The RPR64 algorithm is very similar to RBR64. It also uses all the 64 CPEs of the whole CG for the update of the data blocks of matrix $C$. In addition, data distribution methods for matrices $A$ and $C$ as well as the parallelization strategy for the update of the data blocks are also the same. Major differences between the two methods are the data layout of matrix $B$ and the corresponding data exchange scheme. The RBR64 algorithm transfers all the elements of matrix $B$ needed for the computation to the corresponding CPEs before the multiplication in each step, while the RPR64 algorithm applies a further partition to matrix $B$ and requires additional data exchange during the computation. In RPR64, a data block of matrix $B$ is partitioned to eight column strips, and each column strip is further divided to row blocks as well. We thus also have $p_m = m/8$, $p_n = b_n/8$, and $p_k = k/8$. We use the PE.MODE for the data transfer of matrix $B$. And for the sake of correctness, the data distributions of matrix $B$ also need to fit the data thread mapping of matrices $A$ and $C$, and the data should be transferred under ROW.MODE. CPE($I, J$) fetches a sub-block of $p_k \times p_n$ from the $I$th column strip. Each thread updates its local sub-block of matrix $C$, requiring eight steps of data exchange for matrices $A$ and $B$. Sub-block matrix multiplications are done by invoking the highly efficient assembly kernels. In the data exchange process of the $I$th step, threads in the $I$th row of the CPE cluster load elements of matrix $A$ and broadcast to the CPEs in the same column, while other threads receive the elements of $A$ from the corresponding threads. Besides, threads in the $I$th column of the CPE cluster load elements of matrix $B$ and broadcast to the CPEs in the same row, while other threads receive the elements of $B$ from the corresponding threads.

4.3 SRPR64

The SRPR64 algorithm performs matrix multiplications of large sizes that may exceed the LDM capacity when applying the other four surrogate algorithms. It relies on an iterative application of a highly optimized dense matrix multiplication kernel, which was studied in detail in a previous work [31]. For completeness, we only briefly introduce the basic idea about how this algorithm is designed. In each matrix multiplication, a two-dimensional blocking is applied on the main memory level. Matrices $C$, $A$, and $B$ are logically partitioned into blocks of $b_m \times b_n$, $b_m \times b_k$, and $b_k \times b_n$, respectively. In particular, a fixed blocking strategy on the main memory level with $b_m = 128$, $b_n = 256$, and $b_k = 768$ is applied to achieve high performance. Matrix $B$ is taken as the resident matrix, and the blocks of matrix $C$ are updated in an $n$-$k$-$m$ triple-nested loop. Memory access operations for prefetching the data blocks of $C$ and $A$ and storing the updated data blocks of $C$ can proceed in parallel with the computation in the $m$-loop with the help of a double buffering scheme. In each iteration, a data block $C_{i,j}$, and the corresponding data blocks from matrices $A$ and $B$ are loaded to the LDM of each CPE. All the blocks of the three matrices are further divided evenly into $8 \times 8$ sub-blocks on the LDM level and mapped to the 64 CPEs in a one-to-one manner. Hence, we have $p_m = b_m/8$, $p_n = b_n/8$, and $p_k = b_k/8$. Then the update of data block $C_{i,j}$ is accomplished by using a strategy similar to that in the RPR64 algorithm.

4.4 Data Storage Format

There are three typical data storage formats for matrix multiplications: the P2P format, the strided format, and the interleaved format [19, 24, 30]. The P2P format uses arrays whose elements are
Enabling Highly Efficient Batched Matrix Multiplications on SW26010 Many-core Processor 3:11

pointers to memory locations containing matrices, and the pointer arrays are passed as kernel parameters. The P2P format is flexible, allowing users to allocate memory randomly, but needs to transfer pointer arrays from main memory to LDM. The overhead of this data transfer is usually high, taking around 30% of the execution time as suggested by our tests. The strided format initially allocates one large array containing all the matrices in contiguous memory, with a fixed stride that gives the number of memory locations between matrices. The strided format, though requiring a linear memory allocation, can avoid the data transfer of pointer arrays. This format is provided in some popular vendor libraries such as cuBLAS 8.0 [16]. The interleaved format is a permutation of the strided format, which helps enable vectorization when matrix size is very small and is not the multiple of the vectorization length. But this format may not be helpful in this work, because the computation overhead can be mostly overlapped by the memory access in the PPP1 algorithm. Based on the above analysis, we take the strided format as the default matrix storage format throughout this study.

