Automatically Generating High-performance Matrix Multiplication Kernels on the Latest Sunway Processor

Xiaohan Tao
txh_0119@126.com
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China

Yu Zhu
i_zhuyu@126.com
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China

Boyang Wang
w_boyang1997@163.com
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China

Jinlong Xu
longkaizh@126.com
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China

Jianmin Pang
jianmin_pang@126.com
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China

Jie Zhao
yaozhujiajie@gmail.com
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China

ABSTRACT

We present an approach to the automatic generation of efficient matrix multiplication code on the latest Sunway processor, which will be employed by the next-generation machine of Sunway TaihuLight, one of the fastest supercomputers on earth. The method allows users to write simple C code and automatically generates high-performance matrix multiplication kernels. It uses polyhedral transformations to implement rapid compute decomposition, data exchanges across memory hierarchy and memory latency hiding. An assembly routine is finally integrated into the generated kernels. While achieving up to 90.14% of the theoretical peak performance, our method surpasses a highly tuned library by 9.44%. Compared with existing techniques, our approach reduces the software development life cycle to generate efficient matrix code from months to seconds. We also take into account batched matrix multiplication and some fusion patterns for deep learning (DL), outperforming the library-based implementations by 1.30× and 1.67×.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; • **Computer systems organization** → *Parallel architectures.*

KEYWORDS

polyhedral compilation, matrix multiplication, basic linear algebra subprograms, Sunway TaihuLight, high-performance computing

ACM Reference Format:
Xiaohan Tao, Yu Zhu, Boyang Wang, Jinlong Xu, Jianmin Pang, and Jie Zhao. 2022. Automatically Generating High-performance Matrix Multiplication Kernels on the Latest Sunway Processor. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3545008.3545031

1 INTRODUCTION

Sunway TaihuLight [6] had ever crowned the TOP500 list four times and remains one of the fastest supercomputers. It is used to execute high-performance computing applications [5] and DL workloads [7]. These programs involve massive linear algebra operations, among which general matrix multiplication (Gemm) is of the leading importance. The Linpack benchmark used to rank supercomputers also relies heavily on the efficient implementation of Gemm to solve linear equations. These together make the effective execution of Gemm supreme for the Sunway systems.

The Sunway supercomputer family implements the Sunway architecture, which cannot directly use existing BLAS (basic linear algebra subprograms) libraries for CPU/GPU [8, 19]. Some manual efforts incurring high engineering cost [11, 12] or tedious tuning overhead [7] have been devoted to the SW26010 processor [6], but these approaches are not portable to the latest processor, SW26010Pro [29], that will be adopted by the next generation of the Sunway supercomputers. A practical compilation approach to generate efficient Gemm kernels for SW26010Pro is still missing.

However, automating the generation of high-performance Gemm code is not straightforward, even those approaches for general-purpose chips had to expose the architectural information to the programming model [26] or only generated the innermost loop after Gemm compute decomposition [17], leaving data movements across memory hierarchy and memory latency hiding as a cumbersome duty of the programmer. This situation is further exacerbated by the Sunway architecture, whose potential can be fully exploited only when the programming challenges [25] are well modeled.

We prefer to believe that both the programmer and the vendor of a supercomputer expect a compilation tool to manage the more complex tasks including compute decomposition, data movements and the overlap between computation and communication, *etc.*, but

ICPP '22, August 29-September 1, 2022, Bordeaux, France
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545031

also to be of the ability to collaborate with the highly optimized routines provided by the vendor. The automatically generated Gemm code can thus achieve promising performance, and the human efforts can also be maximally saved. In fact, there already exists prior work for CPU/GPU [1, 13, 18] doing so by integrating the assembly or micro kernels of Gemm into the compilation framework: the high-level loop transformations and memory managements are modeled using an optimizing compiler, and low-level optimizations are implemented by hand and packed in an inline function call.

Inspired by this, we present an automatic code generation approach for Gemm on SW26010Pro. Unlike existing techniques that require user annotations [26], our method allows the programmer to write a naïve 3D loop nest of Gemm code using a general-purpose language, thus simplifying the programmability issue. The user code is lowered to the *schedule tree* [9], a widely used internal representation of the polyhedral model [20, 22], on top of which systematically analysis and transformations are performed.

We use the *isl* library [21] to determine the parallelism and tilability of the 3D loop nest of Gemm, and perform compute decomposition through hierarchical tiling: on the basis of classical tiling as implemented in existing polyhedral compilers [4, 22], we strip-mine [14] the reduced loop dimension of the Gemm code to allow for follow-up memory optimizations. We analytically model the best tile sizes by considering the shape configuration of the assembly micro kernel, avoiding tedious tuning overhead [2, 24]. Data movements across the memory hierarchy are also implemented on top of schedule trees. In addition to performing memory promotions like the GPU case [22], we also hide the memory access latency through software pipelining and double buffering, automating the complex optimization criteria described in [25].

The Gemm code is finally decomposed into a micro kernel, of which the accessed matrix elements can be stored in faster scratchpad memory. Low-level optimizations like data distribution from the scratchpad memory to registers, loop unrolling, instruction scheduling and vectorization should be performed within this micro kernel, which are non-trivial to model in polyhedral compilation [15, 18]. Instead of producing a naïve implementation of this micro kernel, we tailor the optimized schedule tree to generate an invocation of an inline assembly function that has been optimized by the vendor.

The seamless integration between the polyhedral model and inline assembly instructions makes it possible to model the fusion between linear algebra operations not considered by the manual approaches [11, 12]. In particular, we consider the fusion patterns between Gemm and element-wise operations that happen frequently in DL, widening the applicability of the approach.

In summary, our work makes the following contributions:

- This paper is the first work introducing how to systematically generate high-performance Gemm code on SW26010Pro [29], and it also offers insights into Gemm code generation for other heterogeneous supercomputers.
- Our method requires no extra programming efforts [26] or tedious tuning heuristics [7, 24], significantly simplifying the programmability issue.
- We generalize memory latency hiding previously pursued in domain-specific compilers [20, 28] and implement it for the broader context of polyhedral compilation.
- While achieving up to 90.14% of the theoretical performance, our compiler greatly reduces the cost [11, 12] to generate efficient Gemm code on Sunway systems.

Our approach generates the code executable on SW26010Pro. The results demonstrate that our code outperforms a BLAS library *x*Math [10] by 9.44% and effectively make use of the SW26010Pro processor. We also obtain 1.30× and 1.67× speedups for batched Gemm and fusion patterns over the *x*Math-based implementations.

2 BACKGROUND AND OVERVIEW

There exist different variants of Gemm depending on the precision of the matrix elements. We consider double-precision Gemm or Dgemm [11, 17] that executes the operation $C = \alpha(A \times B) + \beta C$, where α and β are coefficients, and A, B, C are matrices of double-precision elements with sizes $M \times K$, $K \times N$ and $M \times N$. We study Dgemm because the Linpack benchmark uses double-precision matrices to rank the supercomputers, and other Gemm variants share the same structure with Dgemm [17]. There are no fundamental reasons impeding our approach from being applied to other Gemm variants.

2.1 The Sunway Architecture

The Sunway architecture is a heterogeneous system adopted by Sunway TaihuLight [6], which organizes its 10,649,600 cores into 40 cabinets, with each made of four super nodes. Each super node includes 256 Sunway processors, connected to others through the system interface. A Sunway processor includes multiple clusters or core groups, the communication between which is delivered by the network on chip. Each cluster is composed of one management processing element (mpe) and 64 compute processing elements (cpes).

All levels of the compute hierarchy except the clusters employ a non-uniform memory access or distributed system. The communications between clusters and higher levels can be implemented using the MPI protocol. Due to the well-nested compute structure and simple access manners of Gemm, one can gradually break down a Gemm routine into independent smaller ones until each piece can be handled by a cluster. Writing MPI messages will thus not incur too much engineering cost. We study Gemm code generation for each cluster that requires more complex memory managements. Note that automatically generating MPI code using the polyhedral model is also possible [3]; integrating this approach into our compiler is not difficult and we leave it as the future work.

A cluster connects its own 16GB DDR4 memory space to its single mpe and the 8×8 cpe mesh using a memory controller. The 64 cpes work together in an asynchronous manner. mpe with a two-level Cache memory hierarchy is usually used for communications, though it can also execute code inefficiently. Each cpe manages a software-controlled data scratchpad memory (spm) and multiple registers. The current TaihuLight machine uses the SW26010 processor [6] composed of four clusters, but SW26010Pro [29] that will be adopted on the next-generation machine can include six. The architecture of SW26010Pro is depicted in Fig.1.

The reasons that make prior methods [11, 12] ineffective are two-folded. First, the memory size of a cpe's spm has increased to 256 KB [29], making manually defined tile sizes [11] and data movements not optimal. Second, the cpes can exchange register data on SW26010, but they are allowed to share larger spm data tiles through a Remote Memory Access (RMA) mechanism on SW26010Pro, which aggravates the difficulty to develop manual approaches.

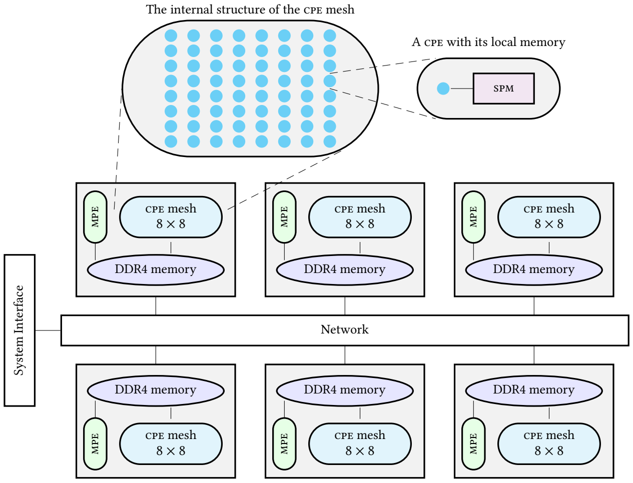

The internal structure of the CPE mesh
A CPE with its local memory
SPM
MPE
CPE mesh 8 × 8
DDR4 memory
Network
System Interface

Figure 1: The architecture of the SW26010Pro processor.

