ALGORITHMS: MATRIX MULTIPLICATION AND GEMM OPTIMIZATION

KAMI: Communication-Avoiding General Matrix Multiplication within a Single GPU

Hemeng Wang, Yang Du, Sidu Li, Xiaowen Tian, Qingxiao Sun, Weifeng Liu
SSSLab, Dept. of CST, China University of Petroleum-Beijing

St. Louis, MO · NOV 20

https://github.com/SuperScientificSoftwareLaboratory/KAMI

SC25 ST. LOUIS hpc ignites.

OUTLINE

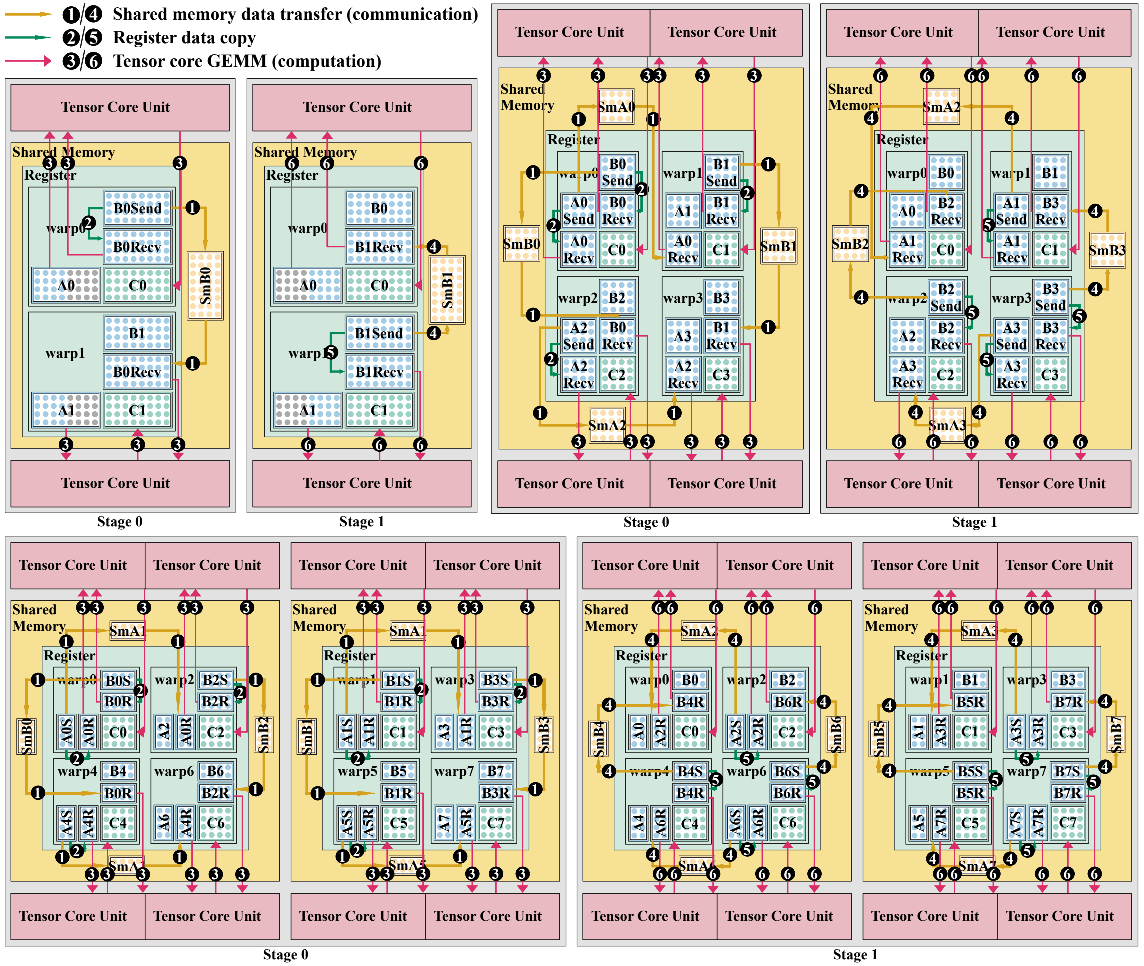
1 Introduction

2 Motivation

3 Method

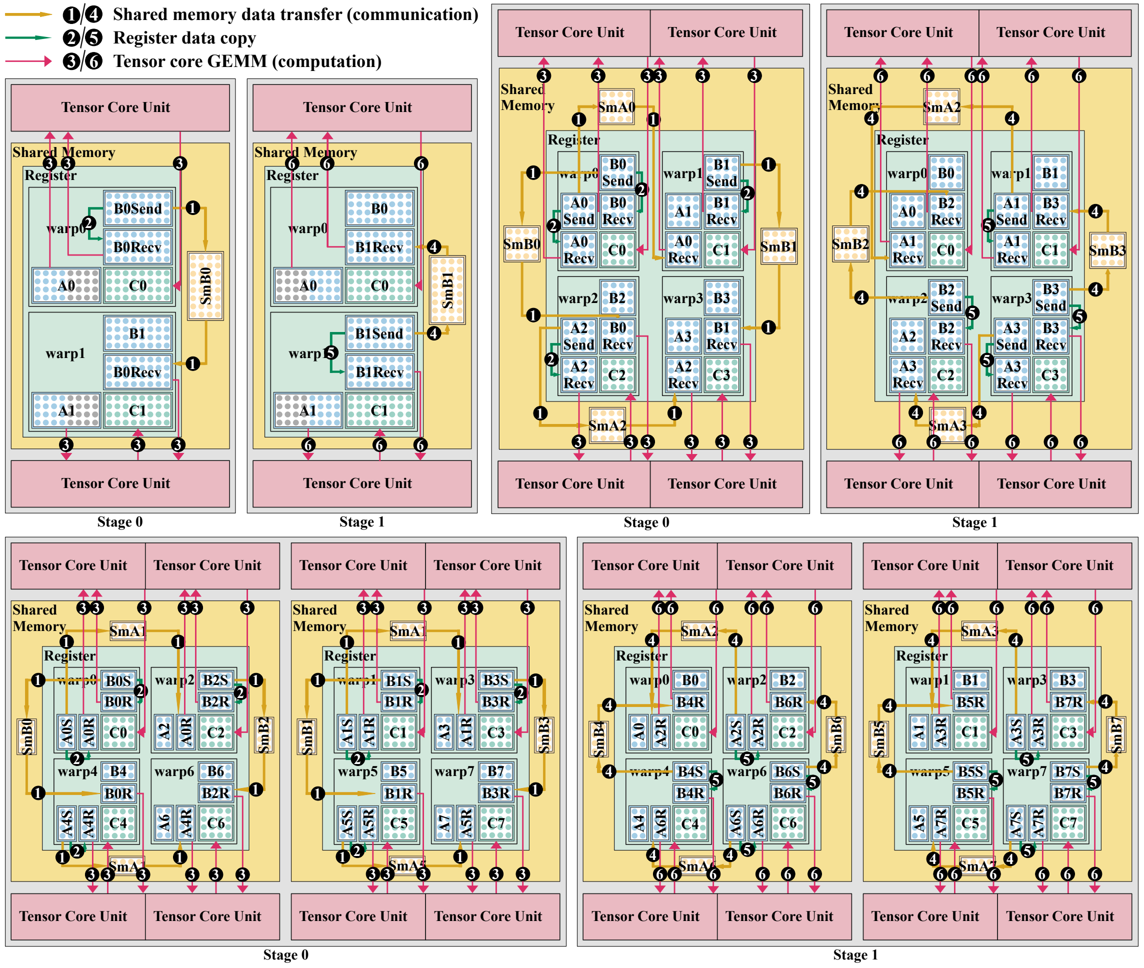
4 Experiment

5 Conclusion



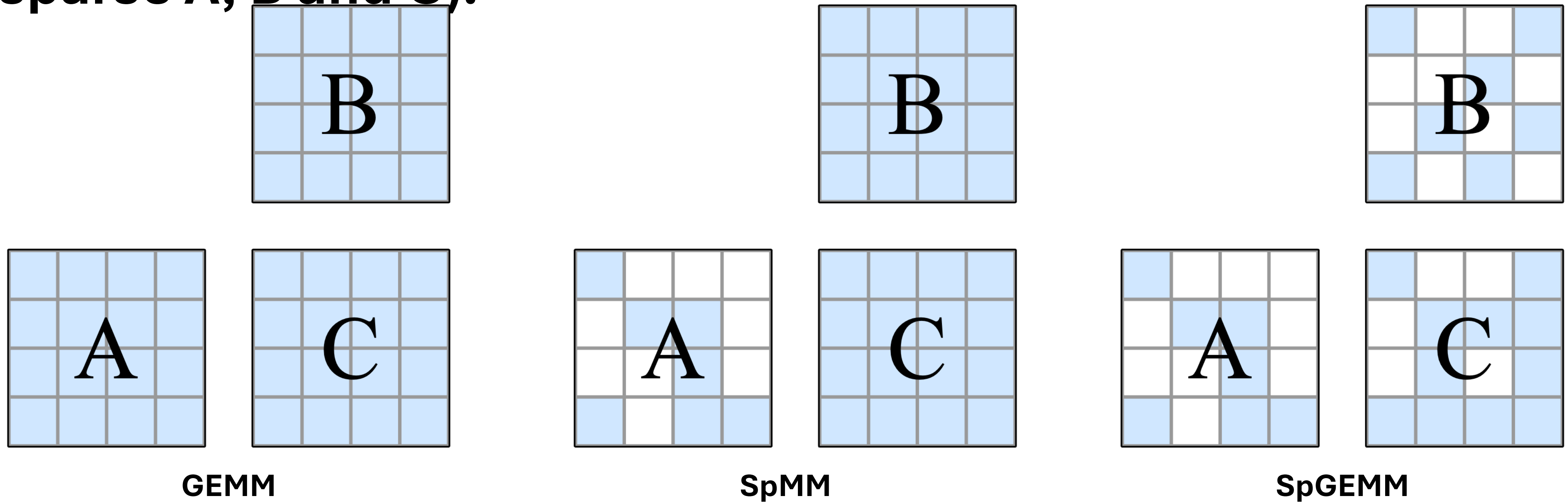
OUTLINE

1. Introduction
2. Motivation
3. Method
4. Experiment
5. Conclusion



Introduction

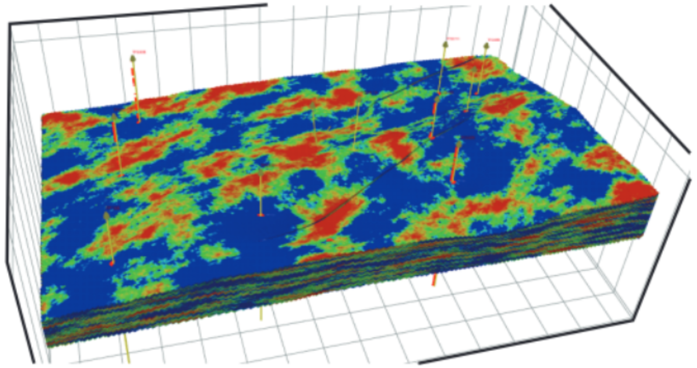
GEMM operation multiplies a dense matrix A of size m-by-k with a dense matrix B of size k-by-n, and gives a resulting dense matrix C of size m-by-n. When accounting for sparsity, GEMM can become SpMM (sparse A, dense B and C) and SpGEMM (sparse A, B and C).



Different variants of matrix multiplication.

Introduction

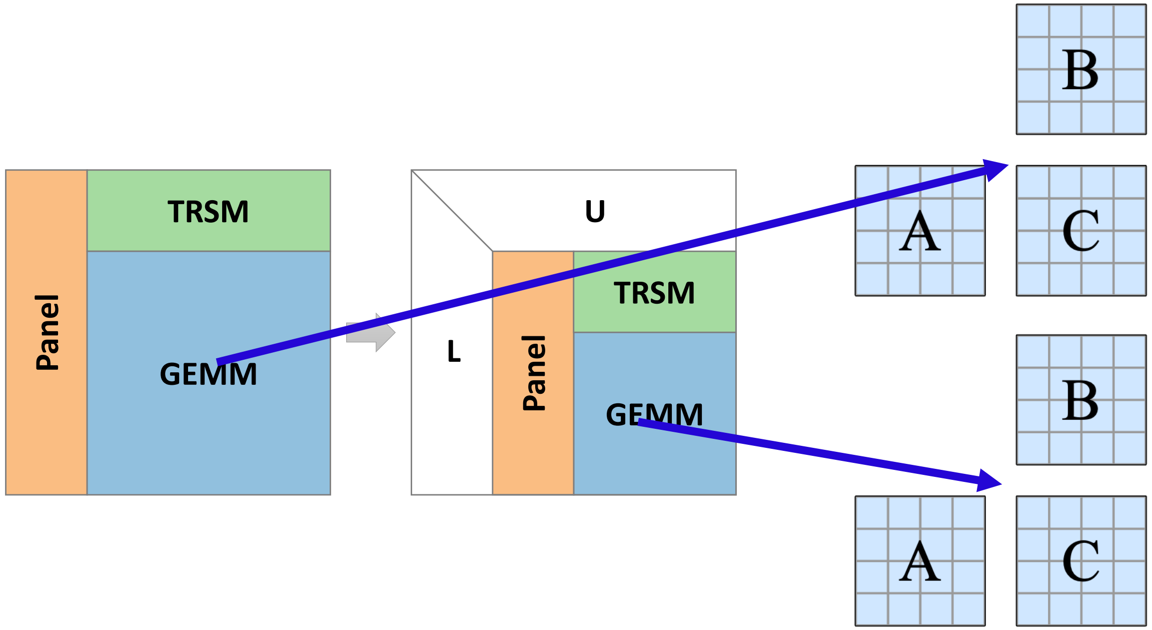
Matrix multiplication is a core operation in high performance computing. Such as reservoir simulation, matrix multiplication is widely employed for PDE discretization and iterative solution.



PDEs in reservoir numerical simulation

$$Ax = b$$

Solving linear systems

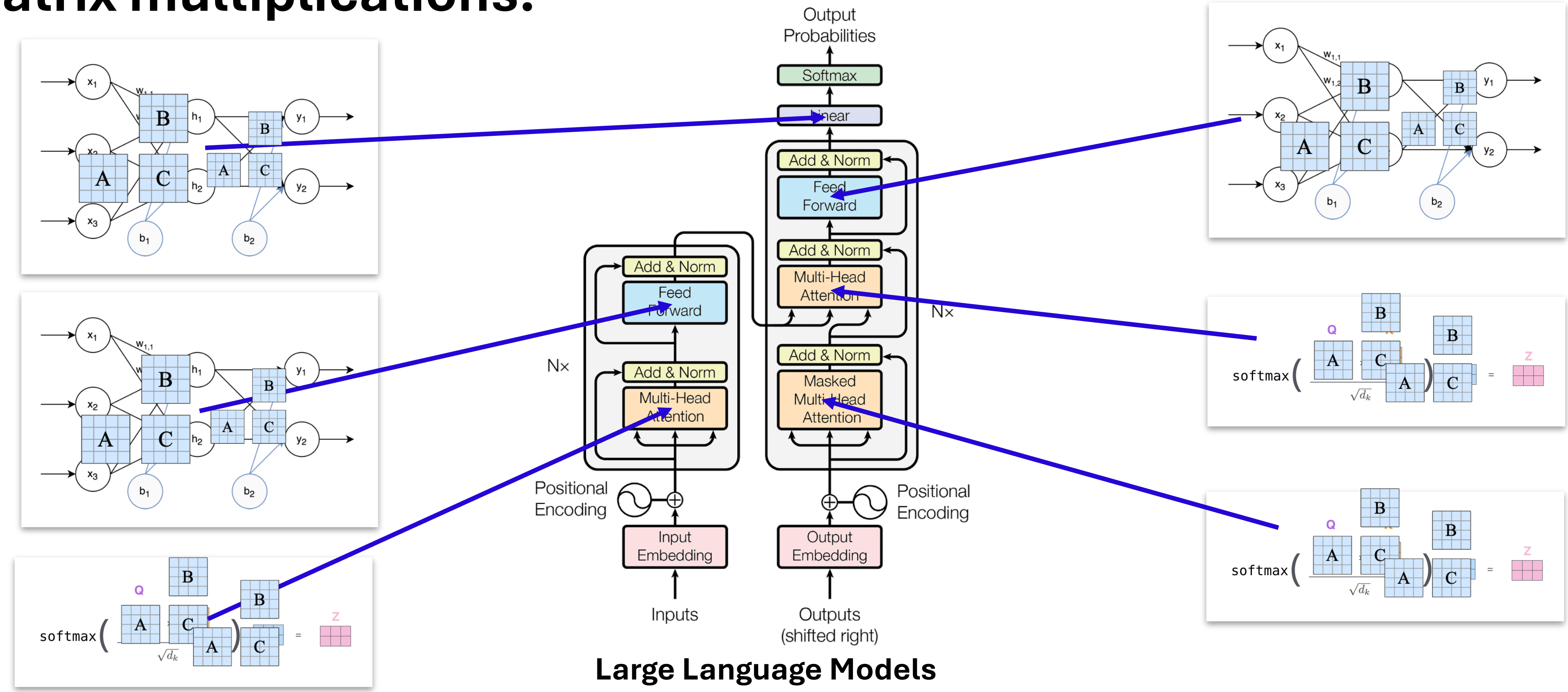


LU decomposition of a matrix

Matrix multiplication

Introduction

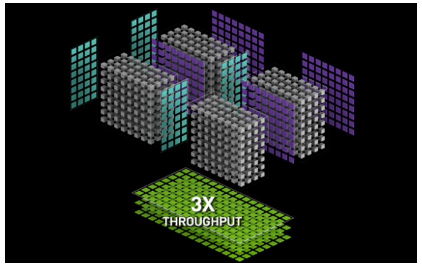
In large language models, the attention mechanism and linear layers in the Transformer architecture are both implemented as matrix multiplications.



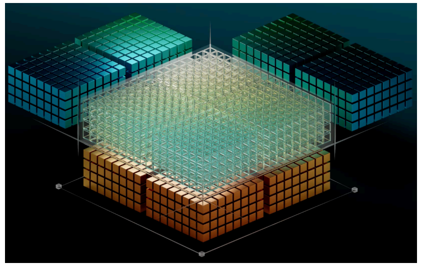
Large Language Models

Introduction

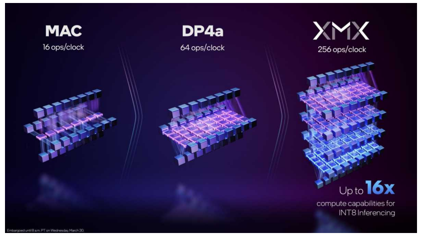
Major hardware vendors have introduced specialized acceleration units to enhance matrix multiplication performance. Numerous efforts have also been dedicated to improving matrix multiplication performance through software.



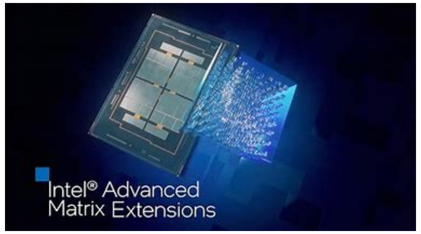
NVIDIA Tensor Core



AMD Matrix Core



Intel Xe Matrix eXtension



Advanced Matrix eXtension

Hardware optimization of GEMM

cuBLAS | cuSPARSE | ROCm | oneAPI

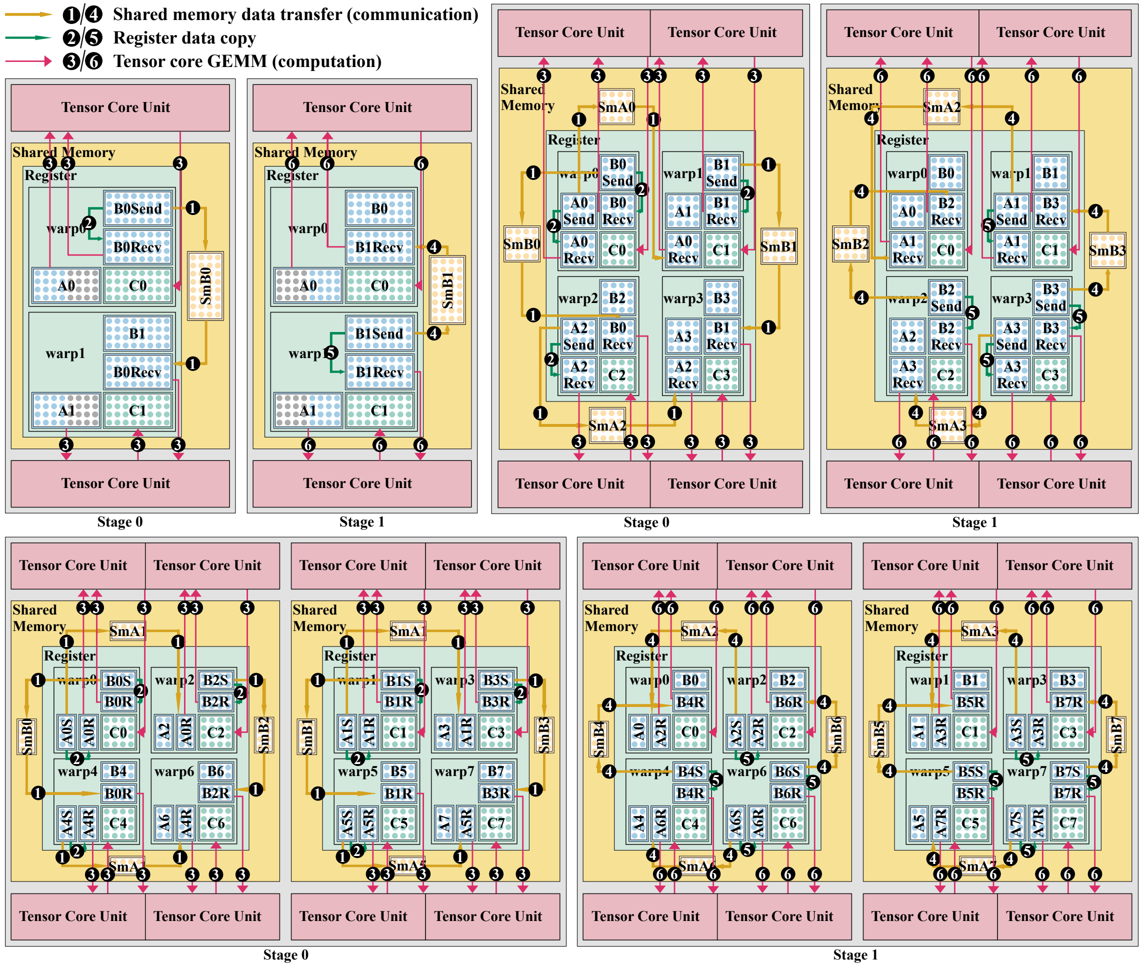
OpenBLAS | MAGMA | TACO

Eigen | BLIS | SLATE | BeidouBLAS

High-Performance Matrix Computation Libraries

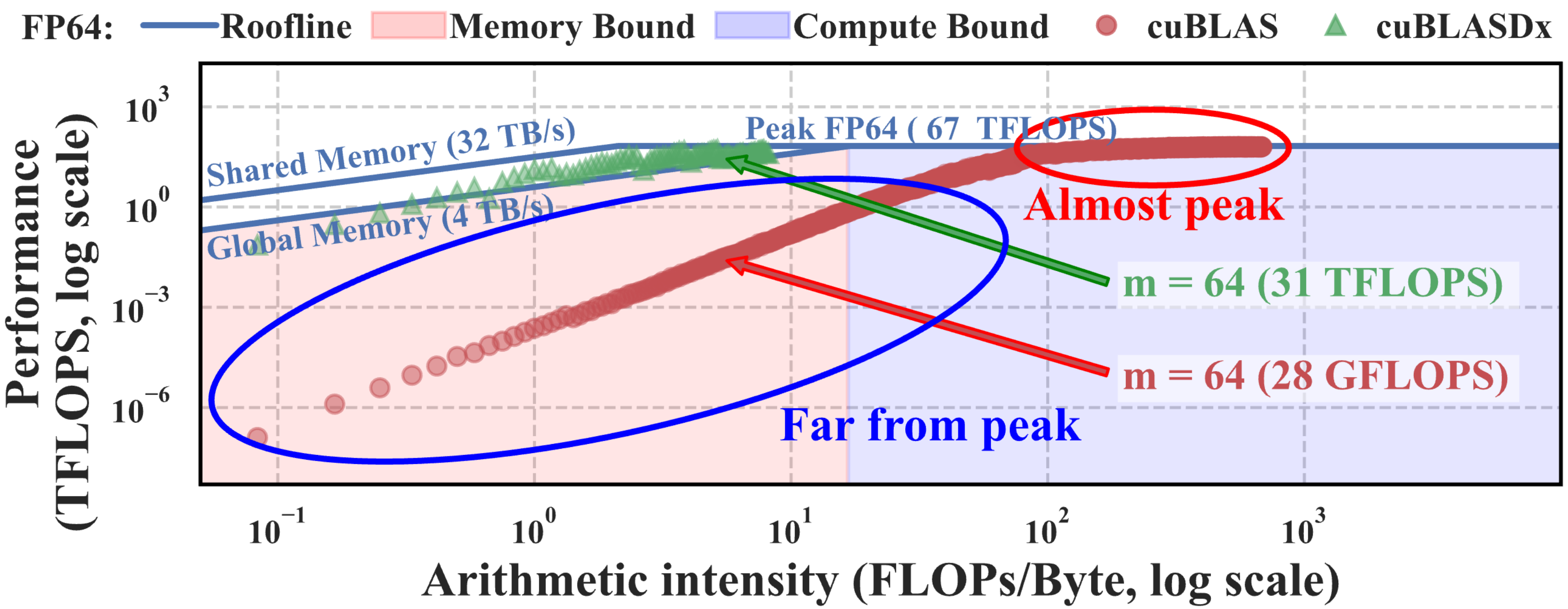
OUTLINE

1. Introduction
2. **Motivation**
3. Method
4. Experiment
5. Conclusion



Motivation

When the matrix dimensions are small, their performance still falls significantly short of the theoretical upper bound, indicating substantial room for optimization.

