CoE 163

Computing Architectures and Algorithms

Matrix-Matrix Multiplication (part 2)

Recall our MMM "ijk" algorithm

```
for i = 1 to n
   for j = 1 to n
        for k = 1 to n
             Cij = Cij+ Aik*Bkj
        end for
   end for
end for
```

Load row i of A into fast memory

Recall our MMM "ijk" algorithm

```
for i = 1 to n
   for j = 1 to n
          for k = 1 to n
                 Cij = Cij+ Aik*Bkj
          end for
   end for
end for
```

Load row i of A into fast memory

Load C_{ij} into fast memory

Recall our MMM "ijk" algorithm

```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      Cij = Cij+ Aik*Bkj
    end for
  end for
end for
```

Load row i of A into fast memory

Load C_{ij} into fast memory

Load column j of B into fast memory

Recall our MMM "ijk" algorithm

```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C_ij = C_ij + A_ik * B_kj
    end for
  end for
end for
```

Load row i of A into fast memory

Load C_{ij} into fast memory

Perform operation

Load column j of B into fast memory

Recall our MMM “ijk” algorithm

```
for i = 1 to n
   for j = 1 to n
      for k = 1 to n
           C_ij = C_ij + A_ik*B_kj
      end for
   end for
end for
```

Load row i of A into fast memory

Load C_{ij} into fast memory

Perform operation

Load column j of B into fast memory

Write C_{ij} back to slow memory

Is there a way to make our MMM algorithm more efficient in terms of memory use?

First let's analyze the performance of our algorithm

Assumptions about computer archi

- **2 levels of memory:** slow and fast
- **Slow memory**
  - Assume **column major**
  - Large enough to store 3 $n \times n$ matrices, A, B, and C
- **Fast memory**
  - Only contains M words where $2n < M \ll n^2$
  - Cannot contain an entire $n \times n$ matrix
  - Can contain at least 2 matrix columns or rows

Slow memory can contain 2 rows of A in fast memory

- Suppose n = 10, and M = 64
- Example shows 4-word cache lines

a_{11}
a_{21}
a_{31}
…
a_{71}
a_{81}
a_{91}
$a_{10\,1}$
…

Matrix A stored column-wise in slow memory

Line number	4 words per cache line			
x	a_{11}	a_{21}	a_{31}	a_{41}
x+1	a_{91}	$a_{10\,1}$	a_{12}	a_{22}
x+2	a_{13}	a_{23}	a_{33}	$a_{4\,3}$
x+3	a_{93}	$a_{10\,3}$	a_{14}	a_{24}
x+4	a_{15}	a_{25}	a_{35}	a_{45}
x+5	a_{95}	$a_{10\,5}$	a_{16}	a_{26}
x+6	a_{17}	a_{27}	a_{37}	a_{47}
x+7	a_{97}	$a_{10\,7}$	a_{18}	a_{28}
x+8	a_{19}	a_{29}	a_{39}	a_{49}
x+9	a_{99}	$a_{10\,9}$	$a_{1\,10}$	$a_{2\,10}$
x+10				
x+12				
x+13				
x+14				
x+15				

Fast memory with 64 words: greater than 2n, but much less than n^2

Total number of memory references?

- n^2 : Move n elements per row of A ($n \times n$) into fast memory, keep it there until no longer needed
- n^3 : Move n elements per column of B ($n \times n$), n times (for each value of i)
- $2n^2$: Move each element of C into fast memory until computation completes, then move back into slow memory (2 transfers per element)
- **Thus, this algorithm involves $\mathbf{3n^2 + n^3}$ memory references**

What does this say about the performance?

Total number of memory references?

- n^2 : Move n elements per row of A ($n{\times}n$) into fast memory, keep it there until no longer needed
- n^3 : Move n elements per column of B ($n{\times}n$), n times (for each value of i)
- $2n^2$: Move each element of C into fast memory until computation completes, then move back into slow memory (2 transfers per element)
- **Thus, this algorithm involves $\boldsymbol{3n^2 + n^3}$ memory references**

Execution time grows approx. cubically as n increases

How efficient is the algorithm?