4.5 Assembly Kernels

Highly optimized assembly kernels are crucial to achieving high performance in matrix multiplications [25, 26, 47]. In this work, we implement four basic assembly kernels to deal with various matrix shapes. The four kernels are $16 \times 4, 12 \times 4, 8 \times 4, 4 \times 4$, where $\text{subm} \times \text{subn}$ represents the size of the sub-block of matrix $C$ in the innermost loop of the kernel. Combinations of kernels are performed in accordance to the blocking factor $p_m$, where $p_m = m$ in the PPP1 algorithm, $p_m = 16$ in the SRPR64 algorithm, and $p_m = m/8$ in the other three surrogate algorithms. The computation of a sub-block of matrix $C$ is composed of two parts, corresponding to the updates of the sub-block of matrix $C$ of size $16 \lceil p_m/16 \rceil \times p_n$ and size $(p_m \% 16) \times p_n$, respectively. For the two parts, kernel $16 \times 4$ and kernel $(p_m \% 16) \times 4$ are invoked accordingly. When $p_m \% 16 = 4$, we try to replace the kernel combination of $16 \times 4$ and $4 \times 4$ with that of $12 \times 4$ and $8 \times 4$ whenever allowed, because the efficiency of kernel $4 \times 4$ is relatively low. To fully exploit the computation capability of the SW26010 processor, further optimization techniques, such as register blocking and instruction scheduling, are applied in the assembly kernels; more details can be found in, e.g., Reference [31].

5 ALGORITHM SELECTOR

Due to the existence of different surrogate algorithms for the batched matrix multiplications, it is important to understand the characteristics of each algorithm and design a proper algorithm selector to automatically choose the optimal one to adapt with the variety of application scenarios. In this section, we will first conduct some analysis on how the performance could be influenced in the five surrogate algorithms, and then we will present a low-overhead highly accurate machine learning method based on a decision tree model to help us select the appropriate surrogate algorithm in an efficient way.

5.1 Algorithm Analysis

Since the PPP1 algorithm simply processes small matrix multiplications that can be held in the LDM of a single thread, and the SRPR64 algorithm relies on a fixed blocking strategy for large matrix multiplications [31], the analysis of these two algorithms is relatively transparent. For the other three surrogate algorithms, we focus on analyzing the following major factors:

LDM Usage. The LDM usage can be used to examine whether an algorithm exceeds the constraint of the LDM capacity. It is worth noting that although the increase of the LDM usage might not bring performance improvement, an excessively low LDM usage will definitely cause performance deterioration. For instance, as compared to $RBR8$, the $RBR64$ algorithm uses a finer LDM blocking
and utilizes more CPE resources to compute each matrix multiplication. Unless the LDM usage of RBR64 is large enough, the performance of RBR64 might not be necessarily higher than that of RBR8. Similar analysis applies to RBR64 and RPR64 as well. The LDM usage of the three surrogate algorithms can be calculated as Equations (1) to (3).

\[
ldm_{RBR8} = 1 \times \frac{m}{8} k + 2 \times kb_n + 3 \times \frac{m}{8} b_n, \tag{1}
\]

\[
ldm_{RBR64} = 2 \times \frac{m}{8} k + 2 \times \frac{k b_n}{8} + 3 \times \frac{m b_n}{8}, \tag{2}
\]

\[
ldm_{RPR64} = 1 \times \frac{m}{8} k + 2 \times \frac{k b_n}{8} + 3 \times \frac{m b_n}{8}. \tag{3}
\]

Here, we use the notations \(ldm_{RBR8}, ldm_{RBR64}, \) and \(ldm_{RPR64}\) to represent the LDM space required by the three algorithms, respectively. In terms of memory access cost, matrices \(A\) and \(B\) are read-only, while matrix \(C\) requires both read and write. In the equations, we emphasize the effecting factors of the double buffering technique in each term with numbers followed by “\(\times\)” For example, in the first equation, the factor 1 in the first term means that the double buffering technique is not applied for the data transfer of matrix \(A\), the factor 2 in the second term means that the double buffering technique is applied to overlap the read of matrix \(B\) with computation, and the factor 3 in the third term means that the double buffering technique is applied to overlap both the read and the write of matrix \(C\) with computation, respectively. We remark here that for matrix \(A\), the double buffering technique is only applied in the RBR64 algorithm, but not the other two, based on some experimental verifications.

**Blocking factors.** The blocking factors have significant impact on the performance of matrix multiplications and have been carefully analyzed and tuned in many previous works, such as References [9, 40, 47]. The evaluation of the blocking factors depends on the methods of parallelization and constraints of LDM capacity and can be used to extract important information such as the on-chip data reuse and the amount of data transfer between the main memory and the LDM. An off-line method is used to help automatically find the near-optimal blocking factors of the three surrogate algorithms. The auto-tuning code can perform functionalities such as search space initialization, code variant generation, macro substitution, parameter instantiation, and result collection. Since the three surrogate algorithms are designed to deal with medium-sized matrices, the search space can be simply set to 1 to \(n\). Statistics on the performance of different blocking factors applied to different matrix shapes show that the superior blocking factors for various cases appear to be different. Hence, we apply one-dimensional dynamic blocking strategy to the three surrogate algorithms on the main memory level. In the dynamic blocking strategy, factor \(b_n\) is initialized to \(n\), then decreases by half each time till the maximal LDM usage predicted by each of the Equations (1) to (3) can fit into the 64 KB capacity limit. In general, the algorithm with a larger blocking factor tends to be less affected by the constraint of the LDM capacity and could achieve higher performance.

