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

Xiaohan Tao **_Yu Zhu_**
Boyang Wang Jinlong Xu Jianmin Pang Jie Zhao

State Key Laboratory of Mathematical Engineering and Advanced Computing
i_zhuyu@126.com

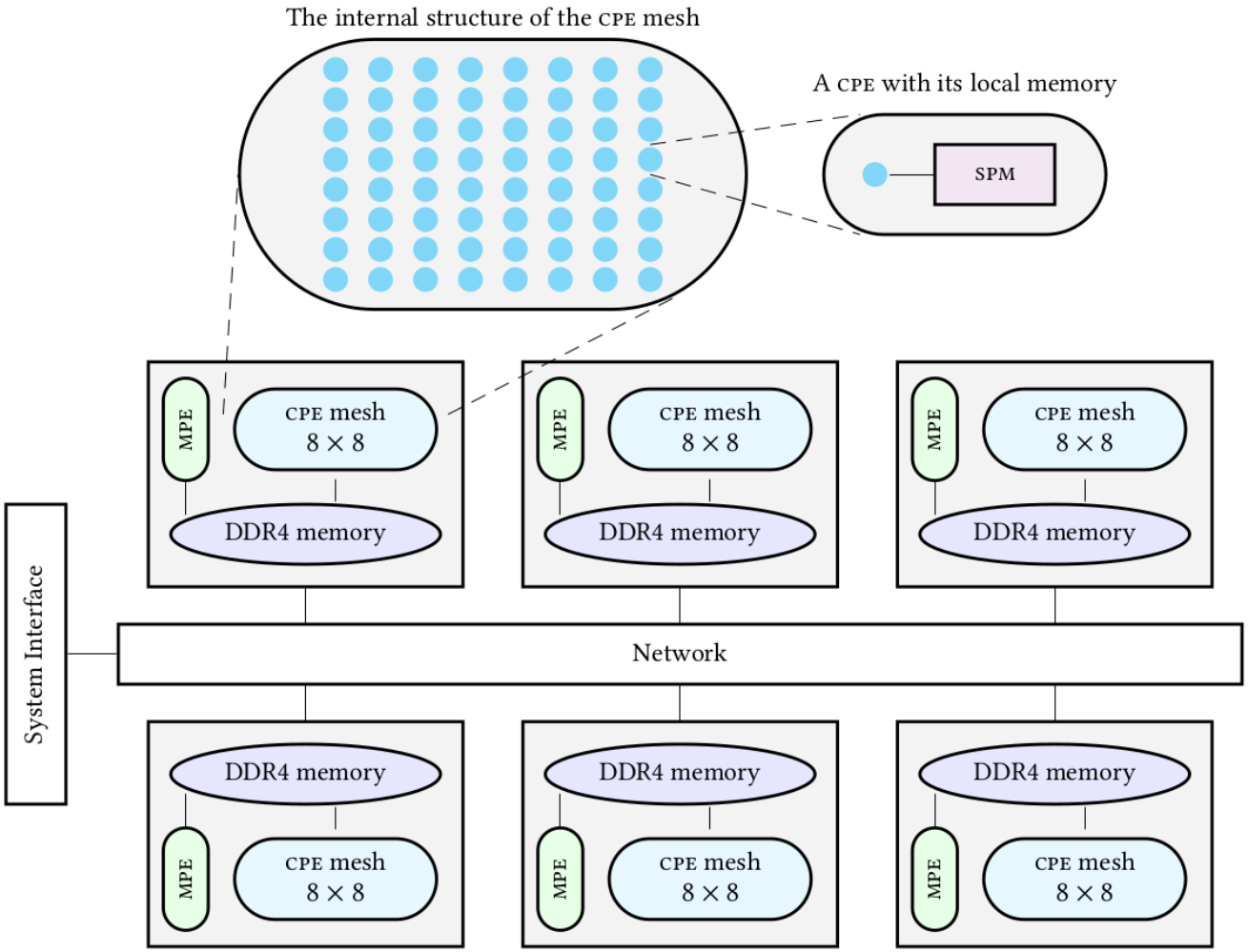
51st International Conference on Parallel Processing
August 31,2022

Background and Motivation

- The Sunway TaiHuLight supercomputer
  - Run high-performance computing applications, which involve massive linear algebra operations
- GEMM: Essential BLAS kernel for AI/ML
- Problem and Motivation
  - Manual efforts incurring high cost
  - Compilation approach is still missing
- Existing methods
  - Have to expose the architecture information
  - Only generated the innermost loop
- Our Method
  - An Automatic Code Generation Approach for GEMM on SW26010Pro

Background and Motivation

The Sunway Architecture



Polyhedral Model

A mathematical abstraction to reason about loop transformations and memory optimizations using integer sets and affine relations.