- f - number of floating point operations
  - 3 nested loops that iterate from 1 to n, 2 operations at innermost loop, thus $\boldsymbol{f = 2n^3}$
- Let $q = $ ratio of f to memory references
- $q = 2n^3/(3n^2 + n^3)$
  - If n is very large, $\boldsymbol{q \approx 2}$ (try solving for q when $n = 500$)
- Approx only 2 operations per memory reference

Is there a way to improve this?

MMM algorithm with blocking

Costly: row traversal on row-major memory

- 2 columns of B involves data that are close to each other – OK!
- Use up many cache lines for 2 rows of A – NOT OK!

- MMM operation has inherent problem:
  - One matrix is traversed row-wise, the other column-wise
  - Whether memory is row- or column-major, we do costly cache transfers

Line number	4 words per cache line			
x	a_{11}	a_{21}	a_{31}	a_{41}
x+1	a_{91}	$a_{10\ 1}$	a_{12}	a_{22}
x+2	a_{13}	a_{23}	a_{33}	$a_{4\ 3}$
x+3	a_{93}	$a_{10\ 3}$	a_{14}	a_{24}
x+4	a_{15}	a_{25}	a_{35}	a_{45}
x+5	a_{95}	$a_{10\ 5}$	a_{16}	a_{26}
x+6	a_{17}	a_{27}	a_{37}	a_{47}
x+7	a_{97}	$a_{10\ 7}$	a_{18}	a_{28}
x+8	a_{19}	a_{29}	a_{39}	a_{49}
x+9	a_{99}	$a_{10\ 9}$	$a_{1\ 10}$	$a_{2\ 10}$
x+10	b_{11}	b_{21}	b_{31}	b_{41}
x+12	b_{51}	b_{61}	b_{71}	b_{81}
x+13	b_{91}	$b_{10\ 1}$	b_{12}	b_{22}
x+14	b_{32}	b_{42}	b_{52}	b_{62}
x+15	b_{72}	b_{82}	b_{92}	$b_{10\ 2}$

Fast memory with 64 words: greater than 2n, but much less than n^2

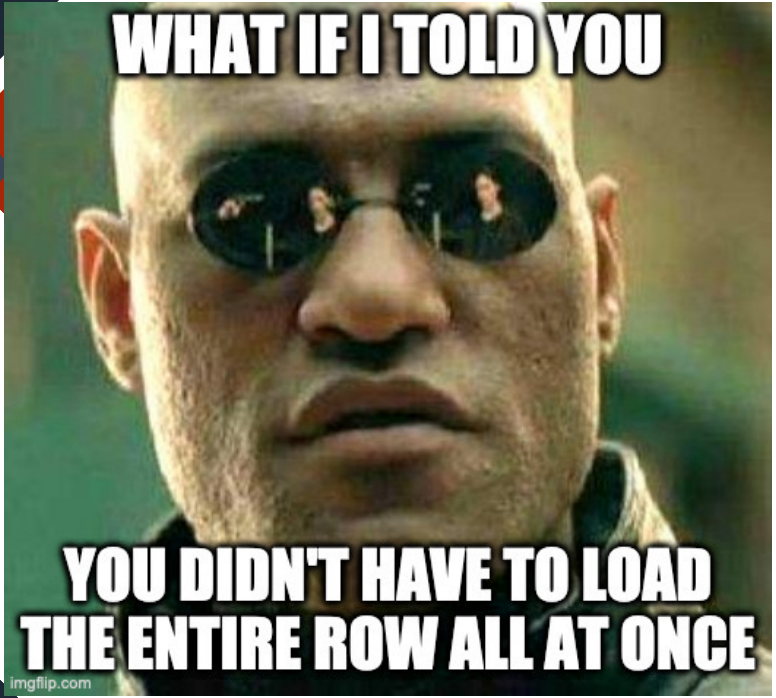
Costly: traversal with long *strides*

- Innermost loop of algorithm uses an **entire row** of matrix A and **entire columns** of matrix B – Long strides
- Uses up many cache lines for a few operations
- **Shorter strides are often better**