**Data access granularity.** As discussed earlier, the data access granularity may have a strong influence on the achieved memory bandwidth. The specific memory access methods of the three surrogate algorithms mainly differ in the data transfer scheme of matrix \(B\). We list the data access granularity of matrix \(B\) of the three surrogate algorithms in Equations (4) to (6), denoted as \(mem_{RBR8}, mem_{RBR64}, \) and \(mem_{RPR64}\), respectively.

\[
mem_{RBR8} = 8kb_n, \tag{4}
\]

\[
mem_{RBR64} = kb_n, \tag{5}
\]

\[
mem_{RPR64} = k. \tag{6}
\]
Enabling Highly Efficient Batched Matrix Multiplications on SW26010 Many-core Processor 3:13

Fig. 4. Illustration of the decision tree–based algorithm selector.

Since the RBR8 algorithm uses the contiguous BROW_MODE, if mem_{RBR8} is larger than 3KB, better performance could be achieved. Similar principles can apply to the other two surrogate algorithms. That is, if mem_{RBR64} is larger than 3KB and mem_{RPR64} is larger than 512B, better performance of the corresponding algorithms could be achieved, respectively.

5.2 Decision Tree Model

The five surrogate algorithms are designed for different application scenarios of the batched matrix multiplications. It is of critical importance to find the appropriate surrogate algorithm for the computation of a certain batch size and matrix shapes. More precisely, once the batch size and the input matrix dimensions are known, the values of the blocking factors, the LDM usage, and the data access granularity of each surrogate algorithm can be determined. For square matrices, due to the simplicity of the parameter spaces, there exist clear boundaries of matrix sizes, within which a certain surrogate algorithm could achieve superior performance for batched square matrix multiplications. However, the boundaries are not clear for the other numerous rectangular matrices. We have tried to use a simple analytical model to help us select the appropriate algorithm surrogate but can only achieve around 80% prediction accuracy. Therefore, we abstracted the algorithm selection into a machine learning classification problem, which can be solved by using a decision tree [15], an ANN [49], or a KNN [12] model, among which we chose the decision tree approach due to the relatively low overhead.

In this work, we employ a Cart decision tree model [15], which is a popular supervised machine learning algorithm for both classification and regression problems. Decision tree builds models in the form of a tree structure, whose internal nodes are features, and leaf nodes correspond to labels. The tree model is built by splitting a dataset into smaller and smaller subsets based on a certain criterion. In our tree model, we use the Gini index as the splitting criterion. To find solutions, a decision tree makes sequential and hierarchical decisions along the tree model. In our implementation, a preprocessing is done to calculate the numerical values of the LDM usage and the blocking factors, as well as the logical values for the comparison of the granularities between the data access and the corresponding DMA modes. Then one can use these preprocessed values as the input features of the decision tree model to predict the appropriate algorithm surrogate for a given set of \( m, n, k, \) and \( b \). Note that the SRPR64 algorithm uses a fixed blocking strategy; the performance-related values are the same to all inputs, which are uninformative to the decision tree model. Overall, we take 11 features as the input to the decision tree model, as depicted in Figure 4. The predicted results are the label of the selected algorithm of the five surrogates.
Table 3. Test Results of the Decision Tree–based Algorithm Selector

| Item                  | Value            |
|-----------------------|------------------|
| Number of total samples | 1,886,950        |
| Number of training samples | 1,415,212    |
| Number of testing samples | 471,738        |
| Range of batch size $b$ | [1, 1024]      |
| Range of dimension $m$ | [1, 6144]       |
| Range of dimension $n$ | [1, 6144]       |
| Range of dimension $k$ | [1, 6144]       |
| Prediction accuracy    | 98.9%            |
| Averaged performance loss | 9.2%            |
| Prediction overhead    | 3.63 μs          |

6 EXPERIMENTAL RESULTS

In this section, all experiments are performed on a single CG of the SW26010 processor. Parallelizations across different CGs are usually handled by the users on a higher programming level, which is a common practice in many libraries [3, 4]. The batched matrix multiplication for general dense matrices in double precision—which is the most frequently utilized among several variants—is taken as the focus of all tests. In particular, we use the strided matrix storage format throughout the experiments due to the flexibility and performance of this format. We will examine the performance of the algorithm selector as well as the optimized surrogate kernels and test the effects of different batch sizes and matrix shapes.

6.1 Performance of the Decision Tree Model

The first thing we do in the experiments is to examine the performance of the machine learning model as the algorithm selector, which is, more precisely, an 11-featured decision tree classifier to automatically select the appropriate algorithm surrogate for a given input set of batch size and matrix dimensions. To prepare data samples for the machine learning model, we execute all the five surrogate algorithms and label the fastest one as the classification output for each configuration. Then a preprocessing is done (as introduced in the previous section) so the 11 features, such as the LDM usages, the blocking factors, and the memory access characteristics of the surrogate algorithms, are extracted as the input of the machine learning model. We prepare a dataset of 1,886,950 samples, in which the batch size $b$ varies in the range of [1, 1024] and the matrix dimensions $m$, $n$, and $k$ are set to change within [1, 6144]. We use 75% of data samples as the training dataset and 25% as the testing dataset. Among the testing datasets, 156,470 cases have more than one surrogate that can execute correctly. And compared with the surrogate algorithm with the worst performance, 47.4% average performance gain can be achieved if the surrogate algorithm with the best performance is used. The experiment results, as provided in Table 3, show that the machine learning–based algorithm selector can automatically choose the proper algorithm with an averaged accuracy of 98.9%. For those cases, the algorithm selector chose the wrong algorithms, the averaged performance loss is as low as 9.2%, and the maximal performance loss is below 20%. In addition, the overhead of the decision tree classifier is only 3.63 μs, which is ignorable as compared to the matrix operations.
6.2 Performance with Various Batch Sizes