2.2 Polyhedral Compilation

We leverage the polyhedral model [4, 22] to implement an automatic GEMM code generator for the Sunway architecture and address the issues encountered by manual approaches. The polyhedral model, a mathematical abstraction to reason about loop transformations and memory optimizations using integer sets and affine relations [21], can take as input the 3D loop nest of GEMM code shown in Fig.2a and represent it as a schedule tree depicted in Fig.2b.

The root is referred to as a *domain* node, expressed as a group of integer sets with each representing an individual statement. Nested loops are denoted using a so-called *band* node, with which GEMM compute decomposition is performed through loop tiling. A dynamic statement execution can be instantiated using a tuple composed of the loop iteration values, and an affine relation between them can be used to express the *schedule* of each statement enclosed in the domain node. An affine relation between a statement instance and an array can be inspected as a read or write *access relation*, depending on the accessed manner. A *sequence* node can be introduced to enforce the sequential execution order between statements that are enclosed in the same basic block, which contributes a scalar dimension to the schedule tuple composed of the loop iteration variables. An *extension* node is used to introduce statements not covered by the domain node, usually used to insert auxiliary statements for data movements and synchronizations. Many other node types are also supported; we invite the readers to refer to the schedule tree publication [9] for more details.

We take GEMM code generation for GPU in PPCG [22], which also relies on *isl* [21] to build an initial schedule tree as shown in Fig.2b. While *isl* does not change the structure of the original schedule tree, it automatically attaches two attributes to the band node that indicating the outer two loops are parallelizable and the 3D loop nest is tilable. A rectangular tiling with sizes 32 × 32 × 32 is performed, resulting in the schedule tree shown in Fig.2c, with the band node split into two variants, the outer iterating between tiles and the inner iterating within a tile.

The GPU thread/block parameters are introduced to replace their corresponding expressions in the band nodes (Fig.2d), and an extension node for implementing memory promotion statements from global memory to shared memory is inserted into the schedule tree in Fig.2e. Each affine relation within the extension node represents a data copy statement, whose domain, (d_0, d_1, d_2), carries the outer dimensions after tiling while the range, $readA[d_3, d_4]$ or $readB[d_3, d_4]$, is the memory footprint allocatable on shared memory.

```
for i in [0, M) and j in [0, N) and k in [0, K)
  C[i, j] = C[i, j] + A[i, k] * B[k, j] /* S1 */
```

(a) A 3D loop nest of GEMM code.

DOMAIN: $\{S_1(i, j, k) : 0 \leq i < M \wedge 0 \leq j < N \wedge 0 \leq k < K\}$
  BAND: $[S_1(i, j, k) \rightarrow (i, j, k)]$

(b) The initial schedule tree.

DOMAIN: $\{S_1(i, j, k) : 0 \leq i < M \wedge 0 \leq j < N \wedge 0 \leq k < K\}$
  BAND: $[S_1(i, j, k) \rightarrow (\lfloor \frac{i}{32} \rfloor, \lfloor \frac{j}{32} \rfloor, \lfloor \frac{k}{32} \rfloor)]$
    BAND: $[S_1(i, j, k) \rightarrow (i - 32\lfloor \frac{i}{32} \rfloor, j - 32\lfloor \frac{j}{32} \rfloor, k - 32\lfloor \frac{k}{32} \rfloor)]$

(c) The tiled schedule tree with sizes 32 × 32 × 32.

DOMAIN: $\{S_1(i, j, k) : 0 \leq i < M \wedge 0 \leq j < N \wedge 0 \leq k < K\}$
  BAND: $[S_1(i, j, k) \rightarrow (blockIdx.y, blockIdx.x, \lfloor \frac{k}{32} \rfloor)]$
    BAND: $[S_1(i, j, k) \rightarrow (threadIdx.y, threadIdx.x, k - 32\lfloor \frac{k}{32} \rfloor)]$

(d) The schedule tree with GPU block and thread parameters.

DOMAIN: $\{S_1(i, j, k) : 0 \leq i < M \wedge 0 \leq j < N \wedge 0 \leq k < K\}$
  BAND: $[S_1(i, j, k) \rightarrow (blockIdx.y, blockIdx.x, \lfloor \frac{k}{32} \rfloor)]$
    EXTENSION: $[(d_0, d_1, d_2) \rightarrow readA[d_3, d_4]; (d_0, d_1, d_2) \rightarrow readB[d_3, d_4]]$
      BAND: $[S_1(i, j, k) \rightarrow (threadIdx.y, threadIdx.x, k - 32\lfloor \frac{k}{32} \rfloor)]$
        SEQUENCE:
          FILTER: $\{ readA[d_3, d_4] \}$
          FILTER: $\{ readB[d_3, d_4] \}$
          FILTER: $\{ S_1(i, j, k) \}$

(e) The schedule tree with shared memory promotion statements.

Figure 2: GEMM code and its schedule trees for GPU.

The transformations on schedule trees can be borrowed to develop our approach, but we have to refine the process for our target for compute decomposition and data movements from the main memory of an SW26010Pro's cluster to the SPM of a CPE. Besides, implementing RMA and memory latency hiding using schedule trees were not studied before; how the optimized schedule tree can work with the inline assembly instructions should also be addressed.

2.3 Overview of the Approach

Our approach takes as input GEMM code written in C language and represents it as a schedule tree, and the *isl* library is used to determine the parallelism and tilability. Compute decomposition (§3) realized through tiling and strip-mining is first performed to break down the input GEMM code. We also consider the batched GEMM pattern that takes place frequently in DL models.

The compute decomposition results in local array references that can be allocated on the SPM of a CPE, which is computed within the polyhedral model and delivered to implement the DMA (Direct Memory Access) and RMA mechanisms. DMA (§4) refers to data movements from the main memory of SW26010Pro to the SPMs;

RMA (§5) is implementing data communication within the CPE mesh. Memory latency hiding (§6) is realized, with the generated schedule tree scanned to produce abstract syntax tree (AST) using *isl*. Finally, an assembly micro kernel is integrated into the code generator (§7), with fusion patterns for DL models also considered.

3 COMPUTE DECOMPOSITION

Compute decomposition should break down the GEMM code into smaller independent blocks such that (1) the 8×8 CPEs can work on them in parallel, and (2) each of the resulted blocks fits the shape configuration of the micro kernel. The parallelization of the outer two loops can be implemented as explained in §2.2. The difficulty is to find a group of optimal tile sizes.

Before we proceed to the next step, we first isolate the batched dimension from the combined band node when given a batched GEMM code, since we choose not to decompose the batch dimension. We may obtain a schedule tree as shown in Fig.3, for which we isolate the b dimension from the remaining. Our approach for GEMM is still applicable to the second band node. Without loss of generality, we focus on the discussion of GEMM in the following context.

```
DOMAIN: $\{S_1(b, i, j, k) : 0 \le b < B \wedge 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$
    BAND: $[S_1(b, i, j, k) \rightarrow (b)]$ /* batch dimension is isolated. */
        BAND: $[S_1(b, i, j, k) \rightarrow (i, j, k)]$ /* This band represents a GEMM loop nest. */
```

Figure 3: Isolate the batch dimension of batched GEMM.

3.1 Tiling All Dimensions

Compute decomposition is achieved by performing loop tiling. One can perform rectangular tiling along each dimension of the 3D GEMM code as illustrated in Fig.2. However, the tile size selection issue has not yet been modeled by *isl* or other polyhedral tools. Instead, existing approaches [4, 20, 22] resort to tedious auto-tuners to search optimal tile sizes. A practical tuning heuristic is vital for general-purpose compilers, but analytically modeling is sufficient for GEMM code generation [16]. The objective of our analytical model is to match the shape configuration of the assembly micro kernel. As will be introduced in §7.2, the micro kernel is configured as $64 \times 64 \times 32$, which results in one output matrix tile C_τ of size 64×64 and two input matrix tiles A_τ of size 64×32 and B_τ of size 32×64 for each CPE. As the GEMM code is only tiled once, $64 \times 64 \times 32$ can be used as the tile sizes, producing the schedule tree in Fig.4a.

```
DOMAIN: $\{S_1(i, j, k) : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$
    BAND: $[S_1(i, j, k) \rightarrow (\lfloor \frac{i}{64} \rfloor, \lfloor \frac{j}{64} \rfloor, \lfloor \frac{k}{32} \rfloor)]$
        BAND: $[S_1(i, j, k) \rightarrow (i - 64\lfloor \frac{i}{64} \rfloor, j - 64\lfloor \frac{j}{64} \rfloor, k - 32\lfloor \frac{k}{32} \rfloor)]$
```

(a) The schedule tree after tiling.

```
DOMAIN: $\{S_1(i, j, k) : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$
    BAND: $[S_1(i, j, k) \rightarrow (Rid, Cid, \lfloor \frac{k}{32} \rfloor)]$
        BAND: $[S_1(i, j, k) \rightarrow (i - 64\lfloor \frac{i}{64} \rfloor, j - 64\lfloor \frac{j}{64} \rfloor, k - 32\lfloor \frac{k}{32} \rfloor)]$
```

(b) The schedule tree with CPE mesh parameters.