Line number	4 words per cache line			
x	a_{11}	a_{21}	a_{31}	a_{41}
x+1	a_{91}	$a_{10\ 1}$	a_{12}	a_{22}
x+2	a_{13}	a_{23}	a_{33}	$a_{4\ 3}$
x+3	a_{93}	$a_{10\ 3}$	a_{14}	a_{24}
x+4	a_{15}	a_{25}	a_{35}	a_{45}
x+5	a_{95}	$a_{10\ 5}$	a_{16}	a_{26}
x+6	a_{17}	a_{27}	a_{37}	a_{47}
x+7	a_{97}	$a_{10\ 7}$	a_{18}	a_{28}
x+8	a_{19}	a_{29}	a_{39}	a_{49}
x+9	a_{99}	$a_{10\ 9}$	$a_{1\ 10}$	$a_{2\ 10}$
x+10	b_{11}	b_{21}	b_{31}	b_{41}
x+12	b_{51}	b_{61}	b_{71}	b_{81}
x+13	b_{91}	$b_{10\ 1}$	b_{12}	b_{22}
x+14	b_{32}	b_{42}	b_{52}	b_{62}
x+15	b_{72}	b_{82}	b_{92}	$b_{10\ 2}$

Fast memory with 64 words: greater than $2n$, but much less than n^2

Costly: traversal with long *strides*



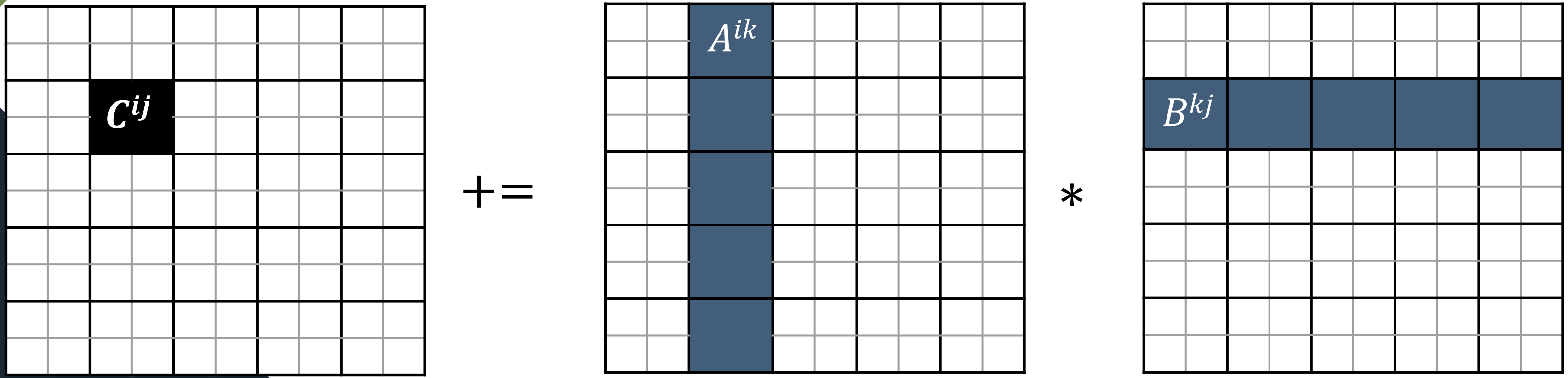
Morpheus, from "The Matrix"

Line number	4 words per cache line			
x	a_{11}	a_{21}	a_{31}	a_{41}
x+1	a_{91}	$a_{10\ 1}$	a_{12}	a_{22}
x+2	a_{13}	a_{23}	a_{33}	$a_{4\ 3}$
x+3	a_{93}	$a_{10\ 3}$	a_{14}	a_{24}
x+4	a_{15}	a_{25}	a_{35}	a_{45}
x+5	a_{95}	$a_{10\ 5}$	a_{16}	a_{26}
x+6	a_{17}	a_{27}	a_{37}	a_{47}
x+7	a_{97}	$a_{10\ 7}$	a_{18}	a_{28}
x+8	a_{19}	a_{29}	a_{39}	a_{49}
x+9	a_{99}	$a_{10\ 9}$	$a_{1\ 10}$	$a_{2\ 10}$
x+10	b_{11}	b_{21}	b_{31}	b_{41}
x+12	b_{51}	b_{61}	b_{71}	b_{81}
x+13	b_{91}	$b_{10\ 1}$	b_{12}	b_{22}
x+14	b_{32}	b_{42}	b_{52}	b_{62}
x+15	b_{72}	b_{82}	b_{92}	$b_{10\ 2}$

