MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Tile size selection using cache organization and data layout

EECS 583 Paper Presentation

Original authors: Stephanie Coleman and Kathryn S. McKinley

Presenters: Group 20: Boren Ke, Peijing Li, Yongyi Yang

Monday, December 4, 2023

Presentation Outline

1. Introduction
   a. Why study matrix multiplication
   b. Cache terminologies
   c. Matrix tiling demonstration
2. Methods
   a. Finding the height of the tile given its width
   b. Finding the optimal width (and height) of the tile
3. Results
4. Commentary

Why study matrix multiplication?

- **Wide applicability** across many modern algorithms, e.g., machine learning
- **High computational complexity**, no asymptotically fast algorithms
- Other algorithms that don't necessarily involve matrices utilize **similar computational patterns**.

Objective: accelerate matrix multiplication through exploiting spatial locality of its large number of elements in caches.

Terminology and Definitions

1. Cache misses:
   a. Compulsory miss
   b. Capacity miss
   c. Interference a.k.a. conflict miss
      i. **Self-interference**: conflicts with elements of the same matrix
      ii. **Cross-interference**: conflicts with elements of different matrix
2. **Spatial locality** and temporal locality
3. Describing matrix and tile dimensions…

Why is MM a problem?

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Our goal:

1. **Eliminate self-interference misses**
2. **minimize cross-interference misses of B caused by A or C**

Naive matrix multiplication

```
int A[N][N];
int B[N][N];
int C[N][N];
init();

for(int j = 0;j < N;j++){
    for(int k = 0;k < N;k++){
        int X = C[k][j];
        for(int i = 0;i < N;i++){
            A[i][j] = A[i][j] + X * B[i][k];
        }
    }
}
```

Between each use of....

