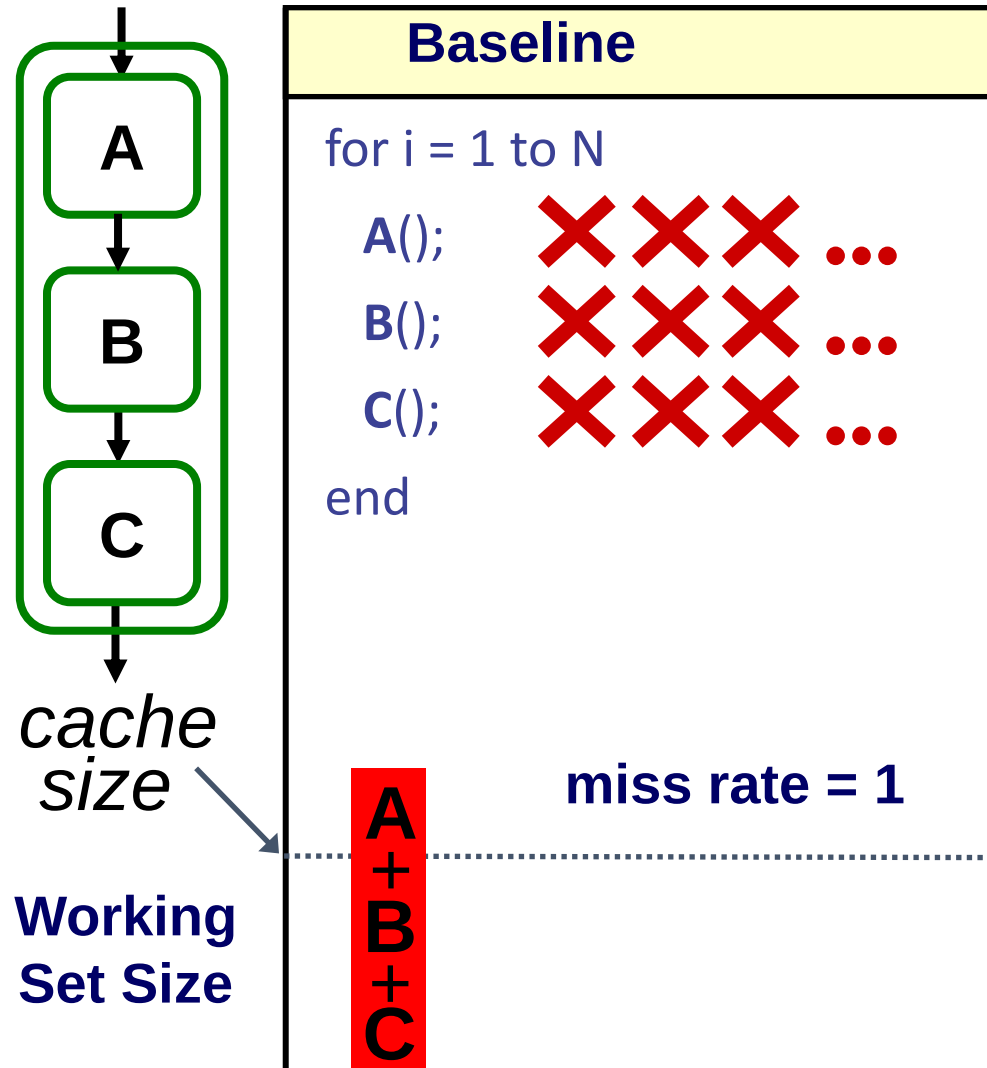




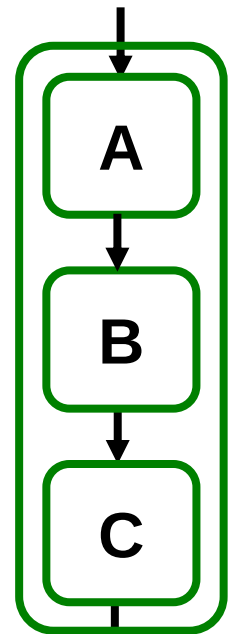
Memory Optimizations

Ozcan Ozturk – Sabancı University

Instruction Locality