Fast memory with 64 words: greater than 2n, but much less than n^2

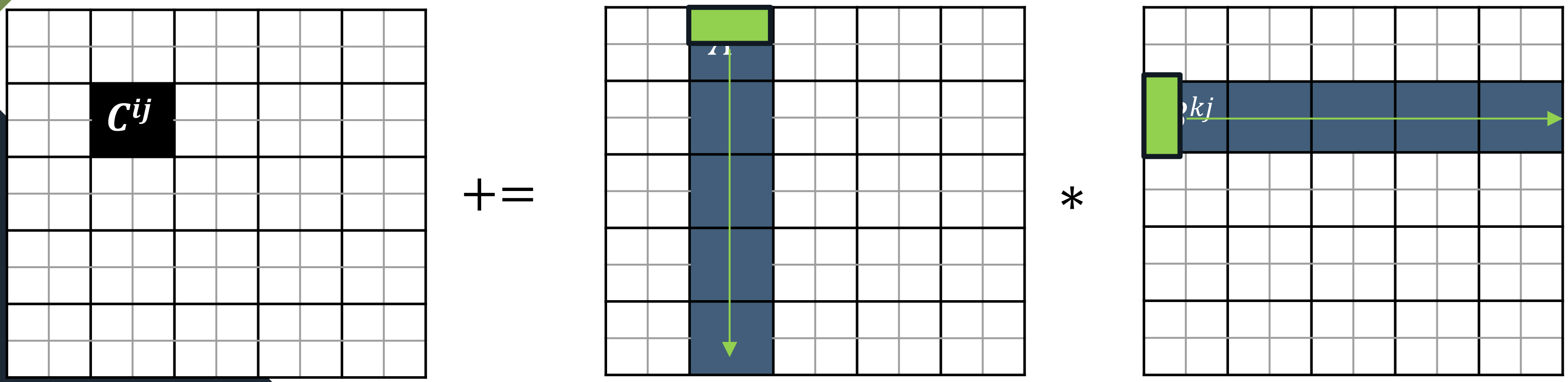
Let's use blocking

- Let's break C into an $(N \times N)$ block matrix with $\left(\frac{\boldsymbol{n}}{\boldsymbol{N}} \times \frac{\boldsymbol{n}}{\boldsymbol{N}}\right)$blocks
- C^{ij}, and A and B are similarly partitioned
- Example below when $\mathrm{N} = 5$ **and** $\boldsymbol{n} = \mathbf{10}$



Let's use blocking

- $\frac{n}{N}$ length slices of rows of A are multiplied with $\frac{n}{N}$ height segments of columns of B as shown below



Blocking gives us shorter strides

- We break up the MMM computation into smaller chunks
- Traverse with shorter strides across our rows and columns
- Diagram shows 2x2 sub-blocks for A, B, and C in cache
- We don’t waste so many cache lines per operation!

