Dense Dynamic Blocks: Optimizing SpMM for Processors with Vector and Matrix Units Using Machine Learning Techniques

SERIF YESIL*, JOSE MOREIRA+, AND JOSEP TORRELLAS*
*UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
+IBM RESEARCH

IBM-ILLINOIS DISCOVERY ACCELERATOR INSTITUTE
Multiplying Matrices

IBM POWER10 Matrix-Multiply Assist instructions (MMA)
  ◦ Successfully utilized for dense matrix multiply operations in ML domain
  ◦ Functional unit rich processors

Sparse Matrix Dense Matrix Multiply
  ◦ Building block for many complex applications
    ◦ Linear solvers, graph neural networks, recommender systems
  ◦ Many of these applications are iterative
    ◦ SpMM consumes many execution cycles
  ◦ Irregular, unpredictable sparsity structure
    ◦ Hard to utilize matrix-multiply capabilities
Our Approach

Our target: Functional unit rich processors with vector and matrix units

Dense Dynamic Blocks (DDB)
- Utilizes matrix and vector units synergistically to maximize floating-point throughput

A performance prediction tool (SpMM-OPT) capable of selecting
- Functional unit strategy
- Register reuse and cache optimization strategies
In This Talk

- POWER10 Matrix Multiply Facilities (MMA)
- Dense Dynamic Blocks to utilize POWER10 MMA units
- SpMM-Optimizer for selecting SpMM strategy
- Conclusions
POWER10 Matrix Multiply Assist (MMA)

Provides double (single) precision
- 4 × 2 (4 × 4) outer product operations
- Eight 4 × 2 (4 × 4) accumulators

Operation of MMA:
- Execute a sequence of outer product instructions

![Diagram showing the operation of MMA with matrices A and B, and accumulators Acc0 and Acc1. The diagram illustrates the row major and column major operations, and how the results are accumulated.](attachment:image.png)
Blocking the Sparse Matrix for SpMM

Previously blocking the sparse matrix
- $r \times c$ blocks to improve the irregular accesses $\rightarrow$ Hard to obtain
- Introduces zero padding, underutilizes matrix-multiply units

Our proposal for POWER10: Dense Dynamic Blocks (DDB)
- Form dynamic blocks from $r \times 1$ blocks
- Utilizes matrix and vector units synergistically to maximize floating-point throughput
Using 4x4 Blocks

• Creating $4 \times 4$ blocks
• 128 FLOPs executed (only 40 FLOPs are necessary)

Example 4x8 Sparse Matrix
Dense Dynamic Blocks (DDB-MM)

- DDB-Matrix Multiply (DDB-MM): Ignore c and create $4 \times 1$ blocks
- 80 FLOPs executed (only 40 FLOPs are necessary) $\rightarrow$ 38% reduction

Example Sparse Matrix
Not All Blocks Are Equal

MMA Throughput
- Double precision 32 Flops/cycle

Effective FP Throughput with MMA
- 32 Flops/cycle
- 24 Flops/cycle
- 16 Flops/cycle
- 8 Flops/cycle
Not All Blocks Are Equal

MMA Throughput
- Double precision 32 Flops/cycle

VSX Throughput
- Double precision 16 Flops/cycle

Effective FP Throughput with MMA
- 32 Flops/cycle
- 24 Flops/cycle
- 16 Flops/cycle
- 8 Flops/cycle

Effective FP Throughput with VSX
- 16 Flops/cycle
- 16 Flops/cycle
- 16 Flops/cycle
- 16 Flops/cycle
Dense Dynamic Blocks (DDB-HYB)

- DDB-Hybrid (DDB-HYB): Utilize both matrix and vector units
- 44 FLOPs executed, only 40 FLOPs are necessary

**Example Sparse Matrix**

Matrix Units

Vector Units
Performance of DDB for SpMM

Tested on IBM POWER10 with 30 SMT4 cores

440 matrices from SuiteSparse

Maximum FLOPs/s

- 1.15 TFLOPs/s for double-precision (DP)
- 2.5 TFLOPs/s for single-precision (SP)

DDB-HYB fastest for 247 of the matrices for DP
DDB-MM fastest for 211 of the matrices for SP
Other SpMM Optimizations

\[ C = A \times B \rightarrow A \text{ (sparse)}, \ B \text{ and } C \text{ (dense)} \]

Improving reuse for A or C

- For compressed portion: CSR-A and CSR-C
- For blocked portion: MMA-A and MMA-C

Cache slicing

- Execute \( C = A \times B \) in multiple phases
- 64 columns C and B can be sliced into 16 columns slices
SpMM Optimizations Search Space

1) Choosing vector or matrix units
   ◦ FU St: The functional unit strategy
     ◦ \{CSR, DDB-HYB, DDB-MM\}

2) Improving reuse for A or C
   ◦ Reuse approach for the blocked portion
     ◦ \{MMA-A, MMA-C\}
   ◦ Reuse strategy for the compressed portion
     ◦ \{CSR-A, CSR-C\}

3) Cache slicing
   ◦ The slicing factor
     ◦ \{16, 32, 64, 128, 256\}
SpMM-Optimizer

SpMM-Optimizer: An ML-based approach to select best SpMM strategy

- Functional unit, reuse, slicing strategies for a sparse matrix
- Detecting Sparse Matrices with High and Low Potential for MMA utilization
  - Average floating-point throughput (AFT) metric
- Features to summarize matrix characteristics
  - Size, locality, and blocking characteristics
- Machine learning models
  - Separate models for High and Low Potential matrices with different # cols in dense matrices
  - 10 different models: {High Potential, Low Potential} x {16, 32, 64, 128, 256}
  - Each model uses Support Vector Machines with the linear kernel
Average Floating-Point Throughput (AFT)

Calculate the potential to utilize MMA for a given sparse matrix

What would be the average FLOP throughput for the matrix?