To show how the batch size affects the performance of batched matrix multiplications, we conduct the experiments for the case of square matrices that $m = n = k$ and change the batch size $b$ for each of the five surrogate algorithms. In each set of the tests, we turn off the algorithm selector to fix the experiments to the target algorithm we want to examine. The test results are drawn in Figure 5. When different algorithms are tested, the ranges of the batch size might be different due to the different behaviors of the algorithms. We can see from the figure that as the batch size increases, each tested case has a common trend of achieving monotonically higher performance and then saturating at some certain number of batch size. In particular, the batch size has the greatest impact on the performance of the PPP1 algorithm, because it performs a simple and direct parallelization of individual matrix multiplications. The influence of the batch size on the RBR8 algorithm is greater than that of the RBR64 algorithm, since the former surrogate exploits a hybrid parallelism of individual matrix multiplications and block computations. We illustrate two test results for the RPR64 algorithm, corresponding to the memory-bound case when $m = n = k = 384$ and the compute-bound case when $m = n = k = 480$, based on the analysis of the roofline model [50].

Our test shows that when the batch size $b$ is small, the performance of the former case is lower than that of the latter, and when $b$ is increased, the former quickly catches up. It suggests that the optimization techniques applied to matrix multiplications of batched forms are more profitable to the memory-bound case than that of the compute-bound ones. We have not shown the effect of batch size on the SRPR64 algorithm in the figure, since the impact is negligible on this kernel.

6.3 Performance with Square Matrices

To further examine the performance with square matrices, we carry out a series of experiments for the five surrogate algorithms and compare the results with the roofline model [50] as the performance upper bound, as well as with that of a non-batched baseline version based on the fine-tuned SRPR64 kernel [31] as the performance lower bound, respectively. Again in the tests the algorithm selector is turned off so the test can be manually configured to different cases. Figure 6
Fig. 6. Performance of batched matrix multiplications with increasingly larger square matrices.

illustrates the test results of batched matrix multiplications with square dimension in the range of [1, 6144]. Because different surrogate algorithms exhibit their own performance advantages in different ranges of matrix dimensions and batch sizes, we calibrate the batch size and the matrix dimensions to be $b = 131072$, $m = n = k \in [1, 32]$ for PPP1; $b = 4096$, $m = n = k \in [48, 144]$ for RBR8; $b = 2048$, $m = n = k \in [160, 256]$ for RBR64; $b = 512$, $m = n = k \in [288, 512]$ for RPR64; and $b = 4$, $m = n = k \in [786, 6144]$ for SRPR64, respectively. It can be observed from the results that the former four surrogate algorithms exhibit significant performance advantage over that of the baseline. For example, when $m = n = k = 128$, the performance improvement is around $7.0 \times$, and when $m = n = k = 64$, the performance improvement is over $100.0 \times$. This is because the sophisticated blocking strategy applied to the SRPR64 algorithm, though working well when the matrix dimensions are large (here, larger than 512), could lead to insufficient parallelism and poor bandwidth with small matrices. The other four surrogate algorithms, on the contrary, can take into account the parallelism of independent matrix multiplications and adequately exploit efficient mixed DMA modes and suitable data blocking in individual matrix multiplications for small matrices. When the matrix dimensions are sufficiently large, i.e., larger than 512 in this test, the proper algorithm for this case is indeed SRPR64, and the results coincide with the baseline with a very slight improvement due to the reduced number of kernel invocations.

As compared to the performance predicted by the roofline model, the five surrogate algorithms are able to achieve on average 84.8% of the performance upper bound. This is particularly remarkable for the memory-bound case when the matrix dimensions are small, thanks to the utilization of mixed DMA modes to transfer data and gradually refined dynamic blocking strategies to improve data reuse. By further measuring the sustained memory bandwidth, we know that for the memory-bound case when using the PPP1, RBR8, RBR64, and RPR64 algorithms with $m = n = k \leq 384$, the averaged memory bandwidth can reach 23.8 GB/s, corresponding to around 85.0% of the measured bandwidth of 28 GB/s. For the compute-bound case when using RPR64 with $m = n = k > 384$ or using SRPR64 with $m = n = k > 512$, the sustained performance incrementally
approaches the estimated performance upper bound, since the overhead of preprocessing and postprocessing of the double buffering method and the partial unhidden local memory access and on-chip communication may occupy a considerable portion of the total execution time, but can get smaller when the computational intensity gets larger. In particular, when the matrix dimension is 6,144, the measured performance can reach 691 Gflops, which is 93.1% of the peak performance.

