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.
Background and Motivation

Polyhedral Compilation

Polyhedral Model

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

\[
\text{for } i \in [0, M) \text{ and } j \in [0, N) \text{ and } k \in [0, K) \\
C[i, j] = C[i, j] + A[i, k] \cdot B[k, j] \quad /\ast S_1 \ast/ \\
\]

(a) A 3D loop nest of \texttt{GEMM} code.

(b) The initial schedule tree.

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

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

\[
\text{DOM\text{A}IN}: \{S_1(i, j, k) : 0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K\} \\
\text{BAN\text{D}}: [S_1(i, j, k) \rightarrow (\lfloor \frac{i}{64} \rfloor, \lfloor \frac{j}{64} \rfloor, \lfloor \frac{k}{32} \rfloor)] \\
\text{BAN\text{D}}: [S_1(i, j, k) \rightarrow (i - 64\lfloor \frac{i}{64} \rfloor, j - 64\lfloor \frac{j}{64} \rfloor, k - 32\lfloor \frac{k}{32} \rfloor)]
\]

(a) The schedule tree after tiling.

\[
\text{DOM\text{A}IN}: \{S_1(i, j, k) : 0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K\} \\
\text{BAN\text{D}}: [S_1(i, j, k) \rightarrow (Rid, Cid, \lfloor \frac{k}{32} \rfloor)] \\
\text{BAN\text{D}}: [S_1(i, j, k) \rightarrow (i - 64\lfloor \frac{i}{64} \rfloor, j - 64\lfloor \frac{j}{64} \rfloor, k - 32\lfloor \frac{k}{32} \rfloor)]
\]

(b) The schedule tree with **CPE mesh parameters**.
Each CPE can buffer a size of $64 \times 64$ tile of the output matrix and $64 \times 32$ tiles of the input matrices on its own SPM (allowed by RMA).
Each CPE can buffer a size of $64 \times 64$ tile of the output matrix and $64 \times 32$ tiles of the input matrices on its own SPM (allowed by RMA).

**Domain:** $\{S_1(i, j, k) : 0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K\}$

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

**Band:** $[S_1(i, j, k) \rightarrow ([k/256]) ]$

**Band:** $[S_1(i, j, k) \rightarrow ([k/32] - 8[ k/256 ])]$

**Band:** $[S_1(i, j, k) \rightarrow (i - 64[ i/64 ], j - 64[ j/64 ], k - 32[ k/32 ])]$
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

```plaintext
DOMAIN: \{S_1(i, j, k) : 0 \leq i < M \land 0 \leq j < N \land 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:\{ readB[d_3, d_4] \}
- FILTER:\{ S_1(i, j, k) \}
```
Automating DMA Communication

The DMA Mechanism

- each CPE mesh executes a GEMM kernel of $512 \times 512 \times 256$.
- \textit{dma iget} (&Matrix[0][0], &Matrix[r][c], $X_τ \times Y_τ$, $Y_τ$, $Y - Y_τ$, &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

\[
\begin{align*}
\text{DOMAIN: } & \{S_1(i, j, k) : 0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K\} \\
\text{BAND: } & [S_1(i, j, k) \rightarrow (R_{id}, C_{id})] \\
\text{EXTENSION: } & [(d_0, d_1, d_2) \rightarrow \text{getC}(d_3, d_4)/\text{get_replyC}()] \\
& [(d_0, d_1, d_2) \rightarrow \text{putC}(d_3, d_4)/\text{put_replyC}()] \\
\text{SEQUENCE: } & \\
& \text{FILTER:}\{\text{getC}(d_3, d_4)\} \otimes \text{FILTER:}\{\text{get_replyC}()\} \\
& \text{FILTER:}\{S_1(i, j, k)\} \\
& \text{BAND: } [S_1(i, j, k) \rightarrow ([k/256])] \\
& \text{EXTENSION: } [(d_0, d_1, d_2) \rightarrow \text{getA}(d_3, d_4)/\text{get_reply_A()}] \\
& [(d_0, d_1, d_2) \rightarrow \text{getB}(d_3, d_4)/\text{get_reply_B()}] \\
& \text{SEQUENCE: } \\
& \text{FILTER:}\{\text{getA}(d_3, d_4)\} \oplus \text{FILTER:}\{\text{get_replyA}()\} \\
& \text{FILTER:}\{\text{getB}(d_3, d_4)\} \oplus \text{FILTER:}\{\text{get_replyB}()\} \\
& \text{FILTER:}\{S_1(i, j, k)\} \\
& \text{BAND: } [S_1(i, j, k) \rightarrow ([k/32] - 8[k/256])] \\
& \text{EXTENSION: } [(d_0, d_1, d_2, d_3) \rightarrow \text{rbcastA}(d_4, d_5)/\text{rbcast_replyA}()] \\
& [(d_0, d_1, d_2, d_3) \rightarrow \text{cbcastB}(d_4, d_5)/\text{cbcast_replyB}()] \\
& \text{SEQUENCE: } \\
& \text{FILTER:}\{\text{rbcastA}(d_4, d_5)\} \oplus \text{FILTER:}\{\text{rbcast_replyA}()\} \\
& \text{FILTER:}\{\text{cbcastB}(d_4, d_5)\} \oplus \text{FILTER:}\{\text{cbcast_replyB}()\} \\
& \text{FILTER:}\{S_1(i, j, k)\} \\
& \text{BAND: } [S_1(i, j, k) \rightarrow (i - 64[j/64]), j - 64[j/64], k - 32[k/32])] \\
& \text{FILTER:}\{\text{putC}(d_3, d_4)\} \otimes \text{FILTER:}\{\text{put_replyC}()\}
\end{align*}
\]
Implementing RMA Broadcast