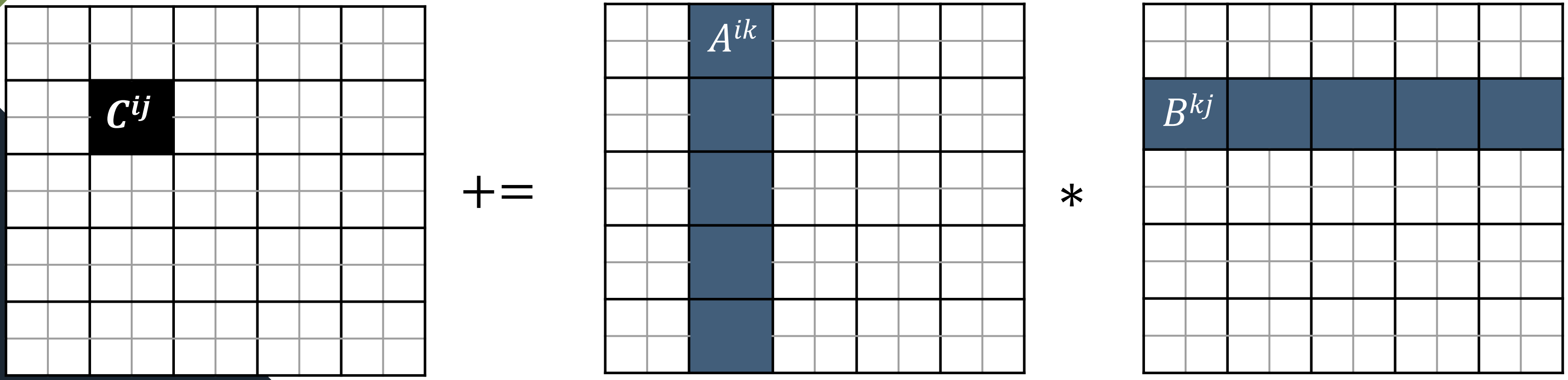
$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} += \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Line number	4 words per cache line			
x	a_{11}	a_{21}	a_{31}	a_{41}
x+1	a_{91}	$a_{10\,1}$	a_{12}	a_{22}
x+2	b_{11}	b_{21}	b_{31}	b_{41}
x+3	b_{91}	$b_{10\,1}$	b_{12}	b_{22}
x+4	C_{11}	C_{21}	C_{31}	C_{41}
x+5	C_{91}	$C_{10\,1}$	C_{12}	C_{22}
x+6				
x+7				
x+8				
x+9				
x+10				
x+12				
x+13				
x+14				
x+15				

Fast memory with 64 words: greater than 2n, but much less than n^2

Our algorithm with blocking becomes:

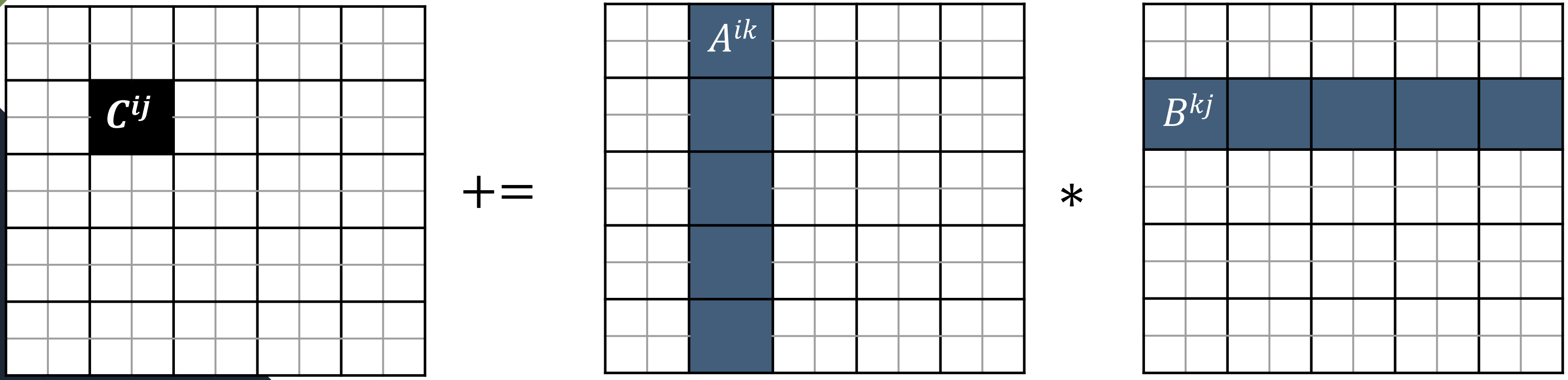
```
for i = 1 to N
   for j = 1 to N
      for k = 1 to N
            Cij = Cij + Aik · Bkj
      end for
   end for
end for
```



Our algorithm with blocking becomes:

```
for i = 1 to N
   for j = 1 to N
       for k = 1 to N
              Cij = Cij + Aik · Bkj
       end for
   end for
end for
```