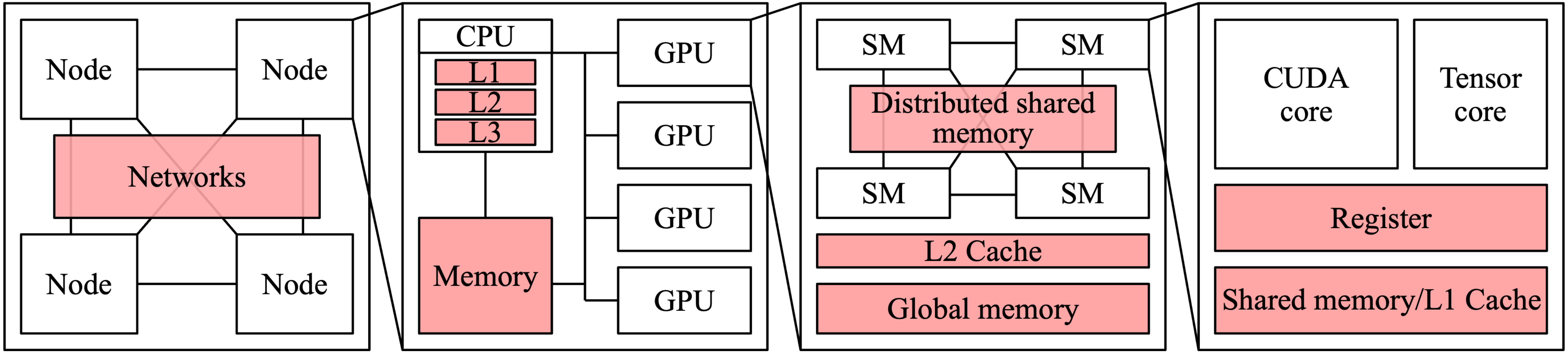


46% of peak

Roofline model of GEMM performance on the NVIDIA GH200 GPU.

Motivation

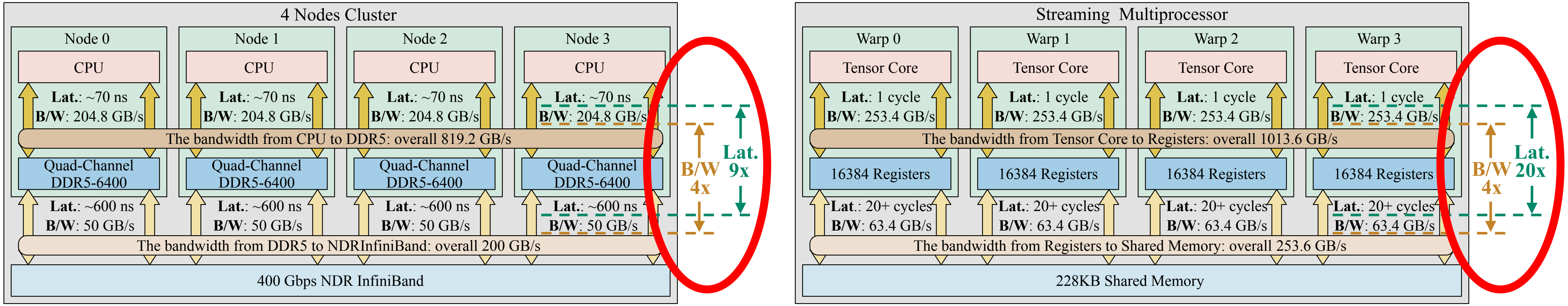
With the continuous evolution of computing architectures, the memory hierarchy has become increasingly deep, encompassing multiple layers from on-chip registers, various levels of cache, and shared memory, to main memory and remote storage.



Hierarchical memory architecture in common computing clusters.

Motivation

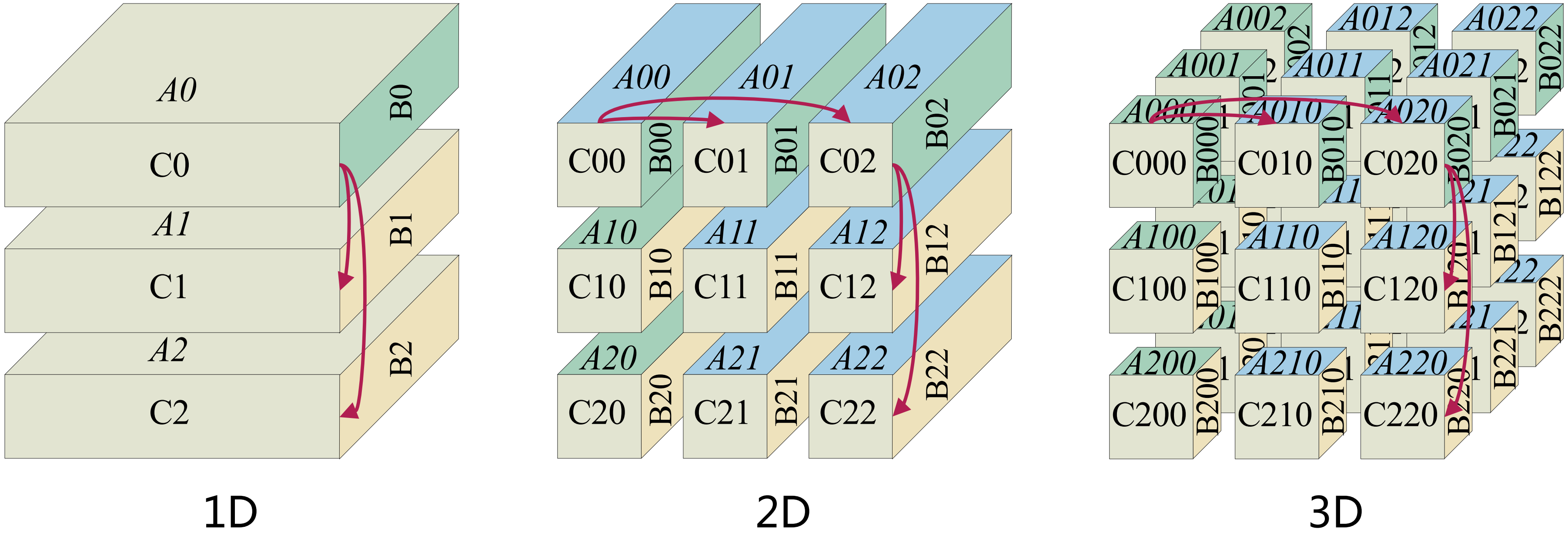
We found similar memory hierarchy characteristics between distributed platforms and single GPUs, with comparable latency (order-of-magnitude) and bandwidth (4x) gaps between local and remote storage.



Latency and bandwidth comparison of the memory hierarchy of a 4-node cluster and a 4-warp SM.

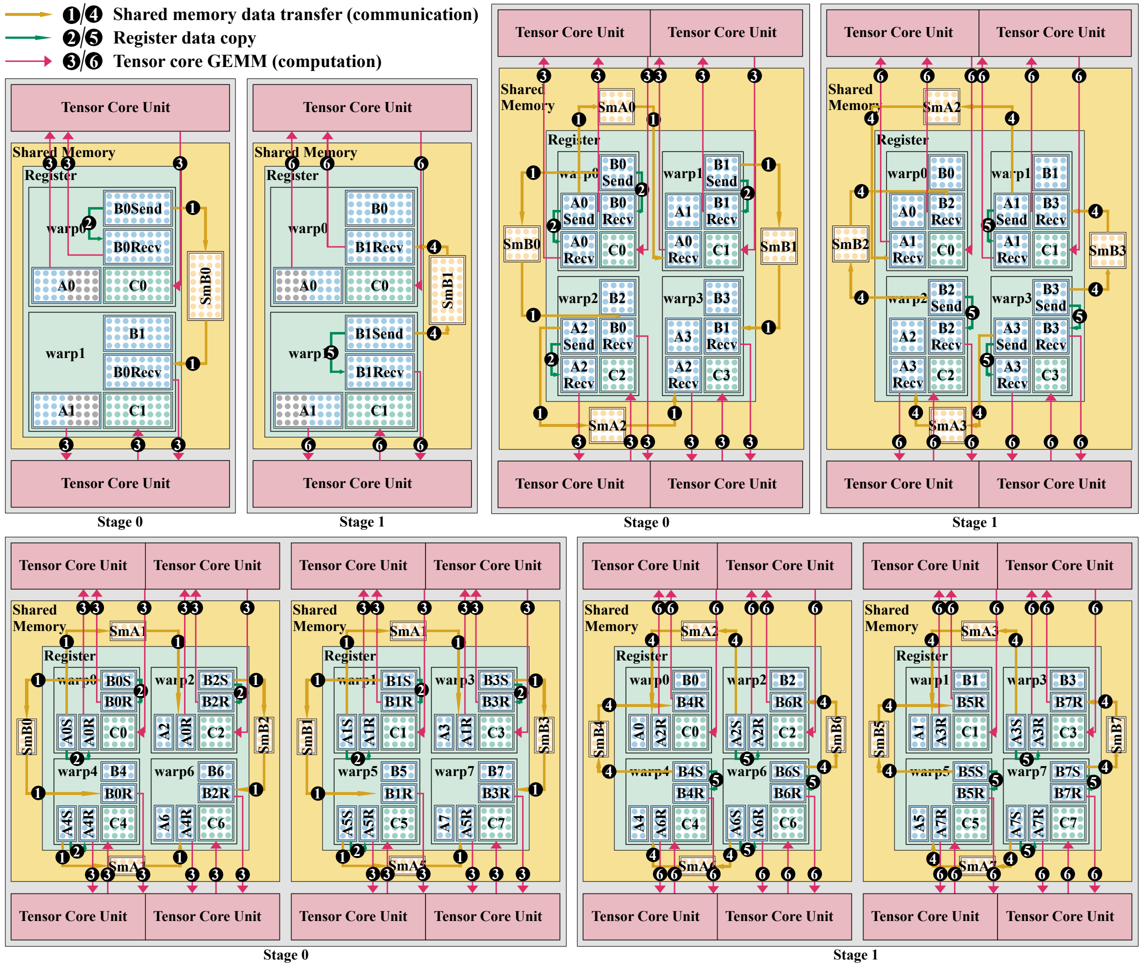
Motivation

For large-scale matrix multiplication executed on distributed platforms, communication often becomes the bottleneck. Communication-Avoiding (CA) algorithms[1-3] aim to reduce data transfer between compute nodes, thereby alleviating this issue.



[1] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing Communication in Numerical Linear Algebra," SIAM J. Matrix Anal. 2011.
[2] E. Georganas, J. Gonzalez-Dominguez, E. Solomonik, Y. Zheng, J. Tourino, and K. Yelick, "Communication avoiding and overlapping for numerical linear algebra," in SC12.
[3] J. Demmel. "Communication-Avoiding Algorithms for Linear Algebra and Beyond." In IPDPS13.

OUTLINE



1/4 Shared memory data transfer (communication)
2/5 Register data copy
3/6 Tensor core GEMM (computation)
Tensor Core Unit
Shared Memory
Register
warp0
warp1
Stage 0
Stage 1

Method: Concept

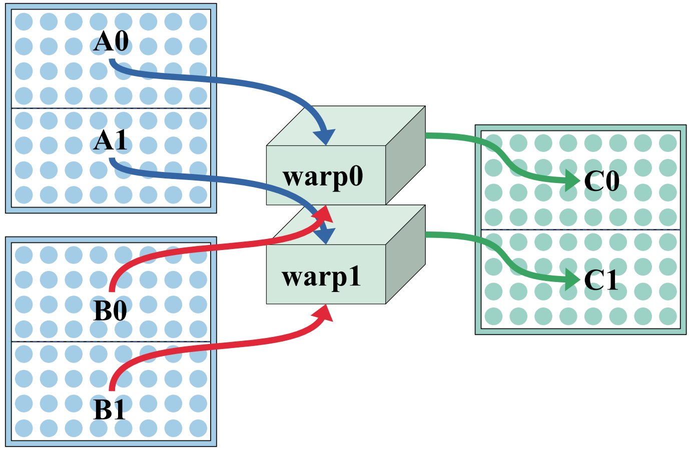
In this paper, we propose KAMI, a set of 1D, 2D, and 3D CA algorithms accelerating small-scale GEMM, SpMM and SpGEMM within a single GPU.

Concept	Classic CA	KAMI (our work)
Compute unit	Process on CPU/GPU	Warp on tensor core
Local storage	DRAM	Thread register
Communication	Send/Recv by network	LD/ST on shared mem.
Perf. metric	Execution time	GPU clock cycle

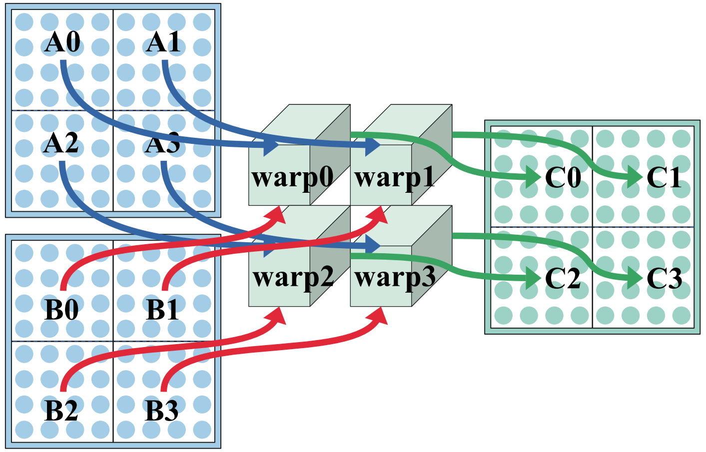
Concept of classic CA and KAMI

Method: Data Layout

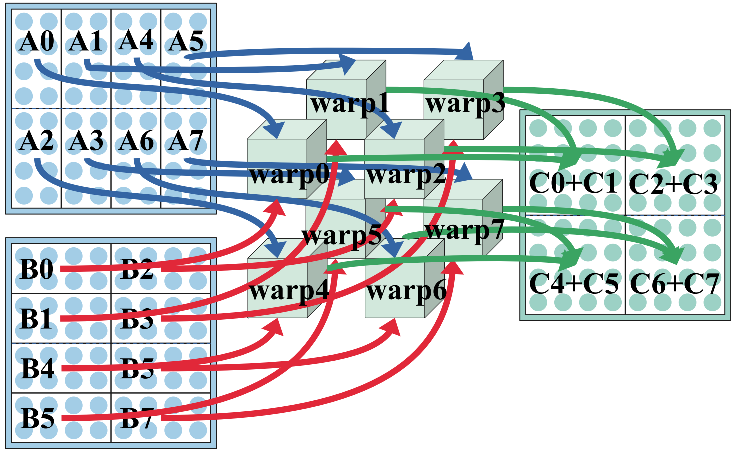
We designed a set of CA algorithms to accelerate matrix multiplication on a single GPU. Depending on the specific algorithm, the matrices are partitioned into sub-matrices in different ways.



1D algorithm
(2 warps, A,B and C are partitioned into 2 sub-matrix)



2D algorithm
(4 warps, A,B and C are partitioned into 4 sub-matrix)



3D algorithm
(8 warps, A and B are partitioned into 8 sub-matrix, C is partitioned into 4 sub-matrix)

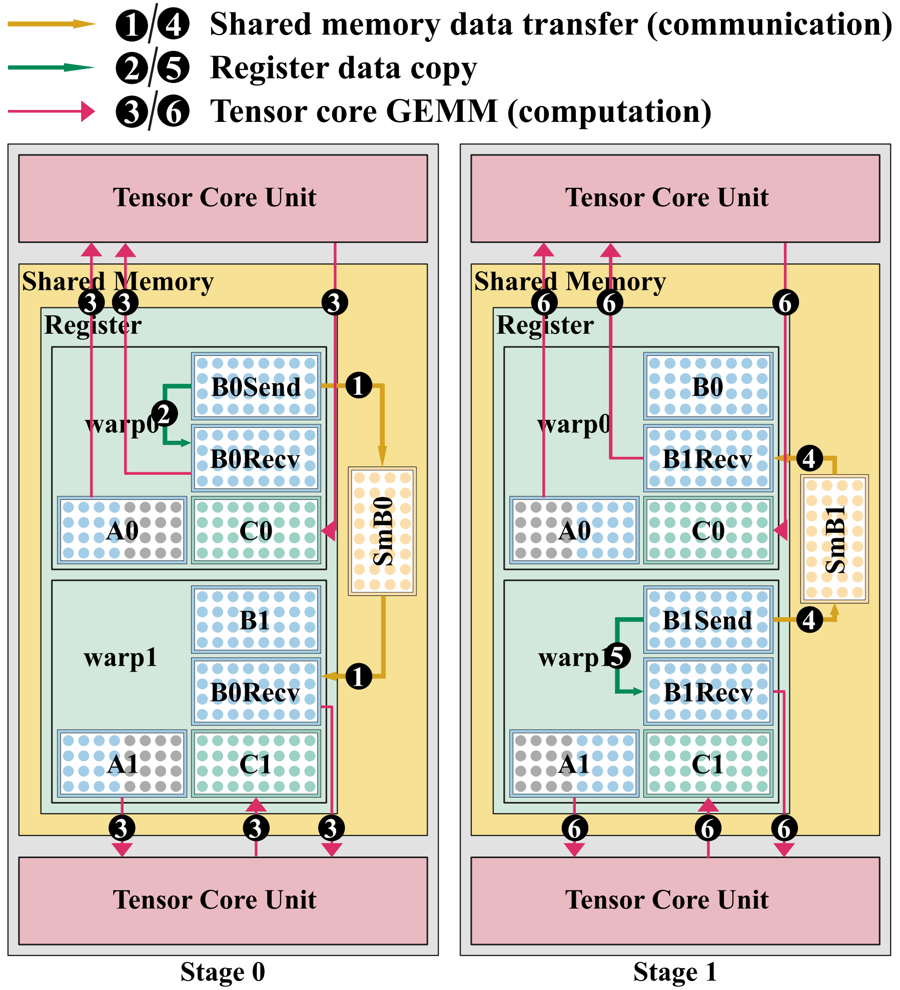
Method: 1D Algorithm

In the 1D algorithm, p warps perform a MM, with each warp (warpi, where 0 ≤ i < p) holding sub-matrix Ai (of size m/p×k) and sub-matrix Bi (of size k/p×n). The GPU warps operate in an

Algorithm 1 1D algorithm by p warps

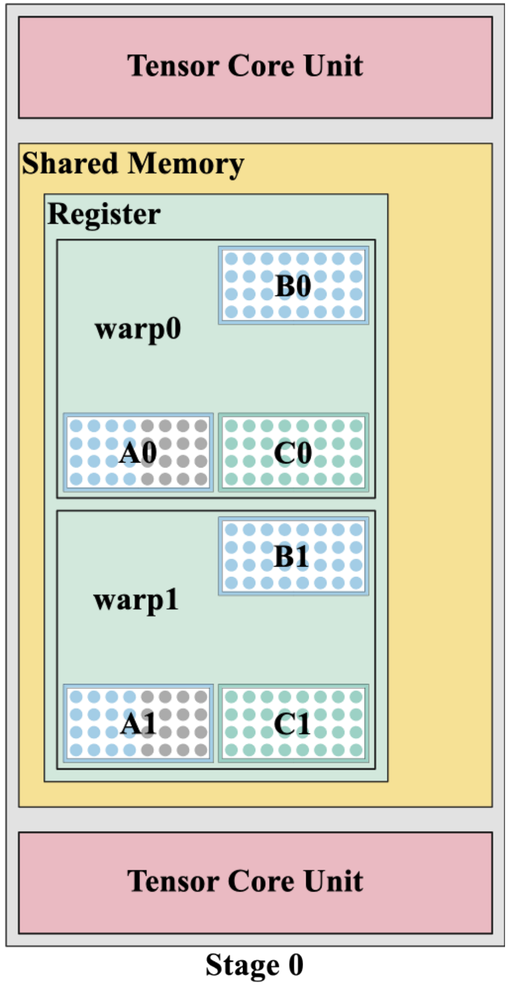
$i \leftarrow$ warpID
GMem2Reg(Ai ← A, Bi ← B, Ci ← C)
for $z = 0$ **to** p **do** ▷ The algorithm consists of p stages.
  if i = z **then**
    Reg2SMem(**SmB** ← **BSend**) ▷ Write **BSend** to shared memory.
    Reg2Reg(**BRecv** ← **BSend**) ▷ Copy **BSend** within registers.
  else if $i \neq$ z **then**
    DTransSMem2Reg(**BRecv** ← **SmB**) ▷ Read **SmB** from shared memory.
  Ci ← TensorCoreGEMM(Ai[:][$z \times \frac{k}{p}$: $(z+1) \times \frac{k}{p}$], BRecv)
  ▷ Part of Ai and BRecv multiplied by Tensor Core.
Reg2GMem(C ← Ci)