Figure 4: Tiling and introduce CPE mesh parameters.

Now we introduce the CPE mesh parameters for hardware binding. Similar to Fig.2, we substitute the first two integer divisions in the outer band node using *Rid* and *Cid* ($0 \le Rid, Cid < 8$), which represent the row and column index variables of the 8×8 CPE mesh. The result is shown in Fig.4b. For batched GEMM, the subtree rooted at the band node of the batch dimension is used to generate the code executed by the CPE mesh, which still preserves the above hardware binding and iterates the batch dimension in a CPE, reducing the frequency of synchronizations, as will be demonstrated in §8.3.

3.2 Strip-mining the Reduced Dimension

Our tiling strategy produces the matrix tiles with the expected sizes of the target micro kernel. These matrix tiles will be promoted using DMA (§4). However, the micro kernel only computes a partial result when the size of the reduced dimension K is greater than 32, since each C_τ is the accumulation of all products along the reduced dimension, *i.e.*, $A_{64 \times K} \times B_{K \times 64}$. Fig.5 illustrates the distribution of matrix elements on the memory hierarchy of an SW26010Pro's cluster, where we assume $\alpha = \beta = 1$ for the sake of simplicity.

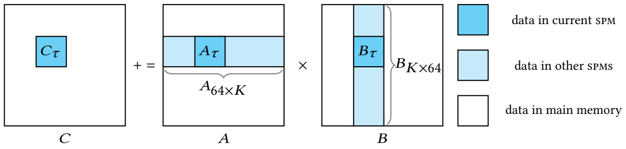


Figure 5: Data distribution across the memory hierarchy.

A possible solution is to promote $A_{64 \times K}$ and $B_{K \times 64}$ to the SPM. Each CPE along the same mesh row or column will keep the same copy of $A_{64 \times K}$ or $B_{K \times 64}$, incurring a great waste of the faster SPM memory. Worse yet, such a buffering strategy will make the decomposition strategy incompatible with the micro kernel, whose shape configuration has been optimized to maximize the utilization of SPM. The allocatable space for the output matrix tile on each SPM is reduced with the increasing buffered sizes of input matrix tiles, compelling a CPE to work on a working set smaller than C_τ.

Fortunately, SW26010Pro allows for the RMA communication of SPM data between CPEs, which we use to address the aforementioned issue. The CPE mesh provides three communication manners as will be introduced in §5, among which we leverage the row/column-broadcast RMA mechanism to share the SPM data along the same row/column of the current CPE. Each CPE can still buffer a size of 64×64 tile of the output matrix and 64×32 tiles of the input matrices on its own SPM to preserve the compatibility with the assembly micro kernel, which in turn is executed in a sequential manner along the reduced dimension. The low-latency RMA communications can be introduced before each execution of the micro kernel to guarantee that the correct A_τ and B_τ are ready on the SPM.

As the movements of 64 input matrix tiles from main memory to SPMs is performed in parallel through DMA, every eight A_τ's/B_τ's along the horizontal/vertical direction are simultaneously buffered. We thus strip-mine the reduced dimension K to enforce the sequential communication of A_τ/B_τ along the horizontal/vertical direction. The resulted schedule tree is shown in Fig.6.

Strip-mining [14] does not involve loop permutation and is thus always valid. To perform strip-mining, we need to isolate the reduced dimension from the combined band node representing the tile loops. This step has also been shown in Fig.6.

DOMAIN: $\{S_1(i, j, k) : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$
    BAND: $[S_1(i, j, k) \rightarrow (Rid, Cid)]$ /* This band is mapped to the 2D CPE mesh. */
        BAND: $[S_1(i, j, k) \rightarrow (\lfloor \frac{k}{256} \rfloor)]$
            BAND: $[S_1(i, j, k) \rightarrow (\lfloor \frac{k}{32} \rfloor - 8\lfloor \frac{k}{256} \rfloor)]$
                BAND: $[S_1(i, j, k) \rightarrow (i - 64\lfloor \frac{i}{64} \rfloor, j - 64\lfloor \frac{j}{64} \rfloor, k - 32\lfloor \frac{k}{32} \rfloor)]$

Figure 6: Strip-mine the reduced dimension by a factor of 8.

4 AUTOMATING DMA COMMUNICATION

The *athread* programming model for SW26010Pro provides a suite of communication interfaces for DMA that exchanges data between the main memory and each SPM of the CPE. The two non-blocking interfaces we use are dma_iget and dma_iput, whose syntax is:

dma_iget (*void *dst*, *void *src*, *int size*, *int len*, *int strip*, *int *reply*)
dma_iput (*void *dst*, *void *src*, *int size*, *int len*, *int strip*, *int *reply*)

where only the interface names are different. *dst* and *src* represent the destination and source addresses of the DMA message. For dma_iget, *dst* expresses the starting address of the matrix tile in SPM and *src* is the starting address of the matrix in the main memory; or vice versa for dma_iput. *size* denotes the total size of the transferred data. *len* and *strip* will be explained later. *reply* is a signal indicating the completion of the DMA transfer. It is always initialized as zero and increases by one each time a DMA message is launched. The following pair of statements

reply = 0;
dma_wait_value (&*reply*, 1);

always appear before and after a non-blocking DMA message to guarantee that the transferred data is always ready before accessed. dma_iget and dma_iput can also be invoked without the *strip* argument. We will introduce the difference later together with the meaning of this argument.

Inserting a foreign statement in schedule trees is realized by extension nodes. We can follow the implementation in PPCG [22] to determine the correct position where an extension node should be introduced, and emitting an instruction for this DMA message is straightforward using the pretty-print strategy. The difficulty is to compute the values for each argument of the DMA syntax, which can be inferred from the affine relation passed to an extension node.

To make use of extension nodes, we need to provide an affine relation like $[d_0, d_1, d_2] \rightarrow readA[d_3, d4]$ as shown in Fig.2e. It can be inspected as the relation between a compute tile $[d_0, d_1, d_2]$ and the read matrix tile footprint $readA[d_3, d_4]$. In our case, the domain of this affine relation can be instantiated using $[Rid, Cid, \lfloor \frac{k}{32} \rfloor]$, as depicted in Fig.4b. d_3 and d_4 should be affine functions of Rid, Cid and $\lfloor \frac{k}{32} \rfloor$. They form the rectangular shape of the tiled memory footprint. As a result, data copying for a matrix is implemented within a nest of two loops in PPCG [22]. The bounds and strip of each data copying loop can be inferred from this affine relation. This can be borrowed by our work, but we only need to determine the lower bound of each data copying loop, because they will be used to instantiate *dst* and *src* of the DMA communication interfaces.

We thereafter assume *dst* is the starting address in SPM and *src* is the starting address in the main memory. We always assign one *reply* value for each DMA message. In addition to simplifying the implementation, this also allows for the independent scheduling of individual DMA messages, which benefits the double buffering strategy in §6. As each matrix tile is promoted to the SPM of a CPE, we can safely deliver the address of $local\_Matrix[0][0]$ to *dst*, where the prefix *local* represents the buffered address in SPM and *Matrix* can be instantiated using *A*, *B* or *C*.

The compute decomposition strategy makes each CPE compute a smaller GEMM compute tile with shape of $64 \times 64 \times 32$. All of the CPEs are distributed into an 8×8 mesh organization. Each CPE mesh thus executes a GEMM kernel of $512 \times 512 \times 256$. We suppose that *Matrix* is of size $X \times Y$. The sizes of *Matrix* executed by the CPE mesh and a CPE are represented as $\widehat{X} \times \widehat{Y}$ and $X_\tau \times Y_\tau$. How these parameters are instantiated is depicted on the right of Fig.7.

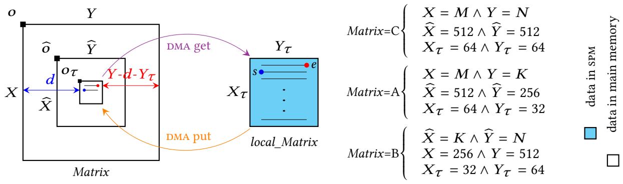

o
Y
$\widehat{o}$
$\widehat{Y}$
o_τ
d
Y-d-Y_τ
X
$\widehat{X}$
Matrix
DMA get
DMA put
Y_τ
X_τ
s
e
local_Matrix
Matrix=C $X = M \wedge Y = N$, $\widehat{X} = 512 \wedge \widehat{Y} = 512$, $X_\tau = 64 \wedge Y_\tau = 64$
Matrix=A $X = M \wedge Y = K$, $\widehat{X} = 512 \wedge \widehat{Y} = 256$, $X_\tau = 64 \wedge Y_\tau = 32$
Matrix=B $\widehat{X} = K \wedge \widehat{Y} = N$, $X = 256 \wedge Y = 512$, $X_\tau = 32 \wedge Y_\tau = 64$
data in SPM
data in main memory

Figure 7: The DMA mechanism.

A dma_iget message copies $X_\tau \times Y_\tau$ matrix elements from the main memory to SPM, *i.e.*, *size* should be set as equal to $X_\tau \times Y_\tau$. These matrix elements can be continuously addressed in SPM, but they are gathered from non-continuous locations in main memory. Fortunately, every Y_τ elements in the same row are still continuous, and *len*, which is used to represent the length of this continuous address. We thus set *len* using Y_τ. The *strip* argument of the DMA interfaces is used to represent the strip value between the ending point *e* of the previous row to the starting point *s* of the next; it can be omitted in more general scenarios but is mandatory in GEMM case. Suppose that the distance from the start of the row that *s* resides in to *s* be *d*. The distance from *e* to the end of the row where *e* locates can be expressed as $Y - d - Y_\tau$. *strip* should therefore be the sum of these two distances, $Y - Y_\tau$.