Polyhedral Model

A mathematical abstraction to reason about loop transformations and memory optimizations using integer sets and affine relations.

```
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: $[S_1(i, j, k) \rightarrow (blockIdx.y, blockIdx.x, \lfloor \frac{k}{32} \rfloor)]$
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.

Compute Decomposition

Tiling All Dimentions

tile size selection issue has not yet been modeled by isl or other polyhedral tools.

Our Objective

match the shape configuration of the assembly micro kernel.

Compute Decomposition

Tiling All Dimentions

tile size selection issue has not yet been modeled by isl or other polyhedral tools.

Our Objective

match the shape configuration of the assembly micro kernel.

$$\begin{aligned}
&\text{DOMAIN: } \{S_1(i,j,k) : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\} \\
&\quad\text{BAND: } [S_1(i,j,k) \to (\lfloor \tfrac{i}{64} \rfloor, \lfloor \tfrac{j}{64} \rfloor, \lfloor \tfrac{k}{32} \rfloor)] \\
&\quad\quad\text{BAND: } [S_1(i,j,k) \to (i - 64\lfloor \tfrac{i}{64} \rfloor, j - 64\lfloor \tfrac{j}{64} \rfloor, k - 32\lfloor \tfrac{k}{32} \rfloor)]
\end{aligned}$$

(a) The schedule tree after tiling.

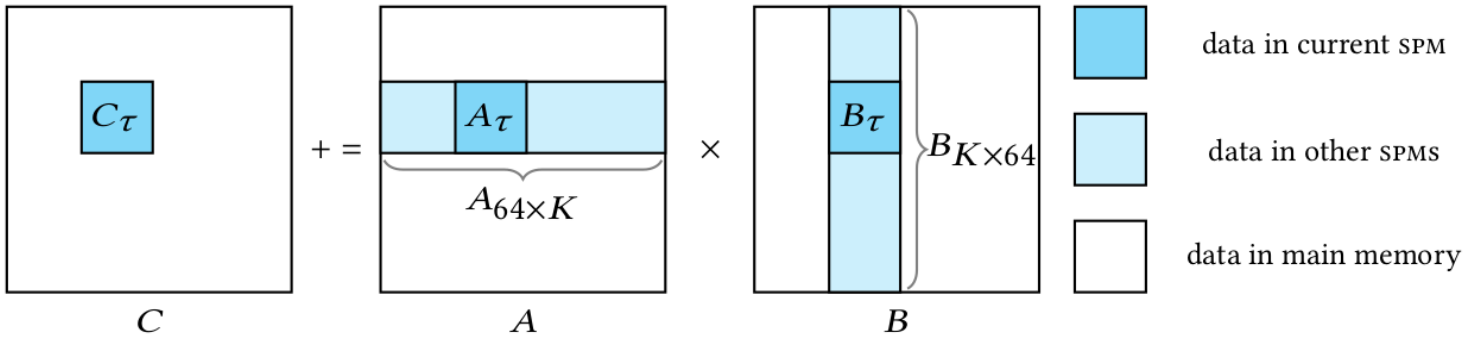
$$\begin{aligned}
&\text{DOMAIN: } \{S_1(i,j,k) : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\} \\
&\quad\text{BAND: } [S_1(i,j,k) \to (Rid, Cid, \lfloor \tfrac{k}{32} \rfloor)] \\
&\quad\quad\text{BAND: } [S_1(i,j,k) \to (i - 64\lfloor \tfrac{i}{64} \rfloor, j - 64\lfloor \tfrac{j}{64} \rfloor, k - 32\lfloor \tfrac{k}{32} \rfloor)]
\end{aligned}$$

(b) The schedule tree with CPE mesh parameters.