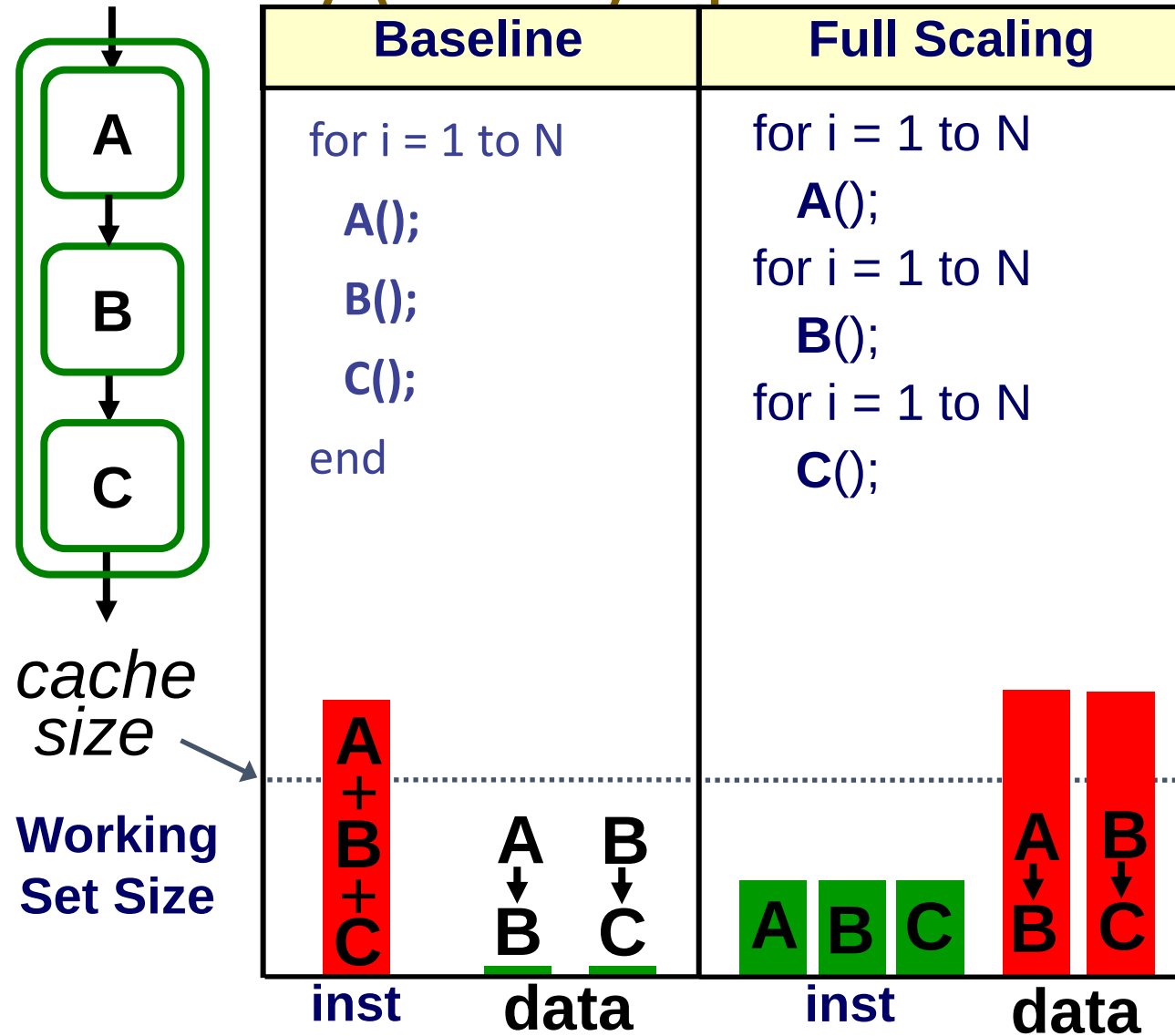
Instruction Locality



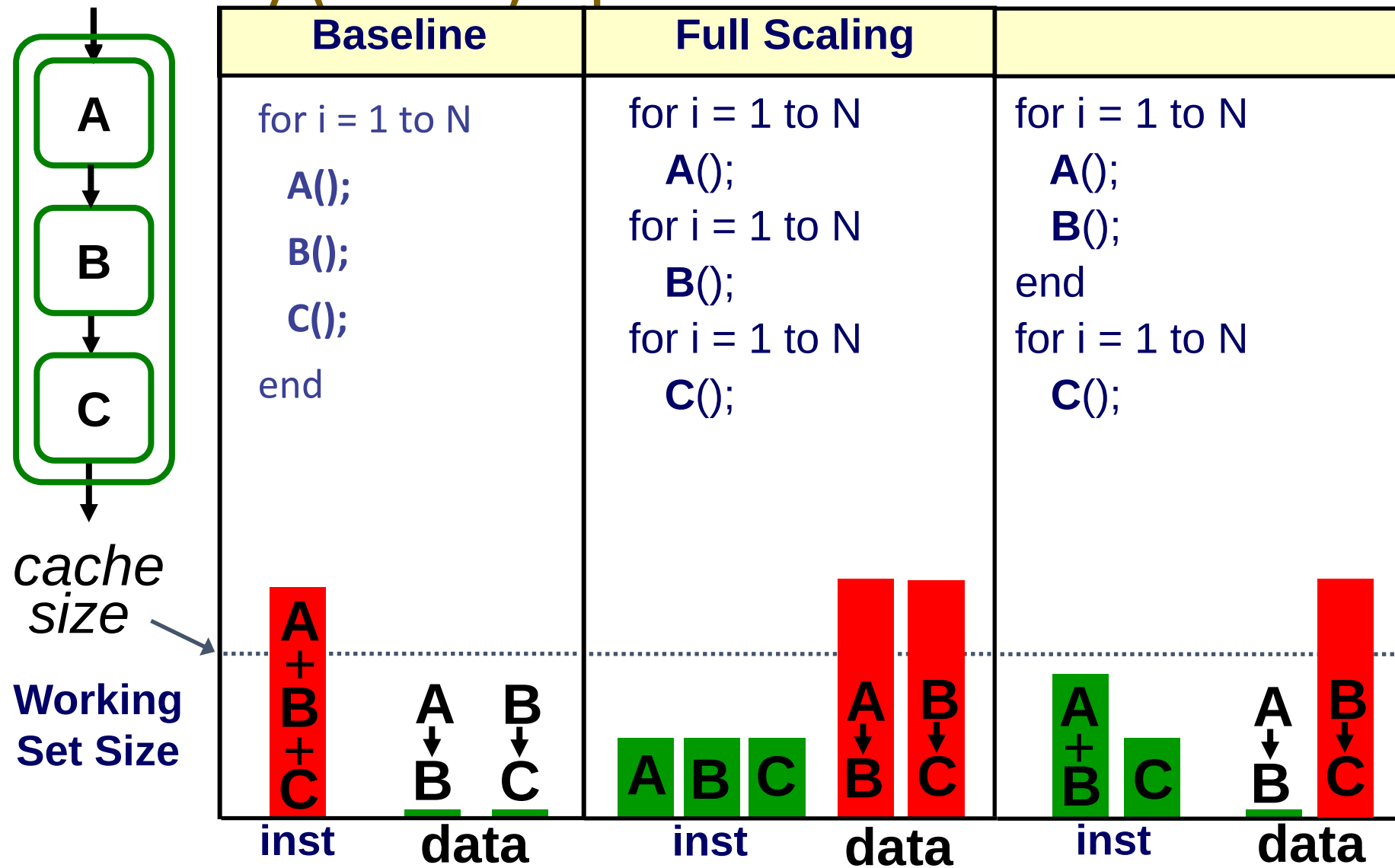
Working Set Size

Baseline	Full Scaling
<pre>for i = 1 to N A(); B(); C(); end</pre>	<pre>for i = 1 to N A(); for i = 1 to N B(); for i = 1 to N C();</pre>
A + B + C miss rate = 1	A B C miss rate = 1 / N