All of the above arguments are constants with respect to Rid, Cid and $\lfloor \frac{k}{32} \rfloor$. We now deduce the relation between the global coordinate of *src* or o_τ in main memory. To achieve this, we first compute the relative position of the matrix tile $X_\tau \times Y_\tau$ within $\widehat{X} \times \widehat{Y}$ that starts at $\widehat{o}$, whose address should in turn be measured as an offset from *o*, the starting address $Matrix[0][0]$ of the whole matrix $X \times Y$ in the main memory. The global coordinate of each matrix element including o_τ is indexed as $Matrix[x][y]$ in the GEMM code, where x, y should be instantiated using i, j or k depending on which *Matrix* represents. r and c can thus be expressed as:

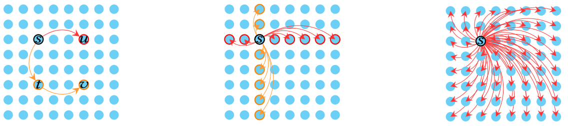
$$r = \widehat{X}\left\lfloor \frac{x}{\widehat{X}} \right\rfloor + X_\tau \left\lfloor \frac{x}{X_\tau} \right\rfloor, c = \widehat{Y}\left\lfloor \frac{y}{\widehat{Y}} \right\rfloor + Y_\tau \left\lfloor \frac{y}{Y_\tau} \right\rfloor \quad (1)$$

where r and c denote the relative row and column indexes of x and y with respect to $\lfloor \frac{x}{X_\tau} \rfloor$ and $\lfloor \frac{y}{Y_\tau} \rfloor$, which will be instantiated using Rid, Cid or $\lfloor \frac{k}{32} \rfloor$ depending on what *Matrix* represents. The address of $Matrix[r][c]$ is passed to *src*. The final dma_iget is

dma_iget(&$Matrix[0][0]$, &$Matrix[r][c]$, $X_\tau \times Y_\tau$, Y_τ,$Y - Y_\tau$, &*reply*)

5 IMPLEMENTING RMA BROADCAST

SW26010Pro provides three RMA manners. Point-to-point (Fig.8a) is used to communicate between two CPEs. It sends the SPM data of *s* to *u* along the same row, or it must resort to a transit point *t* to deliver the message from s to v. Row/Column-wise broadcast (Fig.8b) shares the SPM data of s to those CPEs along the same row/column, and the row and column broadcasts can take place simultaneously. The third manner (Fig.8c) broadcasts the SPM data of s to every other CPE in the mesh, which is internally implemented using the combination of row and column broadcasts.



(a) Point to Point. (b) Row/column broadcast. (c) All broadcast.

Figure 8: The communication manners between CPEs.

The RMA interfaces for row/column-wise broadcast in the athread programming model are `rma_row_ibcast` and `rma_col_ibcast`, whose syntax can be written as:

`rma_row_ibcast`(*void* *dst, *void* *src, *int* $size$, *int* *$reply_s$, *int* *$reply_r$)

`rma_col_ibcast`(*void* *dst, *void* *src, *int* $size$, *int* *$reply_s$, *int* *$reply_r$)

These RMA interfaces are with the same set of arguments but the interface names are different. *dst* and *src* still represent the destination and source addresses of a RMA message, and *size* is also used to denote the total size of the transferred data. *len* and *strip* are no longer required for RMA because the buffered matrix elements in SPM are always continuous. Unlike `dma_iget` or `dma_iput`, `rma_row_ibcast` and `rma_col_ibcast` require two signal indicators−$reply_s$ and $reply_r$: the former increases by one each time an RMA data is sent out by the current CPE, and the latter increases by one when an RMA data is successfully received by the current CPE. The RMA interfaces also implement non-blocking ones, whose correctness is ensured by

$reply_s = 0$;
$reply_r = 0$;
$synch()$;
`rma_wait_value` ($\&reply_s$, 1);
`rma_wait_value` ($\&reply_r$, 1);

One may notice that a synchronization statement is also introduced before launching an RMA message. This is required by the athread programming model that can also be used to execute other types of parallelism on a CPE mesh.

The affine relation of the extension node in Fig.2e is still applicable, whose constraints imply the shape of the tile delivered by RMA. As each of them requires the starting address of the buffered matrix tile in its own SPM, one can pass $local\_Matrix[0][0]$ to both of them, but they will be instantiated by different matrices through double buffering (§6.3) to guarantee the semantic. The value of *size*, $X_\tau \times Y_\tau$, can be easily inferred as the matrix in SPM is continuously addressed. The generated row RMA instruction looks like

`rma_row_ibcast`($\&local\_Matrix[0][0]$, $\&local\_Matrix[0][0]$, $X_\tau \times Y_\tau$, $\&reply_s$, $\&reply_r$)

and the column RMA broadcast is with the same set of argument values. Note that each CPE has its own SPM space for *local_Matrix*. The first *local_Matrix* is the starting address in each receiver CPE, and the second is that of the sender. The schedule tree after inserting extension nodes for DMA and RMA is depicted in Fig.9. The helper line is used to show the alignment between two distant filter nodes.

```
DOMAIN: $\{S_1(i, j, k) : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$
 BAND: $[S_1(i, j, k) \rightarrow (Rid, Cid)]$
  EXTENSION: $[(d_0,d_1,d_2) \rightarrow getC(d_3,d_4)/get\_replyC()]$
  EXTENSION: $[(d_0,d_1,d_2) \rightarrow putC(d_3,d_4)/put\_replyC()]$
   SEQUENCE:
  ┌ FILTER:$\{ getC(d_3, d_4) \} \otimes$ FILTER:$\{ get\_replyC() \}$
  │ FILTER:$\{ S_1(i, j, k) \}$
  │  BAND: $[S_1(i, j, k) \rightarrow (\lfloor \frac{k}{256} \rfloor)]$
  │   EXTENSION: $[(d_0,d_1,d_2) \rightarrow getA(d_3,d_4)/get\_reply\_A()]$
  │   EXTENSION: $[(d_0,d_1,d_2) \rightarrow getB(d_3,d_4)/get\_reply\_B()]$
  │    SEQUENCE:
  │     FILTER:$\{ getA(d_3, d_4) \} \oplus$ FILTER:$\{ get\_replyA() \}$
  │     FILTER:$\{ getB(d_3, d_4) \} \oplus$ FILTER:$\{ get\_replyB() \}$
  │     FILTER:$\{ S_1(i, j, k) \}$
  │      BAND: $[S_1(i, j, k) \rightarrow (\lfloor \frac{k}{32} \rfloor - 8\lfloor \frac{k}{256} \rfloor)]$
  │       EXTENSION: $[(d_0,d_1,d_2,d_3) \rightarrow rbcastA(d_4,d_5)/rbcast\_replyA()]$
  │       EXTENSION: $[(d_0,d_1,d_2,d_3) \rightarrow cbcastB(d_4,d_5)/cbcast\_replyB()]$
  │        SEQUENCE:
  │         FILTER:$\{ rbcastA(d_4, d_5) \} \oplus$ FILTER:$\{ rbcast\_replyA() \}$
  │         FILTER:$\{ cbcastB(d_4, d_5) \} \oplus$ FILTER:$\{ cbcast\_replyB() \}$
  │         FILTER:$\{ S_1(i, j, k) \}$
  │          BAND: $[S_1(i, j, k) \rightarrow (i - 64\lfloor \frac{i}{64} \rfloor, j - 64\lfloor \frac{j}{64} \rfloor, k - 32\lfloor \frac{k}{32} \rfloor)]$
  └ FILTER:$\{ putC(d_3, d_4) \} \otimes$ FILTER:$\{ put\_replyC() \}$
```

Figure 9: Insert extension nodes for DMA and RMA.

We use two affine relations for each DMA or RMA instruction. For example, the `dma_iget` interface is introduced into the schedule tree using $(d_0, d_1, d_2) \rightarrow getC(d_3, d_4)$ and $(d_0, d_1, d_2) \rightarrow get\_replyC()$. The latter is used to generate the `dma_wait_value` statement and the former is used to producing the `dma_iget` together with the initialization instruction of the reply indicator. We put there filter nodes connected using ⊕ or ⊗ in one row to indicate they should be scheduled together, but their separation will be used to implement memory latency hiding in §6. A_τ is only required by the CPEs along the same row while B_τ has to be broadcast to all CPEs along the same column. The 4D domain of each affine relation for the RMA interfaces is due to the strip-mining of the reduced dimension as explained in §3.2. The extension nodes for output matrix tile C_τ are introduced outside the reduced dimension, and those for DMA are inserted between the outer and inner dimensions of loop k. Such transformations of the schedule tree maximize the reuse of C_τ and ensure the correctness of both DMA and RMA communications.

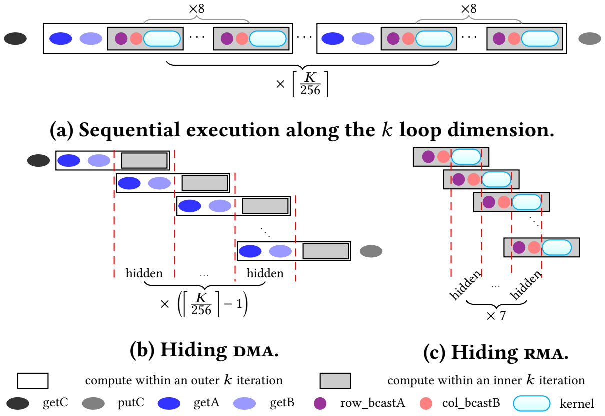
6 MEMORY LATENCY HIDING

Another purpose of the helper line in Fig.9 is used to express the sequential execution within each CPE, which is a mixture of DMA communications, RMA broadcasts and inline assembly kernels. We use Fig.10a to illustrate this sequential execution.