Given an input case, one might want to see the differences between the best and worst performance that can be achieved with different surrogate algorithms. Hence, we compare the performance among the five surrogate algorithms with several square-shaped matrices under the constraints of the LDM capacity. The test results are illustrated in Figure 7, in which the top/middle/bottom line in the boxplot represents the best/average/worst performance that can be achieved in the corresponding matrix dimension. For instance, the difference between the best and worst performance is 63.6 Gflops for matrix size 128, 171.4 Gflops for matrix size 320, and 74.2 Gflops for matrix size 512, respectively. The results strongly indicate that it is necessary to find the optimal or near-optimal surrogate algorithm for various matrix sizes.

### 6.4 Performance with Rectangular Matrices

In the final part of the experiments, we turn on the algorithm selector and run it for a wide range of matrix dimensions, especially for the rectangular cases, to examine the flexibility of the machine learning model as well as the sustained performance of the presented algorithms. The test results are summarized in Figure 8, in which each row shows the performance of the batched matrix multiplications with one of the matrix dimensions fixed and the other two dimensions varying in the range of $[32, 1024]$ with step length 32. The batch size is set to 512 to rule out the randomness of the test results. From Figure 8, we can make the following observations:

- Fixing one of the three dimensions $m$, $n$, $k$ may lead to quite different performance results, depending on which dimension is fixed. This is because different matrix shapes may result in different blocking factors, LDM usage, and data access granularity, which in turn lead to the invocation of different surrogate algorithms. In particular, when $m = 640$, $n = 1,024$, and $k = 256$, the $RPR64$ algorithm is invoked, and a performance of 556.4 Gflops is achieved.
When \( m = 1,024, n = 256, \text{ and } k = 768 \), the SPRP64 algorithm is invoked, and a performance of 574.8 Gflops is achieved. And when \( m = 256, n = 1,024, \text{ and } k = 768 \), the RPR64 algorithm is invoked again, and a performance of 614.2 Gflops is achieved.

- When the dimension \( k \) is fixed, the highest achievable performance is relatively lower than that when one of the other two dimensions is fixed. As analyzed earlier, the \( k \)th dimension could have a strong influence on the data access granularity, which may cause great impact on the sustained memory bandwidth. In general, the batched matrix multiplications with larger \( k \) tend to achieve higher bandwidth usage and thus higher performance. When \( k \) is fixed to a relatively small number, potential performance degradation may occur.

- When the dimension \( m \) is fixed, the highest achievable performance is relatively higher than that when one of the other two dimensions is fixed. Since we transfer the whole matrix \( A \) to the LDM of the CPE cluster, the increase of dimension \( m \) could strongly affect the blocking factor \( b_n \), especially in the three surrogate algorithms RBR8, RBR64, and RPR64. Even for relatively small \( m \), e.g., \( m = 128 \), the batched matrix multiplications may still have large blocking factors, which helps achieve high performance in practice.

7 RELATED WORK

As a recent proposal of important HPC kernels, the batched matrix multiplications have drawn much attention on topics of kernel interfaces and storage formats, as well as optimization methodologies on multi-core CPUs and GPU-based platforms. For example, Dongarra et al. [19] compared different interfaces and data storage formats for the batched matrix multiplications, which were also discussed by us in Section 4.4. Jhurani et al. [30] proposed an interface by adding a second leading dimension and implemented batched square matrix multiplications of sizes under 16 on
the GPUs. In this work, we have applied a similar interface in our implementation with the strided data storage format. A series of previous studies [16, 33, 35] took full use of the stream technique in the invocation of single matrix multiplications for the concurrent execution on the GPUs and achieved promising performance. But when the matrix dimensions are very small, the cost of these methods could be very high. Abdelfattah and collaborators [7, 8, 37] implemented the batched matrix multiplications of sizes under 16, 32, and 512 on the GPUs, respectively, where different optimization techniques are utilized to enhance the bandwidth efficiency and improve parallelism of the corresponding matrix dimensions. Heinecke et al. [26] presented a code generator to generate micro-kernels and employed the Just-In-Time (JIT) compilation technology to build the required kernel variants at runtime with high efficiency. Masliah et al. [38] made use of advanced C++ features, including templates and overloaded functions, to optimize batched matrix multiplications of up to size 32 with performance portability across multiple architectures of the multi-core CPU and GPU. Li et al. [34] proposed a coordinated tiling and batching framework to utilize different tiling strategies for different matrices in variable batched matrix multiplications on the GPUs, which can help in enhancing thread-level and instruction-level parallelism. All these works have inspired the basic ideas and shed some light on the methodologies in our study.

