An Anatomy of Optimized Matrix Multiplication in AArch64

Haochen Wang, Tomasz Czajkowski, Ehsan Amiri
Huawei’s Bisheng compiler team, developed from the llvm-project
Motivation

- There are many mature techniques for matrix multiplication:
  - Tiling and Packing
  - Vectorization and SIMD instructions
  - Outer product expansion

- But their effectiveness doesn’t scale well over a wide range of matrix sizes.
- We present our work on choosing the appropriate techniques for different matrix sizes, and how to best combine the techniques and sizes together.
Matrix Multiplication (MM)

- Widely used in many algorithms
  - Solver of linear equation systems
  - Training machine learning models
  - Rendering computer graphics

- \( C = A \times B \)
- Double precision floating point
- Single-thread
Performance Results

- Theoretical maximum for the testing machine is 10.4 GFLOPs (double precision)
- Single-thread

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>Performance (GFLOPs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>9.92</td>
</tr>
<tr>
<td>512</td>
<td>9.50</td>
</tr>
<tr>
<td>1024</td>
<td>9.12</td>
</tr>
<tr>
<td>1536</td>
<td>9.88</td>
</tr>
<tr>
<td>2048</td>
<td>9.90</td>
</tr>
<tr>
<td>2560</td>
<td>9.93</td>
</tr>
<tr>
<td>3072</td>
<td>9.94</td>
</tr>
<tr>
<td>3584</td>
<td>9.95</td>
</tr>
<tr>
<td>4096</td>
<td>9.97</td>
</tr>
<tr>
<td>32768 (= 2^15)</td>
<td>9.89</td>
</tr>
</tbody>
</table>

* For the ease of tabulating performance results, only square MM performance measurements are shown. Rectangular MM of similar sizes have similar performance.

* GFLOPs = giga \( (10^{9}) \) floating-point operations per second
Why is MM so slow?

- Unnecessary reloads: the same source data undergoing multiple load instructions. $O(n^3)$ loads from each matrix.

- At least one of the source matrices breaks cache locality

Mapping between array `double *a` and the matrix $A$ is:

$$
\begin{pmatrix}
A(0,0) & A(0,1) & A(0,2) \\
A(1,0) & A(1,1) & A(1,2) \\
A(2,0) & A(2,1) & A(2,2)
\end{pmatrix}
= 
\begin{pmatrix}
\end{pmatrix}
$$
Countering reloads

- The more registers you have, the fewer reloads you have to do.
  - Want to exploit all the available registers

- We talk about AArch64 in this talk. It has 32 vector registers available, each can hold 2 doubles (128 bits)
  - NEON

- If matrix is small enough, then all matrix elements can be held inside these registers.
The MM Hierarchy

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x4</td>
<td>- All matrix elements can fit in vector registers entirely&lt;br&gt;- No need for reloads at all</td>
</tr>
<tr>
<td>128x128</td>
<td>- Need reloads into vector registers as we have more matrix elements&lt;br&gt;- Any way to avoid some of these reloads?&lt;br&gt;- Cache locality isn’t that bad yet</td>
</tr>
<tr>
<td>1024x1024 (and higher)</td>
<td>- Cache locality degrades drastically&lt;br&gt;- Tiling/packing are needed</td>
</tr>
</tbody>
</table>

Increasing matrix size
The small: 4x4 MM

Fits entirely in the registers
The 4x4 microcore

- 32 vector registers on AArch64, 2 doubles each
- 8 regs for A, 8 regs for C, 16 regs for B; each element loaded just once
- Reminder: column major order
The 4x4 microcore

dup v30.2d, b00

dup v31.2d, b10

...

load v0.2d, (a00, a10)

load v2.2d, (a01, a11)

load v1.2d, (c00, c10)

...

// every matrix element fits in the vec regs, no reloads whatsoever

fmla v1.2d, v0.2d, v30.2d

fmla v1.2d, v2.2d, v31.2d

......

store v1.2d, (c00, c10)

....

* To accomplish dup, might need help from extractelement, insertelement, shufflevector
The medium: 128x128 MM

Any way to avoid some of these reloads?
Outer Product Expansion (OPE)

- Matrix multiplication can be done in arbitrary blocks, as long as the blocks are of legal dimensions. In the example A, B, C, D can be plain numbers or matrices.

\[
\begin{pmatrix}
A_1 & B_1 \\
C_1 & D_1
\end{pmatrix}
\times
\begin{pmatrix}
A_2 & B_2 \\
C_2 & D_2
\end{pmatrix}
= \\
\begin{pmatrix}
A_1A_2 + B_1C_2 & A_1B_2 + B_1D_2 \\
C_1A_2 + D_1C_2 & C_1B_2 + D_1D_2
\end{pmatrix}
\]
Outer Product Expansion (OPE)

\[
\begin{bmatrix}
1 & 1
\end{bmatrix}
\begin{bmatrix}
1 & -1
\end{bmatrix}
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
= \begin{bmatrix}
1 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 2 & 3
\end{bmatrix}
+ \begin{bmatrix}
1 & -1
\end{bmatrix}
\begin{bmatrix}
4 & 5 & 6
\end{bmatrix}
\]