6.1 Software Pipelining

Each CPE has to iterate $\lceil \frac{K}{256} \rceil$ times over the outer k dimension, since it buffers 256 matrix elements along this direction each time the data is transferred by DMA communication. As C_τ is reused within the k loop, its latency cannot be hidden by software pipelining. However, the DMA communications of A_τ and B_τ of the $(x + 1)$-th iteration can be hidden behind the execution of a gray box of the x-th $(0 \le x < \lceil \frac{K}{256} \rceil - 1)$ iteration, leading to a total number of $\lceil \frac{K}{256} \rceil - 1$ overlaps between DMA communications and computation. The hiding mechanism is depicted in Fig.10b. The overhead of each overlapped part is the greater one between the execution time of the gray box and that of the DMA communications. The movements of A_τ and B_τ can take place simultaneously.

The second level pipelining happens between RMA and a cyan box in Fig.10c. All RMA broadcasts except those of the first iteration can be hidden. As we strip-mine the k loop using a factor of eight, the total number of the overlaps between each pair of RMA broadcasts and the inline assembly kernel should be seven, and the latency of each overlap is determined by the heavier one of its two components. The broadcasts of A_τ and B_τ can be launched together.


×8
×8
$\times \left\lceil \frac{K}{256} \right\rceil$
hidden
...
hidden
$\times \left(\left\lceil \frac{K}{256} \right\rceil - 1 \right)$
hidden
...
hidden
× 7
compute within an outer k iteration
compute within an inner k iteration
getC
putC
getA
getB
row_bcastA
col_bcastB
kernel

(a) Sequential execution along the k loop dimension.

(b) Hiding DMA.

(c) Hiding RMA.

Figure 10: The two-level memory latency hiding strategy.

6.2 Loop Peeling

We implement the software pipelining on top of schedule trees. We always schedule the signal indicator together with its corresponding message interface in Fig.9, but now they can be separated by loop peeling that isolates the first and last iterations of both outer and inner k loops after strip-mining. We define each ⊕ in Fig.9 is separable but ⊗ not. The separated reply indicators are moved after the computation of the next iteration, as shown in Fig.11.

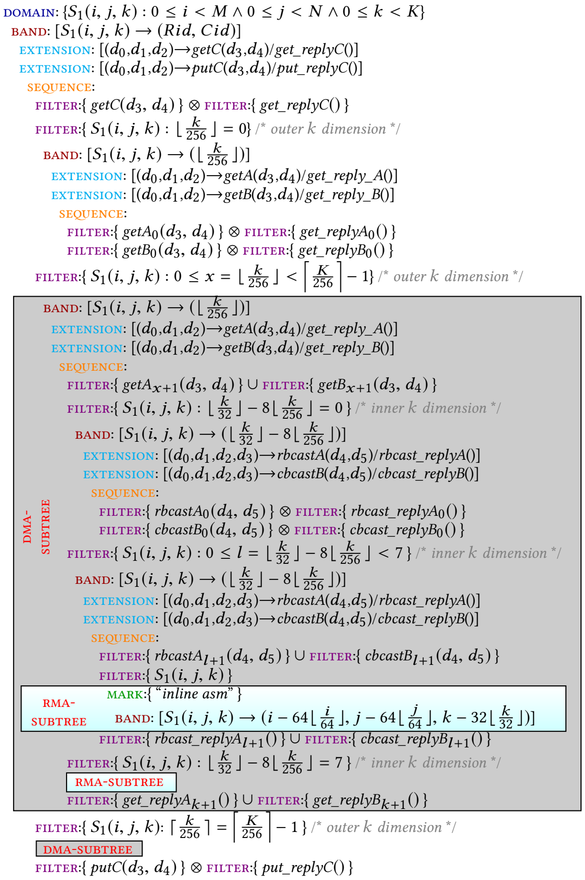
We introduce subscripts to each DMA communication/reply indicator and the execution within the k loop. The inner loop is renamed as l to distinguish from the outer one. Each pair of filter nodes related using ∪ can be executed in parallel. DMA and RMA statements with subscript zero can also be launched simultaneously. The shaded boxes correspond to those in Fig.10. DMA-SUBTREE/RMA-SUBTREE is used to to substitute its replicated presences.

6.3 Double Buffering

The price to pay for enabling the software pipelining is the doubled numbers of local buffers in SPMs. Both A_τ and B_τ have to be buffered twice, and they take part in both levels for hiding DMA and RMA. We allocate four local buffers for them, leading to a total number nine local buffers. Declaring these local buffers in the generated code is implemented by following the approach to allocate shared memory spaces for GPU in PPCG [22].

7 CODE GENERATION

Fig.11 can be scanned to generate the code executable on SW26010Pro. We separate code generation into two phases, with AST first produced using the functionality of *isl* and the athread syntax next printed in a file other than that the main function resides in.



```
DOMAIN: $\{S_1(i, j, k) : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$
 BAND: $[S_1(i, j, k) \to (Rid, Cid)]$
  EXTENSION: $[(d_0,d_1,d_2) \to getC(d_3,d_4)/get\_replyC()]$
  EXTENSION: $[(d_0,d_1,d_2) \to putC(d_3,d_4)/put\_replyC()]$
   SEQUENCE:
    FILTER:$\{ getC(d_3, d_4) \}$ ⊗ FILTER:$\{ get\_replyC() \}$
    FILTER:$\{ S_1(i, j, k) : \lfloor \frac{k}{256} \rfloor = 0\}$ /* outer k dimension */
     BAND: $[S_1(i, j, k) \to (\lfloor \frac{k}{256} \rfloor)]$
      EXTENSION: $[(d_0,d_1,d_2) \to getA(d_3,d_4)/get\_reply\_A()]$
      EXTENSION: $[(d_0,d_1,d_2) \to getB(d_3,d_4)/get\_reply\_B()]$
       SEQUENCE:
        FILTER:$\{ getA_0(d_3, d_4) \}$ ⊗ FILTER:$\{ get\_replyA_0() \}$
        FILTER:$\{ getB_0(d_3, d_4) \}$ ⊗ FILTER:$\{ get\_replyB_0() \}$
    FILTER:$\{ S_1(i, j, k) : 0 \le x = \lfloor \frac{k}{256} \rfloor < \left\lceil \frac{K}{256} \right\rceil - 1\}$ /* outer k dimension */
     BAND: $[S_1(i, j, k) \to (\lfloor \frac{k}{256} \rfloor)]$
      EXTENSION: $[(d_0,d_1,d_2) \to getA(d_3,d_4)/get\_reply\_A()]$
      EXTENSION: $[(d_0,d_1,d_2) \to getB(d_3,d_4)/get\_reply\_B()]$
       SEQUENCE:
        FILTER:$\{ getA_{x+1}(d_3, d_4) \}$ ∪ FILTER:$\{ getB_{x+1}(d_3, d_4) \}$
        FILTER:$\{ S_1(i, j, k) : \lfloor \frac{k}{32} \rfloor - 8\lfloor \frac{k}{256} \rfloor = 0 \}$ /* inner k dimension */
         BAND: $[S_1(i, j, k) \to (\lfloor \frac{k}{32} \rfloor - 8\lfloor \frac{k}{256} \rfloor)]$
          EXTENSION: $[(d_0,d_1,d_2,d_3) \to rbcastA(d_4,d_5)/rbcast\_replyA()]$
          EXTENSION: $[(d_0,d_1,d_2,d_3) \to cbcastB(d_4,d_5)/cbcast\_replyB()]$
           SEQUENCE:
DMA-        FILTER:$\{ rbcastA_0(d_4, d_5) \}$ ⊗ FILTER:$\{ rbcast\_replyA_0() \}$
SUBTREE     FILTER:$\{ cbcastB_0(d_4, d_5) \}$ ⊗ FILTER:$\{ cbcast\_replyB_0() \}$
        FILTER:$\{ S_1(i, j, k) : 0 \le l = \lfloor \frac{k}{32} \rfloor - 8\lfloor \frac{k}{256} \rfloor < 7 \}$ /* inner k dimension */
         BAND: $[S_1(i, j, k) \to (\lfloor \frac{k}{32} \rfloor - 8\lfloor \frac{k}{256} \rfloor)]$
          EXTENSION: $[(d_0,d_1,d_2,d_3) \to rbcastA(d_4,d_5)/rbcast\_replyA()]$
          EXTENSION: $[(d_0,d_1,d_2,d_3) \to cbcastB(d_4,d_5)/cbcast\_replyB()]$
           SEQUENCE:
            FILTER:$\{ rbcastA_{l+1}(d_4, d_5) \}$ ∪ FILTER:$\{ cbcastB_{l+1}(d_4, d_5) \}$
            FILTER:$\{ S_1(i, j, k) \}$
RMA-         MARK:$\{$ "inline asm" $\}$
SUBTREE       BAND: $[S_1(i, j, k) \to (i - 64\lfloor \frac{i}{64} \rfloor, j - 64\lfloor \frac{j}{64} \rfloor, k - 32\lfloor \frac{k}{32} \rfloor)]$
            FILTER:$\{ rbcast\_replyA_{l+1}() \}$ ∪ FILTER:$\{ cbcast\_replyB_{l+1}() \}$
        FILTER:$\{ S_1(i, j, k) : \lfloor \frac{k}{32} \rfloor - 8\lfloor \frac{k}{256} \rfloor = 7 \}$ /* inner k dimension */
         RMA-SUBTREE
        FILTER:$\{ get\_replyA_{k+1}() \}$ ∪ FILTER:$\{ get\_replyB_{k+1}() \}$
    FILTER:$\{ S_1(i, j, k): \lceil \frac{k}{256} \rceil = \left\lceil \frac{K}{256} \right\rceil - 1 \}$ /* outer k dimension */
    DMA-SUBTREE
    FILTER:$\{ putC(d_3, d_4) \}$ ⊗ FILTER:$\{ put\_replyC() \}$
```

Figure 11: The final schedule tree.