A[i,j]: N elements in B

B[i,k]: N * N elements in B, plus

N elements in A

N elements in C

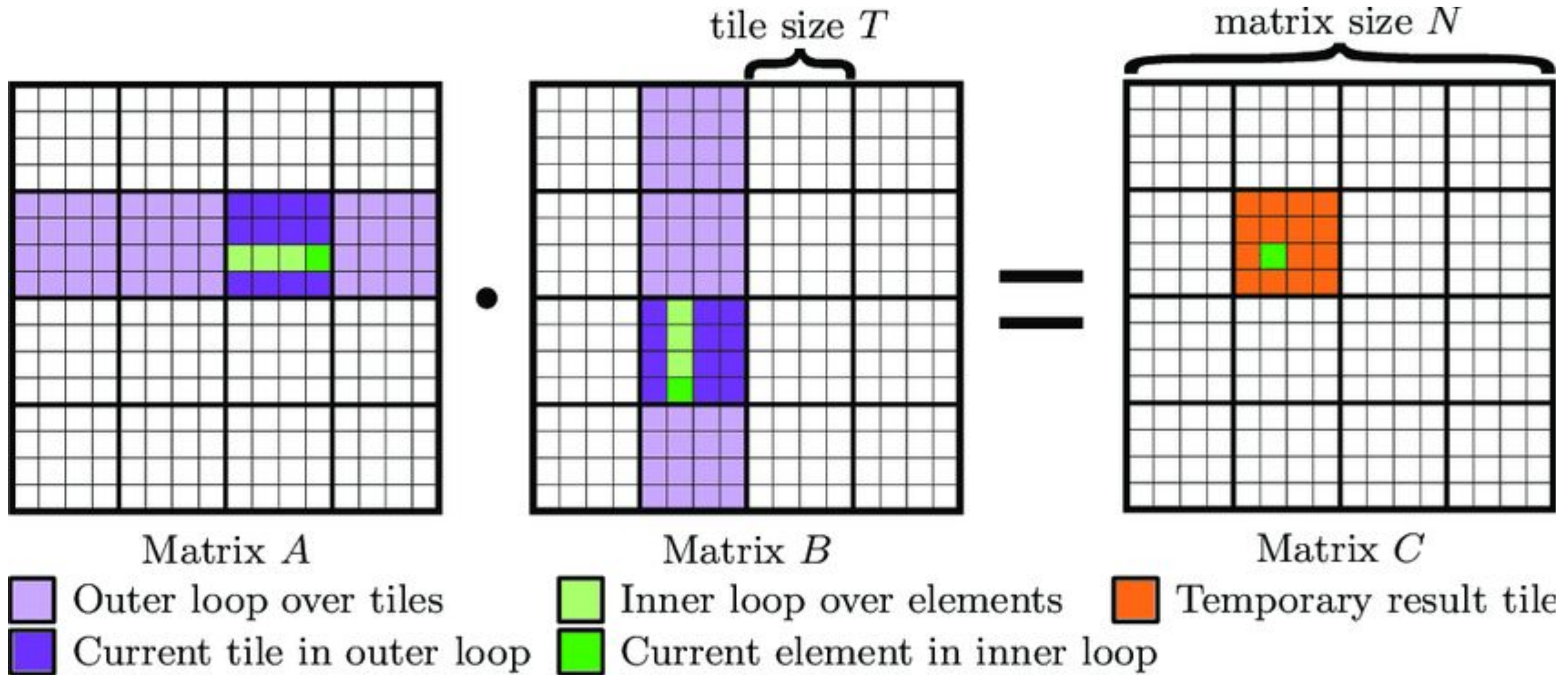
cross-interference misses

self-interference misses

Method: Tiling

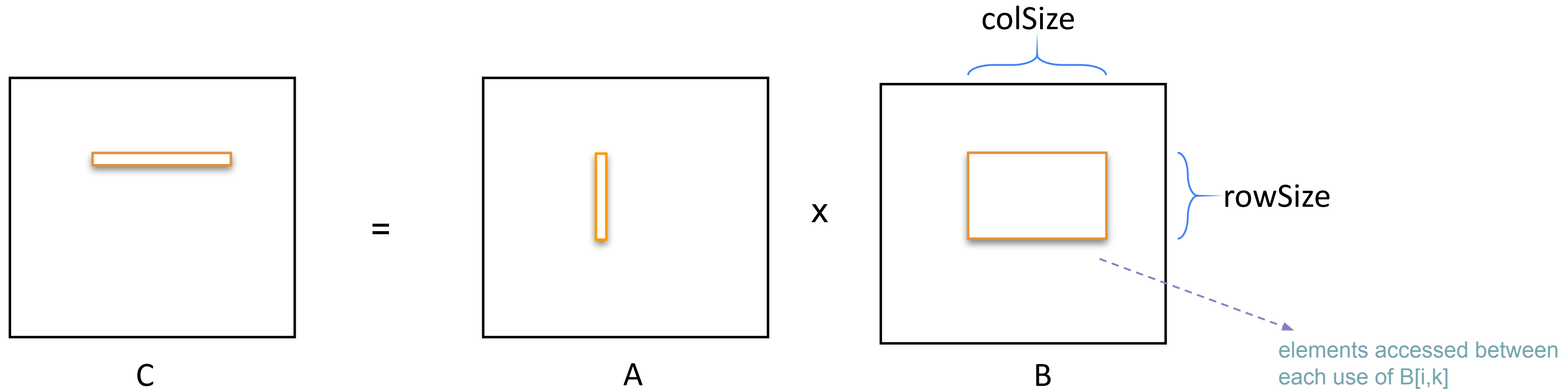
```
int TK = 20;
int TI = 20;
for(int tileK = 0; tileK < N; tileK += TK){
    for(int tileI = 0;tileI < N; tileI += TI){
        for(int j = 0;j < N;j++){
            for(int k = tileK;(k < tileK + TK) && (k < N);k++){

                int X = C[k][j];

                for(int i = tileI;(i < tileI + TI) && (i < N);i++){
                    A[i][j] = A[i][j] + X * B[i][k];
                }

            }
        }
    }
}
```