Compute Decomposition

Strip-minning the Reduced Dimension

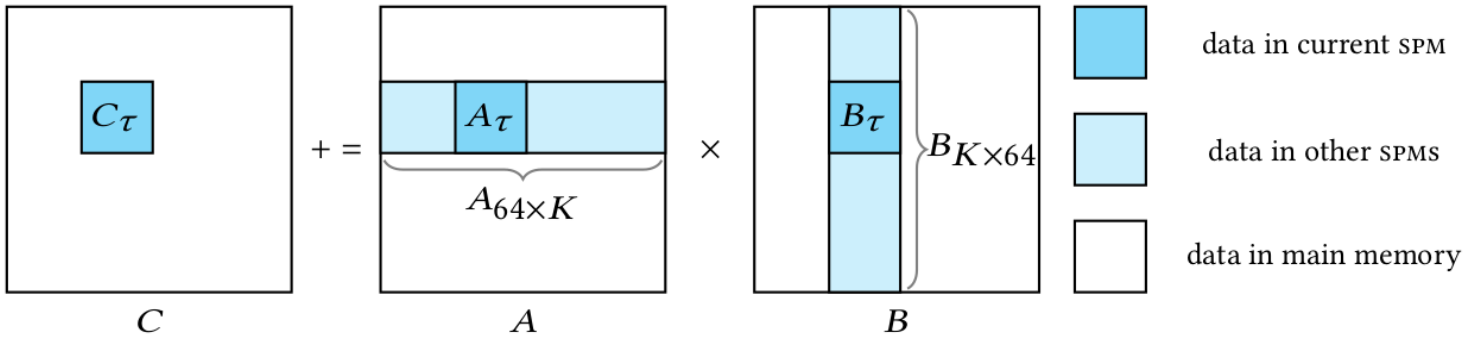
Each CPE can buffer a size of 64×64 tile of the output matrix and 64×32 tiles of the input matrices on its own SPM (allowed by RMA).



Compute Decomposition

Strip-minning the Reduced Dimension

Each CPE can buffer a size of 64×64 tile of the output matrix and 64×32 tiles of the input matrices on its own SPM (allowed by RMA).