Pseudo-code for 1D algorithm



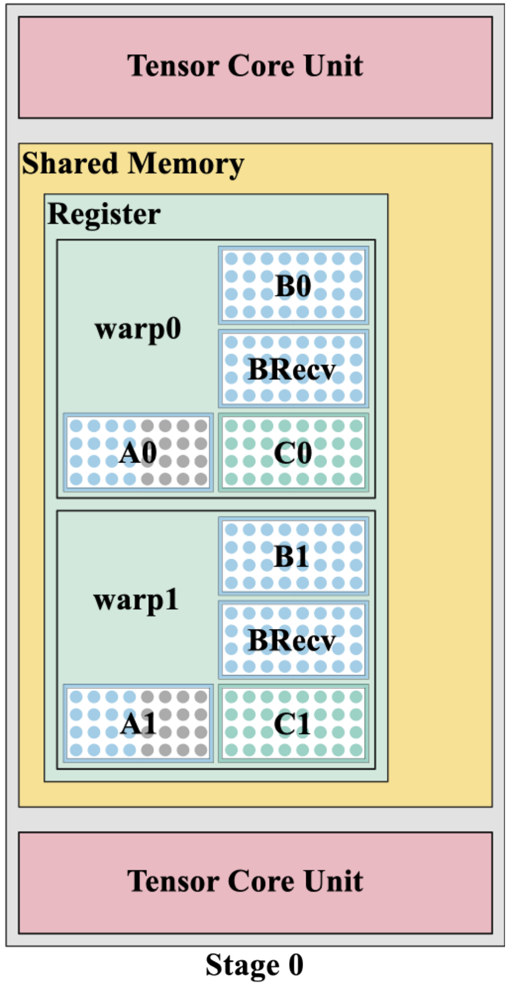
Process of 1D algorithm

Method: 1D Algorithm



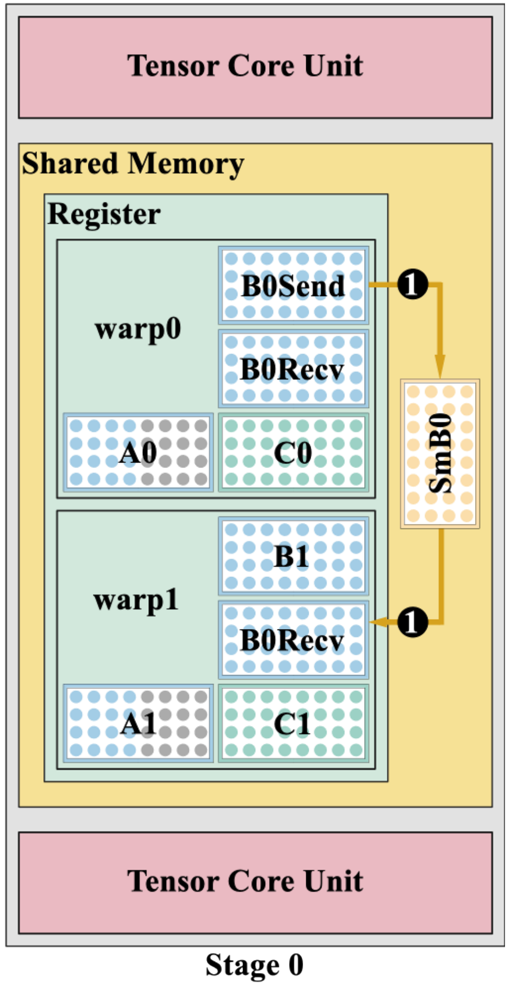
Process of 1D algorithm

Method: 1D Algorithm



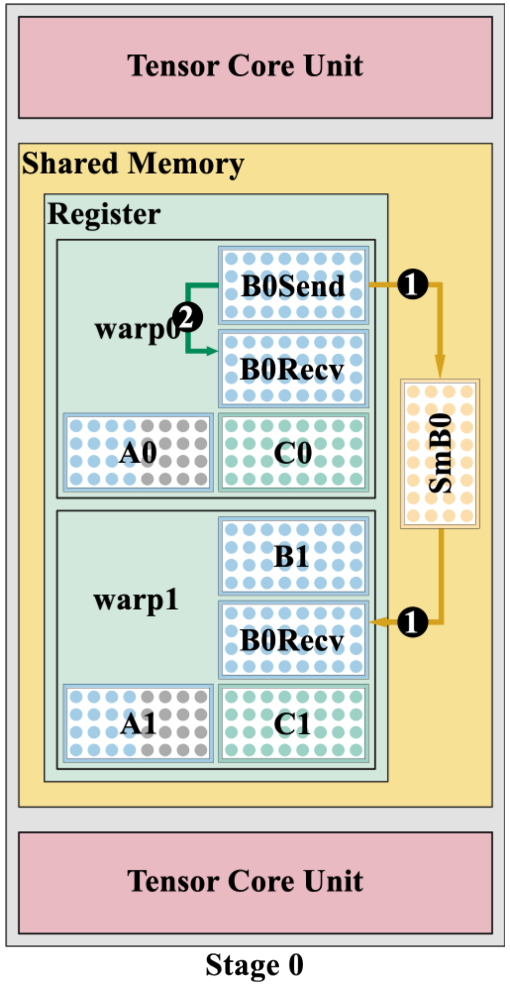
Process of 1D algorithm

Method: 1D Algorithm



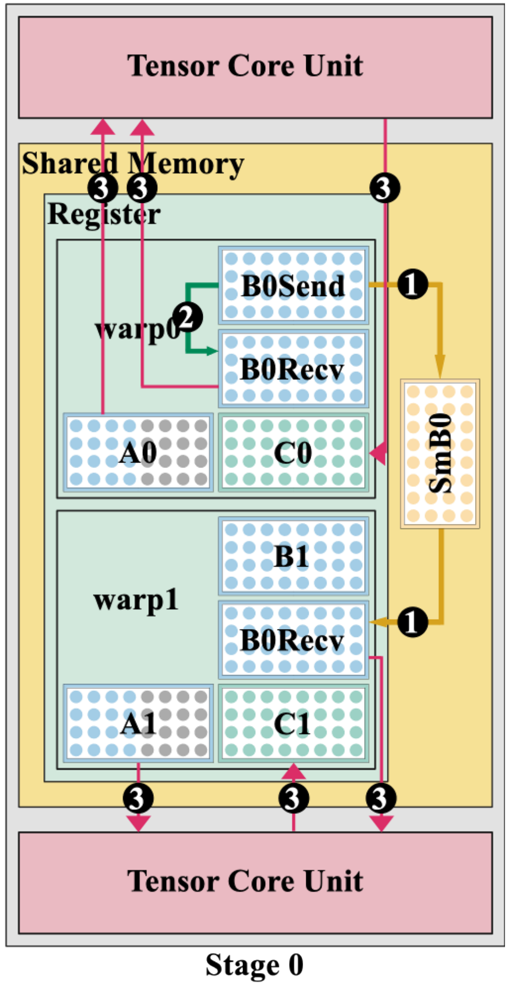
Process of 1D algorithm

Method: 1D Algorithm



Process of 1D algorithm

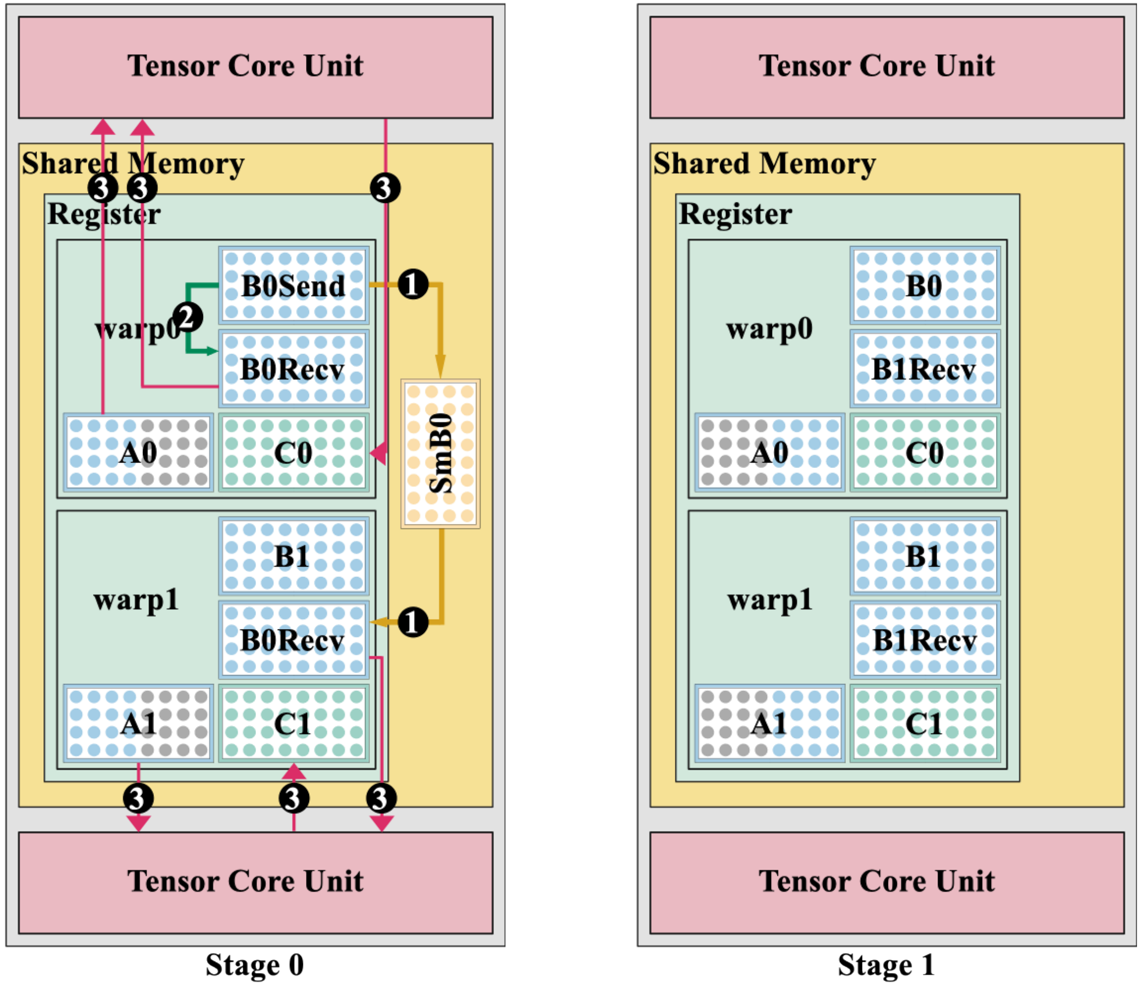
Method: 1D Algorithm



Process of 1D algorithm

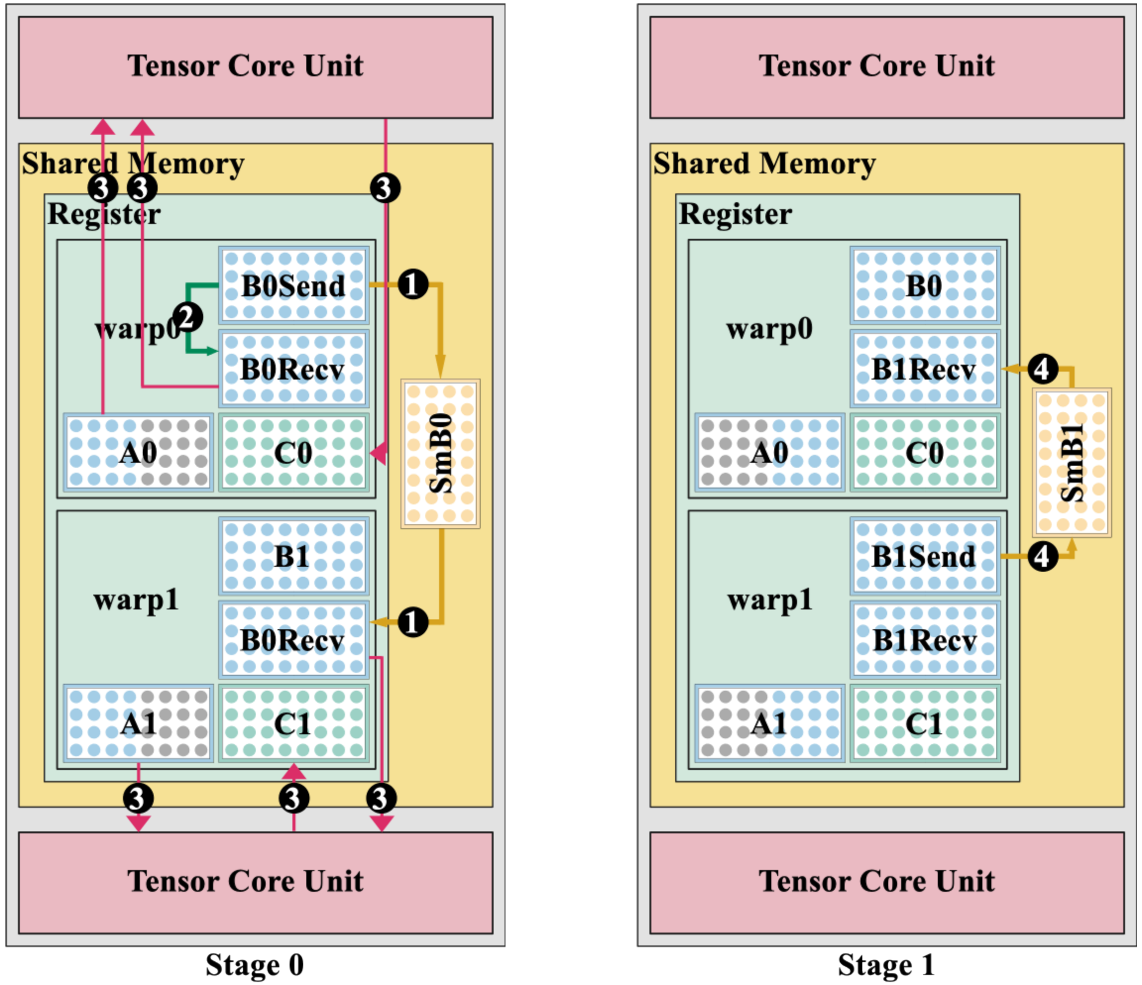
中国石油大学(北京) CHINA UNIVERSITY OF PETROLEUM
Lab 超级科学软件实验室 Super Scientific Software Laboratory

Method: 1D Algorithm



Process of 1D algorithm

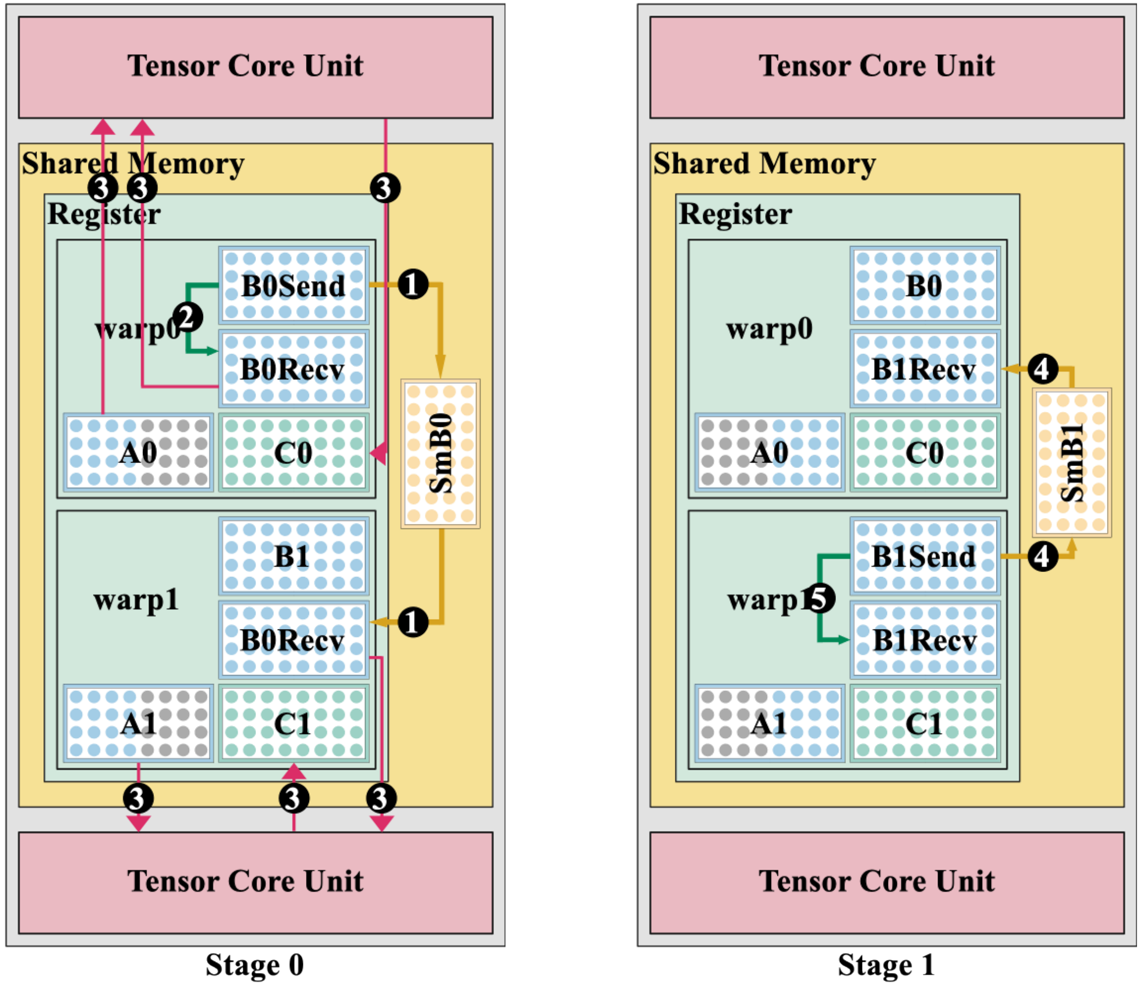
Method: 1D Algorithm



Process of 1D algorithm

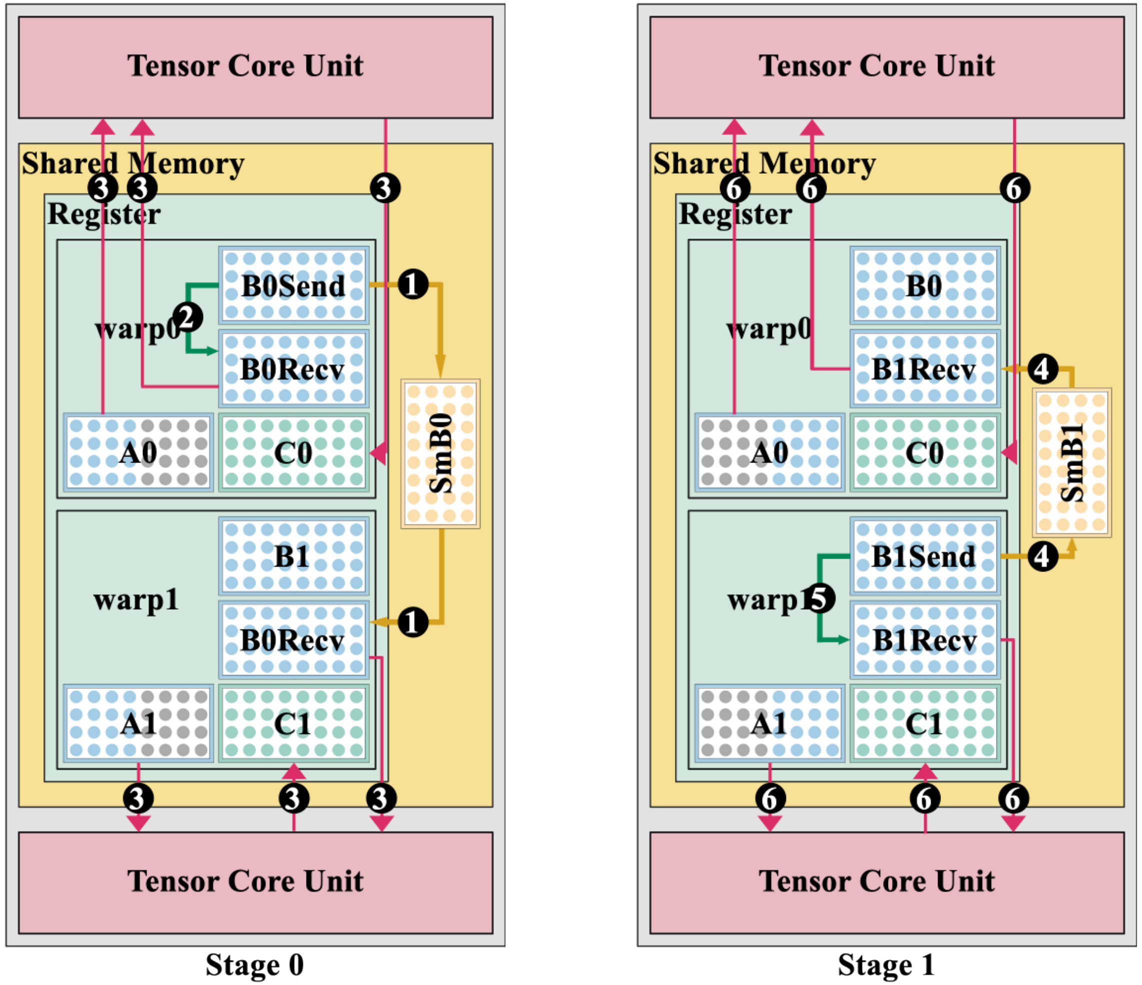
中国石油大学(北京) CHINA UNIVERSITY OF PETROLEUM
Lab 超级科学软件实验室 Super Scientific Software Laboratory

Method: 1D Algorithm



Process of 1D algorithm

Method: 1D Algorithm



Process of 1D algorithm

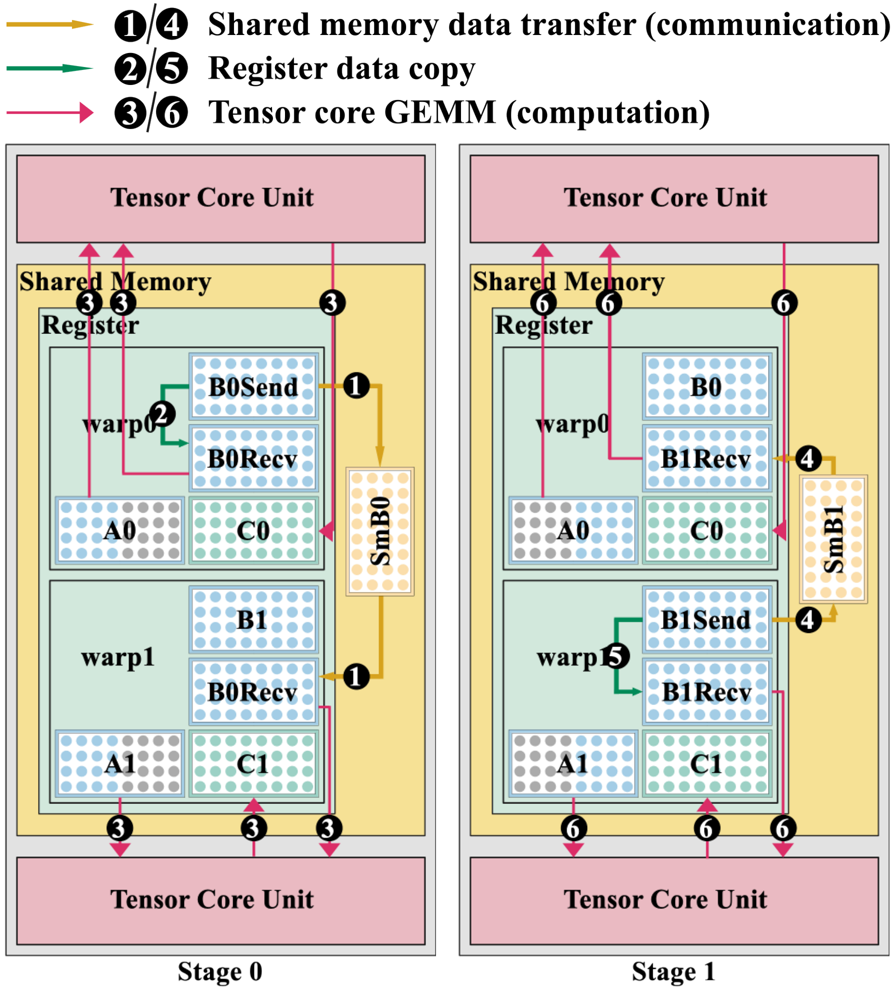
Method: 1D Algorithm

In the 1D algorithm, p warps perform a MM, with each warp (warpi, where 0 ≤ i < p) holding sub-matrix Ai (of size m/p×k) and sub-matrix Bi (of size k/p×n). The GPU warps operate in an

Algorithm 1 1D algorithm by p warps

$i \leftarrow$ warpID
GMem2Reg(Ai ← A, Bi ← B, Ci ← C)
for $z = 0$ **to** p **do** ▷ The algorithm consists of p stages.
  if $i = z$ **then**
    Reg2SMem(**SmB** ← **BSend**) ▷ Write **BSend** to shared memory.
    Reg2Reg(**BRecv** ← **BSend**) ▷ Copy **BSend** within registers.
  else if $i \neq z$ **then**
    DTransSMem2Reg(**BRecv** ← **SmB**) ▷ Read **SmB** from shared memory.
  Ci ← TensorCoreGEMM(Ai[:][$z \times \frac{k}{p}$: $(z+1) \times \frac{k}{p}$], **BRecv**)
    ▷ Part of Ai and **BRecv** multiplied by Tensor Core.
Reg2GMem(C ← Ci)

Pseudo-code for 1D algorithm



Process of 1D algorithm

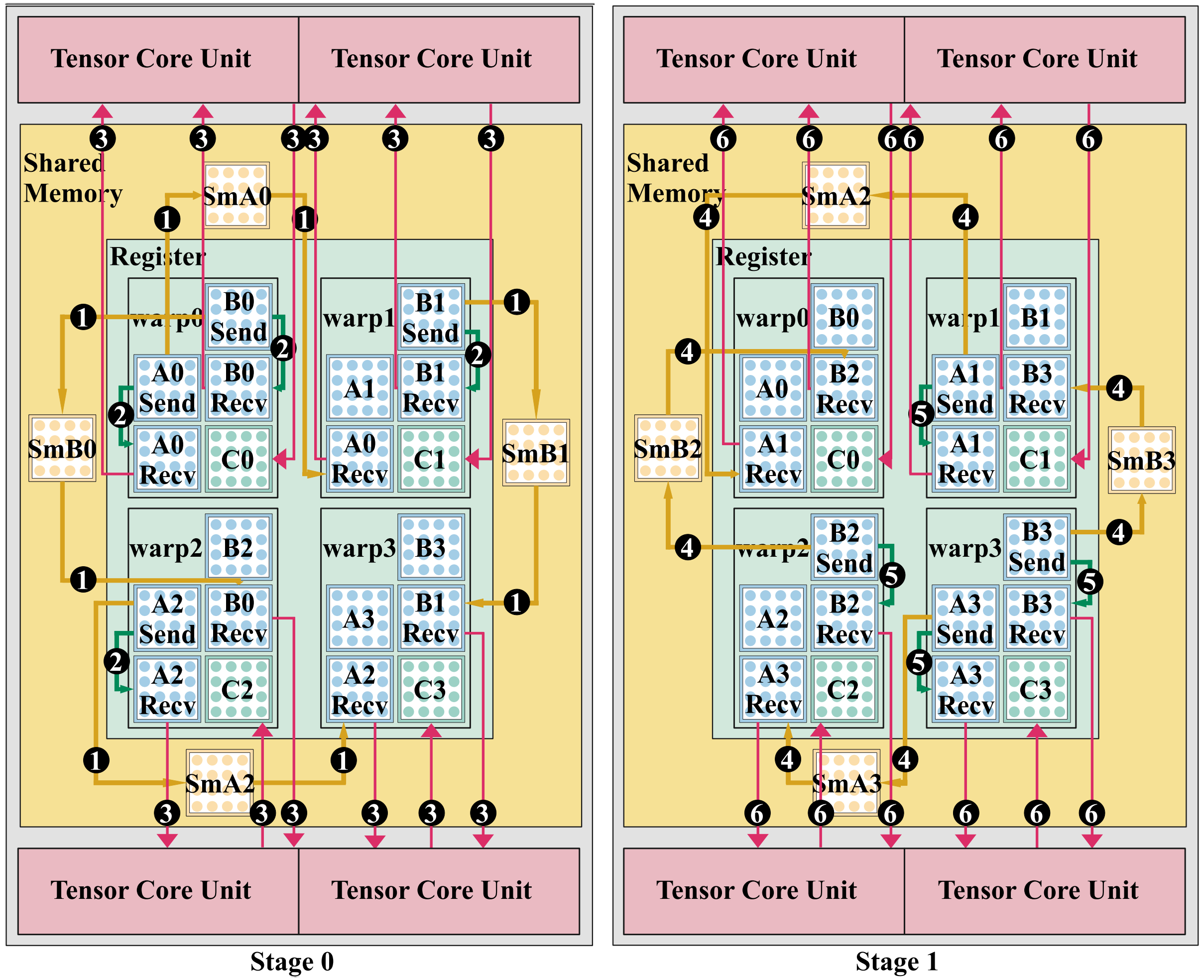
Method: 2D Algorithm

In the 2D algorithm, p warps are organized into a $\sqrt{p}\times\sqrt{p}$ grid for a GEMM, each warp holding A_i $(\frac{m}{\sqrt{p}}\times\frac{k}{\sqrt{p}})$ and B_i $(\frac{k}{\sqrt{p}}\times\frac{n}{\sqrt{p}})$.