Insert extension nodes for DMA and RMA

\[
\text{DOMAIN: } \{ S_1(i, j, k) : 0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K \}
\]
\[
\text{BAND: } [S_1(i, j, k) \rightarrow (Rid, Cid)]
\]
\[
\text{EXTENSION: } [(d_0, d_1, d_2) \rightarrow getC(d_3, d_4)/get\_replyC()]
\]
\[
\text{EXTENSION: } [(d_0, d_1, d_2) \rightarrow putC(d_3, d_4)/put\_replyC()]
\]

\[
\text{SEQUENCE:}
\]
\[
\text{FILTER: } \{ getC(d_3, d_4) \} \otimes \text{FILTER: } \{ get\_replyC() \}
\]
\[
\text{FILTER: } \{ S_1(i, j, k) \}
\]
\[
\text{BAND: } [S_1(i, j, k) \rightarrow (\lfloor \frac{k}{256} \rfloor )]
\]
\[
\text{EXTENSION: } [(d_0, d_1, d_2) \rightarrow getA(d_3, d_4)/get\_reply\_A()]
\]
\[
\text{EXTENSION: } [(d_0, d_1, d_2) \rightarrow getB(d_3, d_4)/get\_reply\_B()]
\]

\[
\text{SEQUENCE:}
\]
\[
\text{FILTER: } \{ getA(d_3, d_4) \} \oplus \text{FILTER: } \{ get\_replyA() \}
\]
\[
\text{FILTER: } \{ getB(d_3, d_4) \} \oplus \text{FILTER: } \{ get\_replyB() \}
\]
\[
\text{FILTER: } \{ S_1(i, j, k) \}
\]
\[
\text{BAND: } [S_1(i, j, k) \rightarrow (\lfloor \frac{k}{32} \rfloor - 8 \lfloor \frac{k}{256} \rfloor)]
\]
\[
\text{EXTENSION: } [(d_0, d_1, d_2, d_3) \rightarrow rcastA(d_4, d_5)/rcast\_replyA()]
\]
\[
\text{EXTENSION: } [(d_0, d_1, d_2, d_3) \rightarrow cbcastB(d_4, d_5)/cbcast\_replyB()]
\]

\[
\text{SEQUENCE:}
\]
\[
\text{FILTER: } \{ rcastA(d_4, d_5) \} \oplus \text{FILTER: } \{ rcast\_replyA() \}
\]
\[
\text{FILTER: } \{ cbcastB(d_4, d_5) \} \oplus \text{FILTER: } \{ cbcast\_replyB() \}
\]
\[
\text{FILTER: } \{ S_1(i, j, k) \}
\]
\[
\text{BAND: } [S_1(i, j, k) \rightarrow (i - 64 \lfloor \frac{i}{64} \rfloor, j - 64 \lfloor \frac{j}{64} \rfloor, k - 32 \lfloor \frac{k}{32} \rfloor)]
\]
\[
\text{FILTER: } \{ putC(d_3, d_4) \} \otimes \text{FILTER: } \{ put\_replyC() \}
\]

• Automatically implementing data communication within CPE mesh
The two-level memory latency hiding strategy

(a) Sequential execution along the \( k \) loop dimension.

(b) Hiding DMA.

(c) Hiding RMA.
Code Generation

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

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

- 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 syntax

- 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).
- 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%
- 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?