= \begin{bmatrix}
1 & 2 & 3 \\
1 & 2 & 3 \\
1 & 2 & 3
\end{bmatrix}
+ \begin{bmatrix}
4 & 5 & 6 \\
-4 & -5 & -6 \\
4 & 5 & 6
\end{bmatrix}

p=0  

p=1
Outer Product Expansion (OPE)

- The \((m,n)\)-th element in the product
  = inner product between \(m\)-th row of \(A\)
    and \(n\)-th col of \(B\)

But this inner product only has one item, so it’s just a single multiplication.
Outer Product Expansion (OPE)

- The \((m,n)\)-th element in the product
  
  \[ = \text{inner product between } m\text{-th row of } A \]
  
  and \(n\)-th col of \(B\)

But this inner product only has one item, so it’s just a single multiplication.
Outer Product Expansion (OPE)

- The \((m,n)\)-th element in the product
  \[= \text{inner product between } m\text{-th row of } A \]
  \[\text{and } n\text{-th col of } B\]

But this inner product only has one item, so it’s just a single multiplication.
Outer Product Expansion (OPE)

- The \((m,n)\)-th element in the product
  = inner product between \(m\)-th row of \(A\) and \(n\)-th col of \(B\)

But this inner product only has one item, so it’s just a single multiplication.
OPE inherently supports loop invariant code motion

Inspect the $p=0$ outer product
for (i in the current B row):
    this_B = B(i,p=0)
    for (j in the current A col):
        $C(i,j) += A(i,j) \times this_B$

- The load of $B(i,0)$ is lifted from the innermost loop
- Each $B(i,p)$ is loaded only once! $O(n^2)$ loads from $B$.
- A single outer product has $O(n^2)$ loads from $A$, so a total of $O(n^3)$ loads from $A$ for $n$ outer products in the whole MM
OPE inherently supports loop invariant code motion

Inspect the p=0 outer product

for (i in the current B row):
    this_B = B(i,p=0)

for (j in the current A col):
    C(i,j) += A(i,j)*this_B

• The load of B(i,0) is lifted from the innermost loop
• Each B(i,p) is loaded only once! O(n^2) loads from B.
• A single outer product has O(n^2) loads from A, so a total of O(n^3) loads from A
   for n outer products in the whole MM
OPE inherently supports loop invariant code motion

Inspect the p=0 outer product
for (i in the current B row):
  this_B = B(i,p=0)
  for (j in the current A col):
    C(i,j) += A(i,j)*this_B

• The load of B(i,0) is lifted from the innermost loop
• Each B(i,p) is loaded only once! O(n^2) loads from B.
• A single outer product has O(n^2) loads from A, so a total of O(n^3) loads from A
  for n outer products in the whole MM
How about loads from C?

Outer product procedure:
1. Do the p-th outer product and store into C in main memory.
   
   for i:
   
   for j:
   
   C(i, j) += ...

2. Switch to the next outer product (p++).

A total of $O(n^3)$ loads from C

\[
\begin{bmatrix}
1 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 2 & 3 \\
-1 & 5 & 6 \\
1 & 4 & 3 \\
\end{bmatrix}
= \begin{bmatrix}
1 & 1 \\
\end{bmatrix} 
\begin{bmatrix}
1 & 2 & 3 \\
\end{bmatrix} + \begin{bmatrix}
1 \\
\end{bmatrix} 
\begin{bmatrix}
1 & 2 & 3 \\
\end{bmatrix}
+ 
\begin{bmatrix}
-1 \\
\end{bmatrix} 
\begin{bmatrix}
4 & 5 & 6 \\
\end{bmatrix}
\]
Benefits from OPE

- $O(n^3)$ loads from A and C, $O(n^2)$ loads from B
- Try further reducing A and C loads
A \((mx4)x(4xn)\) outer product

- 64 loads from A
- 16 loads from B
- 64 loads from C

Total = 64 \times 16 \times 64 = 144 \text{ loads per 16 writes}
- Do 4x4 matrix fma with all entries from A, B and C loaded only once, with the 4x4 microcore. Then the outer product looks like this.
- Of course, lift the loads from B for the same mx4 column of A

![Diagram showing matrix operations and load reductions](image)
Loop Invariant Code Motion, the 4x4 version

move the 4x4 from B into vec reg, takes up 16 regs

for (a 4x4 in the mx4 column of A){
  load the 4x4 from A and C, takes up 8 regs each
  do the 4x4 MM
}

- Exploiting fully out of the 32 vec regs!

unroll, prefetch, …
Results of the first version

- This version is very fast at small sizes but degrades very quickly.
- Use as a 128x128 macrocore!