Algorithm 2 2D algorithm by p warps

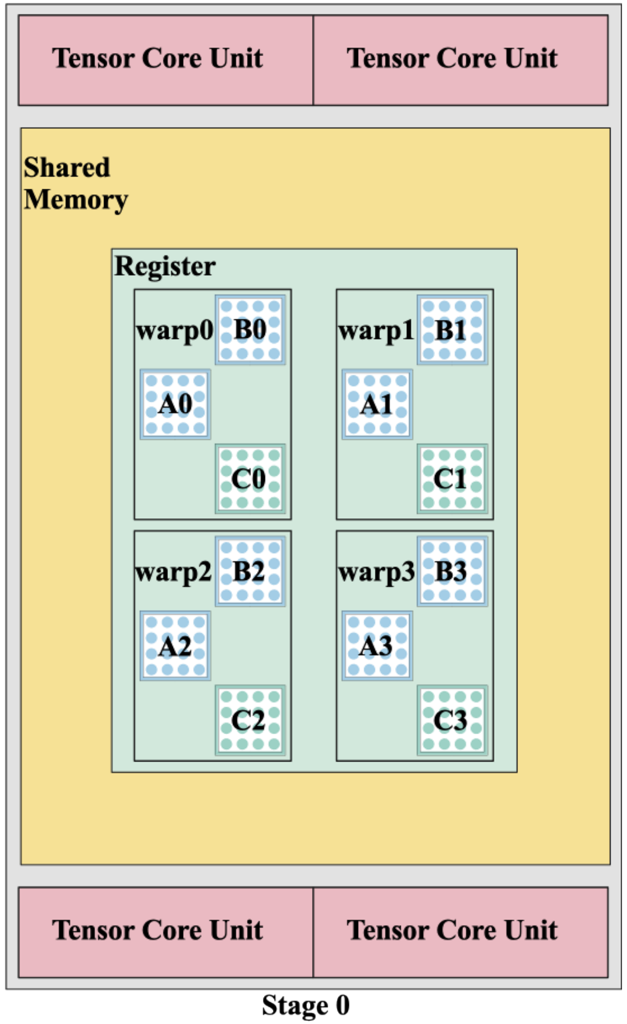
$i \leftarrow$ warpID
GMem2Reg($Ai \leftarrow$ A, $Bi \leftarrow$ B, $Ci \leftarrow$ C)
for $z = 0$ **to** $\sqrt{p}$ **do** ▷ The algorithm consists of $\sqrt{p}$ stages.
  if $i\%\sqrt{p} = z$ **then**
    Reg2SMem(**SmA** $\leftarrow$ **ASend**) ▷ Write **ASend** to shared memory.
    Reg2Reg(**ARecv** $\leftarrow$ **ASend**) ▷ Copy **ASend** between registers.
  if $i/\sqrt{p} = z$ **then**
    Reg2SMem(**SmB** $\leftarrow$ **BSend**) ▷ Write **BSend** to shared memory.
    Reg2Reg(**BRecv** $\leftarrow$ **BSend**) ▷ Copy **BSend** between registers.
  if $i\%\sqrt{p} \neq z$ **then**
    SMem2Reg(**ARecv** $\leftarrow$ **SmA**) ▷ Read **SmA** from shared memory.
  if $i/\sqrt{p} \neq z$ **then**
    SMem2Reg(**BRecv** $\leftarrow$ **SmB**) ▷ Read **SmB** from shared memory.
  $Ci \leftarrow$ TensorCoreGEMM(**ARecv**, **BRecv**) ▷ **ARecv** and **BRecv** multiplied by Tensor Core.
Reg2GMem($Ci \leftarrow$ C)

Pseudo-code for 2D algorithm



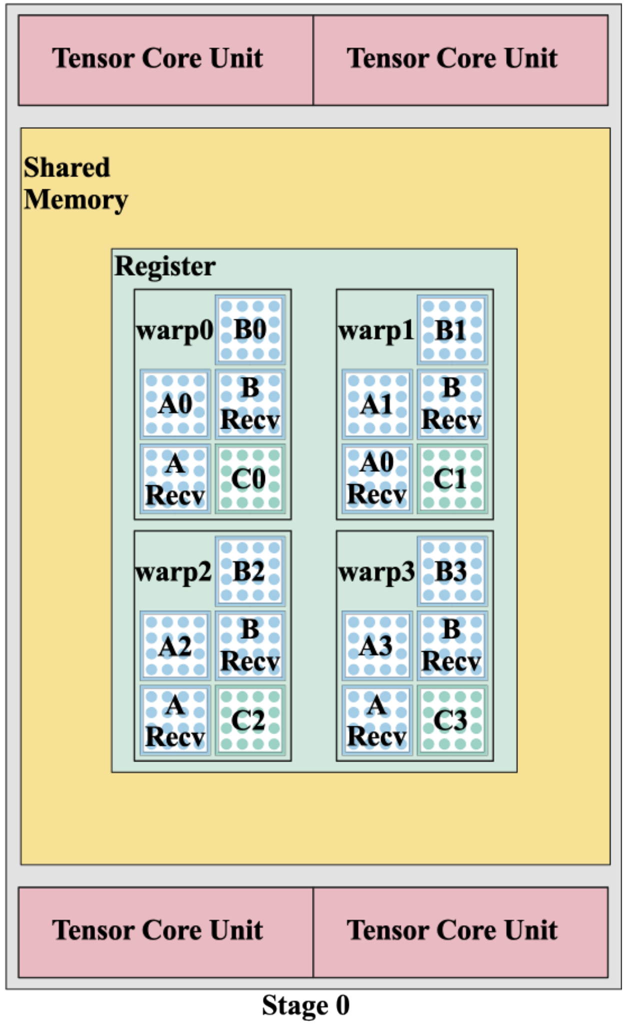
Process of 2D algorithm

Method: 2D Algorithm



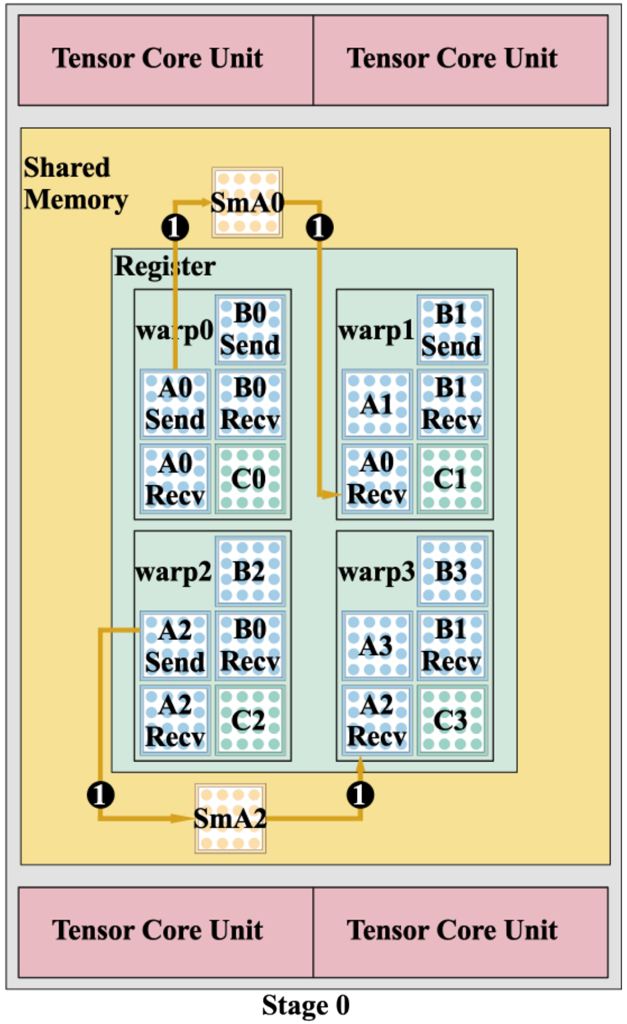
Process of 2D algorithm

Method: 2D Algorithm



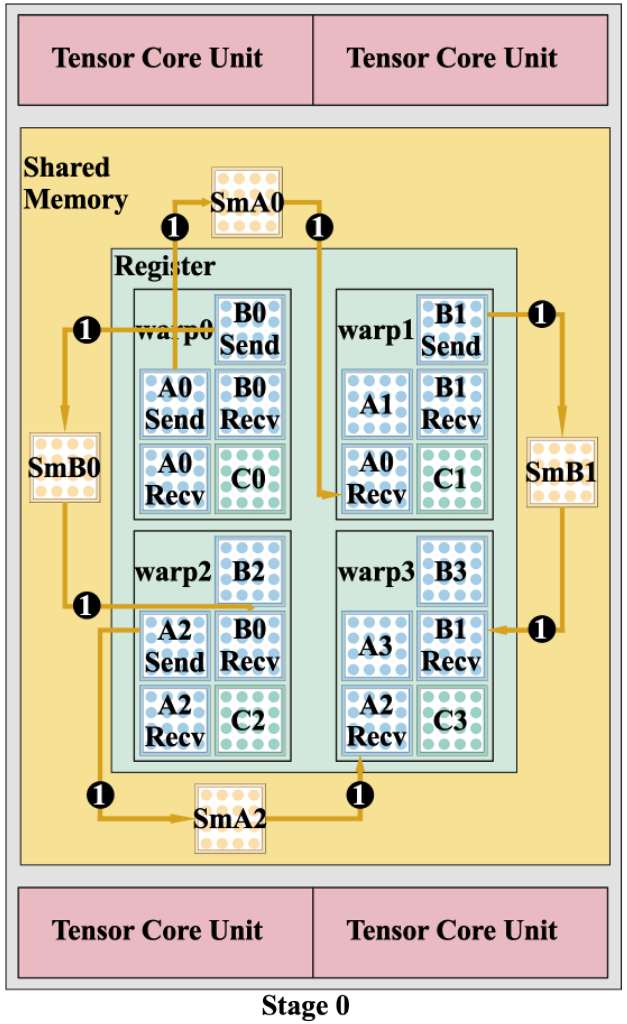
Process of 2D algorithm

Method: 2D Algorithm



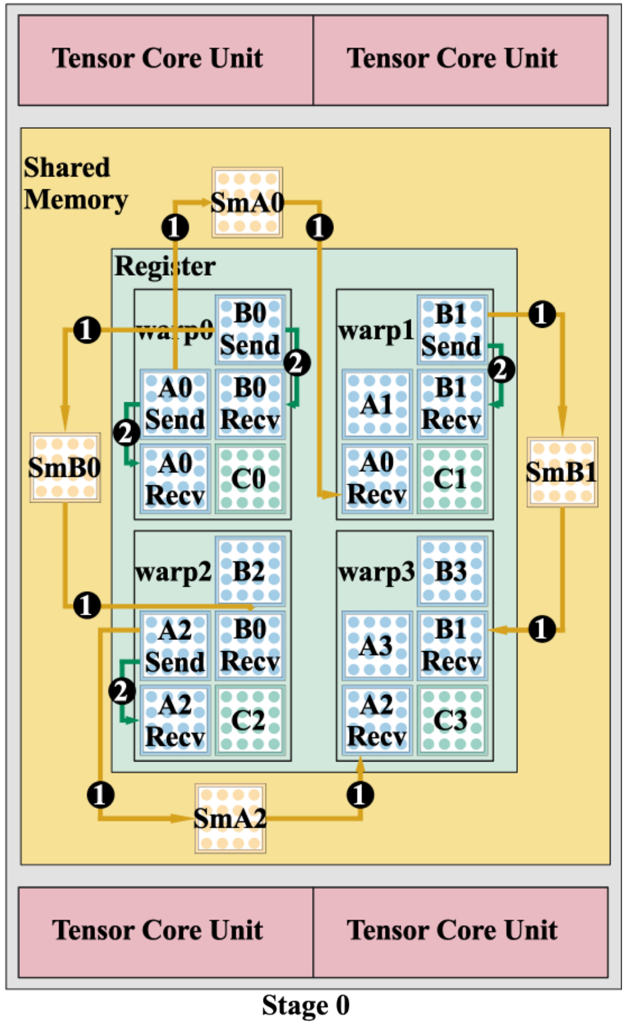
Process of 2D algorithm

Method: 2D Algorithm



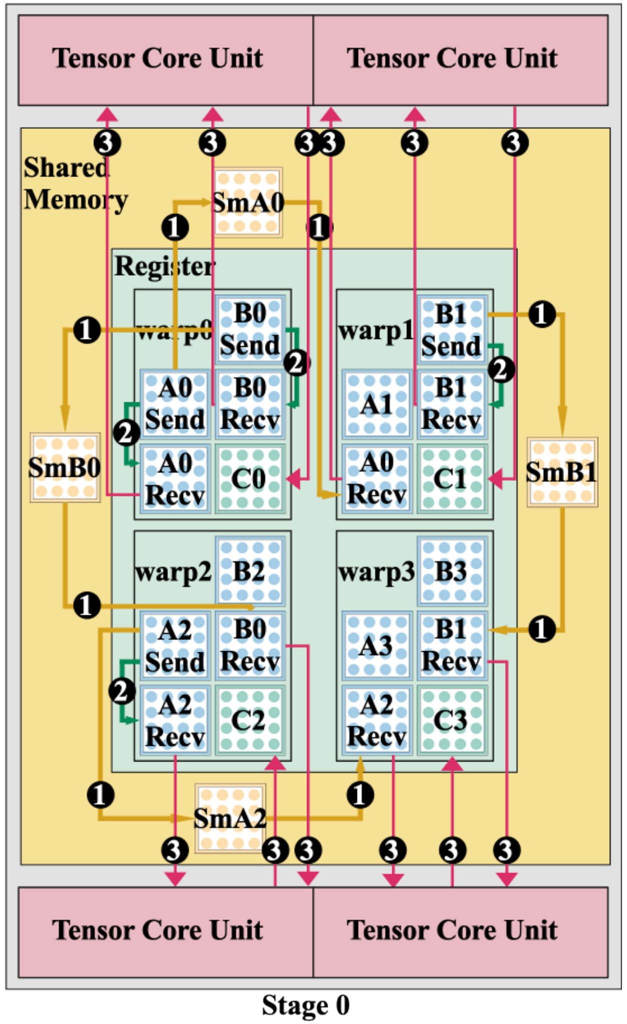
Process of 2D algorithm

Method: 2D Algorithm



Process of 2D algorithm

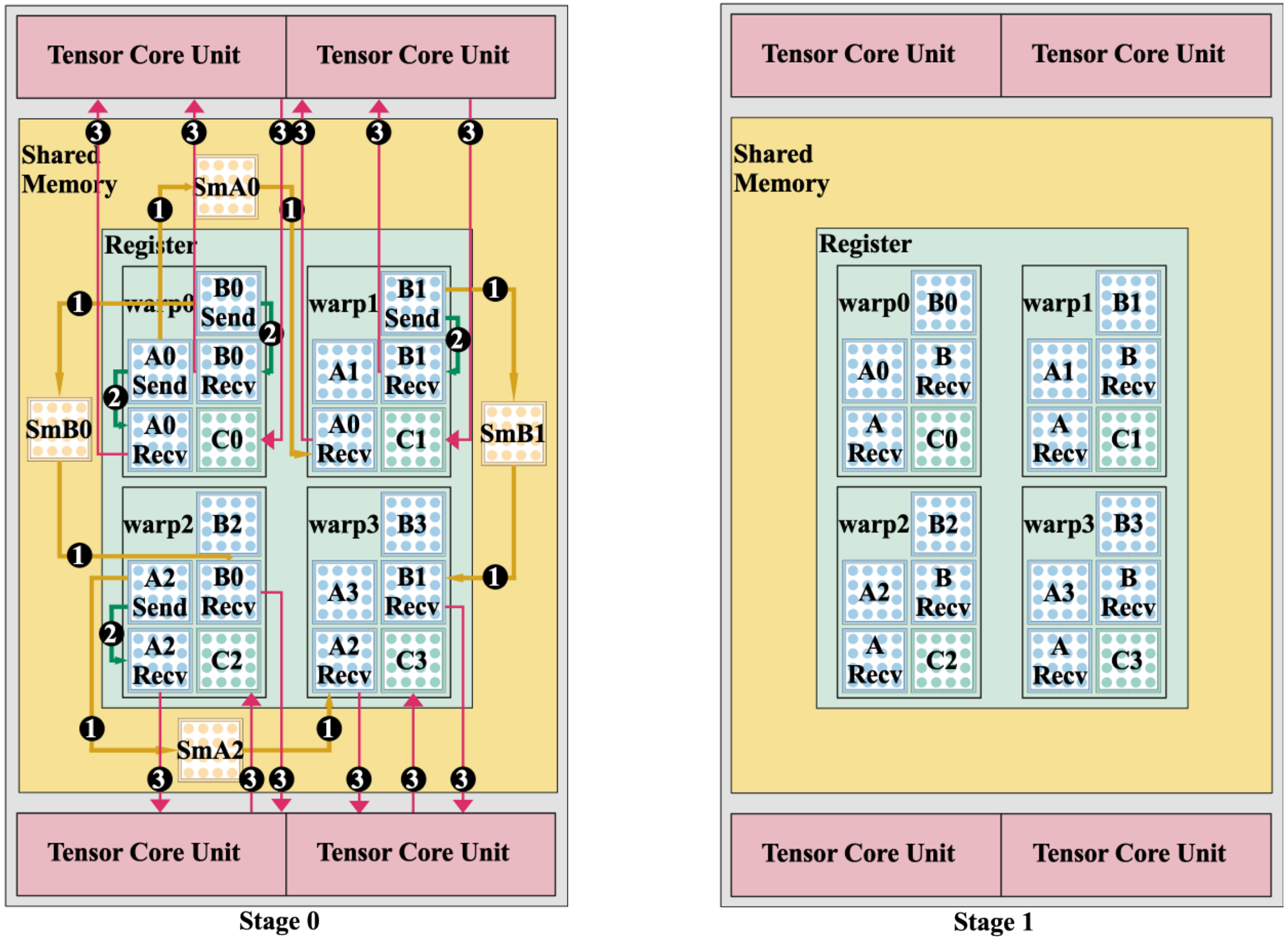
Method: 2D Algorithm



Process of 2D algorithm

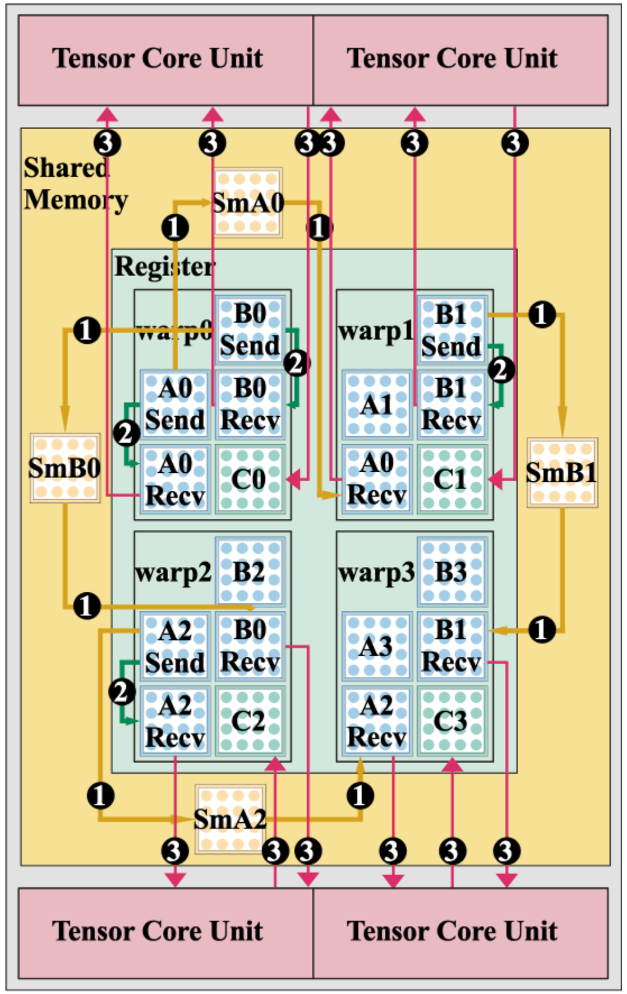
中国石油大学(北京) CHINA UNIVERSITY OF PETROLEUM
Lab 超级科学软件实验室 Super Scientific Software Laboratory