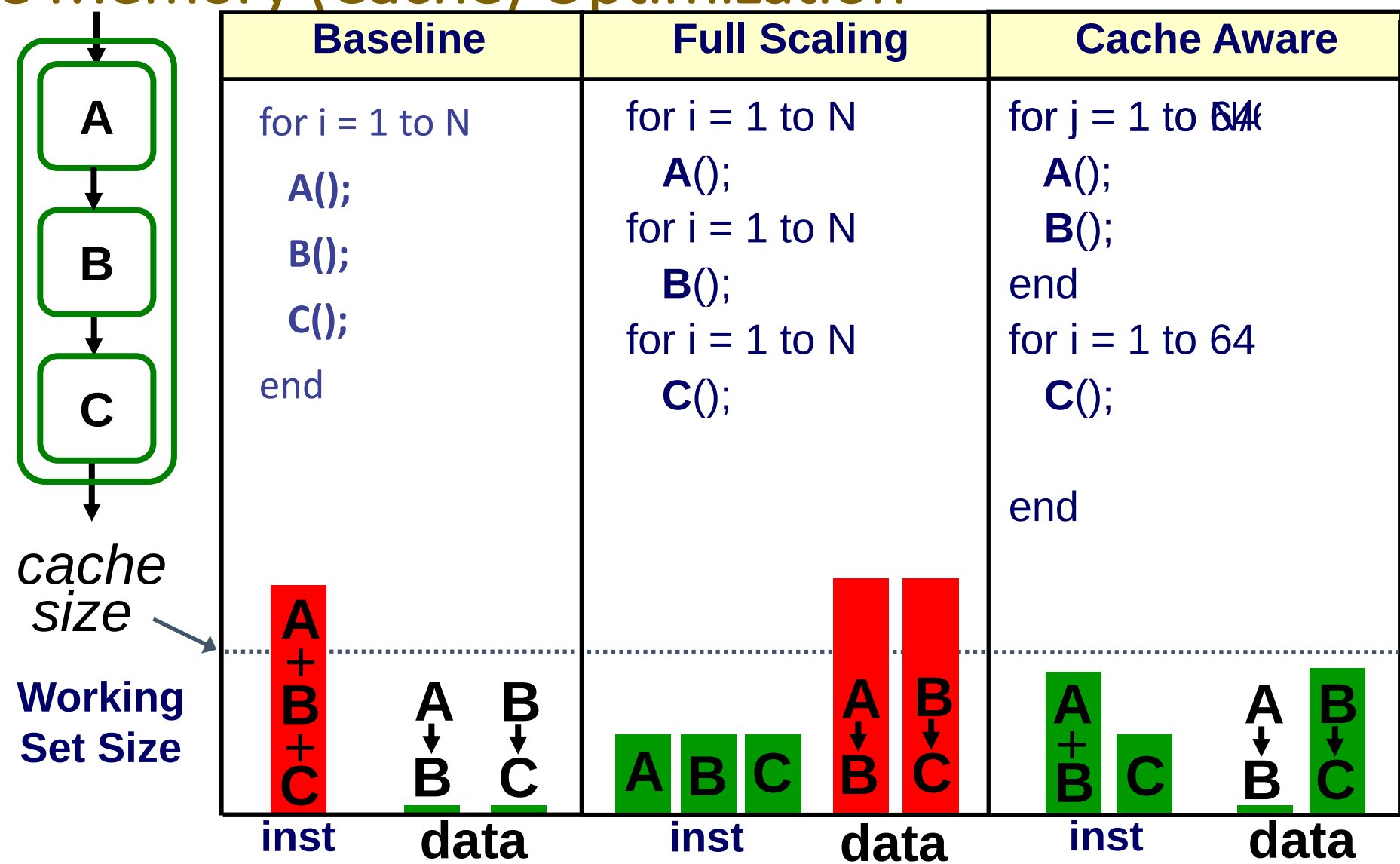
Example Memory (Cache) Optimization



Example Memory (Cache) Optimization



Example Memory (Cache) Optimization



Reuse and Locality

- Consider how data is accessed
 - Data reuse:
 - Same data used multiple times
 - Intrinsic in computation
 - Data locality:
 - Data is reused and is present in “fast memory”
 - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
 - Appropriate data placement and **layout**
 - **Code reordering** transformations

Temporal Reuse in Sequential Code

- Same data used in distinct iterations I and I'

```
for (i=1; i<N; i++)  
  for (j=1; j<N; j++)  
    A[j]= A[j]+A[j+1]+A[j-1]
```

- $A[j]$ has self-temporal reuse in loop i

Spatial Reuse

- Same data transfer (usually cache line) used in distinct iterations I and I'

```

for (i=1; i<N; i++)
    for (j=1; j<N; j++)
        A[j] = A[j] + A[j+1] + A[j-1];
    
```

- $A[j]$ has self-spatial reuse in loop j
- Multi-dimensional array note: C arrays are stored in row-major order

Cache Locality

A[1,1]	A[2,1]	...	A[1,2]	A[2,2]	...	A[1,3]	...
--------	--------	-----	--------	--------	-----	--------	-----

- Suppose array A has column-major layout

```

for i = 1, 100
  for j = 1, 200
    A[i, j] = A[i, j] + 3
  end_for
end_for
    
```

- Loop nest has poor spatial cache locality.