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>Performance (GFLOPs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10.0</td>
</tr>
<tr>
<td>512</td>
<td>8.2</td>
</tr>
<tr>
<td>1024</td>
<td>7.8</td>
</tr>
<tr>
<td>1536</td>
<td>7.7</td>
</tr>
<tr>
<td>2048</td>
<td>6.9</td>
</tr>
</tbody>
</table>

- Theoretical maximum for the testing machine is 10.4 GFLOPs (double precision)
- Single-thread
The large: 1028x1028 MM

Cache considerations and tiling
The problem: column jumps are too big when loading

- Here the (a00, a10) and (a01, a11) register loads are 128/2048 addresses apart in main memory if the matrix size is 128/2048.
- Bad caching due to poor special locality
Using temporary arrays for macrocore

- Therefore we want to do the entire 2048x2048 MM in 128x128 blocks.

- Specifically, we want three temporary arrays:
  - `cur_a = (double *)malloc(sizeof(double)*128*128);`
  - `cur_b = (double *)malloc(sizeof(double)*128*128);`
  - `cur_c = (double *)malloc(sizeof(double)*128*128);`

- … pack the 128x128 blocks into these temporary arrays, and use the first method on these temporary arrays. We want to do this because we know the first method is fast (10G!) on 128x128 arrays.
How to schedule the 128x128 macrocores?

Note: now each cell represents a 128x128
A "load" refers to an entire 128x128 load

8 loads from A (pack) and 64 loads from B per 8 writes to C

Total loads = (8+64) * 8 = 576 loads
How to schedule the 128x128 macrocores?

Note: now each cell represents a 128x128, a "load" refers to an entire 128x128 load.

32 loads from A (pack) and 32 loads from B (pack) per 16 writes to C.

Total loads = (32+32)*4 = 256 loads.
How to schedule the 128x128 macrocores?

Tile size = 512

Note: now each cell represents a 128x128, A "load" refers to an entire 128x128 load

Each tile has 4 macrocore-sized row/column panels

32 loads from A (pack) and 16 writes to C

Total loads = (32+32)*4 = 256 loads

320 loads fewer!
How to schedule the 128x128 macrocores?
Packing the tiles

- Problem with packing of the tiles: need an extra load from the tile’s packing array to temporary 128x128 arrays

- To illustrate, imagine a **macrocore** size of 2x2, and a tile size of 4.
  - Each tile has 2 rows/columns of **macrocore sized panels**.
Packing the tiles
Packing the tiles

```
double * a_pack
(size = 4*8)
```

```
double * b_pack
(size = 8*4)
```
Packing the tiles
Packing the tiles

double * cur_a
(size = 2*2)

double * cur_b
(size = 2*2)
Packing the tiles

double * cur_a (size = 2*2)
double * cur_b (size = 2*2)
Packing the tiles

double * cur_a  
(size = 2*2)

double * cur_b  
(size = 2*2)
Packing the tiles

double * cur_a
(size = 2*2)

double * cur_b
(size = 2*2)
Packing the tiles

Note: the numbers in this figure are addresses within the packing array, not values of matrix elements.

- Need to load from the pack arrays into cur_a and cur_b arrays, since the 2x2 macrocores are not in contiguous locations in the pack.
Packing the tiles

No need to load from the packing arrays into cur_a and cur_b arrays, since the 2x2 macrocores are in contiguous locations in the pack.

To go to the next macrocore, simply increment the pointer!

\[
\text{cur}_a += 2*2; \quad \text{cur}_b += 2*2;
\]

Note: the numbers in this figure are addresses within the packing array, not values of matrix elements.
Packing the tiles

Note: the numbers in this figure are addresses within the packing array, not values of matrix elements.

• No need to load from the packing arrays into cur_a and cur_b arrays, since the 2x2 macrocores are in contiguous locations in the pack.

• To go to the next macrocore, simply increment the pointer!

\[ \text{cur}_a += 2*2; \text{cur}_b += 2*2; \]
Complete summary
1. Tile A into row tiles. In the demo tile size=512
Pack the A row tiles in the zigzag fashion
2. Tile B into column tiles. In the demo tile size = 512.
Pack the B column tiles in the zigzag fashion
3. Within the inner product for a tile (i.e. within the same $jj$), do $(\text{tile size/macrocore size})^2$ macrocore-sized inner products.
4. Within one macrocore-sized inner product, use zigzag packing and iterate by simply incrementing pointers

\[
\text{cur}_a += 128 \times 128
\]

\[
\text{cur}_b += 128 \times 128
\]
Within a 128x128 macrocore:

- Use the first version, i.e. outer product + 4x4_microcore
Thank you

…and thanks to excel that made these otherwise really annoying graphs possible