Method: 2D Algorithm

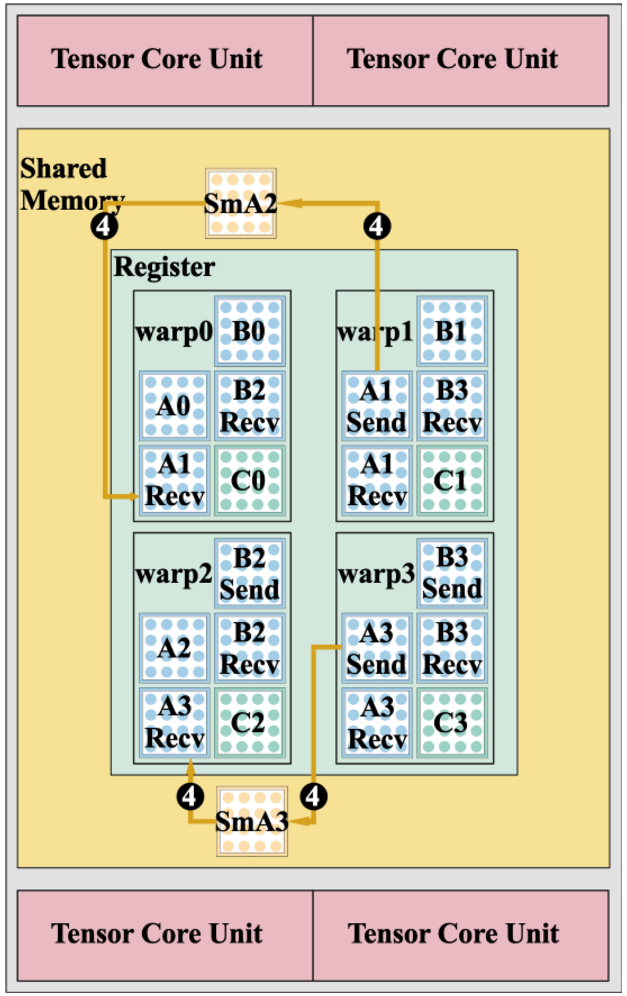


Process of 2D algorithm

Method: 2D Algorithm



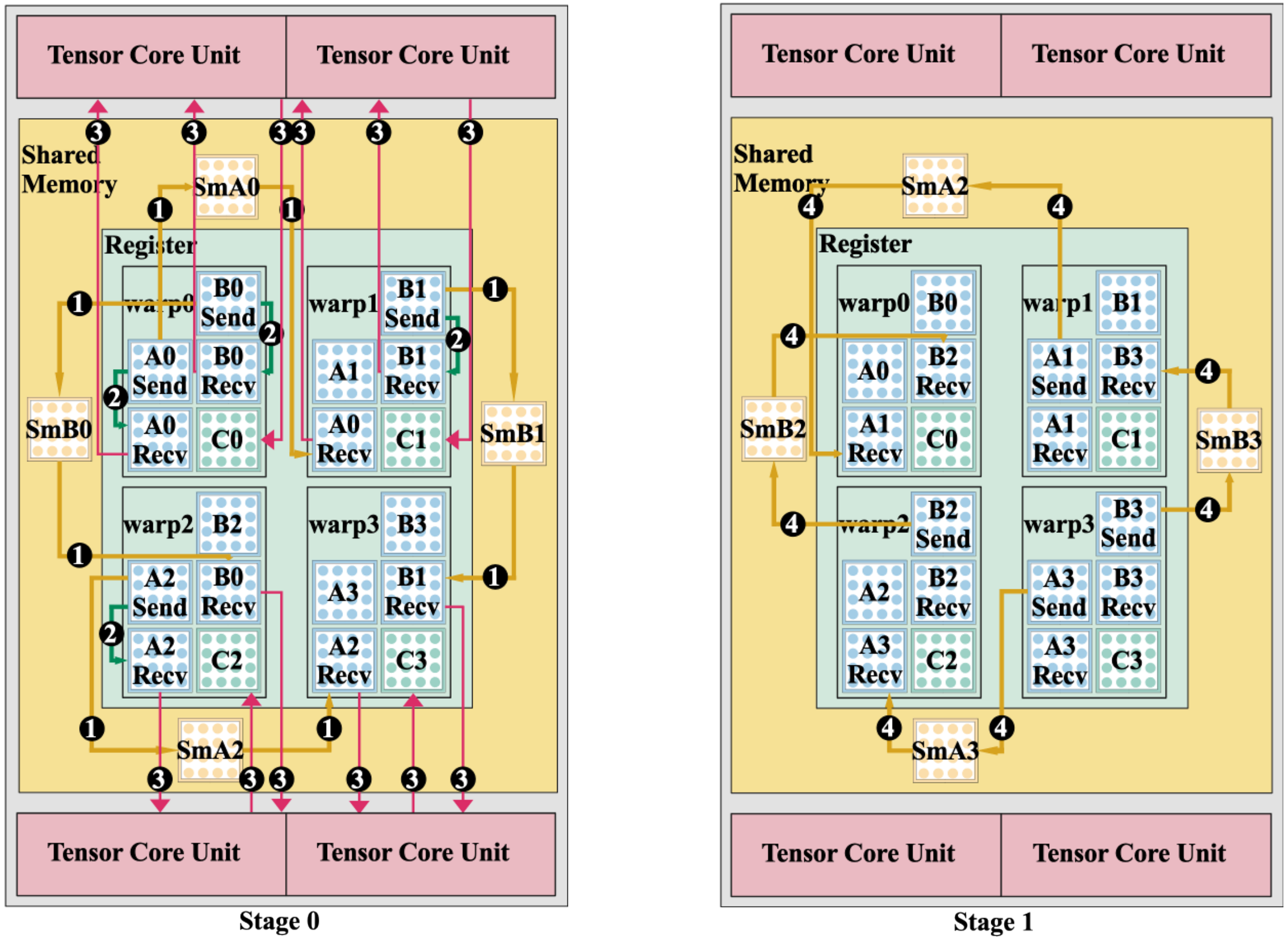
Stage 0



Stage 1

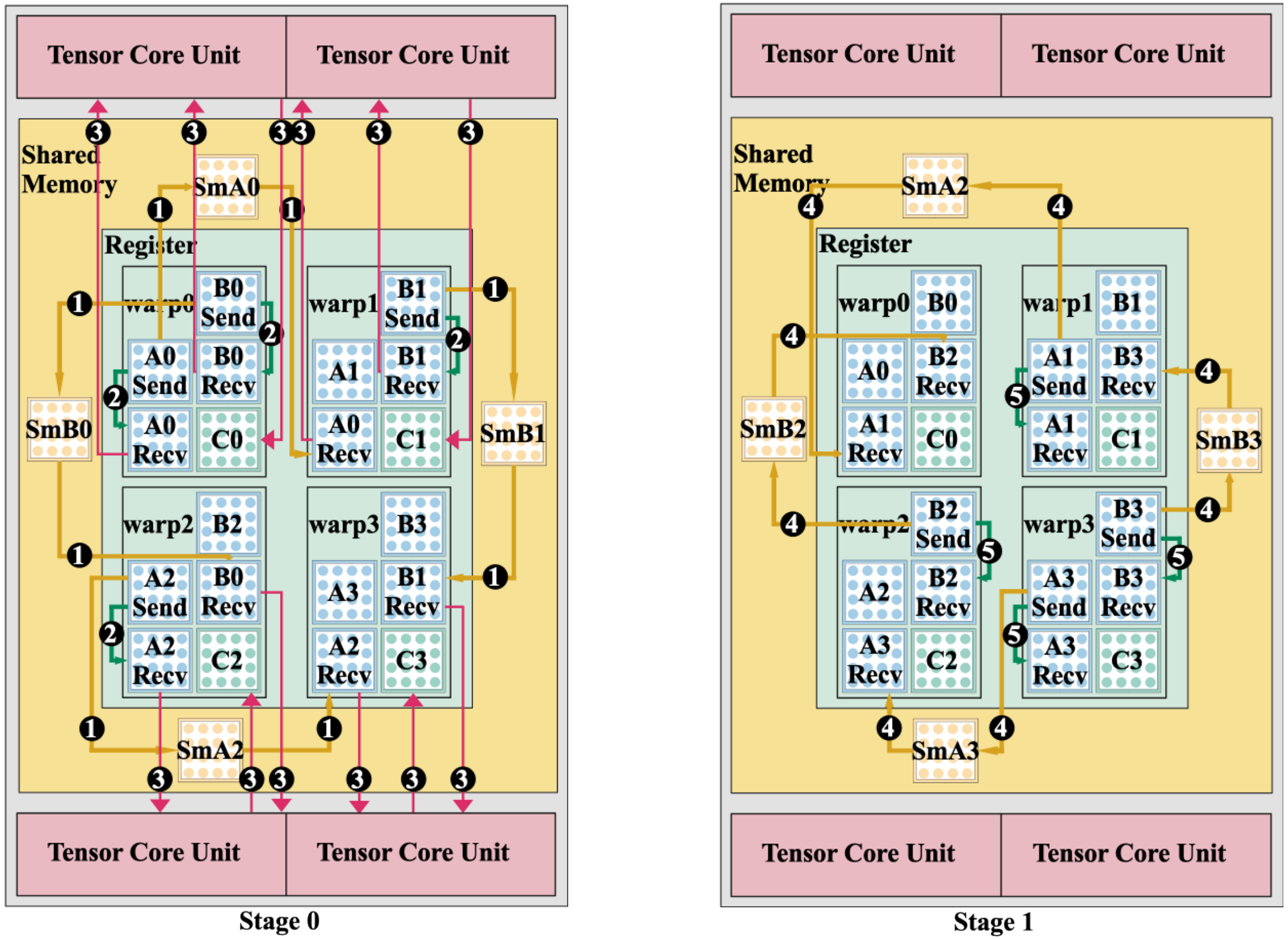
Process of 2D algorithm

Method: 2D Algorithm



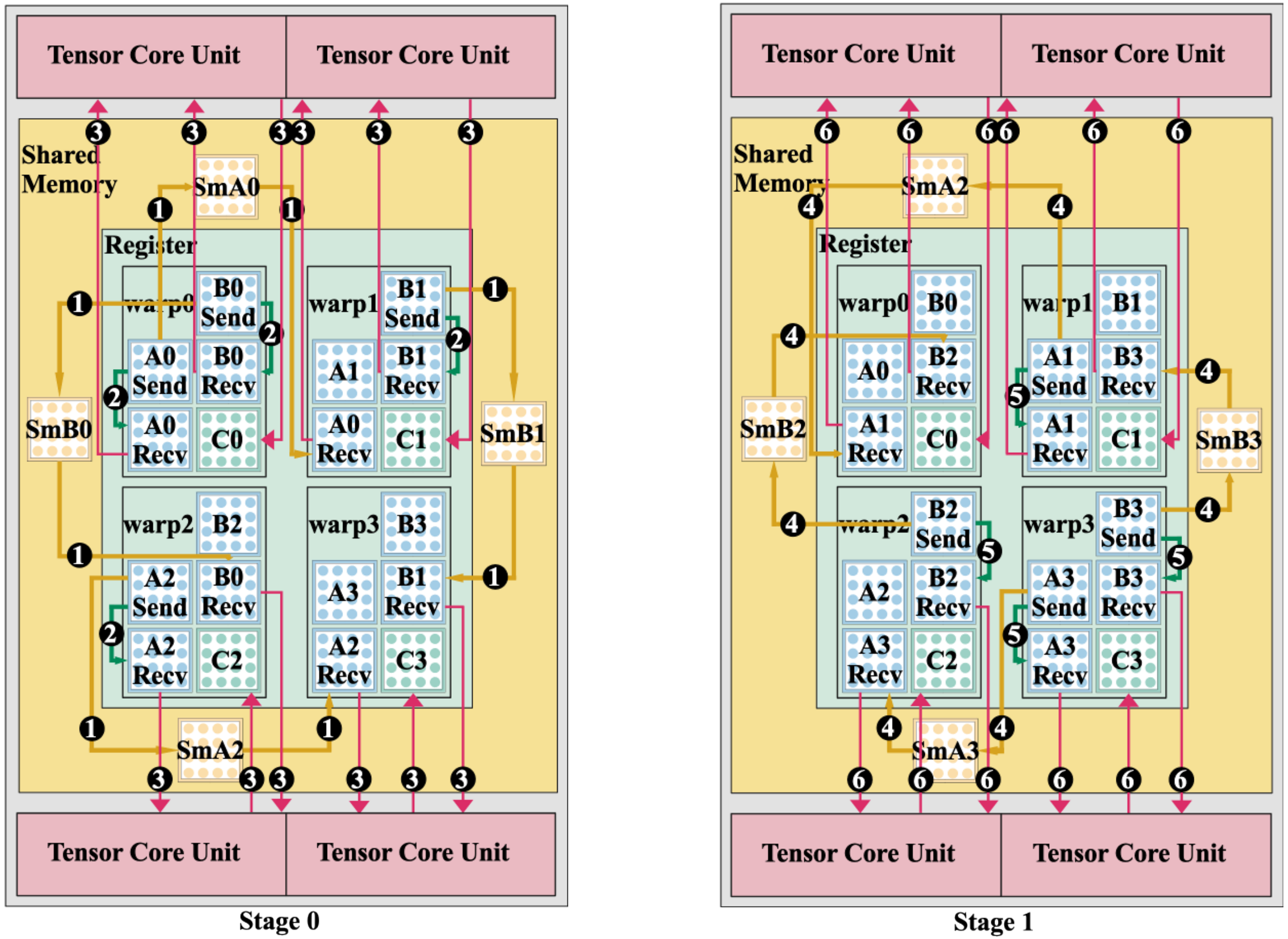
Process of 2D algorithm

Method: 2D Algorithm



Process of 2D algorithm

Method: 2D Algorithm



Process of 2D algorithm

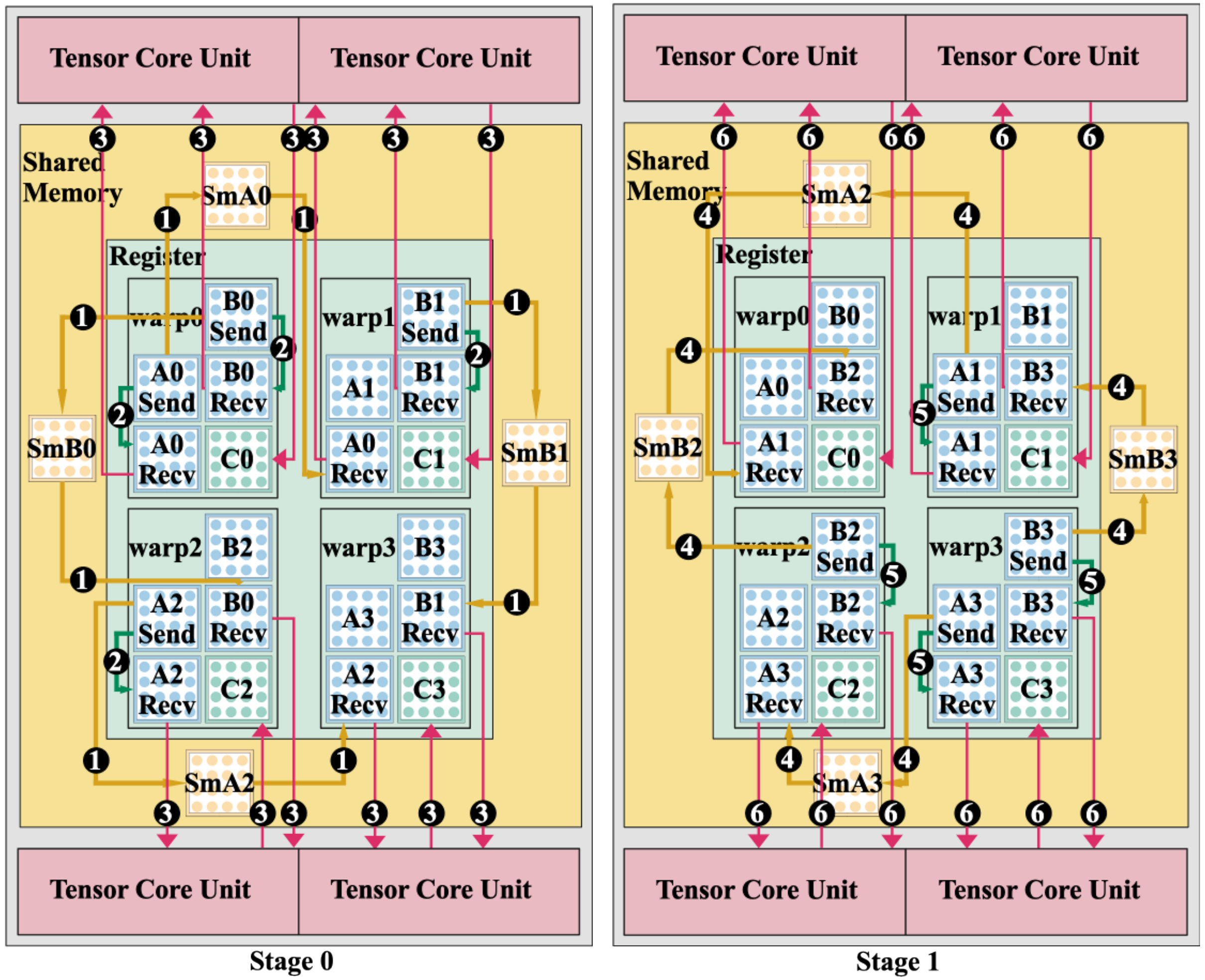
Method: 2D Algorithm

In the 2D algorithm, p warps are organized into a $\sqrt{p} \times \sqrt{p}$ grid for a GEMM, each warp holding A_i $\left(\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}\right)$ and B_i $\left(\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}\right)$.

Algorithm 2 2D algorithm by p warps

$i \leftarrow$ warpID
GMem2Reg($Ai \leftarrow$ A, $Bi \leftarrow$ B, $Ci \leftarrow$ C)
for $z = 0$ **to** $\sqrt{p}$ **do** ▷ The algorithm consists of $\sqrt{p}$ stages.
  if $i \% \sqrt{p} = z$ **then**
    Reg2SMem(**SmA** $\leftarrow$ **ASend**) ▷ Write **ASend** to shared memory.
    Reg2Reg(**ARecv** $\leftarrow$ **ASend**) ▷ Copy **ASend** between registers.
  if $i / \sqrt{p} = z$ **then**
    Reg2SMem(**SmB** $\leftarrow$ **BSend**) ▷ Write **BSend** to shared memory.
    Reg2Reg(**BRecv** $\leftarrow$ **BSend**) ▷ Copy **BSend** between registers.
  if $i \% \sqrt{p} \neq z$ **then**
    SMem2Reg(**ARecv** $\leftarrow$ **SmA**) ▷ Read **SmA** from shared memory.
  if $i / \sqrt{p} \neq z$ **then**
    SMem2Reg(**BRecv** $\leftarrow$ **SmB**) ▷ Read **SmB** from shared memory.
  Ci $\leftarrow$ TensorCoreGEMM(**ARecv**, **BRecv**)
    ▷ **ARecv** and **BRecv** multiplied by Tensor Core.
Reg2GMem(C$i \leftarrow$ C)

Pseudo-code for 2D algorithm



Process of 2D algorithm

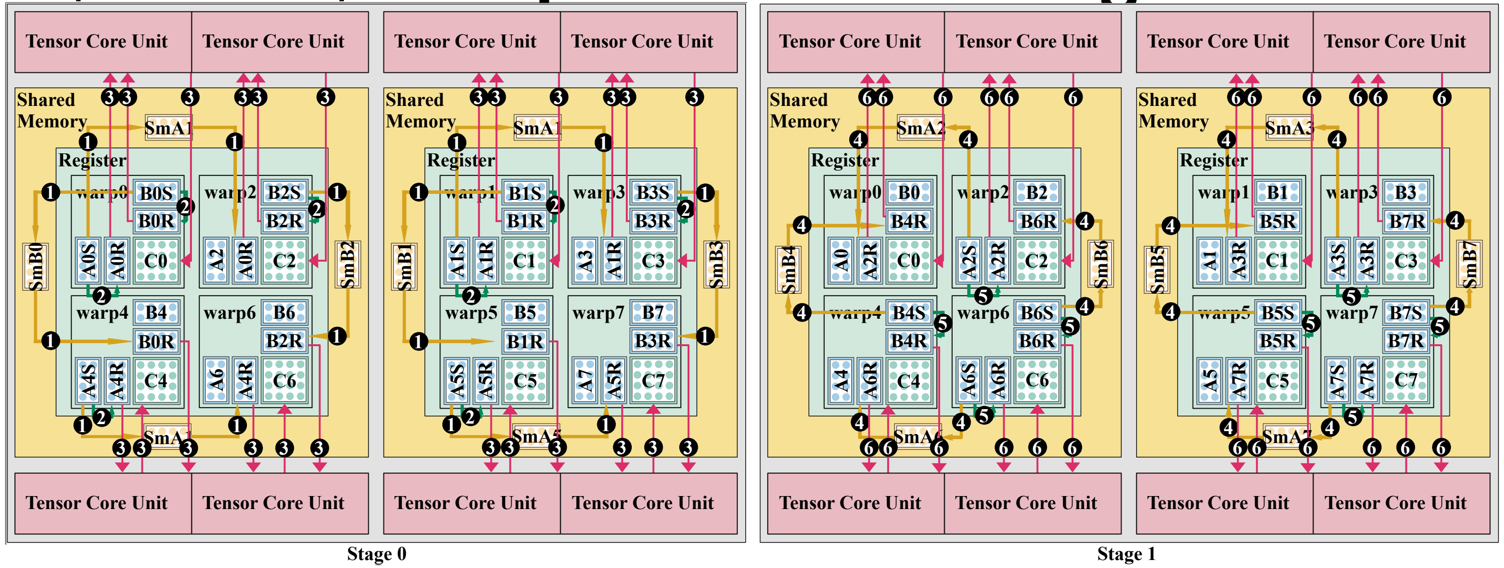
Method: 3D Algorithm

In the 3D algorithm, p warps are organized into a $\sqrt[3]{p}\times\sqrt[3]{p}\times\sqrt[3]{p}$ cube for a GEMM, each warp holding A_i $(\frac{m}{\sqrt[3]{p}}\times\frac{k}{\sqrt[3]{p}})$ and B_i $(\frac{k}{\sqrt[3]{p}}\times\frac{n}{\sqrt[3]{p}})$.This warp cube can be viewed as $\sqrt[3]{p}$ warp grids, each of size $\sqrt[3]{p}\times\sqrt[3]{p}$, where A_i and B_i are partitioned along the k-

Algorithm 3 3D algorithm by p warps

$i \leftarrow$ warpID
GMem2Reg($Ai \leftarrow$ A, $Bi \leftarrow$ B, $Ci \leftarrow$ C)
for $z = 0$ **to** $\sqrt[3]{p}$ **do** ▷ The algorithm consists of $\sqrt[3]{p}$ stages.
  if $i/\sqrt[3]{p}/\sqrt[3]{p} = z$ **then**
    Reg2SMem(SmA $\leftarrow$ ASend) ▷ Write **ASend** to shared memory.
    Reg2Reg(ARecv $\leftarrow$ ASend) ▷ Copy **ASend** between registers.
  if $i/\sqrt[3]{p}\%\sqrt[3]{p} = z$ **then**
    Reg2SMem(SmB $\leftarrow$ BSend) ▷ Write **BSend** to shared memory.
    Reg2Reg(BRecv $\leftarrow$ BSend) ▷ Copy **BSend** between registers.
  if $i/\sqrt[3]{p}/\sqrt[3]{p} \neq z$ **then**
    SMem2Reg(ARecv $\leftarrow$ SmA) ▷ Read **SmA** from shared memory.
  if $i/\sqrt[3]{p}\%\sqrt[3]{p} \neq z$ **then**
    SMem2Reg(BRecv $\leftarrow$ SmB) ▷ Read **SmB** from shared memory.
  $Ci \leftarrow$ TensorCoreGEMM(ARecv, BRecv) ▷ **ARecv** and **BRecv** multiplied by Tensor Core.
Reg2GMem($Ci \leftarrow C[i\%\sqrt[3]{p}]$)
$C \leftarrow C + C[i\%\sqrt[3]{p}]$