In addition to the emerging batched matrix multiplications, the traditional non-batched matrix multiplication is still one of the hottest research areas of performance optimizations. Many classic techniques proposed in this research direction could help us design the hardware-oriented, highly optimized compute kernels in this work. For example, a variety of effective blocking methods for matrix multiplications on modern CPU-based computer systems with memory hierarchies were studied by research groups such as Whaley et al. [48], Gunnels et al. [25], and Goto et al. [24]. Also in Reference [48], a widely cited auto-tuning optimization scheme was proposed for matrix multiplications on general-purpose CPUs in the famous ATLAS package. In recent years, increasingly more efforts have been made to optimize matrix multiplications on many-core processors such as GPUs and Intel Xeon Phis. For example, Nath et al. [41] proposed a register-blocking method to mitigate the bandwidth differences between register files and shared memory of the GPUs. Tan et al. [47] 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 factors. On Intel Xeon Phi, Heinecke et al. [27] optimized matrix multiplications for hybrid CPU and Intel Xeon Phi architecture, with jobs assigned to threads for work stealing, and Gepner et al. [23] maximized the reuse of the on-die cache. Despite some similarities in terms of the data blocking and performance model, the specific methods we presented in this work are quite different from those listed above, due in large part to the differences between the underlying hardware platforms as well as the batched nature of the matrix multiplications.

Here, we highlight the work by Abdelfattah et al. [7], which proposes optimized batched matrix multiplications on GPUs and is adopted by the open source library MAGMA [5]. Some basic architecture information of the GPU and comparisons between the architectures of the SW26010 processor and the GPU can be found in the Appendix. In Reference [7], each matrix \( C \) in the batch is partitioned into blocks in a two-dimensional manner. In a kernel invocation of batched matrix multiplications, a three-dimensional grid is launched whose third dimension is the value of batch size. Each independent matrix multiplication has a unique index of the third dimension of the grid. For the update of each matrix \( C \), the partitioned blocks are mapped to the thread blocks with the same index of the third dimension of the grid. Within a thread block, each thread loads partial data and threads share data through the shared memory. This work achieves promising performance on GPUs and provides inspiration to our work, such as performance-aware factors discussed in the article and block size determination methods. However, one cannot simply port the work from MAGMA to SW26010, because the two architectures have substantially different
thread scheduling mechanisms and memory hierarchies. Besides, several architecture-specific features of the SW26010 processor, such as multiple DMA data transfer modes and register communication mechanism, should be exploited thoroughly, which necessitates this study.

8 CONCLUSION

Batched matrix multiplications are playing an increasingly important role in a large variety of applications. There is an urgent need for highly efficient batched matrix multiplications on modern hardware architectures. In this work, we have presented a systematic methodology to enable highly efficient batched matrix multiplications on the SW26010 many-core processor of the Sunway TaihuLight supercomputer. Five surrogate algorithms, along with a machine learning–based algorithm selector, are proposed to fully exploit the computing capability of the SW26010 processor and cope with the sophisticated algorithm characteristics of batched matrix multiplications. We have shown by experiments that the algorithm selector is able to adaptively choose the appropriate algorithm for various matrix shapes and batch sizes with low overhead and high accuracy. And the optimized batched matrix multiplications can substantially outperform the non-batched version and reach around 84.8% of the performance upper bound.

Possible future work could include the extensions of this work to more general cases with, for example, variable batch sizes, different matrix storage formats, and other floating-point precisions. A more careful and more systematic study on other variants of blocking algorithms might also become necessary. Some fast matrix-multiplication algorithms, such as the Strassen’s method [14, 28, 29, 46], might be worth looking into in the future. Some of the ideas presented in this work, such as the adoption of a low-overhead decision tree model for algorithm selection, might be possible to extend to other many-core–based platforms such as the GPUs for similar tasks. From the decision tree model, we are also interested in doing further analysis on the predicted results to gain some performance insights for future study.

APPENDIX

Table 4 illustrates some major differences between the architectures of the SW26010 many-core processor and the NVIDIA Tesla Volta V100 GPU [42]. A GPU is composed of multiple Stream Multiprocessors (SMs). And each SM contains a large number of SIMD execution units for logical and arithmetic operations. Although the architectures of SW26010 and the GPU both obey SIMT (Single Instruction, Multiple Threads) parallel execution mechanism, the GPU has more complicated scheduling structures. Unlike SW26010, which requires launching at most one thread on each CPE, a GPU kernel usually launches much more threads. Each kernel on a GPU is launched as a thread grid, which is composed of a large number of thread blocks, and each thread block could have hundreds of threads. The thread blocks in a grid could be organized in a three-dimensional format. The threads in a thread block could also be organized in a three-dimensional format. Multiple thread blocks could be scheduled on the same SM, and the smallest scheduling unit on an SM is a warp, which defines the smallest unit executing in a lock-stepped fashion. Besides, the GPU has more levels of storage medium than SW26010. The SMs share the device memory and off-chip L2 cache. Each SM has software-writable shared memory and L1 cache. Threads within a thread block could share data through the shared memory. And the maximum number of registers per thread is 255, which are private to each thread.
Table 4. Comparisons between the Architectures of the SW26010 Processor and the V100 GPU

|                                | SW26010                                                                 | Tesla V100 GPU                                                                 |
|--------------------------------|-------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| **Computing units**            | Four core groups form a processor. A core group contains an MPE and 64 CPEs. | One GPU consists of 80 SMs. Each SM has 168 SIMD execution units.             |
| **Memory hierarchy**           | The MPE and the CPEs share 8 GB main memory. Each CPE is configured with 64 KB LDM and 32 256-bit registers. | SMs share 16 GB global memory and 6 MB off-chip L2 cache. Each SM is configured with shared memory of up to 96 KB, and 64K 32-bit register files. The maximum number of registers per thread is 255. |
| **Execution mechanism**        | Obey SIMT execution mechanism. Each thread executes on its own hardware pipelines. | Obey SIMT execution mechanism. A number of thread blocks run on an SM in shift. And a warp composed of 32 threads is the smallest scheduling unit on an SM. |
| **Data transfer methods**      | Use multiple DMA modes to transfer data between the main memory and the LDM. | Use copy engines to transfer data between the host memory and the device memory. |
| **Data sharing schemes**       | Threads could share data through the register communication mechanism on-chip. | Threads within a thread block could share data through the shared memory. |