7.1 AST Generation

Reusing the AST generator of *isl* not only reduces the engineering cost, but it is also critical to support the generation of DMA and RMA instructions. The injection of extension nodes for DMA and RMA into schedule trees delivers the request of generating necessary communication statements to the AST generator, but there exists no corresponding AST node type for such extension nodes.

We introduce a new AST node type to handle DMA and RMA. The generated AST is thus converted into the expected form. Another benefit brought by bridging schedule trees and the athread code using AST is its portability to other programming models on the Sunway architecture: one can still reuse what we have described up to now but only has to redesign the pretty-print phase.

7.2 Inline Assembly Routine

The assembly micro kernel is provided as a compiled object, which has been highly optimized by the Sunway architects. Some other shapes were also designed before the one used in this work made publicly accessible. The shape 64 × 64 × 32 was empirically demonstrated as the best-performing one, which fully considers the memory sizes of SPMs and registers. It can also maximally exploit the use of each SPM when cooperating with double buffering (§6.3).

What has been done in this assembly routine is not visible, but decompiling the compiled object reveals that data copying from the SPM to the registers of a CPE, together with optimal register allocation, is well performed. Besides, assembly-level transformations including instruction pipelining, making use of SIMD intrinsics and loop unrolling are also considered. Combining this inline micro kernel with polyhedral transformations can thus achieve near-peak performance, which is implemented by introducing a *mark* node in the schedule tree (Fig.11). A mark node can be used to instruct the code generator to print an assembly function call, with the request is delivered by the string of the mark node.

7.3 Fusion Patterns

GEMM is also important for DL, but existing approaches [7] used without fusion with its element-wise prologue or epilogue operations. GEMM can be fused by *isl* scheduler with its prologue, a typical example of which is the element-wise quantization operation on input matrix A or B. However, such fusion does not hold the parallelism along the j or i loop dimension of GEMM. The aggressive fusion heuristic of *isl* thus loses the 2D parallelism required by the CPE mesh. We leverage the post-tiling fusion strategy [27] that uses an extension node to guarantee the 2D parallelism without missing the fusion opportunity. The schedule tree is shown in Fig.12a, where the mark node is used to bypass the original subtree of the prologue operation. The side effect is the recomputation of the prologue. In our case, this fusion performs an element-wise operation over $A_{64\times K}$ as shown in Fig.5, since this data tile is required by each CPE.

GEMM can also be fused with its epilogue that can be an activation function of matrix C. The fused result after tiling is depicted in Fig.12b. One can handle the GEMM subtree using the presented approach since the filter nodes under the innermost sequence node have been distributed. Code generation for prologue/epilogue is simple. Note that we have exploited the memory size of SPM to generate high-performance GEMM code. We restrict our approach to fusion patterns with a prologue/epilogue operation to demonstrate the possibility to support fusion while preserving the near-peak performance achieved for GEMM. The work on more fusion patterns for the Sunway architecture is still ongoing. More general fusion scenarios were well studied by DL compilers [20, 28].

There are no fundamental reasons prohibiting a polyhedral approach from being applied to fusion patterns with more prologue and/or epilogue operations. Two GEMM operations with the same shape configuration feeding an element-wise operation can also be fused for tensor core GPU [1]. Extending our approach to fuse more operations or both prologue and epilogue is trivial, which calls for an inline assembly routine with smaller shape configurations. The transformations described in this paper can still work when given such smaller configurations, since their validity does not depend on the sizes. DMA access latency hiding can still be optimized under the fusion pattern. For example, the DMA access latency of an epilogue can be overlapped with the computation of GEMM.

8 EXPERIMENTS

We implement our approach in PPCG [22]. The open-source code repository is available at https://gitee.com/tao-jinxuan/swcodegen.git. A simple GEMM code like Fig.2a is taken as input by our compiler.

```
DOMAIN: $\{S_0(i,k), S_1(i,j,k) : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$
 SEQUENCE:
  FILTER:{ $S_0(i,k)$ } /* prologue, this part will not be generated. */
   MARK: {"skipped"}
    BAND: $[S_0(i,k) \rightarrow (i,k)]$
  FILTER:{ $S_1(i,j,k)$ } /* GEMM */
   BAND: $[S_1(i,j,k) \rightarrow (\lfloor \frac{i}{64} \rfloor, \lfloor \frac{j}{64} \rfloor, \lfloor \frac{k}{32} \rfloor)]$
    EXTENSION: $[(d_0,d_1) \rightarrow S_0(d_2,d_3)]$
     SEQUENCE:
      FILTER:{ $S_0(d_2,d_3)$ } /* prologue defining $A_{64\times K}$ in Fig5. */
      FILTER:{ $S_1(i,j,k)$ } /* GEMM */
       BAND: $[S_1(i,j,k) \rightarrow (i-64\lfloor \frac{i}{64} \rfloor, j-64\lfloor \frac{j}{64} \rfloor, k-32\lfloor \frac{k}{32} \rfloor)]$
```

(a) The schedule tree of fusion with prologue.

```
DOMAIN: $\{S_1(i,j,k), S_2(i,j) : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$
 BAND: $[S_1(i,j,k) \rightarrow (\lfloor \frac{i}{64} \rfloor, \lfloor \frac{j}{64} \rfloor, \lfloor \frac{k}{32} \rfloor); S_2(i,j) \rightarrow (\lfloor \frac{i}{64} \rfloor, \lfloor \frac{j}{64} \rfloor, \lfloor \frac{K}{32} \rfloor)]$
  SEQUENCE:
   FILTER:{ $S_1(i,j,k)$ } /* GEMM */
    BAND: $[S_1(i,j,k) \rightarrow (i-64\lfloor \frac{i}{64} \rfloor, j-64\lfloor \frac{j}{64} \rfloor, k-32\lfloor \frac{k}{32} \rfloor)]$
   FILTER:{ $S_2(i,j)$ } /* epilogue */
    BAND: $[S_2(i,j) \rightarrow (i-64\lfloor \frac{i}{64} \rfloor, j-64\lfloor \frac{j}{64} \rfloor)]$
```

(b) The schedule tree of fusion with epilogue.

Figure 12: The schedule trees of the fusion patterns.

By default, it generates athread code for a cluster of SW26010Pro. The option *--batch* is used to help our compiler identify the batched GEMM scenarios, and *--no-use-asm* is provided to bypass the inline assembly kernel but generate simple loop code. The code for fusion patterns can be handled in a similar way.

The athread code (CPE code) and the file (MPE code) containing the main function are separately compiled by the native compiler SWGCC version 1307. For each code variant, we pass options *-faddress_align=128 -mhost -msimd -O3* to SWGCC when compiling the MPE code, and *-mslave -msimd -O3* for compiling the CPE code. They are linked together using the *-mhybrid* option. *-faddress_align=128* guarantees that the starting address of a matrix allocated in the main memory always align with 128 bytes, which can maximize the efficiency of DMA communications. *-msimd* allows for the use of SIMD intrinsics of MPE and CPE. *-mhost* and *-mslave* switch the compilation workflow of SWGCC for MPE and CPE.

We compare the performance with xMath version 2.0 [10], the highly tuned BLAS library of the SW26010Pro processor. The code used as the input to our code generator is modified to invoke a function call to this library, and this modified file is compiled by SWGCC with the same set of above options for compiling the MPE code. As xMath is written in Fortran language, we link its compiled object with the main function using SWGFORTRAN with options *-mhybrid*, and the row-major accesses have been converted into column-major required by the Fortran language.

We report Gflops of each code version computed by dividing the number of floating operations within in the code of interest using the execution clock cycles. Each result is the average of 10 executions, and the performance noise is lightweight. Committing a job to SW26010Pro requires a special command, together with which we set the stack space of the main memory as 8 GB.

8.1 Performance Breakdown

We first report the performance breakdown to illustrate how many improvements each optimization contributes to the overall performance. The data is collected in Fig.13 by experimenting on GEMM code with square matrix inputs due to the limited space, but the results are also applied to non-square matrix inputs and fusion patterns. We select matrix sizes by letting M and N be the multiples of 512 and K the multiple of 256. One can manually construct such shapes through zero padding when these requirements are not satisfied. The baseline version colored in red is the code generated by our compiler with automatic DMA communication enabled, since matrices have to be moved to the SPMs to be executed by CPEs.

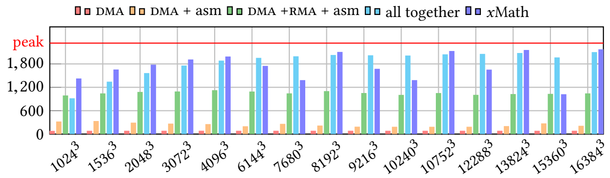


Figure 13: Performance comparison of Gemm with square matrix inputs. x axis: shapes; y axis: Gflops numbers.

The second version (orange bars) substitutes the naïve CPE code of the first version using the inline assembly micro kernel. Its performance can be used to illustrate to what extend the vendor provided assembly routine can impact the execution performance of Gemm. The third code variant is represented using green bars, with the RMA broadcast enabled and only memory latency hiding disabled. It can be used to assess the effectiveness of the two-level software pipelining strategy. Finally, we show the overall performance (in cyan) of our approach that turns on all optimizations.