Loop Interchange

A[1,1]	A[2,1]	...	A[1,2]	A[2,2]	...	A[1,3]	...
--------	--------	-----	--------	--------	-----	--------	-----

- Suppose array A has column-major layout

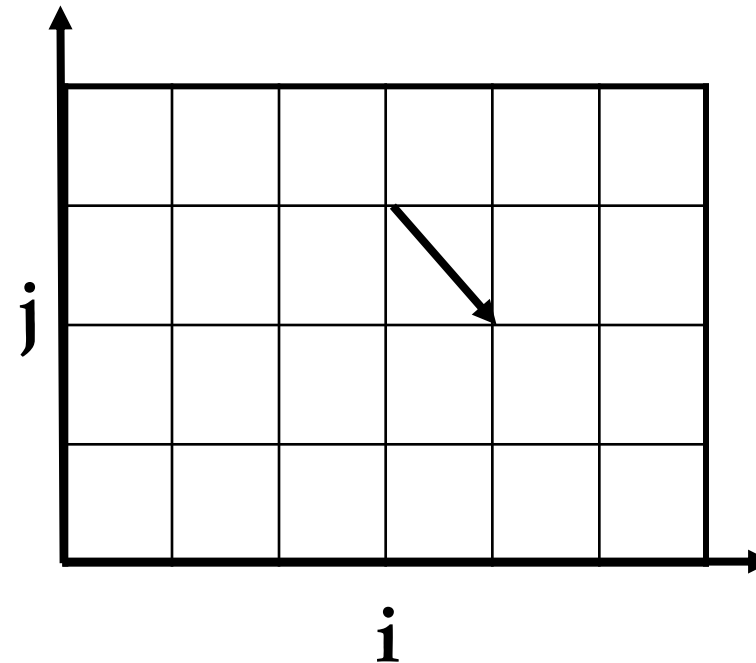
```
for i = 1, 100  
  for j = 1, 200  
    A[i, j] = A[i, j] + 3  
  end_for  
end_for
```

```
for j = 1, 200  
  for i = 1, 100  
    A[i, j] = A[i, j] + 3  
  end_for  
end_for
```

- New loop nest has better spatial cache locality.

Legality of Loop Interchange

```
for i = 2, 100
  for j = 1, 200
    A[i, j] = A[i-1, j+1]+3
  end_for
end_for
```



- e.g. dependence from (3,3) to (4,2)

Loop Fusion

```
for i = 1, 1000  
  A[i] = B[i] + 3  
end_for  
  
for j = 1, 1000  
  C[j] = A[j] + 5  
end_for
```

```
for i = 1, 1000  
  A[i] = B[i] + 3  
  C[i] = A[i] + 5  
end_for
```

- Better reuse between A[i] and A[i]

Loop Distribution

```
for i = 1, 1000  
  A[i] = A[i-1] + 3  
  C[i] = B[i] + 5  
end_for
```

```
for i = 1, 1000  
  A[i] = A[i-1] + 3  
end_for
```

```
for i = 1, 1000  
  C[i] = B[i] + 5  
end_for
```

- 2nd loop is parallel

Register Blocking

```
for j = 1, 2*m  
  for i = 1, 2*n  
    A[i, j] = A[i-1, j] +  
    A[i-1, j-1]  
  end_for  
end_for
```

```
for j = 1, 2*m, 2  
  for i = 1, 2*n, 2  
    A[i, j] = A[i-1, j] + A[i-1, j-1]  
    A[i, j+1] = A[i-1, j+1] + A[i-1, j]  
    A[i+1, j] = A[i, j] + A[i, j-1]  
    A[i+1, j+1] = A[i, j+1] + A[i, j]  
  end_for  
end_for
```

- Better reuse between $A[i, j]$ and $A[i, j]$

Virtual Register Allocation

```
for j = 1, 2*M, 2
  for i = 1, 2*N, 2
    r1 = A[i-1,j]
    r2 = r1 + A[i-1,j-1]
    A[i, j] = r2
    r3 = A[i-1,j+1] + r1
    A[i, j+1] = r3
    A[i+1, j] = r2 + A[i, j-1]
    A[i+1, j+1] = r3 + r2
  end_for
end_for
```

- Memory operations reduced to register load/store
- 8MN loads to 4MN loads

Scalar Replacement

```
for i = 2, N+1  
    = A[i-1]+1  
    A[i] =  
end_for
```

```
t1 = A[1]  
for i = 2, N+1  
    = t1 + 1  
    t1 =  
    A[i] = t1  
end_for
```

- Eliminate loads and stores for array references

Tiling

```
for (j=1; j<M; j++)  
  for (i=1; i<N; i++)  
    D[i] = D[i] + B[j][i];
```