DOMAIN: $\{S_1(i,j,k) : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$

BAND: $[S_1(i,j,k) \to (Rid, Cid)]$ /* *This band is mapped to the 2D CPE mesh.* */

BAND: $[S_1(i,j,k) \to (\lfloor \frac{k}{256} \rfloor)]$

BAND: $[S_1(i,j,k) \to (\lfloor \frac{k}{32} \rfloor - 8\lfloor \frac{k}{256} \rfloor)]$

BAND: $[S_1(i,j,k) \to (i - 64\lfloor \frac{i}{64} \rfloor, j - 64\lfloor \frac{j}{64} \rfloor, k - 32\lfloor \frac{k}{32} \rfloor)]$

Automating DMA Communication

Athread programming model for SW26010Pro

- *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)

Inserting extension nodes in schedule tree

- *the schedule tree with shared memory promotion statements*

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:

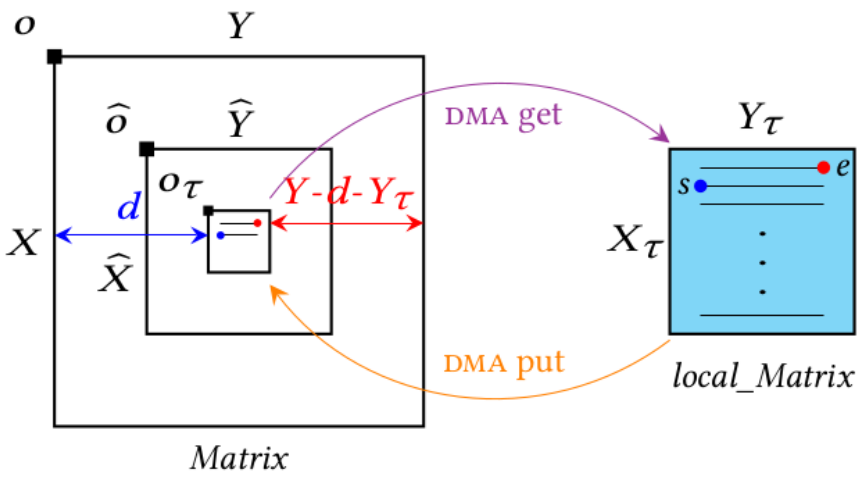
FILTER:$\{ readA[d_3, d_4] \}$

FILTER:$\{ readB[d_3, d_4] \}$

FILTER:$\{ S_1(i, j, k) \}$

Automating DMA Communication

The DMA Mechanism



$$\text{Matrix=C}\begin{cases} X = M \wedge Y = N \\ \widehat{X} = 512 \wedge \widehat{Y} = 512 \\ X_\tau = 64 \wedge Y_\tau = 64 \end{cases}$$

$$\text{Matrix=A}\begin{cases} X = M \wedge Y = K \\ \widehat{X} = 512 \wedge \widehat{Y} = 256 \\ X_\tau = 64 \wedge Y_\tau = 32 \end{cases}$$

$$\text{Matrix=B}\begin{cases} \widehat{X} = K \wedge \widehat{Y} = N \\ X = 256 \wedge Y = 512 \\ X_\tau = 32 \wedge Y_\tau = 64 \end{cases}$$

data in SPM

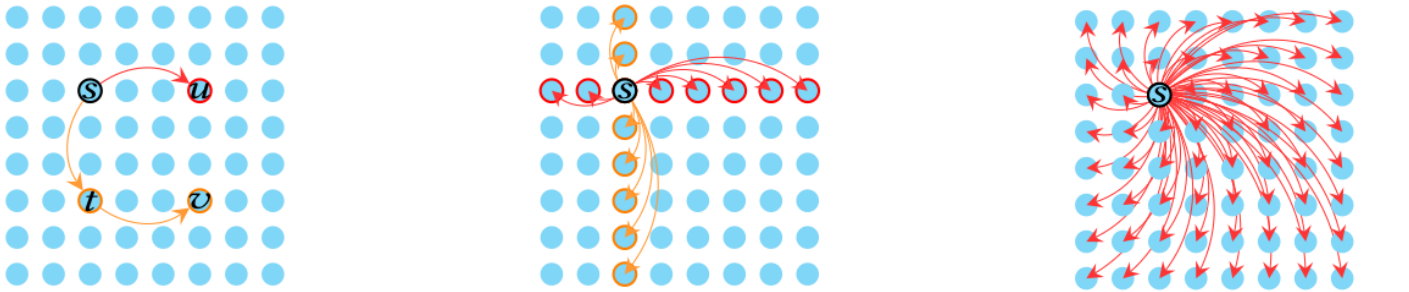
data in main memory

- each CPE mesh executes a GEMM kernel of 512x512x256.