The average performance of the baseline version is 84.89 Gflops almost without fluctuations, achieving a very smaller percentage of the peak performance. The exact Gflops number of the theoretical peak performance currently cannot be released. It will be officially declared soon. Adapting existing polyhedral code generators like PPCG for the Sunway architecture without further optimizations will also observe similar results. One can obtain a mean speedup of 2.83× when combined with the inline assembly routine, with the average performance increasing to 240.39 Gflops. The data of this version is followed by the bars of third version, which demonstrate that the RMA broadcasts are critical to performance improvement, raising the mean number up to 1052.94 Gflops. In other words, RMA can improve the performance by 4.38× on average. We finally turn on memory latency hiding and achieve an average performance of 1849.06 Gflops. On average, memory latency hiding improves the previous code variant by 1.76× and the baseline version by 23.72×.

We observe that our approach is underperforming when given the leftmost four shape configurations, none of which exceeds 1800.00 Gflops. The reason is because the matrix size along the k loop dimension has a heavy impact on the effectiveness of DMA latency hiding. As depicted in Fig.10b, the number of the overlaps is $\lceil \frac{K}{256} \rceil - 1$. A smaller K declines the benefit brought by the DMA latency hiding strategy. The effectiveness of memory latency hiding is significant under the relatively larger shape configurations. The average number of the remaining shape configurations is 2013.70 Gflops. In particular, the Gflops number of the code with the rightmost shape hits 90.14% of the theoretical peak performance. We believe this result should be competitive in practice.

8.2 Performance Comparison of Gemm

Fig.13 also collects the Gflops numbers for xMath, which achieves a mean 1746.97 Gflops number. Our approach outperforms it by 9.62%. As a BLAS library for SW26010Pro, xMath is not an open-source project. We thus have no ideas about its algorithmic implementation. The following analysis is based on our guess.

The superiority of xMath is observed when given the leftmost four smaller square matrix sizes. Why our approach performs less well than expected has been explained in §8.1. xMath beats our approach in these cases. We suspect that the internal implementation of xMath also implements its software pipelining strategy but it might introduce custom optimizations to adapt to these shape configurations. For example, the matrix tile sizes executed by a CPE can be reduced such that the number of overlaps can increase. xMath sometimes suffers from performance degradation when given sizes that are not powers of two. In particular, its performance falls behind our approach for the shape configuration 6144^3 and its performance gets even worse (under 1500.00 Gflops) when given sizes 7680^3, 10240^3 and 15360^3. We thus also surmise that the manual optimizations of xMath might be not mature for such data sizes.

To validate our suspicion, we also collect the Gflops numbers for 36 non-square matrix shapes, with the data plotted in Fig.14. xMath exhibits an average 1846.96 Gflops number. On the contrary, our approach obtains a mean Gflops number of 1911.22. Both xMath and our approach hit their top points, 93.53% and 90.03% of the peak performance, under the shape 4096×16384×16384, and the Gflops numbers of xMath indeed exceed 93.00% of the peak performance multiple times when the size of the k dimension is 16384. However, the performance of xMath falls down to 42.25% for 8192×8192×15360, and similar degradation is observed for nine times, each with the k dimension not being a power of two.

Our approach still performs well under these shapes, performing better than xMath by 58.95%; it is competitive with xMath even when given a size of powers of two along the k dimension, with only a 7.32% performance loss experienced. The above closer study on the data is another evidence that the library might not be mature for matrix sizes of powers of two. In summary, we obtain a mean improvement of 9.25% over the xMath library for these non-square matrix sizes. Coupled with the performance for square shapes, our work outperforms xMath by 9.44% for Gemm. Our method also exhibits a more stable trend than the BLAS library, which, on the contrary, fluctuates significantly with the changes of matrix sizes.

8.3 Performance Comparison of Batched Gemm

The results of batched Gemm are shown in Fig.15, where four batch sizes (2, 4, 8 and 16) are considered, each configured using six shapes. The sizes of the k dimension are selected as powers of two or not evenly. As explained in §3, the batched dimension is not involved in hardware binding, leading to the sequential execution of this dimension within the compiler generated CPE code. Our approach thus starts up the CPE mesh only once. The average performance value of our work is 1949.92 Glops. The highest point, 90.43%, is observed when give batch size 2 and shape 4096×4096×16384.

The batch dimension cannot be embedded into xMath, which results in multiple startups of the CPE mesh and thus introduces redundant coarser-grained synchronizations. This decreases the

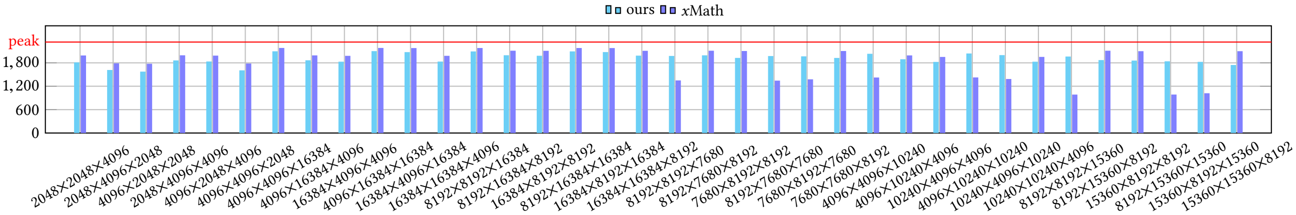


Figure 14: Performance comparison of GEMM with non-square matrix inputs. x axis: shapes; y axis: Gflops numbers.

performance of *x*Math, which obtains a mean number of 1603.26 Glops. Its performance jumps up to 93.52% of the peak performance under the batch size 2 and the shape 4096×4096×16384. On average, our approach outperforms *x*Math for batched GEMM by 1.30×.

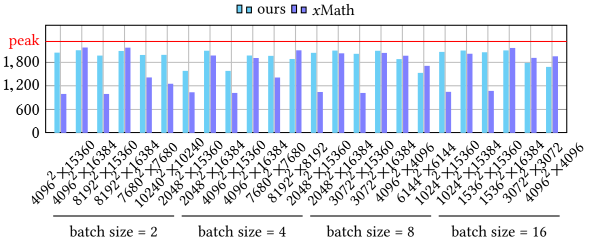


Figure 15: Performance comparison of batched GEMM. x axis: shape configurations; y axis: Gflops numbers.

8.4 Performance Comparison of Fusion

We now study the effectiveness of our work on two fusion patterns, with one fusing GEMM with its prologue, a quantization operation of matrix A, and the other with its epilogue, an activation function of matrix C. The results are depicted in Fig.16. We compare with the baseline version that does not perform fusion between these operations, which invokes *x*Math for GEMM and executes the prologue/epilogue on MPE.

Our work surpasses the baseline by 1.26× for fusion with the prologue. The two versions obtain average Gflops numbers of 1709.81 and 1436.46, respectively. The compiler generated code outperforms the baseline in most cases thanks to fusion that combines GEMM and the prologue in a single loop nest and executes both within each CPE. However, the baseline exhibits better performance in some cases, for example, when given shapes 10752^3 and 8192×16384×8192. This is because fusion with the prologue is achieved at the expense of recomputation of the quantization operation, with the introduced redundancy taking place along the j loop dimension. Our work thus performs less well when the size of this dimension increases. Besides, fusion with the prologue also makes the overhead of each cyan (rounded) rectangle in Fig.10c heavier and is thus negative to software pipelining, which decreases the performance of our code.

Different from the fusion pattern with prologue, our compiler steadily outperforms the *x*Math-based implementation by 2.11× on average when GEMM is fused with its epilogue. The mean Gflops numbers of our work and the *x*Math-based implementation are 1818.24 Gflops and 919.56 Gflops, respectively. Similar to the fusion pattern with prologue, both GEMM and its epilogue are executed on CPEs, but fusion with epilogue does not introduce recomputation since the two operations are fused outside the k dimension, which does not hamper software pipelining of GEMM. These are the reasons why our work always outperforms the library implementation for this fusion pattern. The average speedup of our approach over the *x*Math-based implementation for both fusion patterns is 1.67×.

8.5 Engineering Cost

Our work also greatly reduces the engineering cost to generate high-performance GEMM code for the Sunway architecture. *x*Math was developed by the same research team that proposed the manually optimized approaches for SW26010. They took a couple of months to finish the implementation and another several months to tune the performance. These together result in several years to release a public version: their approach for batched GEMM [12] was published three years later than that for GEMM [11]. On the contrary, our approach only takes several seconds to produce the code, including the overhead of integer linear solver of the polyhedral model. The vendor of SW26010Pro told us their architects cost two to three weeks to implement the inline assembly kernel described in §7.2. Our approach still significantly reduces the software development life cycle when this cost is taken into account.

9 RELATED WORK

While saving their significant engineering cost and overcoming the poor portability, our work also differs from the manual GEMM code generation approaches [11, 12] for SW26010 in many aspects. First, the target platforms are different, with the architectural changes described in §2.1. Second, our tiling strategy (§3) is much simpler than these manual approaches that require users to perform three-level tiling. Third, how DMA should be effectively used was not made clear in these works, which is the most difficult part for the Sunway architecture. We implement it as an automatic optimization (§4). Finally, these manual approaches for SW26010 only considered double buffering for DMA; the register communication latency available in SW26010 was implemented by scheduling assembly instructions. We propose the two-level double buffering strategy in §6, further simplifying the memory management for SW26010Pro.