Strip
mine

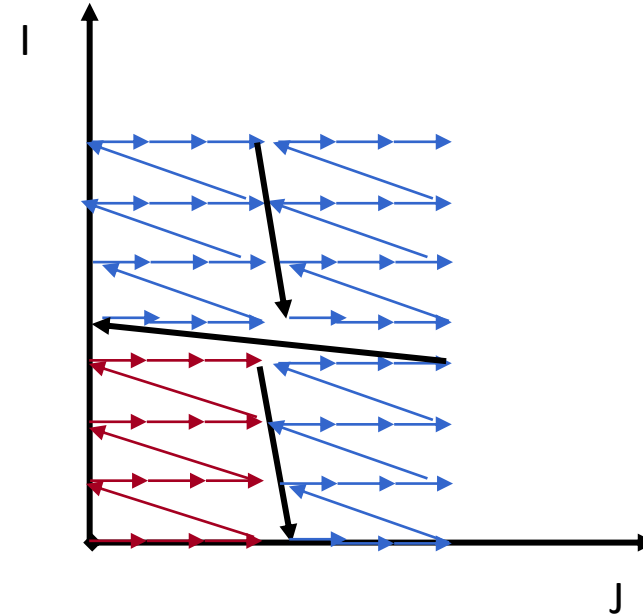
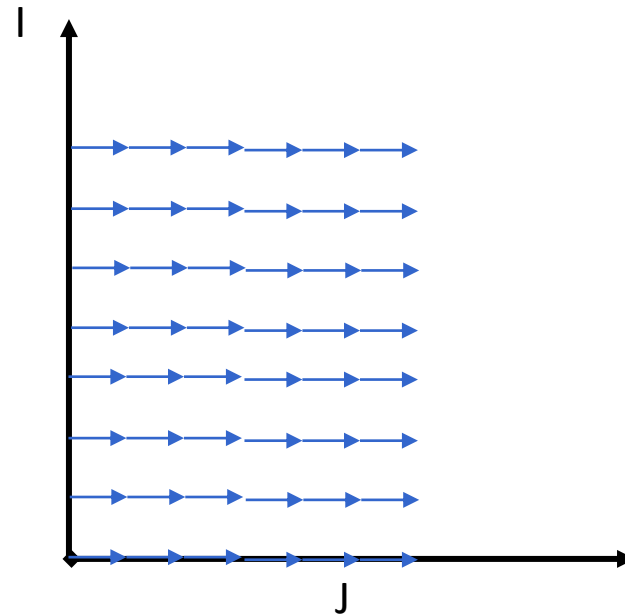
```
for (j=1; j<M; j++)  
  for (ii=1; ii<N; ii+=s)  
    for (i=ii; i<min(ii+s-1,N); i++)  
      D[i] = D[i] + B[j][i];
```

Permute

```
for (ii=1; ii<N; ii+=s)  
  for (j=1; j<M; j++)  
    for (i=ii; i<min(ii+s-1,N); i++)  
      D[i] = D[i] + B[j][i];
```

Tiling

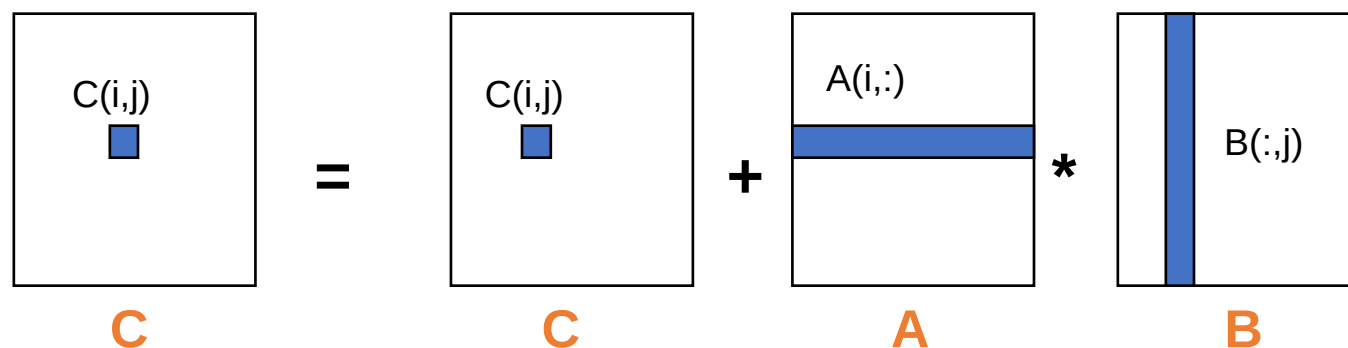
Tiling reorders loop iterations to bring iterations that reuse data closer in time