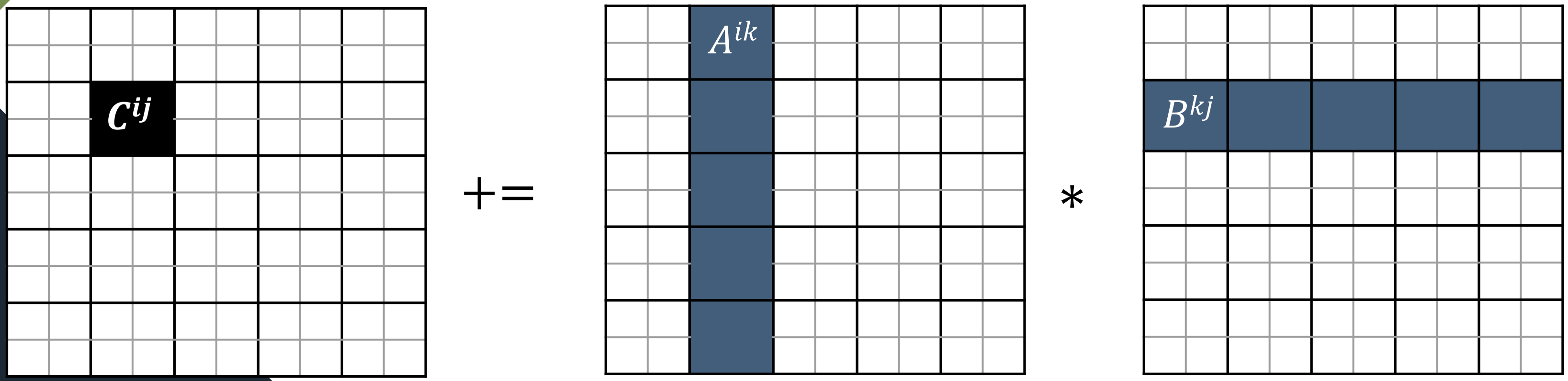
Matrix multiplication of (2×2) block A^{ik} and (2×2) block B^{kj}



Our algorithm with blocking becomes:

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            C^ij = C^ij + A^ik · B^kj
        end for
    end for
end for
```

Load C^{ij} into fast memory

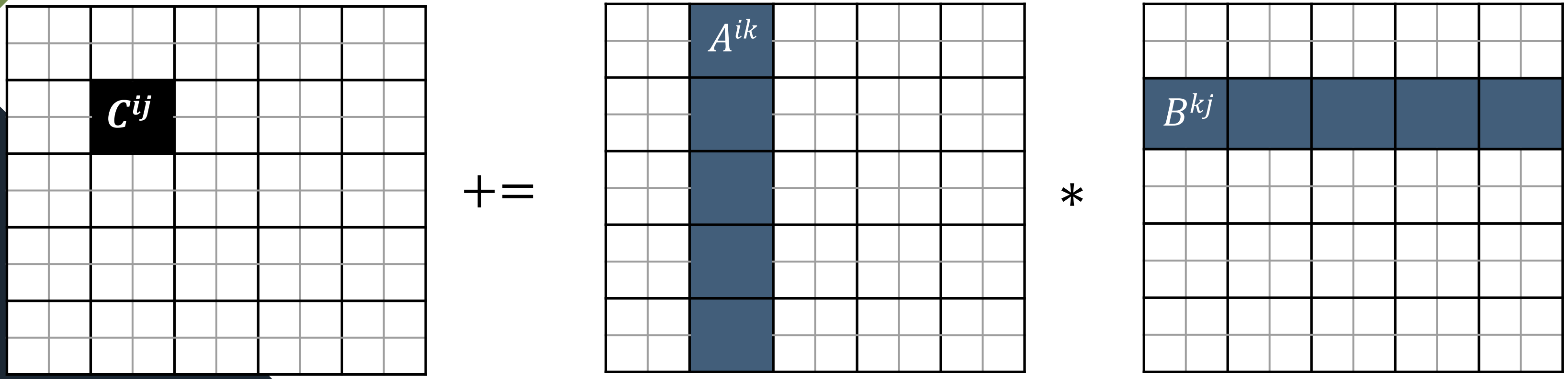


Our algorithm with blocking becomes:

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            C^ij = C^ij + A^ik · B^kj
        end for
    end for
end for
```