Breakdown matrices into **High Potential** and **Low Potential** groups

- Experimentally selected threshold 20
AFT Example

AFT: Estimate the throughput per nnz

- \( H↓i \): Fraction of nonzeros in a block with \( i \) nonzeros
- \( T↓i \): Effective throughput for \( i \) element block
  - \( T↓4 = 32, T↓3 = 24, T↓2 = 16, T↓1 = 16 \) FLOPS/cycle
  - Blocks with 1 nnz assumed → On Vector Units

\[
AFT = \sum_i \left( T↓i \times H↓i \right)
\]

Example Matrix: \( AFT = 24.8 \) → High Potential
ML System Features

- **Size Characteristics**
  - # rows, # nonzeros: Encode the size characteristics of the matrix

- **Blocking Characteristics**
  - Distribution of nonzeros to 4x1 dense blocks

- **Locality Characteristics**
  - Memory access characteristics of 4x1 blocks created
  - Memory access characteristics for a chunk of the sparse matrix assigned to a thread
## Putting it All Together

<table>
<thead>
<tr>
<th>High Potential</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 cols</td>
</tr>
<tr>
<td>32 cols</td>
</tr>
<tr>
<td>64 cols</td>
</tr>
<tr>
<td>128 cols</td>
</tr>
<tr>
<td>256 cols</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low Potential</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 cols</td>
</tr>
<tr>
<td>32 cols</td>
</tr>
<tr>
<td>64 cols</td>
</tr>
<tr>
<td>128 cols</td>
</tr>
<tr>
<td>256 cols</td>
</tr>
</tbody>
</table>
Putting it All Together

1. Calculate AFT
2. Calculate Matrix Features

<table>
<thead>
<tr>
<th>High Potential</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 cols</td>
</tr>
<tr>
<td>32 cols</td>
</tr>
<tr>
<td>64 cols</td>
</tr>
<tr>
<td>128 cols</td>
</tr>
<tr>
<td>256 cols</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low Potential</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 cols</td>
</tr>
<tr>
<td>32 cols</td>
</tr>
<tr>
<td>64 cols</td>
</tr>
<tr>
<td>128 cols</td>
</tr>
<tr>
<td>256 cols</td>
</tr>
</tbody>
</table>
Putting it All Together

1. Calculate AFT
2. Calculate Matrix Features
3. High Potential
   - 16 cols
   - 32 cols
   - 64 cols
   - 128 cols
   - 256 cols
4. Low Potential
   - 16 cols
   - 32 cols
   - 64 cols
   - 128 cols
   - 256 cols
Putting it All Together

1. Calculate AFT
2. Calculate Matrix Features
3. High Potential
   - 16 cols
   - 32 cols
   - 64 cols
   - 128 cols
   - 256 cols
4. CSR (Vector)
   - CSR-A
   - CSR-C
5. DDB-HYB (Vector+Matrix)
   - MMA-A, CSR-A
   - MMA-A, CSR-C
   - MMA-C, CSR-A
   - MMA-C, CSR-C
6. DDB-MM (Matrix)
   - MMA-A
   - MMA-C

Slicing parameters: 16, 32, 64
SpMM-OPT for High Potential Matrices

- **Oracle** and **SpMM-OPT** select: *Functional unit, reuse, slicing* strategies for a sparse matrix
- **SpMM-OPT** can achieve 1.55× average speedup
- **oracle** delivers 1.76× average speedup
- **DDB-HYB** and **DDB-MM** generally achieve the highest performance

**Distribution of speedup for oracle method and SpMM-OPT**
Double-precision SpMM with 64 columns dense matrices

**SpMM-Optimizer**

**Oracle**
SpMM-OPT for Low Potential Matrices

- **Oracle** and **SpMM-OPT** select: *Functional unit, reuse, slicing* strategies for a sparse matrix
- **SpMM-OPT** can achieve 1.3× average speedup
- **Oracle** delivers a 1.64× average speedup
- **Slicing** is the key to achieve high speedups for Low Potential matrices

**Distribution of speedup for oracle method and SpMM-OPT**

Double-precision SpMM with 64 columns dense matrices
More details in the paper

Detailed descriptions of register reuse and cache slicing optimizations
Detailed description of DDB and SpMM-Optimizer
Experiments with 16-256 columns dense matrices
Discussion on effect of slicing
Conclusions

- Optimizing SpMM on processors with vector and matrix units
- Dense Dynamic Blocks
  - A hybrid approach to utilize vector and matrix units for SpMM
  - Observed up to 1.1 TFLOPS/s for DP, 2.5 TFLOPS/s for SP SpMM
- SpMM-Optimizer
  - An ML method to navigate the optimization search space for SpMM
  - An average speedup of up to 2x compared to an optimized CSR baseline
Dense Dynamic Blocks: Optimizing SpMM for Processors with Vector and Matrix Units Using Machine Learning Techniques

SERIF YESIL*, JOSE MOREIRA+, AND JOSEP TORRELLAS*
*UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
+IBM RESEARCH
IBM-ILLINOIS DISCOVERY ACCELERATOR INSTITUTE