Remember – Naive Matrix Multiply

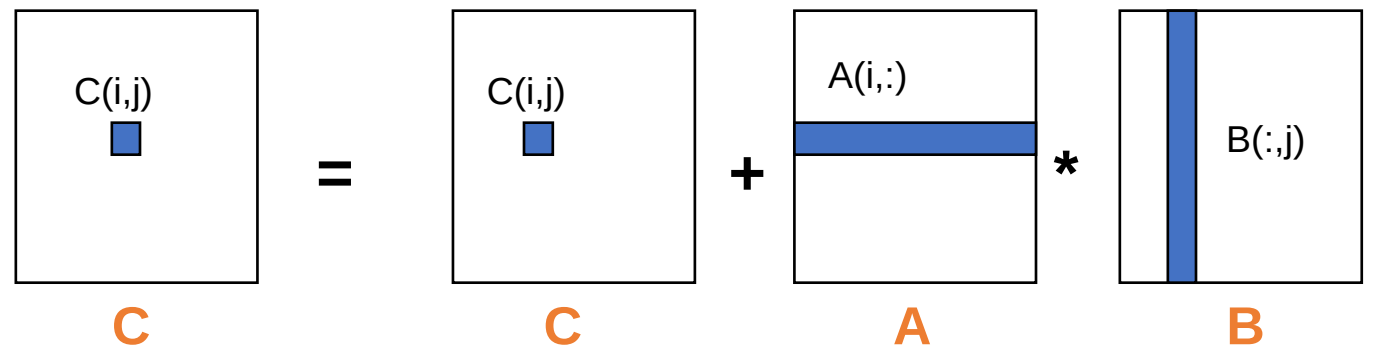
- Number of slow memory references on unblocked matrix multiply
- Algorithm has $2*n^3 = O(n^3)$ Flops and operates on $3*n^2$ words of memory
- Reuse quotient ($q = \text{flops/word}$) in the algorithm is *potentially* as large as

$$2*n^3 / 3*n^2 = O(n)$$



Naive Matrix Multiply – Insufficient Cache

```
{implements  $C = C + A * B$ }  
for i = 1 to n  
  {read row i of A into fast memory}  
  for j = 1 to n  
    {read  $C(i,j)$  into fast memory}  
    {read column j of B into fast memory}  
    for k = 1 to n  
       $C(i,j) = C(i,j) + A(i,k) * B(k,j)$   
    {write  $C(i,j)$  back to slow memory}
```



Naive Matrix Multiply

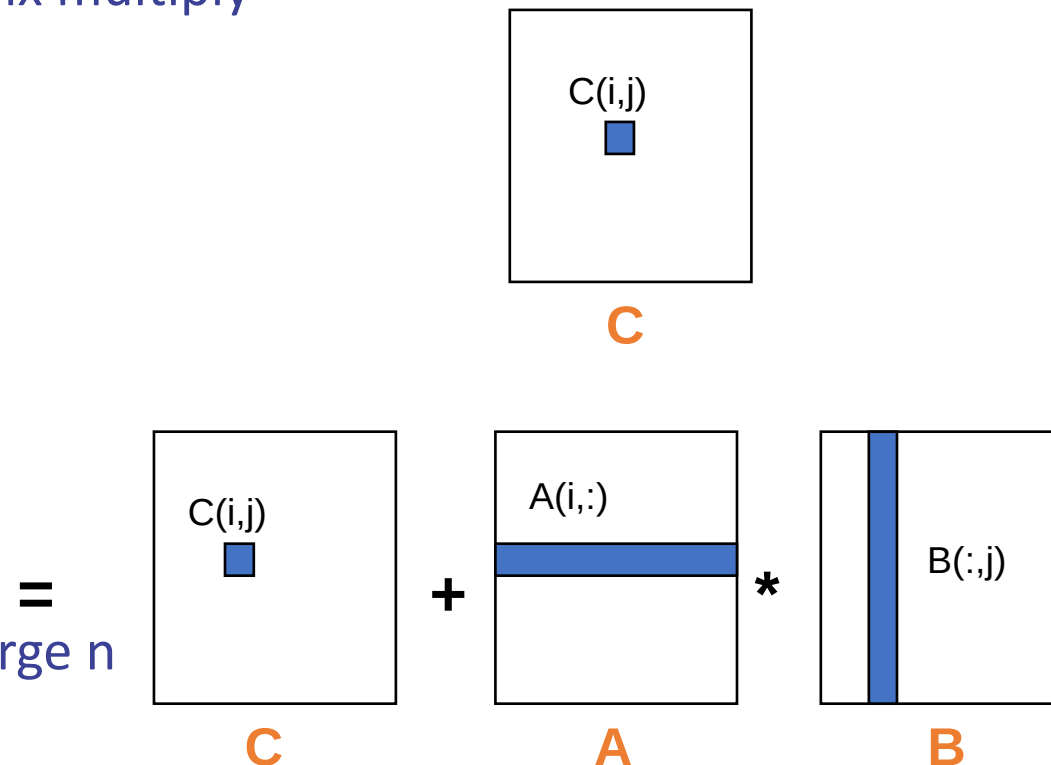
- Number of slow memory references on unblocked matrix multiply

$$\begin{aligned}
 m &= n^3 && \text{to read each column of B } n \text{ times} \\
 &+ n^2 && \text{to read each row of A once} \\
 &+ 2n^2 && \text{to read and write each element of C once} \\
 &= n^3 + 3n^2
 \end{aligned}$$