- *dma_iget* (&Matrix[0][0], &Matrix[r][c], $X_\tau \times Y_\tau$, Y_τ, $Y - Y_\tau$, &reply)

- Automatically implementing data movements from the main memory to the SPMs

Implementing RMA Broadcast

The communication manners between CPEs



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

Implementing RMA Broadcast

Insert extension nodes for DMA and RMA

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:$\{ put\_replyC() \}$

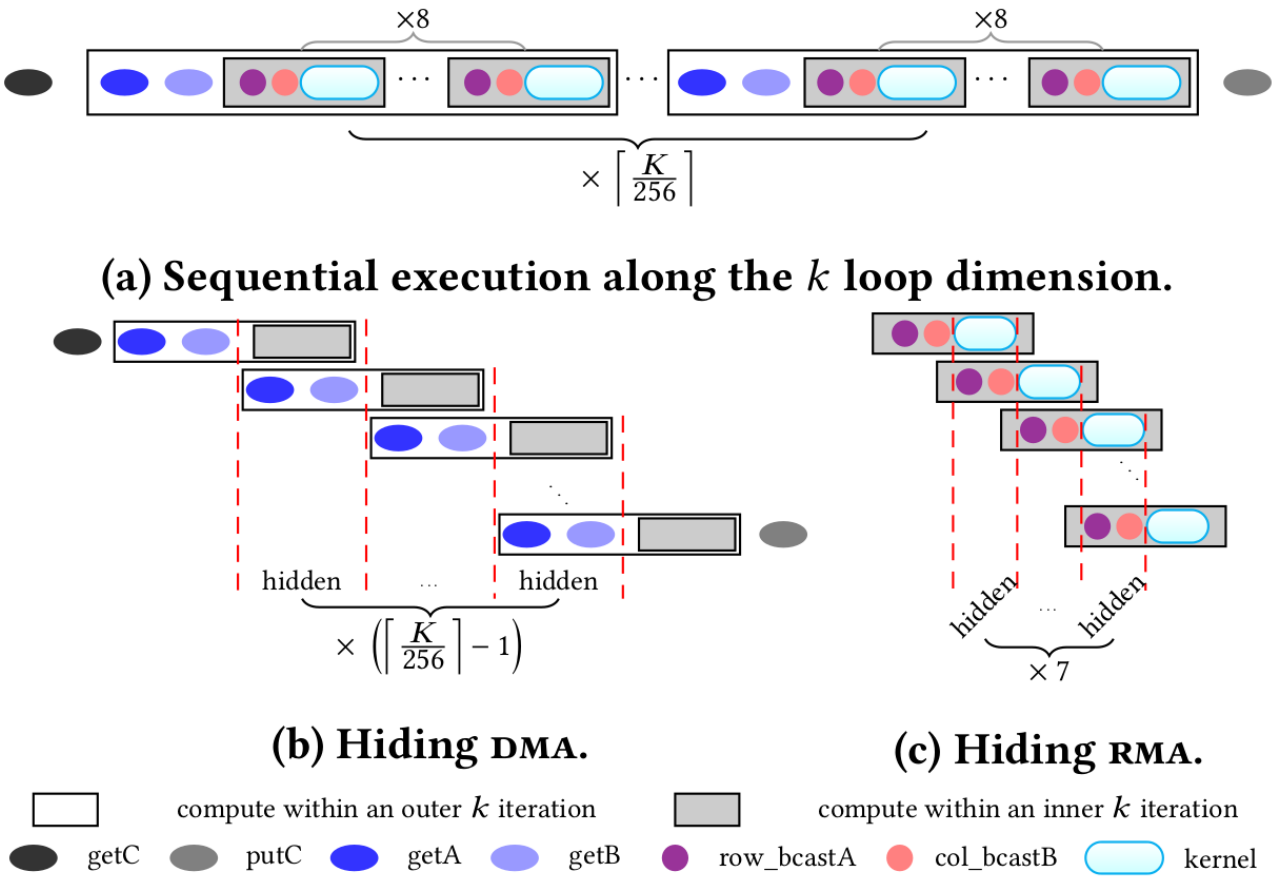
Implementing RMA Broadcast

Insert extension nodes for DMA and RMA

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:$\{ put\_replyC() \}$

- Automatically implementing data communication within CPE mesh

The two-level memory latency hiding strategy



Code Generation

- Code generation phases
  - Reusing the AST generator of isl
  - Printing Athread syntex

Code Generation

- Code generation phases
  - Reusing the AST generator of isl
  - Printing Athread syntex
- Inline Assembly Routine
  - The assembly micro kernel is provided as a compiled object, which has been highly optimized by the Sunway architects
  - We use a mark node in schedule tree to instruct the code generator to print an assembly function call.