REFERENCES

[1] Intel Corporation. 2019. https://software.intel.com/en-us/intel-mkl.
[2] NVIDIA Corporation. 2019. https://docs.nvidia.com/cuda/cublas/.
[3] Eigen project. 2019. http://eigen.tuxfamily.org/index.php?title=Main_Page.
[4] LAPACK project. 2019. http://www.netlib.org/lapack/.
[5] MAGMA project. 2019. http://icl.cs.utk.edu/magma/.
[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin et al. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org (2015). http://dx.doi.org/10.1177/1094342010385729
[7] Ahmad Abdelfattah, Marc Baboulin, Veselin Dobre, Jack Dongarra, Christopher Earl, Joel Falcou, Azzam Haidar, Ian Karlin, Tz Kolev, Ian Marsilah et al. 2016. High-performance tensor contractions for GPUs. Proc. Comput. Sci. 80 (2016). Elsevier, 108–118.
[8] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Performance, design, and autotuning of batched GEMM for GPUs. In Proceedings of the International Conference on High Performance Computing. Springer, 21–38.
[9] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2017. Novel HPC techniques to batch execution of many variable size BLAS computations on GPUs. In Proceedings of the International Conference on Supercomputing. ACM, 5:1–5:10.
[10] Emmanuel Agullo, Luc Giraud, and Mawussi Zounon. 2015. On the resilience of parallel sparse hybrid solvers. In Proceedings of the International Conference on High Performance Computing. IEEE, 75–84.
[11] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints, abs/1605.02688 (2016).
[12] Naomi S. Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. Amer. Statist. 46, 3 (1992), 175–185.
[13] Alexander A. Auer, Gerald Baumgartner, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Siriram Krishnamoorthy, Sandhya Krishnan et al. 2006. Automatic code generation for many-body electronic structure methods: The tensor contraction engine. Molec. Phys. 104, 2 (2006), 211–228.
[14] Austin R. Benson and Grey Ballard. 2015. A framework for practical parallel fast matrix multiplication. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Vol. 50. ACM, 42–53.
[15] Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone. 1984. Classification and regression trees. Belmont, CA: Wadsworth International Group (1984). https://doi.org/10.1201/9781315139470
[16] Cris Cecka. 2017. Pro Tip: cuBLAS Strided Batched Matrix Multiply. Retrieved from https://devblogs.nvidia.com/cublas-strided-batched-matrix-multiply/.

[17] Jack Dongarra. 2016. Sunway TaihuLight supercomputer makes its appearance. Nat. Sci. Rev. 3, 3 (2016), 265–266.

[18] Jack Dongarra, Iain Duff, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J. Higham, Jonathon Hogg, Pedro Valero-Lara, Samuel D. Relton, Stanimire Tomov et al. 2016. A proposed API for batched linear algebra subprograms. Manchester Institute for Mathematical Sciences, University of Manchester (2016). http://eprints.ma.man.ac.uk/2464/.

[19] Jack Dongarra, Sven Hammarling, Nicholas J. Higham, Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon. 2017. The design and performance of batched BLAS on modern high-performance computing systems. Proc. Comput. Sci. 108 (2017), 495–504.

[20] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. 1990. A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. 16, 1 (1990), 1–17.

[21] Haohuan Fu, Conghui He, Bingwei Chen, Zekun Yin, Chenguo Zhang, Wenqiang Zhang, Tingjian Zhang, Wei Xue, Weiguo Liu, Wanwang Yin et al. 2017. 18.9-Pflops nonlinear earthquake simulation on Sunway TaihuLight: Enabling depiction of 18-Hz and 8-meter scenarios. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2:1–2:12.

[22] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao et al. 2016. The Sunway TaihuLight supercomputer: System and applications. Sci. China Inf. Sci. 59, 7 (2016), 072001.

[23] Pawel Gepner, Victor Gamayunov, David L. Fraser, Ludovic Sauge, Damien Declat, and Mathieu Dubois. 2014. Evaluation of DGEMM implementation on Intel Xeon Phi coprocessor. J. Comput. 9, 7 (2014), 1566–1571.

[24] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. 34, 3 (2008), 12.

[25] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. 2001. A family of high-performance matrix multiplication algorithms. In Proceedings of the International Conference on Computational Science. Springer, 51–60.

[26] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press, 84:1–84:11.

[27] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G. Shet, George Chrysos, and Pradeep Dubey. 2013. Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel Xeon Phi coprocessor. In Proceedings of the Parallel and Distributed Processing Symposium (IPDPS’13). IEEE, 126–137.