- So the re-use quotient, $q = f / m = 2n^3 / (n^3 + 3n^2)$

$$\sim 2 \text{ for large } n$$

- This is no better than matrix-vector multiply
- And is far from the “best possible” which is $2/3 * n$ for large n
- And this doesn't take into account cache lines



Blocked (Tiled) Matrix Multiplication

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where

$b = n / N$ is called the **block size**

for $i = 1$ to N

for $j = 1$ to N

{read block $C(i,j)$ into fast memory}

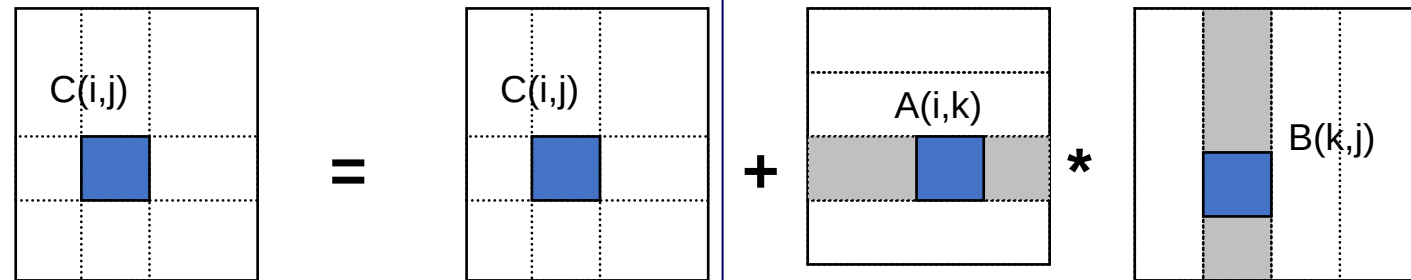
for $k = 1$ to N

{read block $A(i,k)$ into fast memory}

{read block $B(k,j)$ into fast memory}

$C(i,j) = C(i,j) + A(i,k) * B(k,j)$ {do a matrix multiply on blocks}

{write block $C(i,j)$ back to slow memory}



Blocked (Tiled) Matrix Multiplication

- Recall:

m : memory traffic

matrix has $n \times n$ elements

$N \times N$ blocks each of size $b \times b$
($N = n/b$)

f is number of floating point operations, $2n^3$ for this problem

$q = f / m$ is our measure of algorithm efficiency in the memory system

$$\begin{aligned}
 m &= N \cdot n^2 && \text{read each block of B } N^3 \text{ times } (N^3 \cdot b^2 = N^3 \cdot (n/N)^2 = N \cdot n^2) \\
 &+ N \cdot n^2 && \text{read each block of A } N^3 \text{ times} \\
 &+ 2n^2 && \text{read and write each block of C once} \\
 &= (2N + 2) \cdot n^2
 \end{aligned}$$

- Re-use quotient is: $q = f / m = 2n^3 / ((2N + 2) \cdot n^2) \sim n / N = b$ for large n
- We can improve performance by increasing the blocksize b
 - Can be much faster than matrix-vector multiply ($q=2$)

References

Grama, Ananth, Anshul Gupta, George Karypis, and Vipin Kumar. 2003. *Introduction to Parallel Computing*. 2nd ed. Boston: Addison-Wesley.

Sabancı Universitesi Thank you!



**Co-funded by
the European Union**



EuroHPC
Joint Undertaking



EuroCC 3 has received funding from the [European High-Performance Computing Joint Undertaking \(JU\)](#) under Grant Agreement No. 101306701. The JU receives support from the [European Union](#)'s Digital Europe Programme and Germany, Albania, Austria, Belgium, Bosnia and Herzegovina, Bulgaria, Croatia, Cyprus, Czechia, Denmark, Estonia, Finland, France, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, Montenegro, the Netherlands, North Macedonia, Norway, Poland, Portugal, Romania, Serbia, Slovakia, Slovenia, Spain, Sweden, Türkiye, and Kosovo.

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or EuroHPC Joint Undertaking. Neither the European Union nor the EuroHPC Joint Undertaking can be held responsible for them.