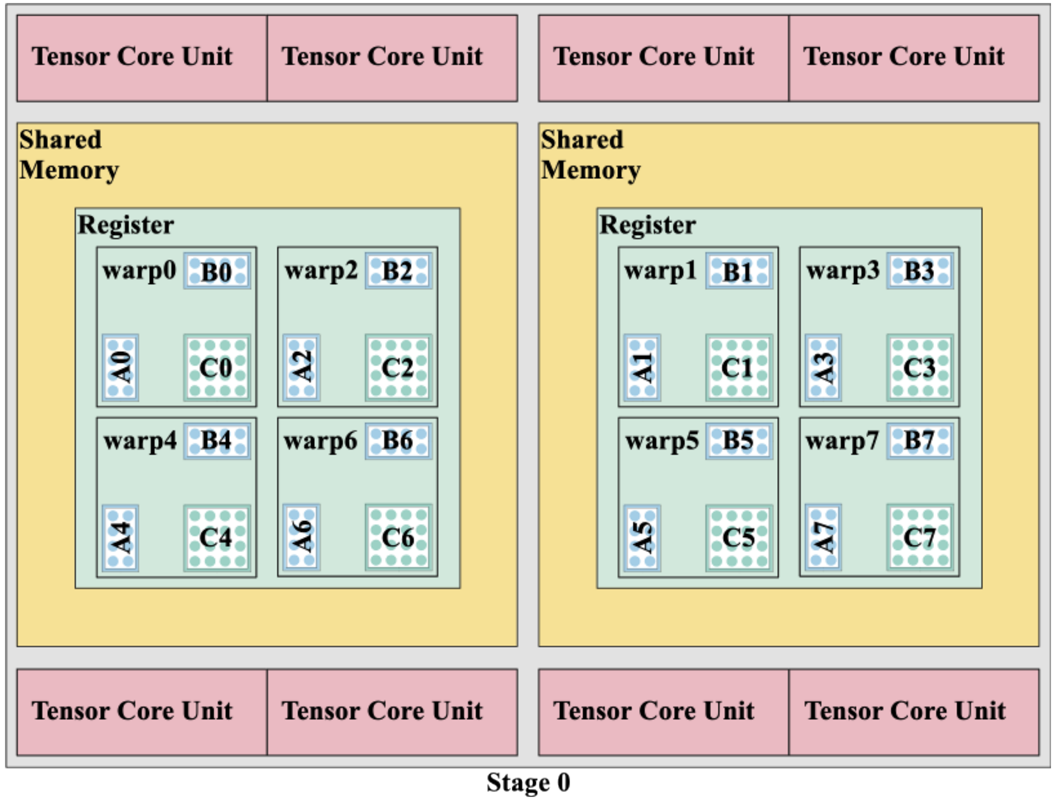
Pseudo-code for 3D algorithm



Process of 3D algorithm

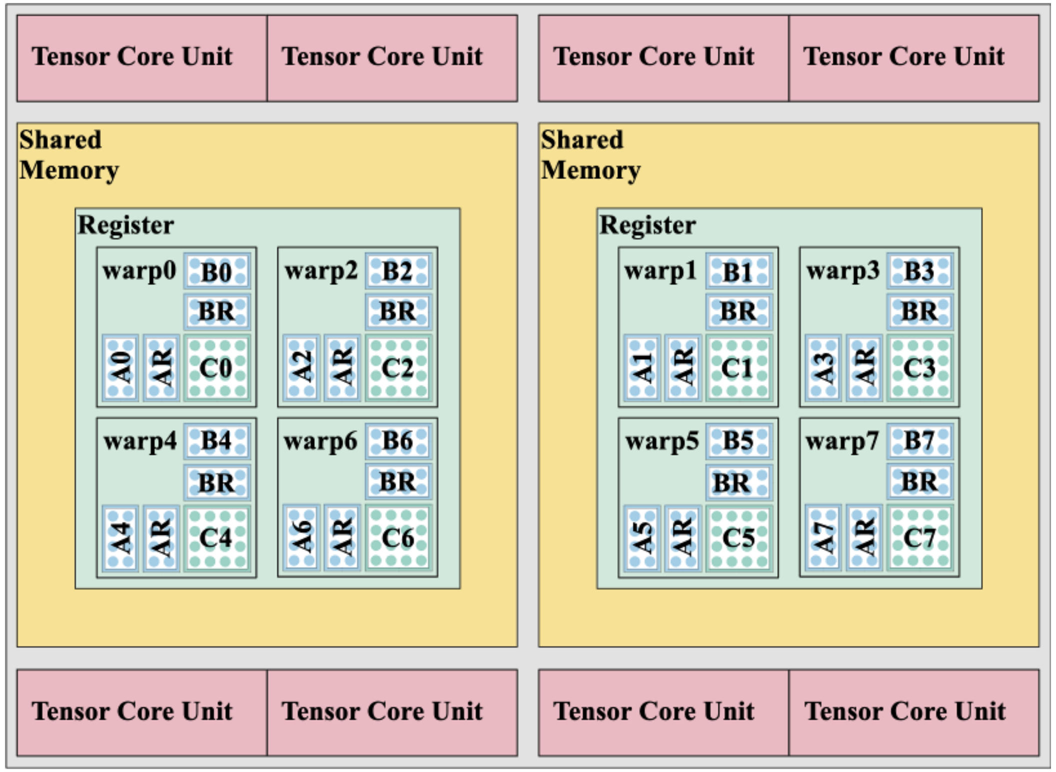
中国石油大学（北京） CHINA UNIVERSITY OF PETROLEUM
Lab 超级科学软件实验室 Super Scientific Software Laboratory