[28] Jianyu Huang, Leslie Rice, Devin A. Matthews, and Robert A. van de Geijn. 2017. Generating families of practical fast matrix multiplication algorithms. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium. IEEE, 656–667.

[29] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. 2016. Strassen’s algorithm reloaded. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press, 59.

[30] Chetan Jhurani and Paul Mullowney. 2015. A GEMM interface and implementation on NVIDIA GPUs for multiple small matrices. J. Parallel Distrib. Comput. 75 (2015), 133–140.

[31] Lijuan Jiang, Chao Yang, Yulong Ao, Wanwang Yin, Wenjing Ma, Qiao Sun, Fangfang Liu, Rongfen Lin, and Peng Zhang. 2017. Towards highly efficient DGEMM on the emerging SW26010 many-core processor. In Proceedings of the International Conference on Parallel Processing (ICPP’17). IEEE, 422–431.

[32] Ali Khodayari, Ali R. Zomorrodi, James C. Liao, and Costas D. Maranas. 2014. A kinetic model of Escherichia coli core metabolism satisfying multiple sets of mutant flux data. Metab. Eng. 25 (2014), 50–62.

[33] Xiuhong Li and Yun Liang. 2016. Efficient kernel management on GPUs. In Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium, 85–90.

[34] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. ACM, 229–241.

[35] Yun Liang, Huynh Huynh, Kyle Rupnow, Rick Siow, Goh, and Deming Chen. 2014. Efficient GPU spatial-temporal multitasking. IEEE Trans. Parallel Distrib. Syst. 25 (2014), 748–760.

[36] James Lin, Zhigeng Xu, Akira Nukada, Naoya Maruyama, and Satoshi Matsuoka. 2017. Optimizations of two compute-bound scientific kernels on the SW26010 many-core processor. In Proceedings of the International Conference on Parallel Processing (ICPP’17). IEEE, 432–441.

[37] Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, Joël Falcou, and Jack Dongarra. 2016. High-performance matrix-matrix multiplications of very small matrices. In Proceedings of the European Conference on Parallel Processing. 659–671.
Enabling Highly Efficient Batched Matrix Multiplications on SW26010 Many-core Processor 3:23

[38] Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, Joël Falcou, and Jack Dongarra. 2019. Algorithms and optimization techniques for high-performance matrix-matrix multiplications of very small matrices. *Parallel Comput.* 81 (2019), 1–21.

[39] O. E. Messer, J. Austin Harris, Suzanne Parete-Koon, and Merek A. Chertkov. 2012. Multicore and accelerator development for a leadership-class stellar astrophysics code. In *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing*. Springer-Verlag, 92–106.

[40] Naohito Nakasato. 2011. A fast GEMM implementation on the Cypress GPU. *ACM SIGMETRICS Perf. Eval. Rev.* 38, 4 (2011), 50–55.

[41] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2010. An improved MAGMA GEMM for Fermi graphics processing units. *Int. J. High Performance Computing Applications* 24, 4 (2010), 511–515.

[42] NVIDIA. 2017. NVIDIA Tesla V100 GPU architecture. [https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf](https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf).

[43] Martin D. Schatz, Robert A. Van de Geijn, and Jack Poulson. 2016. Parallel matrix multiplication: A systematic journey. *SIAM J. Sci. Comput.* 38, 6 (2016), C748–C781.

[44] S. J. Sherwin and G. E. Karniadakis. 2005. Spectral/hp element methods for computational fluid dynamics. *Oxford Sci. Public.* 17 (2005), 18.

[45] Yang Shi, Uma Naresh Niranjan, Animashree Anandkumar, and Cris Cecka. 2016. Tensor contractions with extended BLAS kernels on CPU and GPU. In *Proceedings of the International Conference on High Performance Computing (HiPC’16)*. IEEE, 193–202.

[46] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numer. Math.* 13, 4 (1969), 354–356.

[47] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM, 35:1–35:11.

[48] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 1–27.

[49] Halbert White. 1992. *Artificial Neural Networks: Approximation and Learning Theory*. Blackwell Publishers, Inc.

[50] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009).

[51] Chao Yang, Wei Xue, Haohuan Fu, Hongtao You, Xinliang Wang, Yulong Ao, Fangfang Liu, Lin Gan, Ping Xu, Lanning Wang, et al. 2016. 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 6:1–6:12.

[52] Jian Zhang, Chunbao Zhou, Yangang Wang, Lili Ju, Qiang Du, Xuebin Chi, Dongsheng Xu, Dexun Chen, Yong Liu, and Zhao Liu. 2016. Extreme-scale phase field simulations of coarsening dynamics on the Sunway TaihuLight Supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 4:1–4:12.

[53] Fang Zheng, Hong-Liang Li, Hui Lv, Feng Guo, Xiao-Hong Xu, and Xiang-Hui Xie. 2015. Cooperative computing techniques for a deeply fused and heterogeneous many-core processor architecture. *J. Comput. Sci. Technol.* 30, 1 (2015), 145–162.

Received August 2019; revised October 2019; accepted January 2020