Code Generation

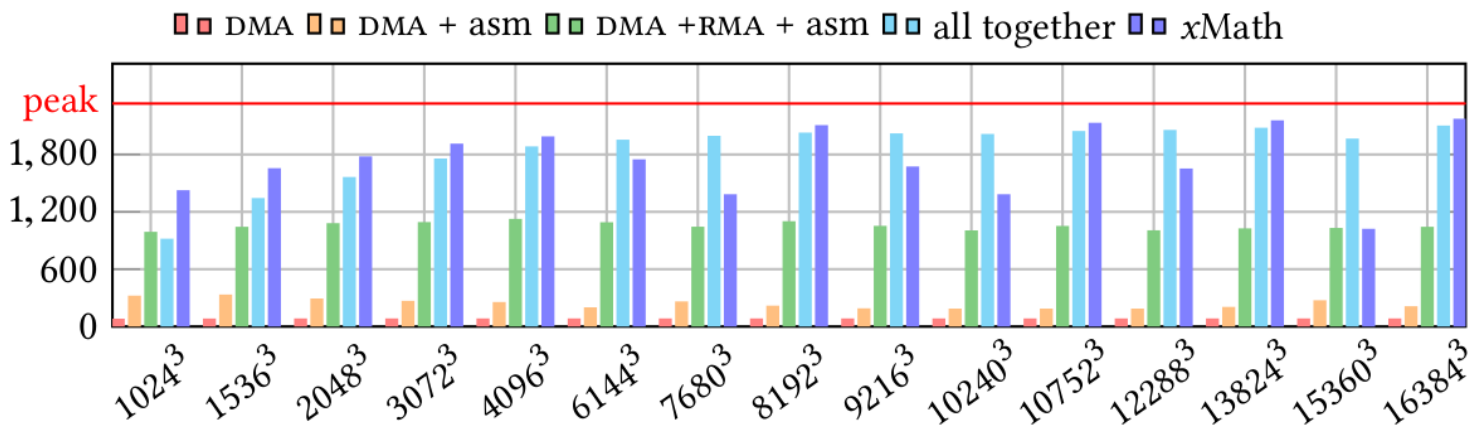
- Code generation phases
  - Reusing the AST generator of isl
  - Printing Athread syntex
- Inline Assembly Routine
  - The assembly micro kernel is provided as a compiled object, which has been highly optimized by the Sunway architects
  - We use a mark node in schedule tree to instruct the code generator to print an assembly function call.
- Fusion Patterns
  - Fusion patterns with a prologue/epilogue operation

Experiments

- Experiment Environment.
  - PPCG (Polyhedral Parallel Code Generation)
  - SW26010Pro Processor
  - SWGCC Compiler
- Comparison
  - xMath version 2.0 (tuned BLAS library of the SW26010 processor).