Method: Tiling



MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

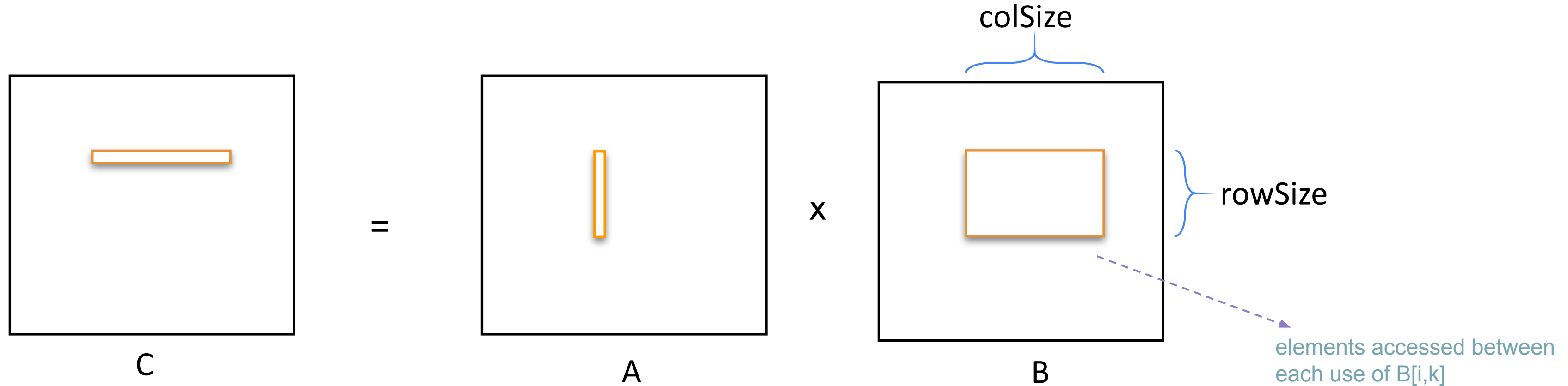
Tile Size Selection



Between each use of B[i,k], we need to access

- colSize * rowSize elements in B, plus
- colSize elements in C, plus
- rowSize elements in A

Tile Size Selection

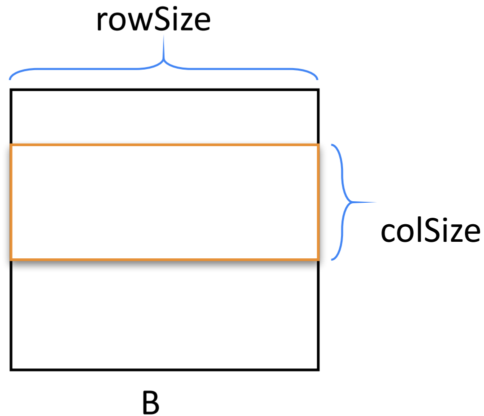


We want colSize and rowSize to be:

- small enough to eliminate self-interference misses of B
- big enough to fully use the cache

Tile Size Selection

A naive method: select as many columns as possible.



$$\text{colSize} = \left\lfloor \frac{\text{cache size}}{N} \right\rfloor$$

unused cache units: $r_1 = \text{cache size} \bmod N$

In the optimal solution, the number of unused cache units should not exceed r1.

Tile Size Selection

Proposed method: Euclidean Algorithm (originally used to calculate GCD)

Input : a, b

$$
\begin{aligned}
a &= q_1 b + r_1 \\
b &= q_2 r_1 + r_2 \\
r_1 &= q_3 r_2 + r_3 \\
&\cdots \\
r_{k-1} &= q_{k+1} r_k + r_{k+1}
\end{aligned}
$$

Each time, calculate the remainder of two remainders, until one remainder becomes 0.