Unlike prior work that requires users to add annotations [26] in the input code, our method does not require additional programming efforts. While not considering the rather simpler BLAS kernels investigated by AUGEM [23], the strategy used for optimizing GEMM can be easily adopted to subrograms like general matrix-vector multiplication. Note that a manual identification of

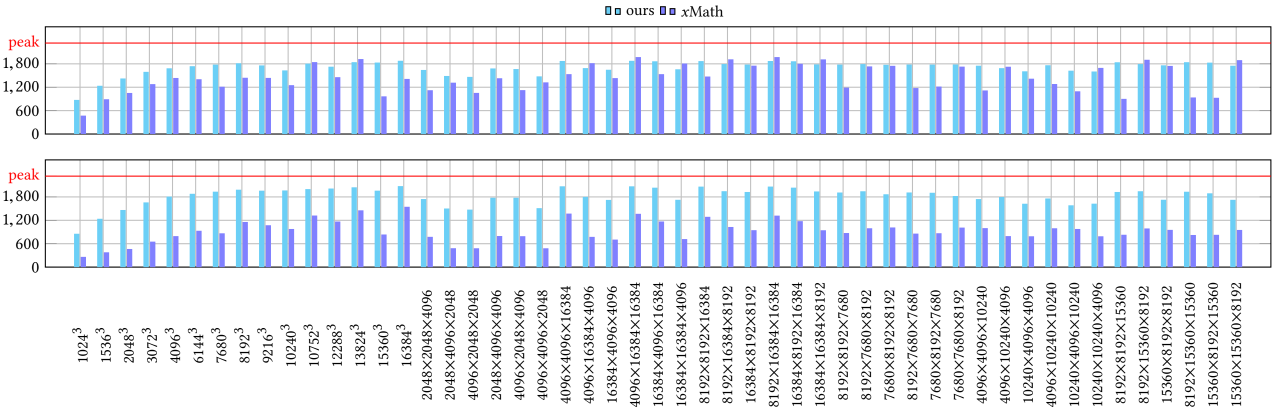


Figure 16: Performance comparison of fusion with prologue (upper) or epilogue (lower). x axis: shapes; y axis: Gflops.

AUGEM's optimization templates is needed but we have no such requirements. Our work is orthogonal with the compilation approach for the automatic generation of BLAS micro kernels [17] by focusing on data movements caused by GEMM compute decomposition. In particular, we generalize some domain-specific transformations and implement them using the polyhedral model, thus going beyond existing polyhedral approaches [4, 20, 22, 28].

Auto-tuners [2, 24] is an alternative to generate efficient GEMM code. As the inline assembly function in our generated code is defined using fixed matrix sizes, we analytically model the optimal tile sizes, which is sufficient for a specific compute pattern like GEMM [16]. We believe a tuning heuristic should be introduced when solving more general scenarios [7].

10 CONCLUSION AND FUTURE WORK

In this paper, we present a method to automatically generate GEMM kernels for SW26010Pro. Complex transformations including compute decomposition, DMA/RMA and memory latency hiding are all carried out as polyhedral transformations. Low-level optimizations are packed in an inline assembly kernel, which is embedded into the compiler generated code. The approach exhibits better performance than xMath for both (batched) GEMM and fusion patterns while significantly reducing the engineering cost to program SW26010Pro. With our method, one can obtain up to more than 90.00% of the theoretical performance within few lines of C code.

Our method also offers insights into the compilation of other supercomputers. RMA is a specialized property of the SW26010Pro processor; its implementation in §5 is thus a Sunway-specific algorithm. Yet the techniques to automate DMA (§4) and memory latency hiding (§6) can be borrowed by approaches for other architectures; the idea of combining polyhedral transformations and inline assembly kernels is also applicable to CPU and GPU [1, 18].

The approach currently has two weaknesses: the performance of small-scale matrix sizes can be enhanced, and the supported fusion patterns are limited. They can be solved if smaller shapes of the inline assembly kernel can be offered. We intend to automate the generation of the inline assembly in the future, which is also achievable through compilation approaches [17]. We also plan to implement MPI code generation like what prior work [3] did, which will realize the fully automation of GEMM code generation for the SW26010Pro processor. Besides, generalizing the approach to a wider scope for SW26010Pro is also under construction.

ACKNOWLEDGMENTS

The authors acknowledge Wenlei Bao for his suggestions on an earlier version of this submission. We also thank the reviewers for their valuable time and constructive comments on reading this paper. Partly supported by the National Natural Science Foundation of China under Grant No. U20A20226, this work corresponds to Jie Zhao. The views and conclusions in this work are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Chinese Government.

REFERENCES

[1] Somashekaracharya G. Bhaskaracharya, Julien Demouth, and Vinod Grover. 2020. Automatic Kernel Generation for Volta Tensor Cores. arXiv:2006.12645 [cs.PL]

[2] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 1997. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the 11th International Conference on Supercomputing* (Vienna, Austria) *(ICS '97)*. Association for Computing Machinery, New York, NY, USA, 340–347. https://doi.org/10.1145/263580.263662

[3] Uday Bondhugula. 2013. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC'13)*. Association for Computing Machinery, New York, NY, USA, Article 33, 12 pages. https://doi.org/10.1145/2503210.2503289

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI'08)*. ACM, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[5] Haohuan Fu, Conghui He, Bingwei Chen, Zekun Yin, Zhenguo Zhang, Wenqiang Zhang, Tingjian Zhang, Wei Xue, Weiguo Liu, Wanwang Yin, Guangwen Yang, and Xiaofei Chen. 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* (Denver, Colorado) *(SC'17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. https://doi.org/10.1145/3126908.3126910

[6] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei Ge, Jian Zhang, Yangang Wang, Chunbo Zhou, and Guangwen Yang. 2016. The Sunway TaihuLight Supercomputer: System and Applications. *Science China Information Sciences* 59, Article 072001 (June 2016), 16 pages. https://doi.org/10.1007/s11432-016-5588-7

[7] Wei Gao, Jiarui Fang, Wenlai Zhao, Jinzhe Yang, Long Wang, Lin Gan, Haohuan Fu, and Guangwen Yang. 2019. SwATOP: Automatically Optimizing Deep

Learning Operators on SW26010 Many-Core Processor. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 89, 10 pages. https://doi.org/10.1145/3337821.3337883

[8] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[9] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 12 (July 2015), 50 pages. https://doi.org/10.1145/2743016

[10] National Supercomputing Center in Wuxi. 2016. xMath User Manual v1.0 (in Chinese). http://www.nsccwx.cn:1337/uploads/595bce0bed1b4537994d927ef6be922d.pdf

[11] 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 *2017 46th International Conference on Parallel Processing (ICPP)*. 422–431. https://doi.org/10.1109/ICPP.2017.51

[12] 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

[13] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. 2022. MLIR-Based Code Generation for GPU Tensor Cores. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/3497776.3517770

[14] Wayne Kelly and William Pugh. 1995. A unifying framework for iteration reordering transformations. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, Vol. 1. 153–162. https://doi.org/10.1109/ICAPP.1995.472180

[15] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When Polyhedral Transformations Meet SIMD Code Generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI'13)*. ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/2491956.2462187

[16] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw.* 43, 2, Article 12 (Aug. 2016), 18 pages. https://doi.org/10.1145/2925987

[17] Xing Su, Xiangke Liao, and Jingling Xue. 2017. Automatic Generation of Fast BLAS3-GEMM: A Portable Compiler Approach. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO'17)*. IEEE Press, 122–133.

[18] Sanket Tavarageri, Alexander Heinecke, Sasikanth Avancha, Bharat Kaul, Gagandeep Goyal, and Ramakrishna Upadrasta. 2021. PolyDL: Polyhedral Optimizations for Creation of High-Performance DL Primitives. *ACM Trans. Archit. Code Optim.* 18, 1, Article 11 (Jan. 2021), 27 pages. https://doi.org/10.1145/3433103

[19] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (June 2015), 33 pages. https://doi.org/10.1145/2764454

[20] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (Oct. 2019), 26 pages. https://doi.org/10.1145/3355606

[21] Sven Verdoolaege. 2010. Isl: An Integer Set Library for the Polyhedral Model. In *Proceedings of the Third International Congress Conference on Mathematical Software* (Kobe, Japan) *(ICMS'10)*. Springer-Verlag, Berlin, Heidelberg, 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

[22] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[23] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2503210.2503219

[24] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing* (San Jose, CA) *(SC'98)*. IEEE Computer Society, USA, 1–27.

[25] Shizhen Xu, Yuanchao Xu, Wei Xue, Xipeng Shen, Fang Zheng, Xiaomeng Huang, and Guangwen Yang. 2018. Taming the "Monster": Overcoming Program Optimization Challenges on SW26010 Through Precise Performance Modeling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 763–773. https://doi.org/10.1109/IPDPS.2018.00086

[26] Qing Yi, Qian Wang, and Huimin Cui. 2014. Specializing Compiler Optimizations through Programmable Composition for Dense Matrix Computations. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 596–608. https://doi.org/10.1109/MICRO.2014.14

[27] Jie Zhao and Peng Di. 2020. Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 427–441. https://doi.org/10.1109/MICRO50266.2020.00044

[28] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*. Association for Computing Machinery, New York, NY, USA, 1233–1248. https://doi.org/10.1145/3453483.3454106

[29] Qianchao Zhu, Hao Luo, Chao Yang, Mingshuo Ding, Wanwang Yin, and Xinhui Yuan. 2021. Enabling and Scaling the HPCG Benchmark on the Newest Generation Sunway Supercomputer with 42 Million Heterogeneous Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC'21)*. Association for Computing Machinery, New York, NY, USA, Article 57, 13 pages. https://doi.org/10.1145/3458817.3476158