Experiments

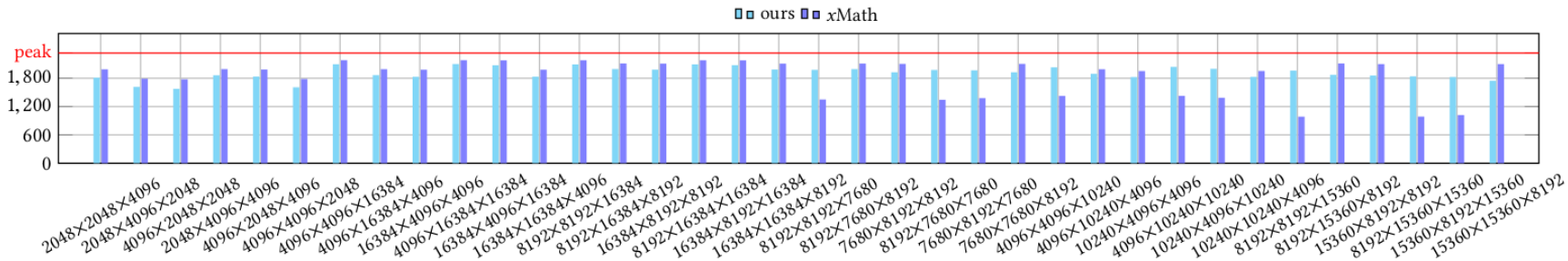
- Performance Breakdown



- DMA (Baseline version): 84.89 Gflops
- DMA+asm: 240.39 Gflops
- DMA+RMA+asm: 1052.94 Gflops
- all together: 1849.06 Gflops

Experiments

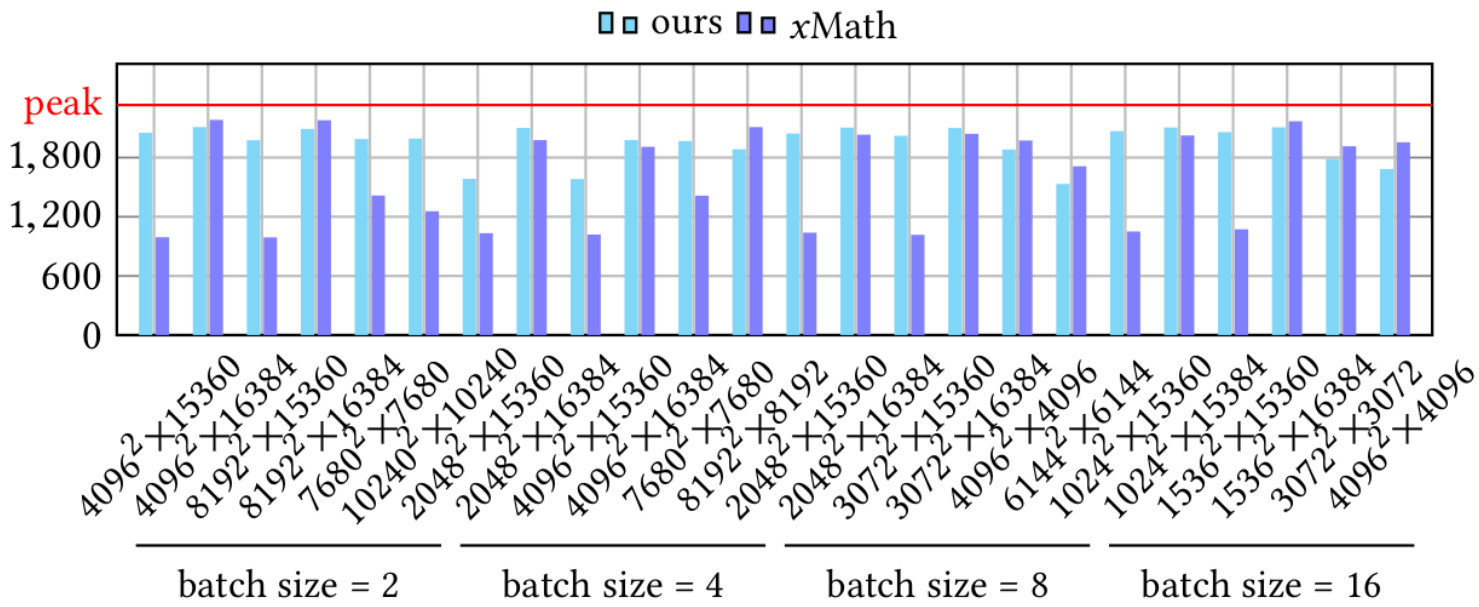
- Performance Comparison of GEMM and xMath



- xMath achieves a mean 1746.97 Gflops
- Our approach outperforms it by 9.62%

Experiments

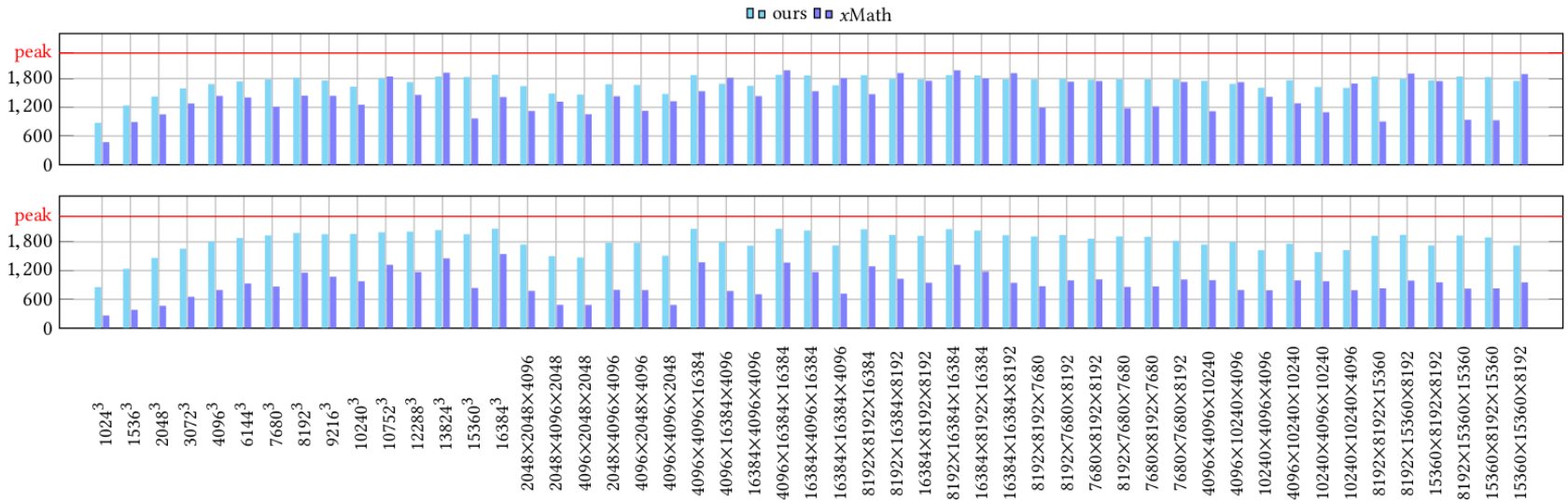
- Performance Comparison of Batched GEMM



- Batch sizes: 2, 4, 8, 16
- Average performance: 1949.92 Glops
- Our approach outperforms xMath by 1.30x

Experiments

- Performance Comparison of Fusion



- 2.11x speedup when fused with epilogue
- 1.26x speedup when fused with prologue

Conclusion

- We present a method to automatically generate matrix multiplication kernels for the latest Sunway processor.

Conclusion

- We present a method to automatically generate matrix multiplication kernels for the latest Sunway processor.

- Polyhedral Transformations
  - Compute decomposition
  - DMA/RMA communication
  - Memory latency hiding
- Low-level optimizations
  - inline assembly kernel

Conclusion

- We present a method to automatically generate matrix multiplication kernels for the latest Sunway processor.
- Polyhedral Transformations
  - Compute decomposition
  - DMA/RMA communication
  - Memory latency hiding
- Low-level optimizations
  - inline assembly kernel
- One can obtain up to more than 90% of the theoretical performance within few lines of C code

The End

Questions? Comments?