Tile Size Selection

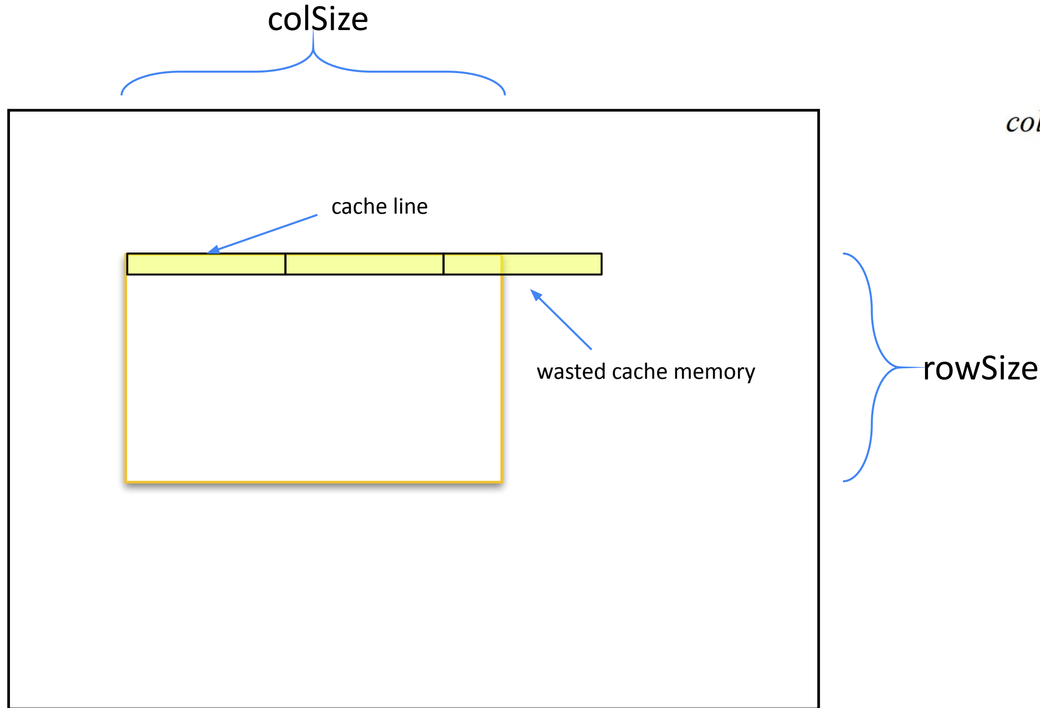
Proposed method: Euclidean Algorithm (originally used to calculate GCD)

Input : a = N, b = cache size

$$
\begin{aligned}
a &= q_1 b + r_1 \\
b &= q_2 r_1 + r_2 \\
r_1 &= q_3 r_2 + r_3 \\
&\cdots \\
r_{k-1} &= q_{k+1} r_k + r_{k+1}
\end{aligned}
$$

Each time, calculate the remainder of two remainders, until one remainder becomes 0.

Tile Size Selection



Take into account cache line size

$$colSize = \begin{cases} colSize & \text{if } colSize \bmod CLS = 0, or \\ & \text{if } colSize = column\ length \\ \left\lfloor \frac{colSize}{CLS} \right\rfloor CLS & otherwise \end{cases}$$

(CLS = cache line size)

Tile Size Selection

Minimizing Cross Interference Misses

worst case number of cross interference misses:

$$\text{CIM} = 2 \times \text{rowSize} + \text{colSize}$$

cross interference ratio:

$$\text{CIR} = \frac{\text{CIM}}{\text{rowSize} \times \text{colSize}}$$

working set size constraint:

$$\text{colSize} \times \text{rowSize} + \text{rowSize} + \text{CLS} < \text{cache size}$$

the algorithm: choose the pair with the best CIR while not violating working set size constraint

Performance vs. Untiled Matrix Multiplication

1. Test over 2D matrix multiplication, successive over relaxation, LU decomposition, and (expanded) Livermore Loop 23 algorithms
2. Select problem size of 256*256, 300*300, and 301*301
3. **Tiling improves average miss rates by a factor of 8.6**
    a. 32-byte cache line: **9.5x**; 64-byte cache line: **7.62x**
4. Cache performance always increases with **set associativity**
    a. Even though algorithm is designed to work with direct-mapped cache

Comparison with Other Tiling Algorithms

1. The competition: different tile shapes
   a. **L**am, **R**othenberg, **W**olf 1991: largest **square tiles** without self interference.
   b. **Esseghir** 1993: fit as many **entire rows** into cache as possible
2. Esseghir has too big **working set** sizes, LRW has too small **working set** sizes
3. For **larger cache sizes**, the benefits of optimizing rectangular cache shape becomes less apparent
4. Performance differences diminish with higher **set associativity**

	Miss rate @ 8KB	Speedup @ 8KB	Miss rate @ 64KB	Speedup @ 64KB
LRW '91	1.03	1.54	0.85	1.06
Esseghir '93	6.66	1.27	1.19	0.98

Commentary

Strengths

- Algorithm fully eliminates **self-interference misses**
- Low **time complexity** for pinpointing tile size (O(log^3))
- Good **benchmarking and comparison data** instead of skewed graphics dump

Limitations

- **Loop order exchange**
- **Fitting cache line size** after determining colSize
- No explicit explanation of using **Euclidean Algorithm** (heuristic)
- Further discussion on **associativity**
- Unintuitive writing and examples
- Inconsistent terminologies

Thanks!