Method: 3D Algorithm



Process of 3D algorithm

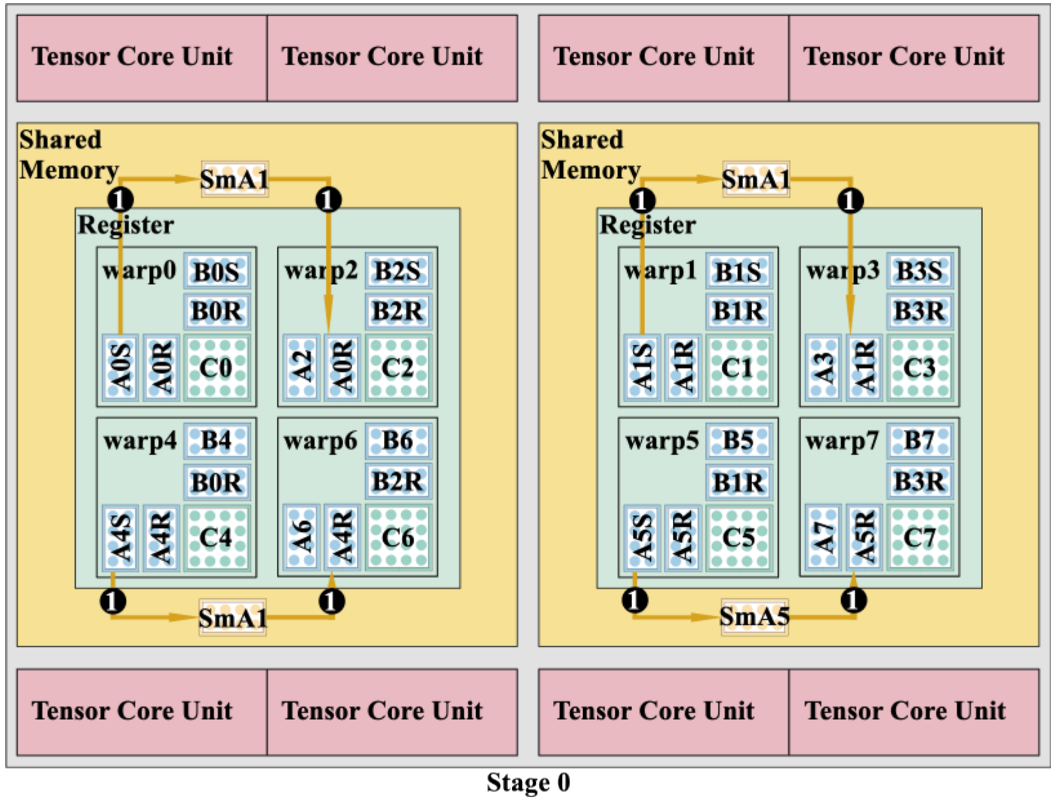
Method: 3D Algorithm



Stage 0

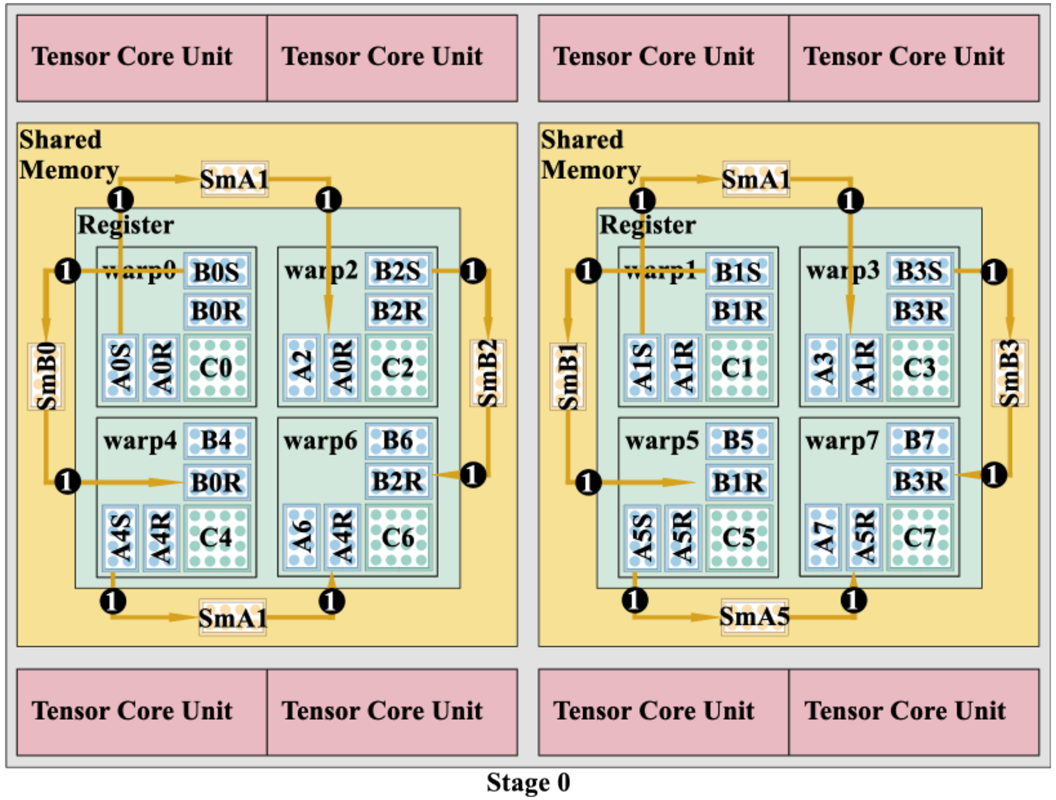
Process of 3D algorithm

Method: 3D Algorithm



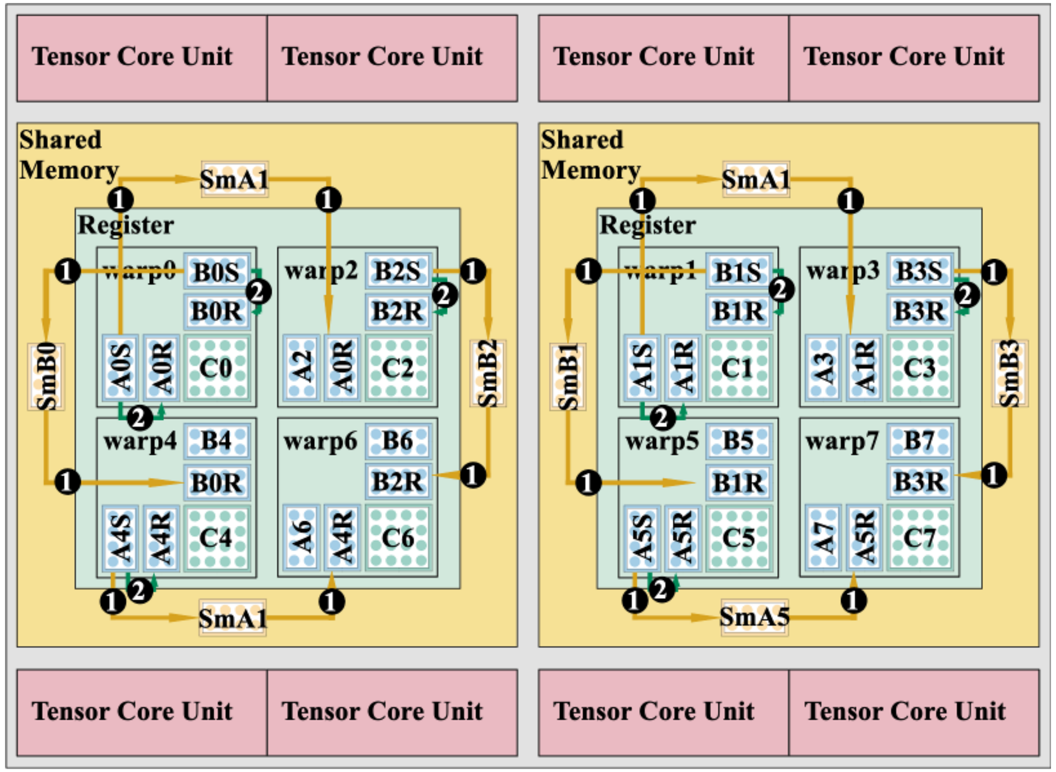
Process of 3D algorithm

Method: 3D Algorithm



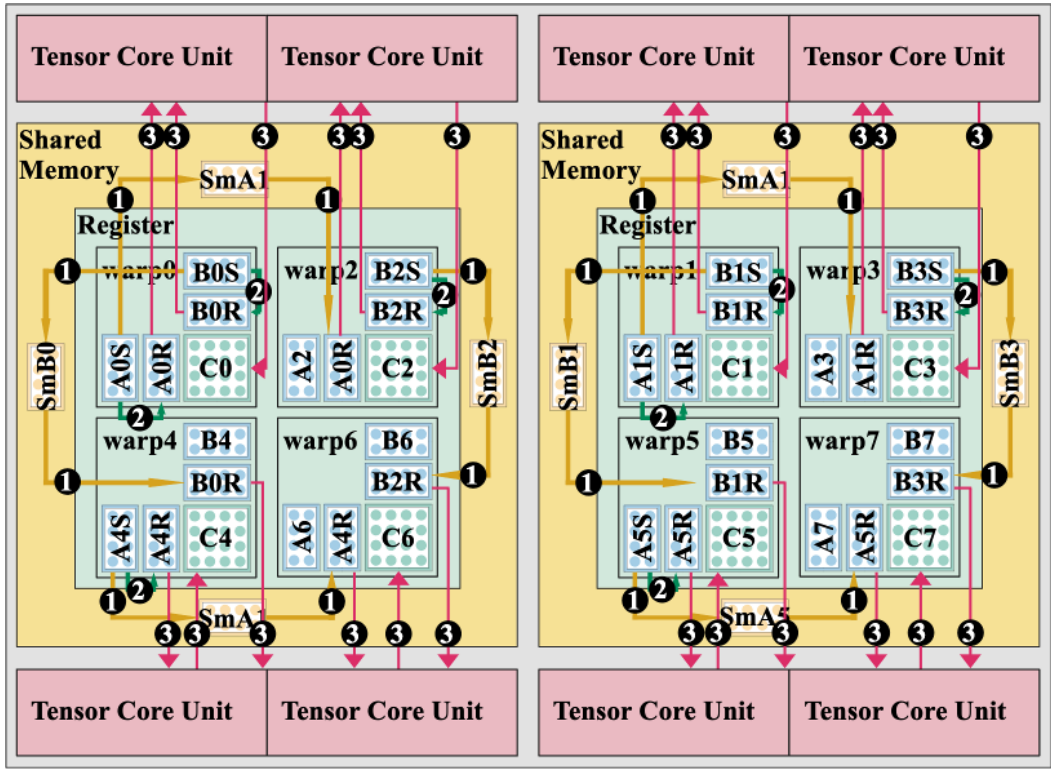
Process of 3D algorithm

Method: 3D Algorithm



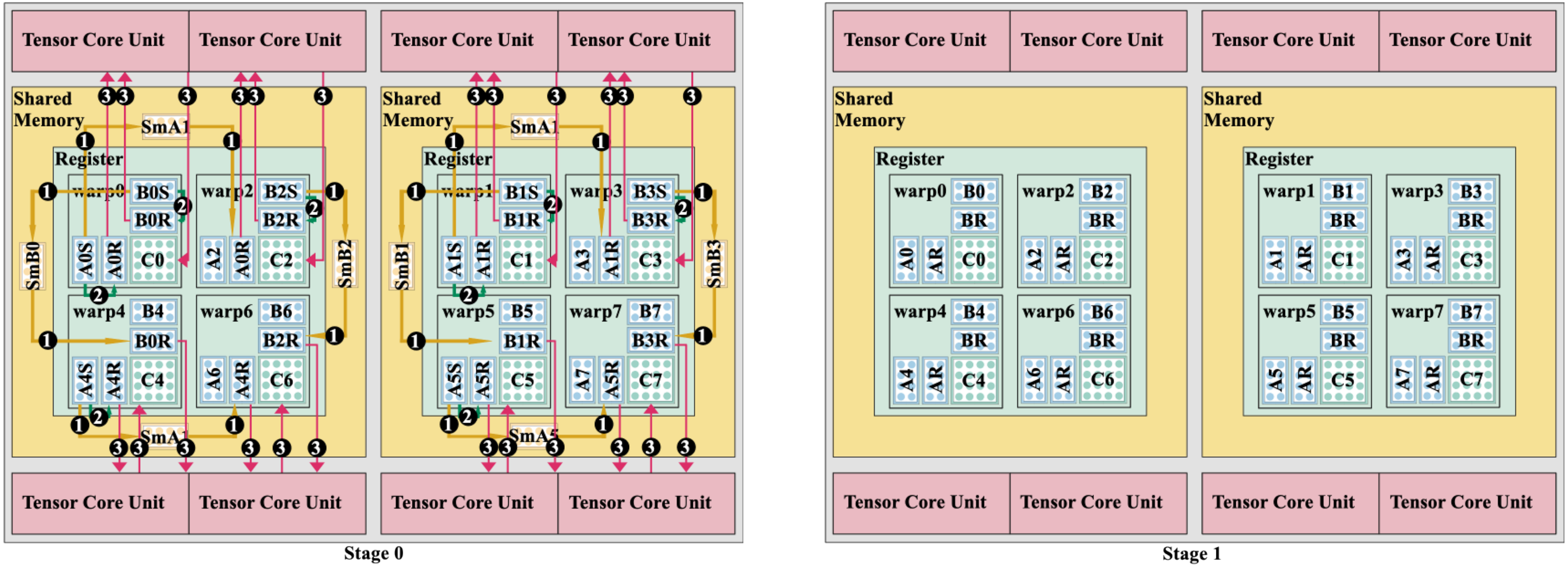
Process of 3D algorithm

Method: 3D Algorithm



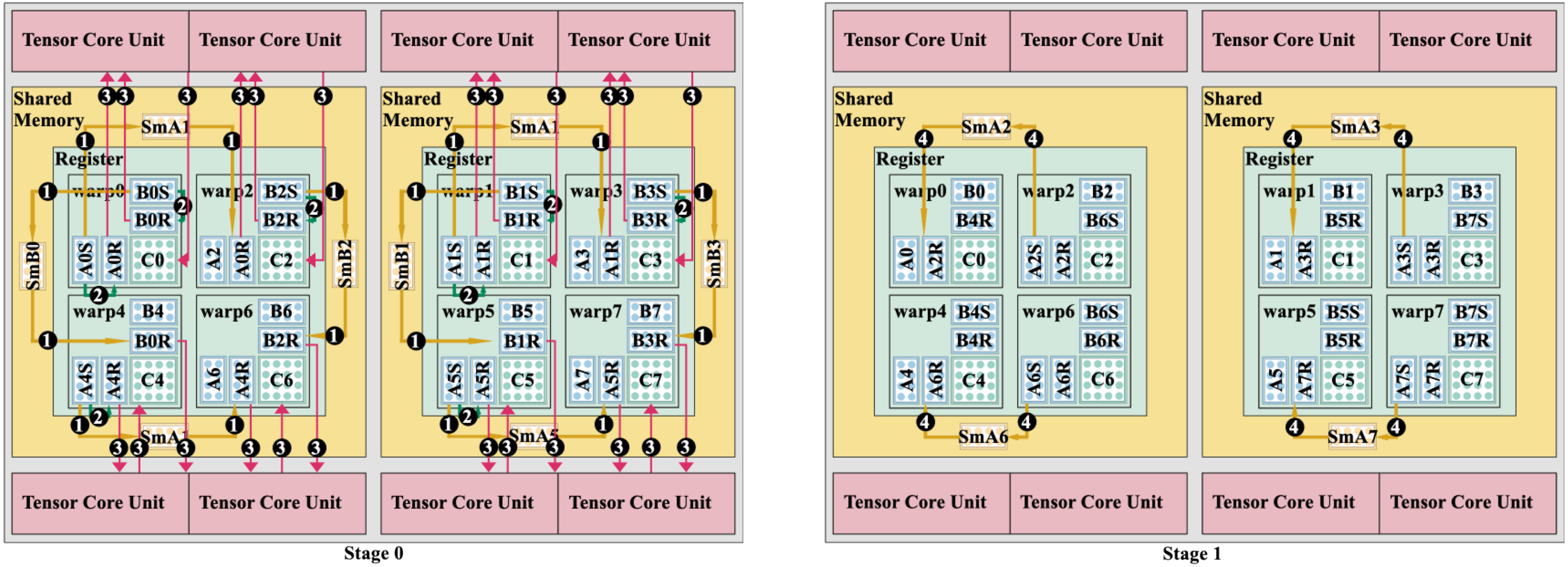
Process of 3D algorithm

Method: 3D Algorithm



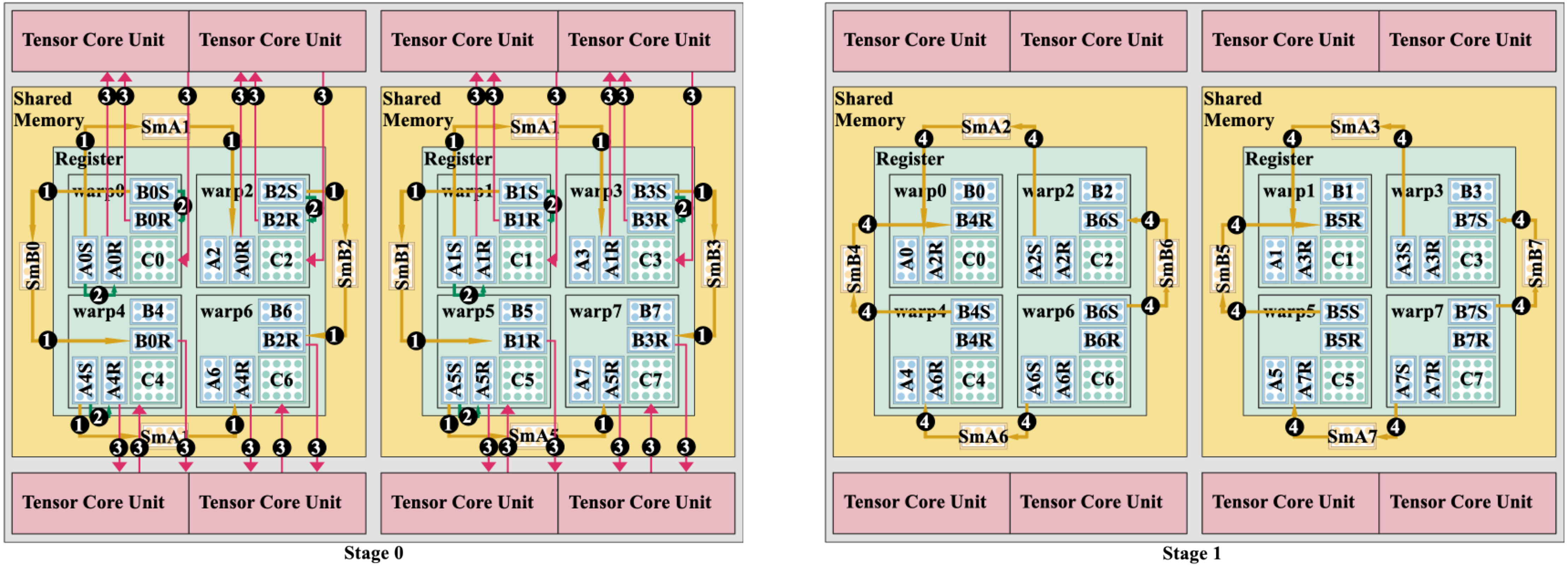
Process of 3D algorithm

Method: 3D Algorithm



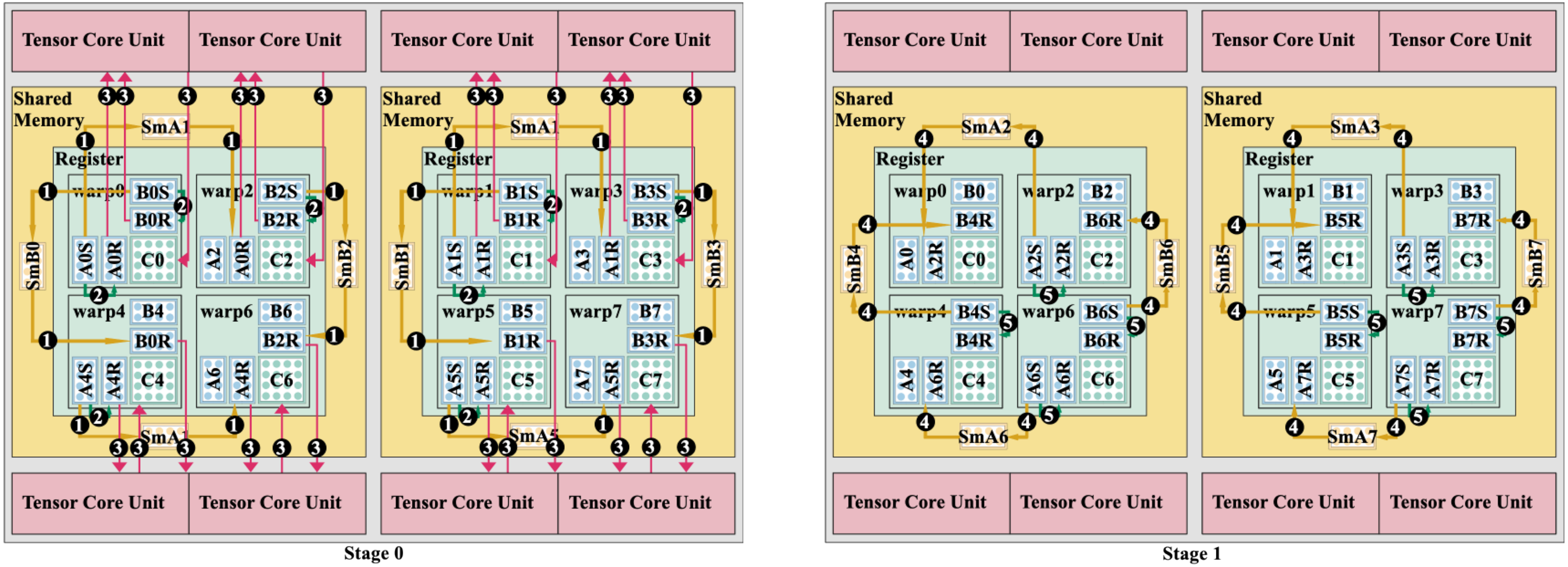
Process of 3D algorithm

Method: 3D Algorithm



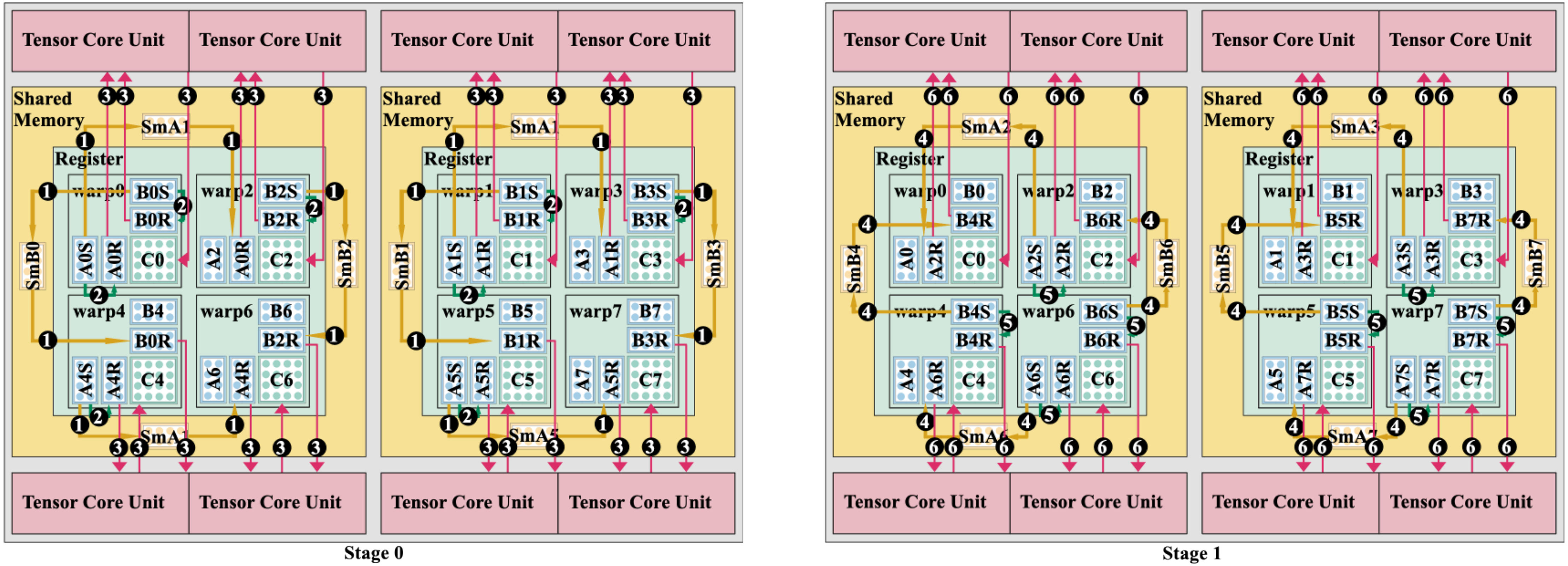
Process of 3D algorithm

Method: 3D Algorithm



Process of 3D algorithm

Method: 3D Algorithm



Process of 3D algorithm

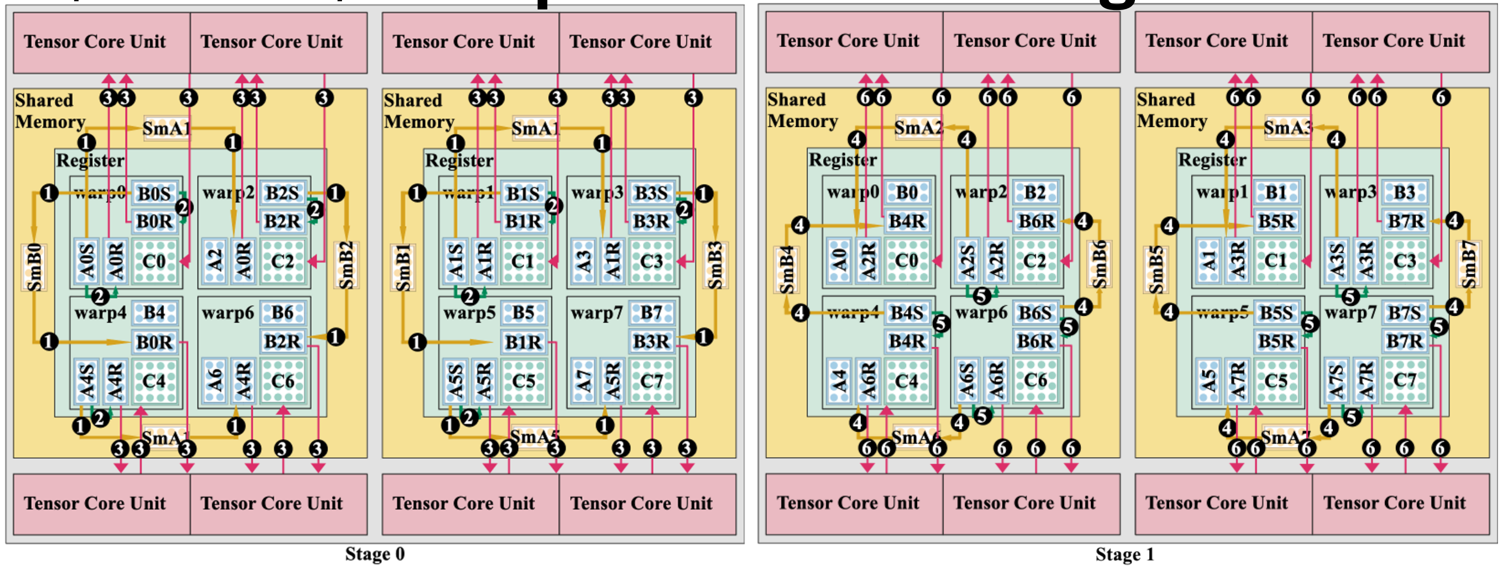
Method: 3D Algorithm

In the 3D algorithm, p warps are organized into a $\sqrt[3]{p}\times\sqrt[3]{p}\times\sqrt[3]{p}$ cube for a GEMM, each warp holding A_i $(\frac{m}{\sqrt[3]{p}}\times\frac{k}{\sqrt[3]{p}})$ and B_i $(\frac{k}{\sqrt[3]{p}}\times\frac{n}{\sqrt[3]{p}})$. This warp cube can be viewed as $\sqrt[3]{p}$ warp grids, each of size $\sqrt[3]{p}\times\sqrt[3]{p}$, where A_i and B_i are partitioned along the k-

Algorithm 3 3D algorithm by p warps

```
i ← warpID
GMem2Reg(Ai ← A, Bi ← B, Ci ← C)
for z = 0 to ∛p do                      ▷ The algorithm consists of ∛p stages.
    if i/∛p/∛p = z then
        Reg2SMem(SmA ← ASend)           ▷ Write ASend to shared memory.
        Reg2Reg(ARecv ← ASend)          ▷ Copy ASend between registers.
    if i/∛p%∛p = z then
        Reg2SMem(SmB ← BSend)           ▷ Write BSend to shared memory.
        Reg2Reg(BRecv ← BSend)          ▷ Copy BSend between registers.
    if i/∛p/∛p ≠ z then
        SMem2Reg(ARecv ← SmA)           ▷ Read SmA from shared memory.
    if i/∛p%∛p ≠ z then
        SMem2Reg(BRecv ← SmB)           ▷ Read SmB from shared memory.
    Ci ← TensorCoreGEMM(ARecv, BRecv)
                                        ▷ ARecv and BRecv multiplied by Tensor Core.
Reg2GMem(Ci ← C[i%∛p])
C ← C + C[i%∛p]
```

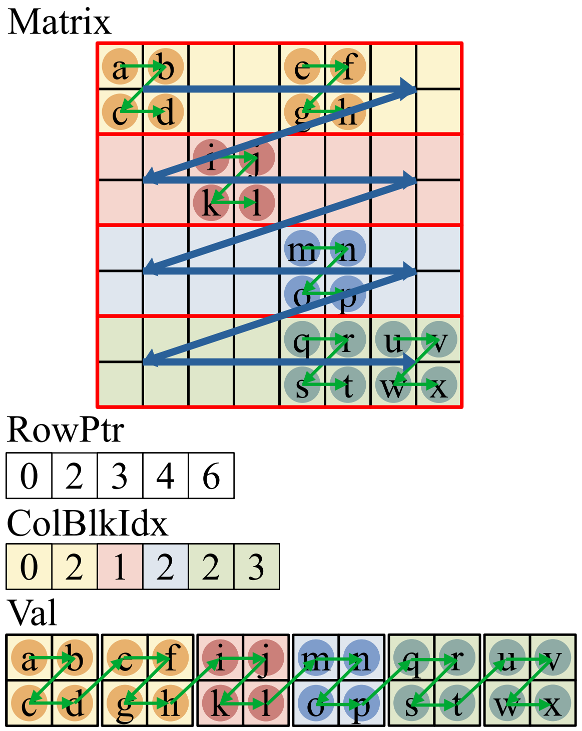
Pseudo-code for 3D algorithm



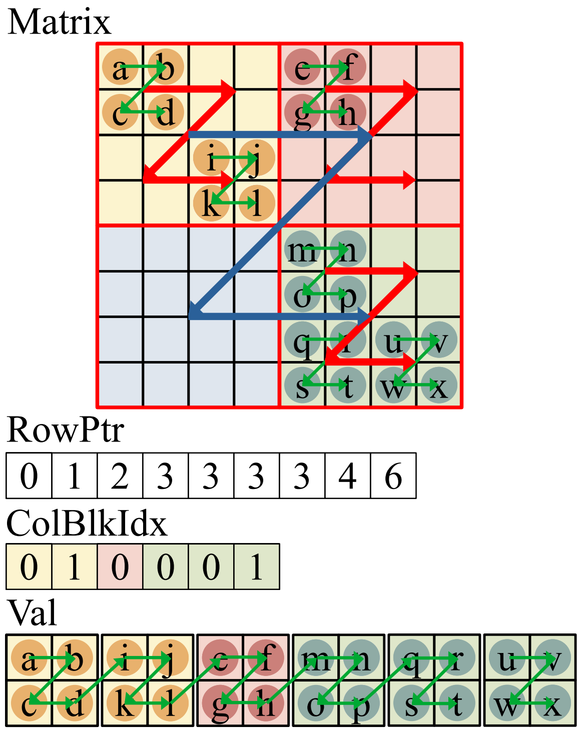
Process of 3D algorithm

Method: SpMM and SpGEMM

We have also extended KAMI to support sparse matrices. The matrices are stored in Z-Morton order as small dense sub-blocks, with a default size of 16×16 to adapt to different matrix computation units.



1D block sparse layout



2D/3D block sparse layout

OUTLINE

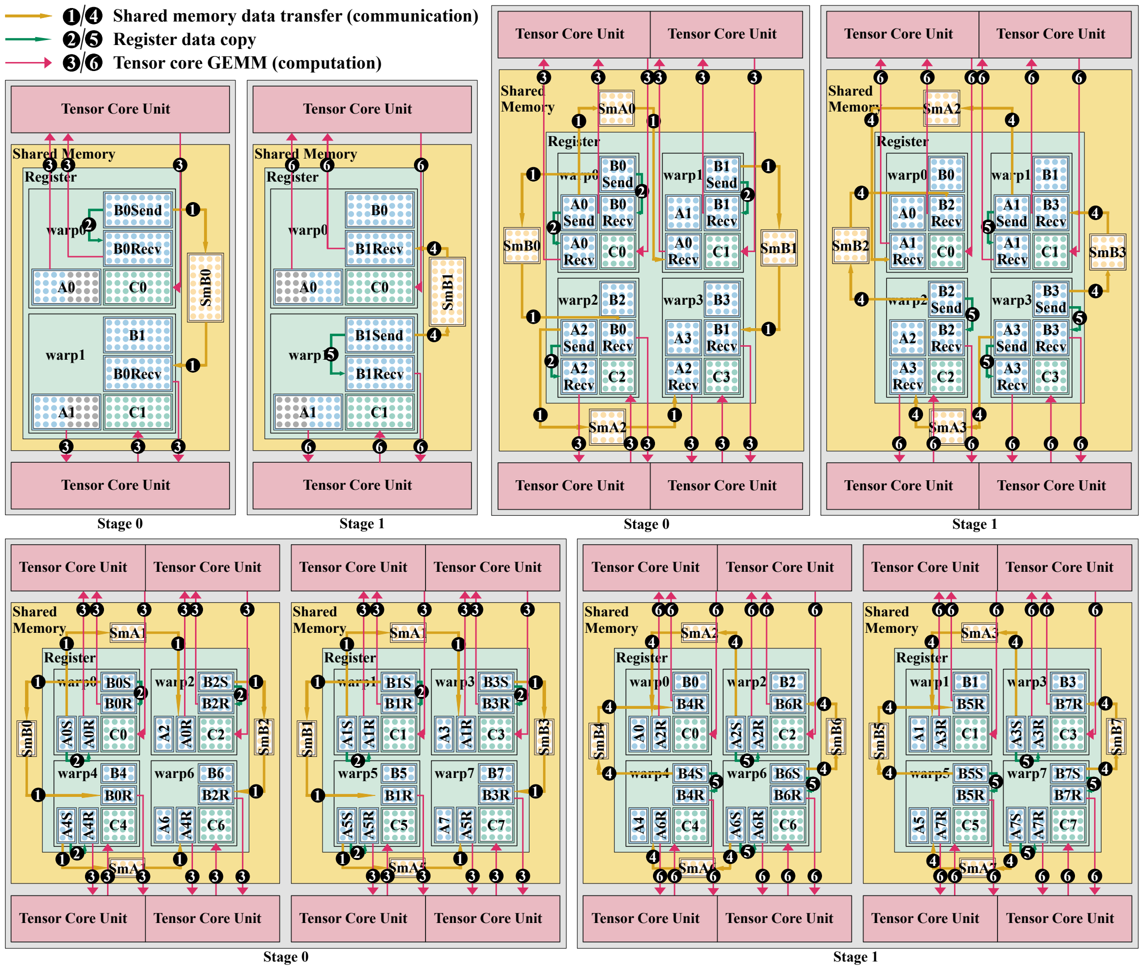
1 Introduction

2 Motivation

3 Method

4 Experiment

5 Conclusion

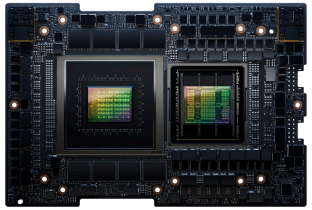


Experiments: setup

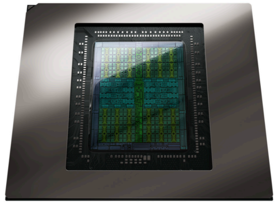
We evaluated KAMI on four GPUs from NVIDIA, AMD, and Intel, using CUDA, HIP, and SYCL respectively. The tests were conducted on the following GPUs: NVIDIA GH200, NVIDIA 5090, AMD 7900 XTX, and Intel Max 1100. We compared KAMI to cuBLASDx, CUTLASS, cuBLAS, MAGMA, and SYCL-Bench.

Vendor	NVIDIA		AMD	Intel
Specifications	GH200	RTX 5090	7900 XTX	Max 1100
Boost clock (MHz)	1980	2655	2498	1550
#Banks × bank width (Bytes)	32×4	32×4	32×4	16×4
#SMs × #tensor cores/SM	132×4	170×4	96×2	448×1
Peak FP16 tensor (TFLOPS)	990	462	123	22
Peak FP64 tensor (TFLOPS)	67	N/A	N/A	N/A

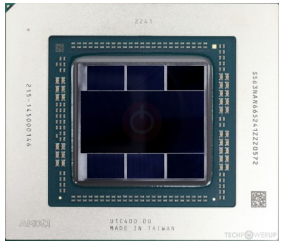
GPU specification.



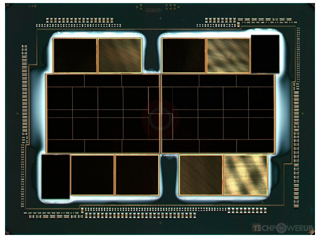
NVIDIA GH200



NVIDIA RTX 5090



AMD 7900 XTX



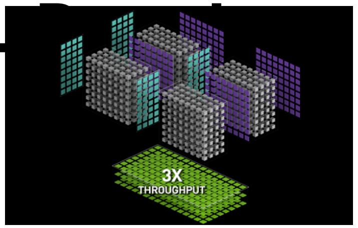
Intel Max 1100

Experiments: setup

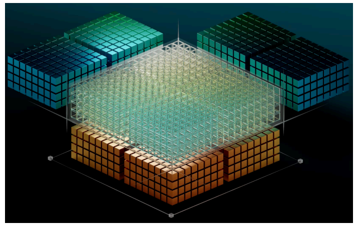
We evaluated KAMI on four GPUs from NVIDIA, AMD, and Intel, using CUDA, HIP, and SYCL respectively. The tests were conducted on the following GPUs: NVIDIA GH200, NVIDIA 5090, AMD 7900 XTX, and Intel Max 1100. We compared KAMI to cuBLASDx, CUTLASS, cuBLAS, MAGMA, and SYCL

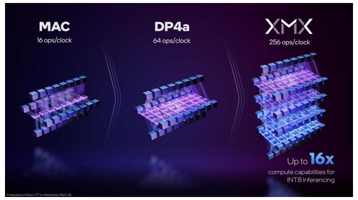
GPU Vendor	NVIDIA	AMD	Intel
Programming API	CUDA	HIP	SYCL
Local storage	Register	fragment	joint_matrix
Communication space	Shared memory	Shared memory	Local memory
Tensor core func.	mma	mma_sync	joint_matrix_mad
Instruction shape	m16n8k8 (FP64) m16n8k16 (FP16)	m16n16k16 (FP16)	m16n16k16 (FP16)

Programming API supported by KAMI.



NVIDIA Tensor Core

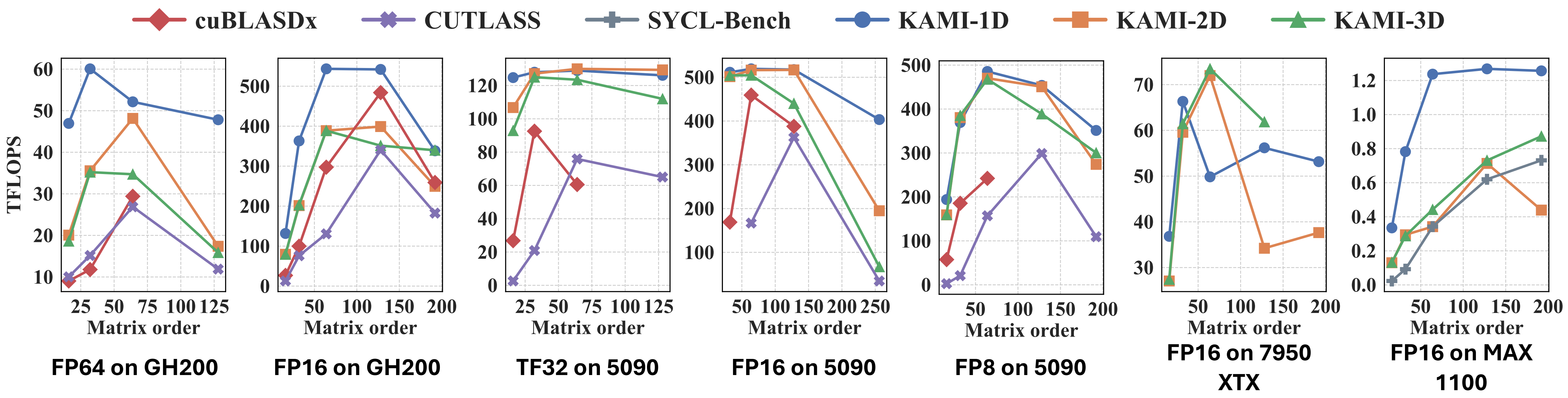

AMD Matrix Core



Intel Xe Matrix eXtension

Experiments: block-level square GEMM

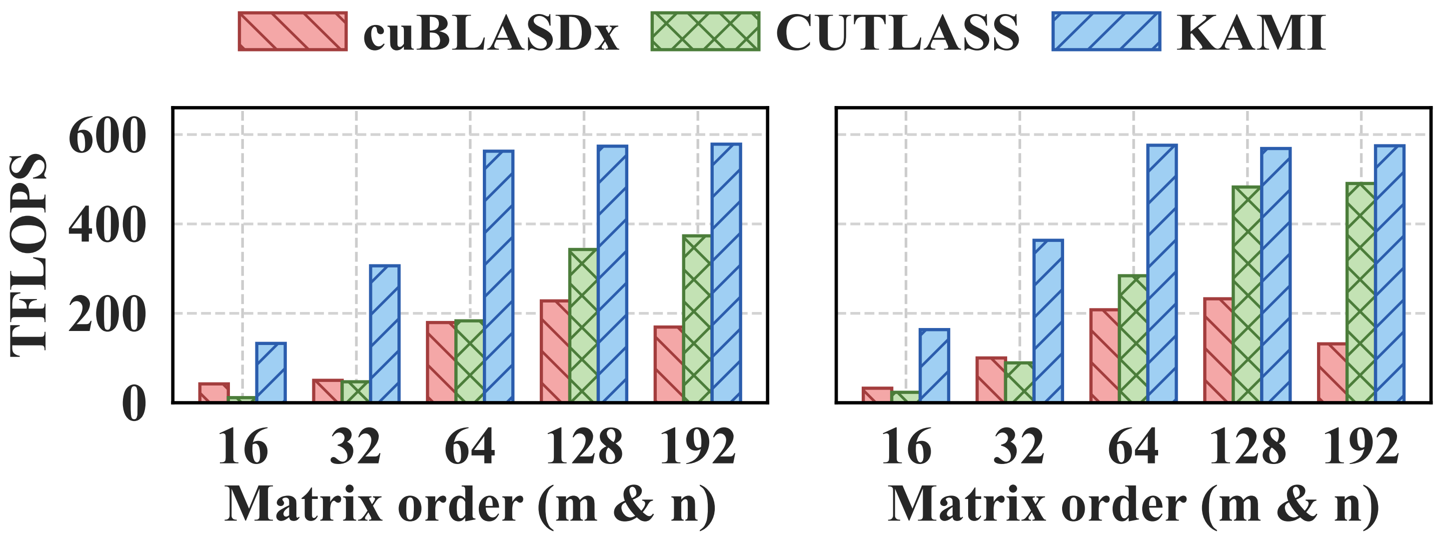
In block-level matrix multiplication, KAMI achieves up to 5.20x, 74.36x, and 14.48x higher throughput compared to cuBLASDx, CUTLASS, and SYCL-Bench, respectively, for square matrix multiplication. Additionally, KAMI utilizes less shared memory than cuBLASDx, enabling support for larger matrix dimensions.