Load C^{ij} into fast memory

Load A^{ik} and B^{kj} into fast memory



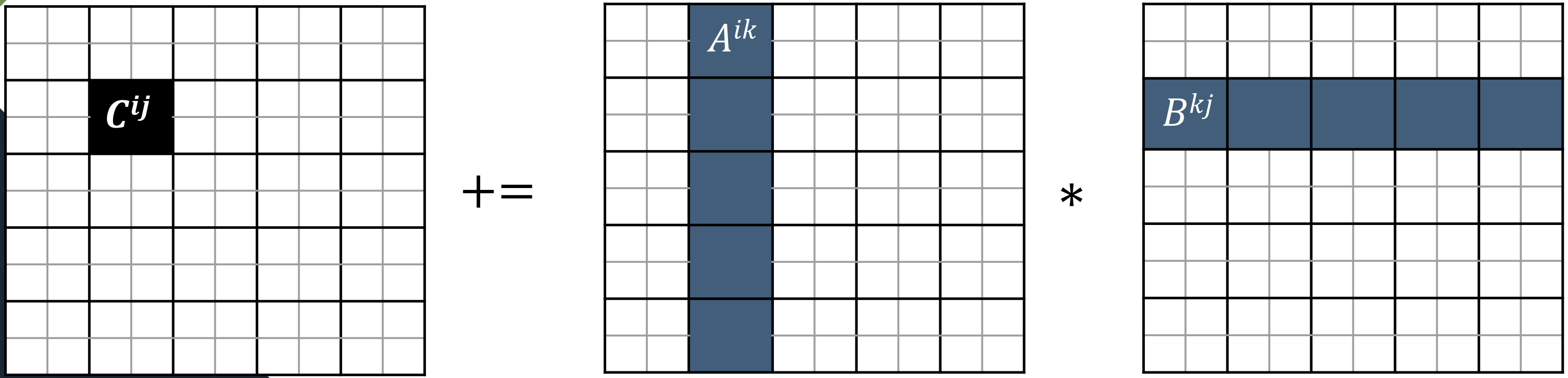
Our algorithm with blocking becomes:

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            C^ij = C^ij + A^ik · B^kj
        end for
    end for
end for
```

Load C^{ij} into fast memory

Load A^{ik} and B^{kj} into fast memory

Perform operation



Our algorithm with blocking becomes:

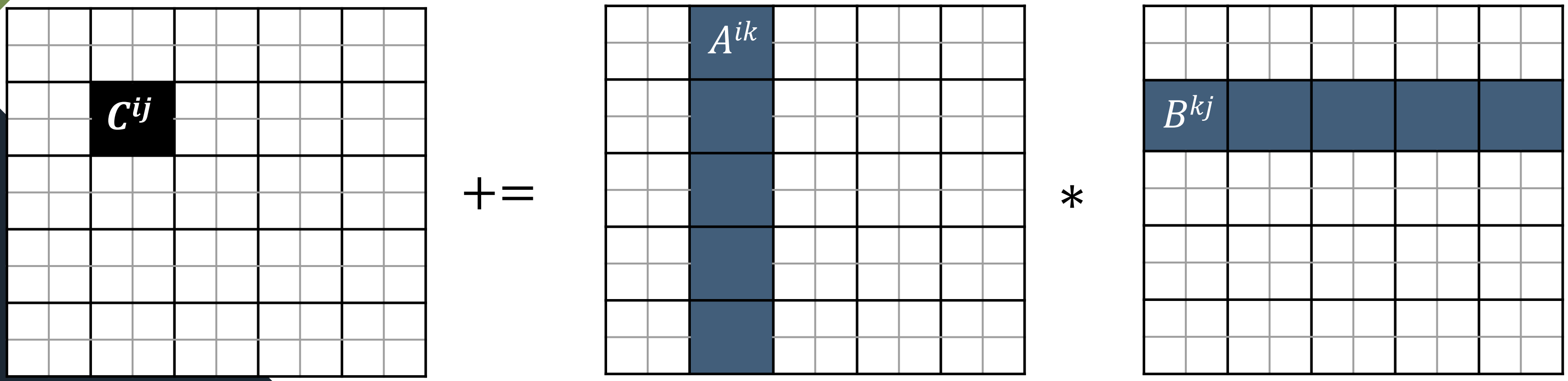
```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            Cij = Cij + Aik · Bkj
        end for
    end for
end for
```

Load C^{ij} into fast memory

Load A^{ik} and B^{kj} into fast memory

Perform operation

Write C^{ij} back to slow memory



How many memory references if blocking is used?

- Read each $(\frac{n}{N} \times \frac{n}{N})$ block of A N^3 times:
  - $N^3(\frac{n^2}{N^2}) = \boldsymbol{Nn^2}$
- Read each $(\frac{n}{N} \times \frac{n}{N})$ block of B N^3 times:
  - $\boldsymbol{Nn^2}$
- Read and write each $(\frac{n}{N} \times \frac{n}{N})$ block of C once
  - n^2 (read) + n^2 (write) = $2n^2$
- Total: $2n^2 + 2Nn^2 = (2 + 2N)n^2 \approx \boldsymbol{2Nn^2}$
  - N is usually much larger than 2, so we get approximately $\boldsymbol{2Nn^2}$ memory references
-

How many memory references if blocking is used?

- Read each $(\frac{n}{N} \times \frac{n}{N})$ block of A N^3 times:
  - $N^3(\frac{n^2}{N^2}) = \boldsymbol{Nn^2}$
- Read each $(\frac{n}{N} \times \frac{n}{N})$ block of B N^3 times:
  - $\boldsymbol{Nn^2}$
- Read and write each $(\frac{n}{N} \times \frac{n}{N})$ block of C once
  - n^2 (read) + n^2 (write) = $2n^2$
- Total: $2n^2 + 2Nn^2 = (2 + 2N)n^2 \approx \boldsymbol{2Nn^2}$
  - N is usually much larger than 2, so we get approximately $\boldsymbol{2Nn^2}$ memory references

Given: $2Nn^2$, how do we minimize memory references?

- Choose as small as possible N (ie larger blocks)
- Constraint for size of N:
  - We should be able to fit one $(\frac{n}{N}\times\frac{n}{N})$ block each for A, B, and C simultaneously
  - This lets us load into fast memory all the data needed to iterate and perform operations at the innermost loop for k=1 to n
  - Thus, $M \geq 3\left(\frac{n}{N}\right)^2$
- $N = \boldsymbol{n}\sqrt{\frac{\mathbf{3}}{\boldsymbol{M}}}$

How efficient is the blocked algorithm?

- Memory references: $\mathbf{2Nn^2}$
- Number of floating point operations: $\mathbf{2n^3}$
- Select N to be approx $\boldsymbol{n\sqrt{\frac{3}{M}}}$
- Thus we get:
  - $q \approx \frac{2n^3}{2Nn^2} \approx \frac{n}{n\sqrt{\frac{3}{M}}} \approx \sqrt{\frac{M}{3}}$

How efficient is the blocked algorithm?

- $q \approx \sqrt{\frac{M}{3}}$
    - $O(\sqrt{M})$
    - q grows as M grows: more efficient with larger cache/fast memory
    - Grows independently of n: fast for any matrix size $n \times n$

Additional remarks on blocked algorithm

- It can be shown that the algorithm is asymptotically optimal
- Real code will have to handle asymmetric matrices – optimal block size may not be square
- Cache and register structure of machine will affect the best shapes of submatrices

Try it yourself!

- Implement MMM with and without blocking
- Use large matrices (say, n=1000)
- Compare performance / runtimes

There are other ways to optimize MMM

- Only a few methods are discussed in the course (arrangement of loops, and blocking)
- Other methods are out there
  - Can we transpose one matrix first then iterate column-wise or row-wise for both?
  - Strassen algorithm with $O(\boldsymbol{n}^{2.807355})$
  - Coppersmith–Winograd algorithm with $O(\boldsymbol{n}^{2.375477})$
- Often, optimizations make code harder to read but improve cache behavior

Final words

- MMM is at the heart of many linear algebra algorithms
- Achieving an optimized MMM will improve performance of many applications