Block-Level GEMM Performance across GPU Architectures.

Experiments: low-rank GEMM

We compare KAMI and cuBLASDx on low-rank GEMM for k=16 and k=32 on GH200 in FP16. KAMI consistently outperforms cuBLASDx and CUTLASS, achieving average speedups of 3.66x, 4.89x for k=16 and 3.65x, 3.09x for k=32.

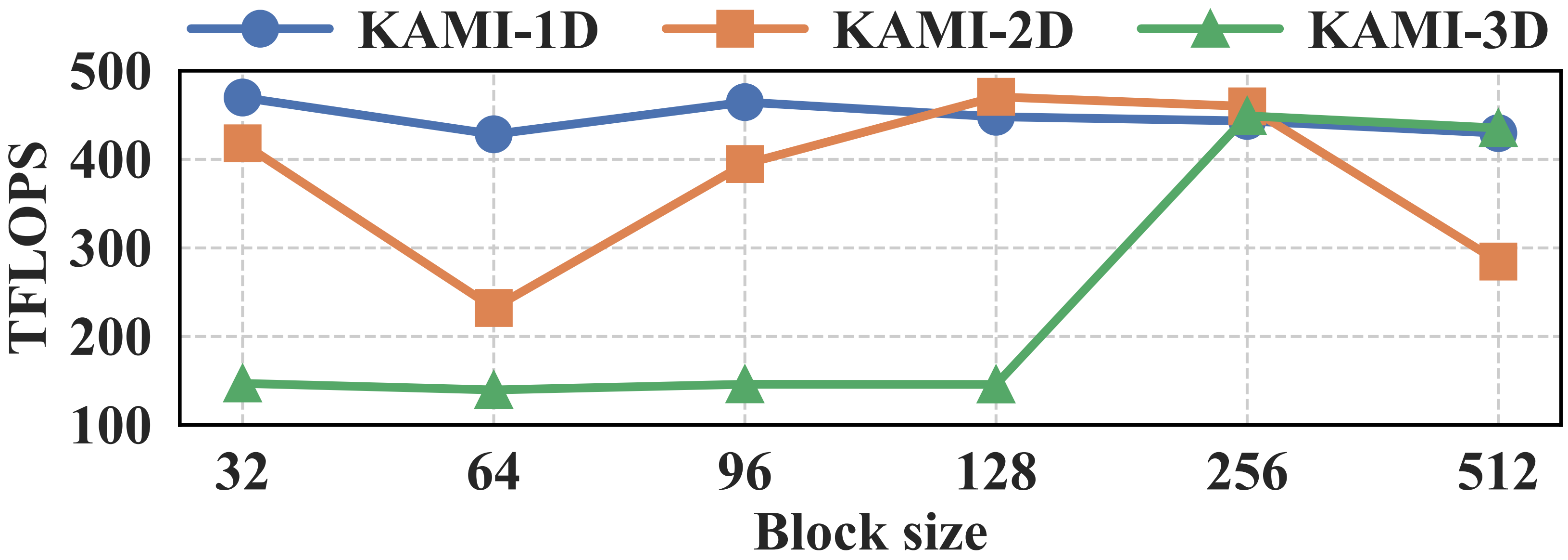


(a) k=16 (b) k=32

Low-rank GEMM in FP16 on GH200.

Experiments: block size effects

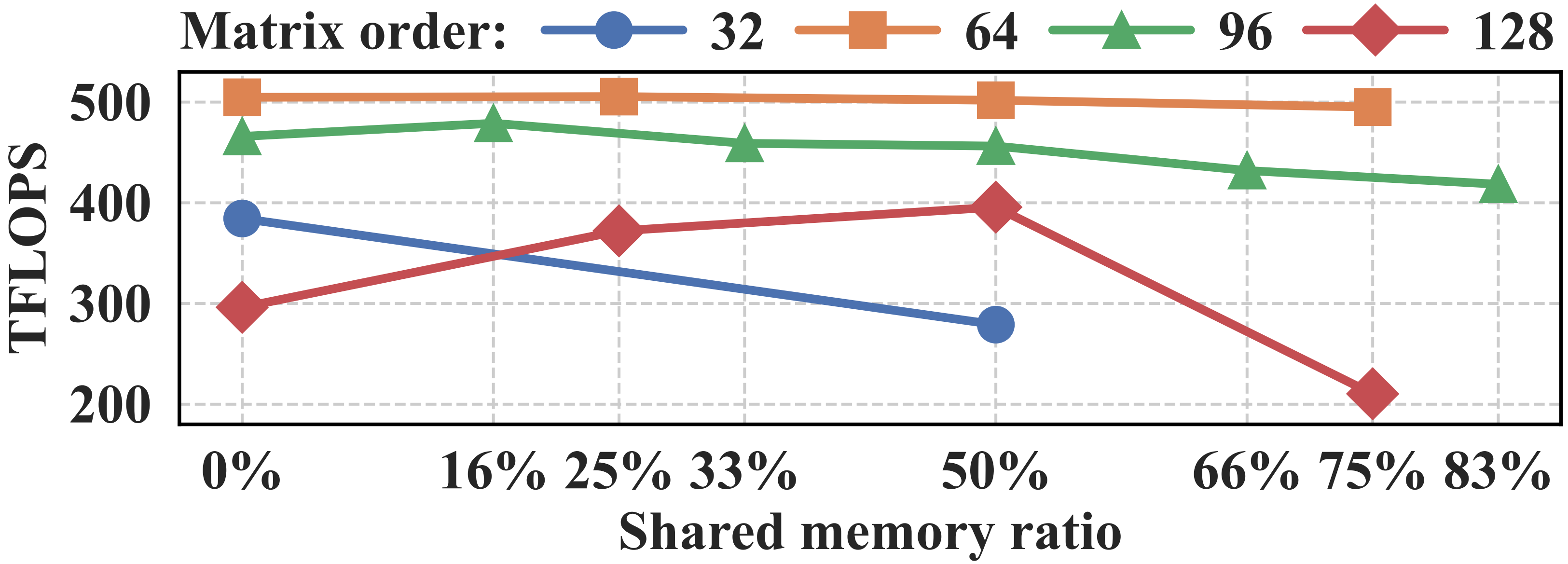
We shows the GEMM performance of two 64×64 matrices by KAMI-1D, KAMI-2D, and KAMI-3D on 5090, with peak performances of 469.80, 470.57, and 449.07 TFLOPS. KAMI-1D is robust under tight block size constraints, while KAMI-2D/3D is preferable when larger block sizes are available.



Impact of block size in FP16 on 5090.

Experiments: shared memory and register cooperation

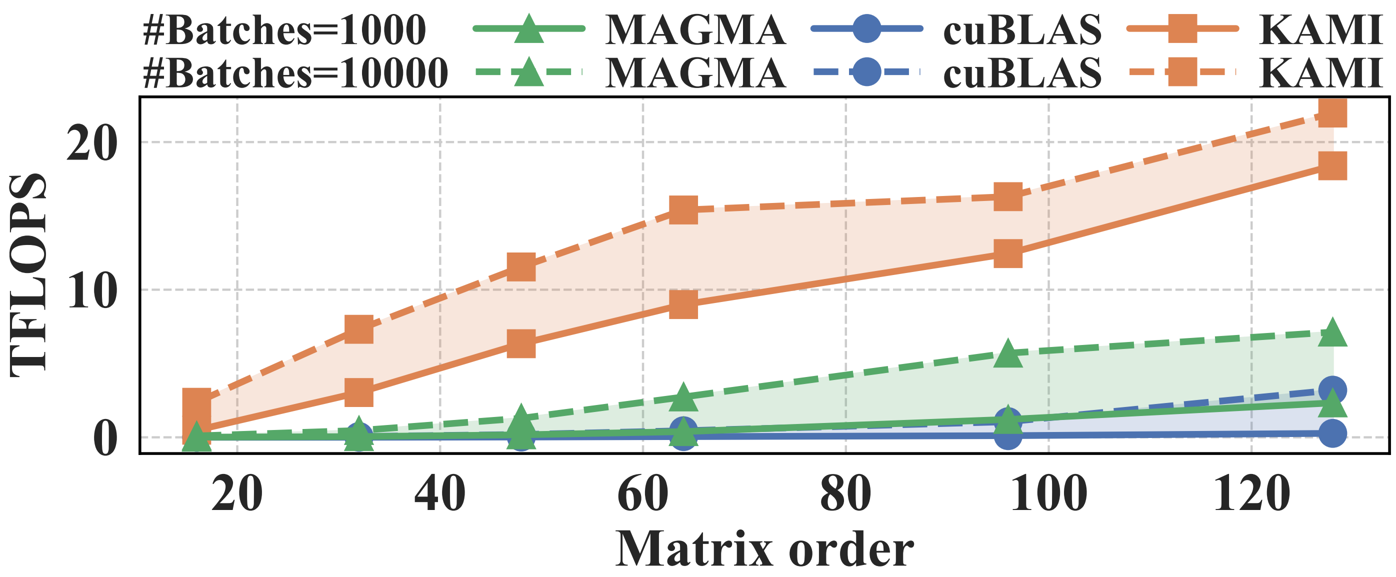
When register usage is excessive, KAMI utilizes both registers and shared memory for matrix storage: it partitions matrices A and B along the k-dimension and stores temporarily unused sub-matrices into shared memory to alleviate register pressure. We tested GEMM performance under different offloading ratios.



Impact of shared memory ratio on block-level.

Experiments: batched GEMM

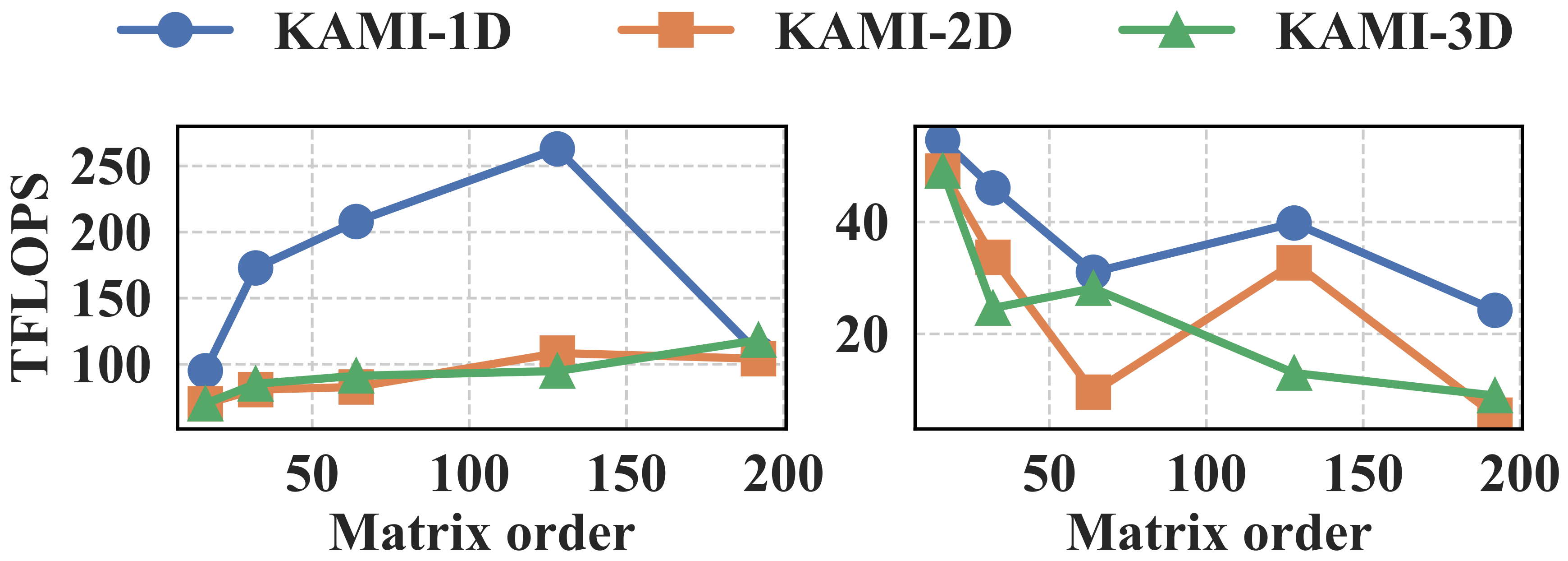
KAMI's batched interface is consistent with cuBLAS and MAGMA. We compare them in a uniform order to focus on the GEMM efficiency. KAMI achieves average speedups of 31.60x and 340.37x for batch sizes of 1000, and 10.23x and 96.17x for batch sizes of 10000, compared with MAGMA and cuBLAS.



Comparison of batched GEMM in FP64 on GH200.

Experiments: SpMM and SpGEMM

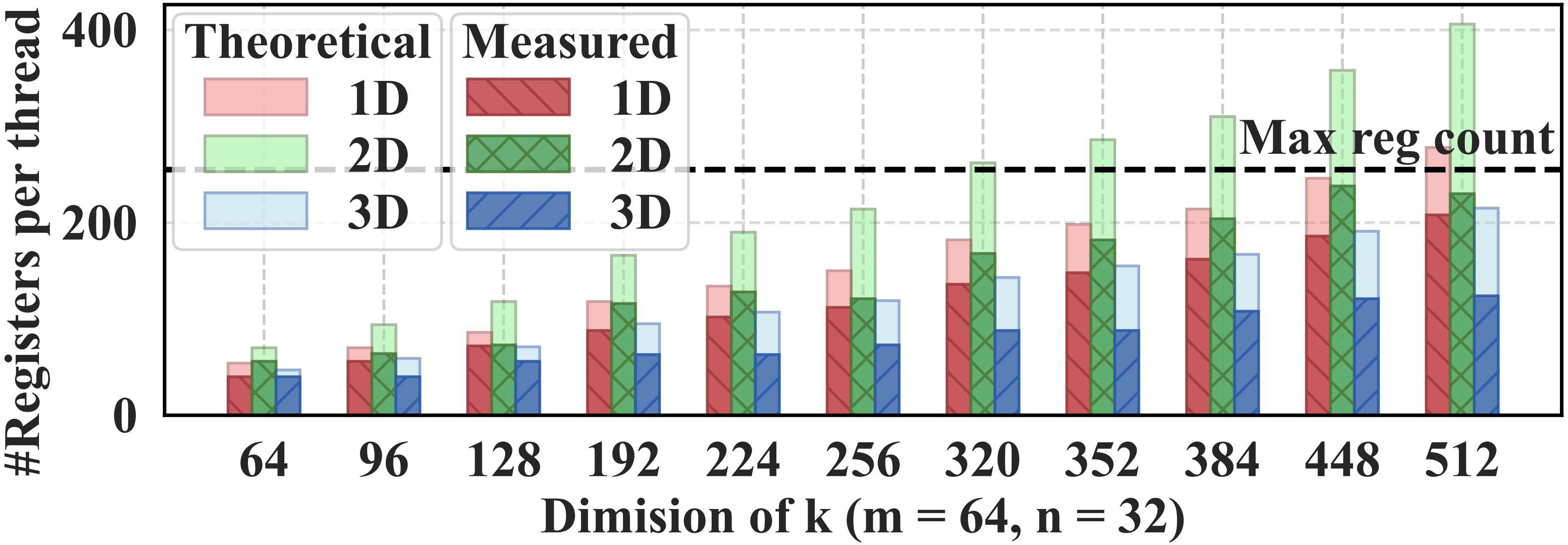
The performance trend of SpMM closely resembles that of GEMM, as the input matrices B and C are dense. In contrast, SpGEMM introduces significantly more complex indexing and results in less predictable memory access patterns due to different sparse structures in both input matrices.



SpMM and SpGEMM in FP16 on GH200.

Experiments: register allocation

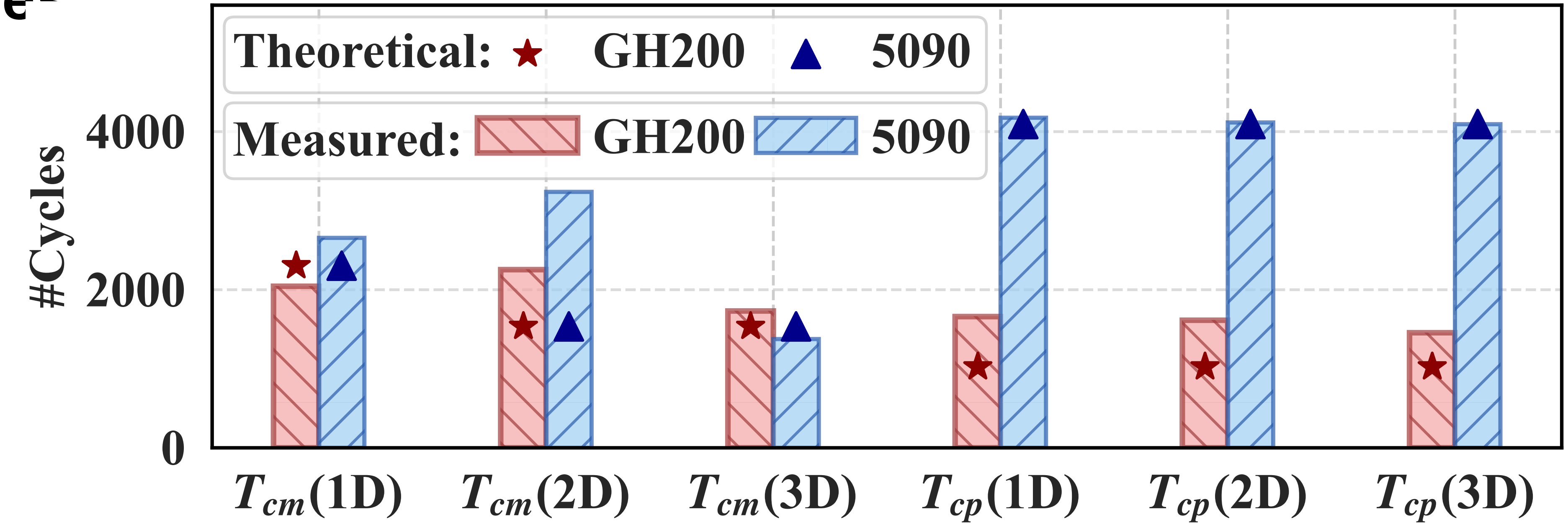
We test KAMI-1D (4 warps), KAMI-2D (4 warps) and KAMI-3D (8 warps), with C fixed at 64x32 and A, B varying with k. Results show that compared with theoretical values, actual register usage reaching 76.86% for 1D, 73.14% for 2D, and 65.67% for 3D.



The register usage of KAMI in FP16.

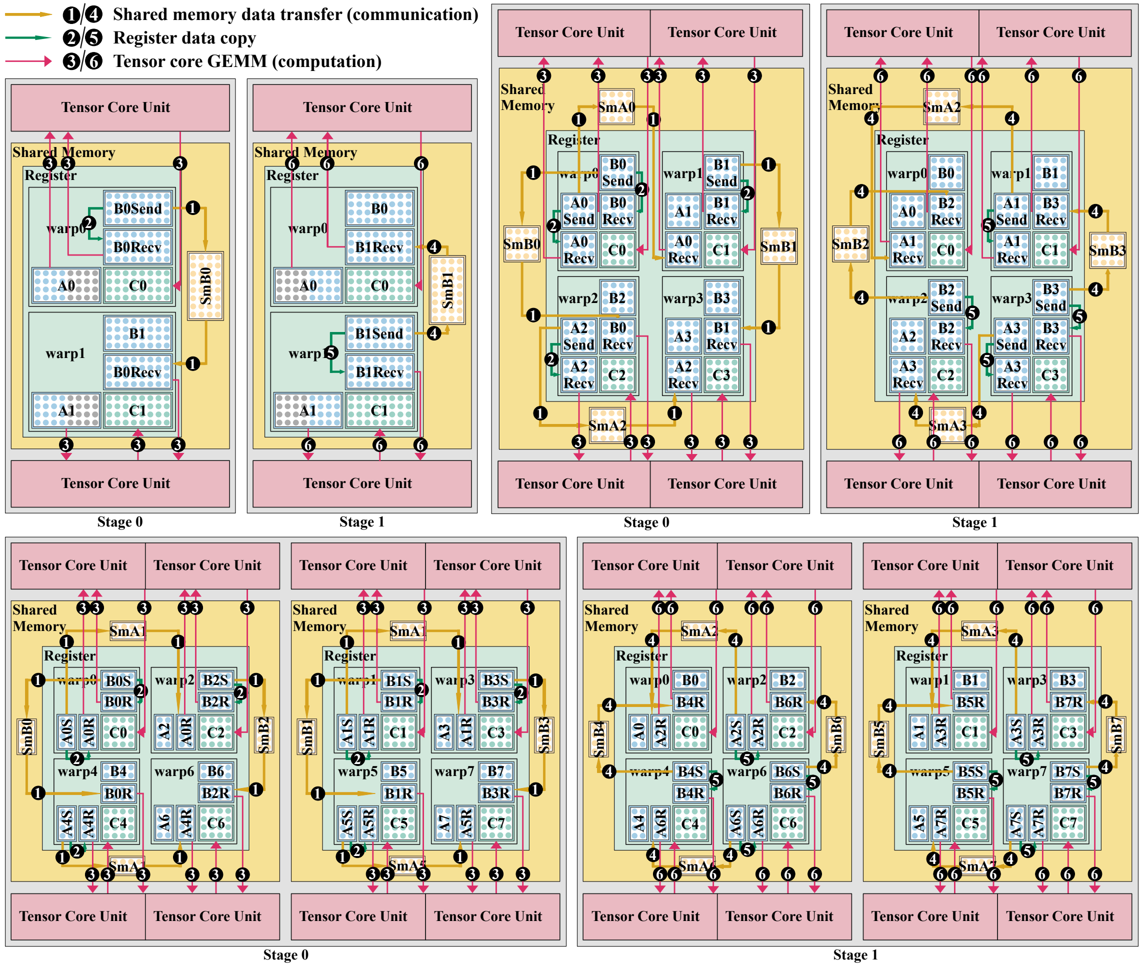
Experiments: cycles breakdown

We break down execution cycles into communication and computation, comparing results with the theoretical model. Overall, the experimental results are largely consistent with the theoretical model, aside from some discrepancies in a few cases.



The theoretical and measured cycles in FP16.

OUTLINE


❶/❹ Shared memory data transfer (communication)
❷/❺ Register data copy
❸/❻ Tensor core GEMM (computation)
Tensor Core Unit
Shared Memory
Register
Stage 0
Stage 1

Conclusion

- We propose KAMI, which extends the communication-avoiding (CA) algorithm to within a single GPU to accelerate small-scale matrix multiplication.
- We present a new theoretical analysis method that models communication and computation based on GPU clock cycles.
- We support SpMM and SpGEMM in the CA method by leveraging sparsity and a block-based Z-Morton storage format.
- We implement KAMI on NVIDIA, AMD, and Intel GPUs and demonstrate better performance than existing methods.
- KAMI is open-sourced at: https://github.com/SuperScientificSoftwareLaboratory/KAMI

Thanks! Q&A

KAMI: Communication-Avoiding General Matrix Multiplication within a Single GPU

https://github.com/SuperScientificSoftwareLaboratory/KAMI






SC25 ST. LOUIS hpc ignites.