



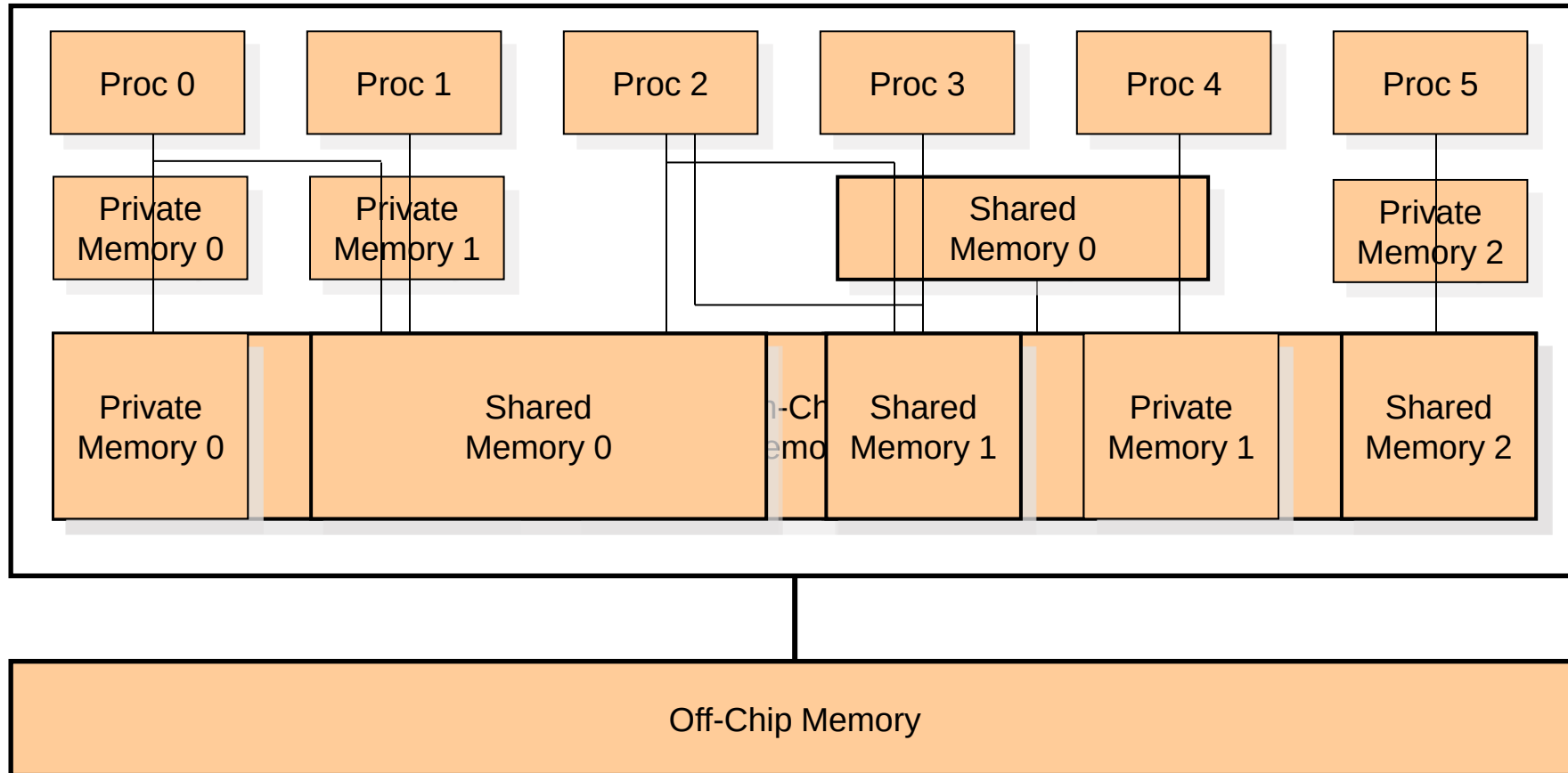
Memory System Introduction

Ozcan Ozturk – Sabancı University

Limitations of Memory System Performance

- Memory system, and not processor speed, is often the bottleneck for many applications.
- Memory system performance is largely captured by two parameters
 - Latency
 - Bandwidth
- **Latency** is the time from the issue of a memory request to the time the data is available at the processor.
- **Bandwidth** is the rate at which data can be pumped to the processor by the memory system.

Memory System



Latency vs. Bandwidth

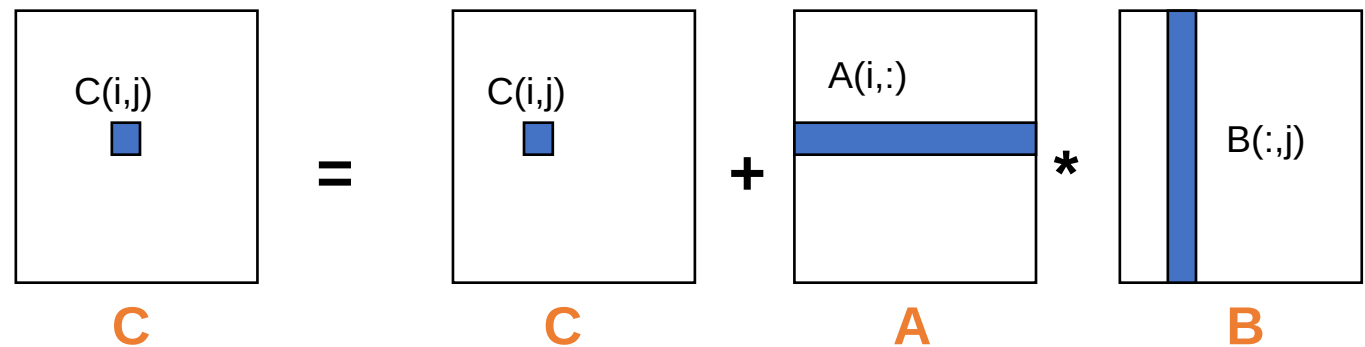
- Consider the example of a fire-hose.
- If the water comes out of the hose two seconds after the hydrant is turned on, the latency of the system is two seconds.
- Once the water starts flowing, if the hydrant delivers water at the rate of 5 gallons/second, the bandwidth of the system is 5 gallons/second.
- If you want immediate response from the hydrant, it is important to reduce latency.
- If you want to fight big fires, you want high bandwidth.

Improving Effective Memory Latency Using Caches

- Caches are small and fast memory elements between the processor and DRAM.
- This memory acts as a low-latency high-bandwidth storage.
- If a piece of data is **repeatedly** used, the effective latency of this memory system can be reduced by the cache.
- The fraction of data references satisfied by the cache is called the **cache hit** ratio of the computation on the system.
- Cache hit ratio achieved by a code on a memory system often determines its performance.

Naive Matrix Multiply

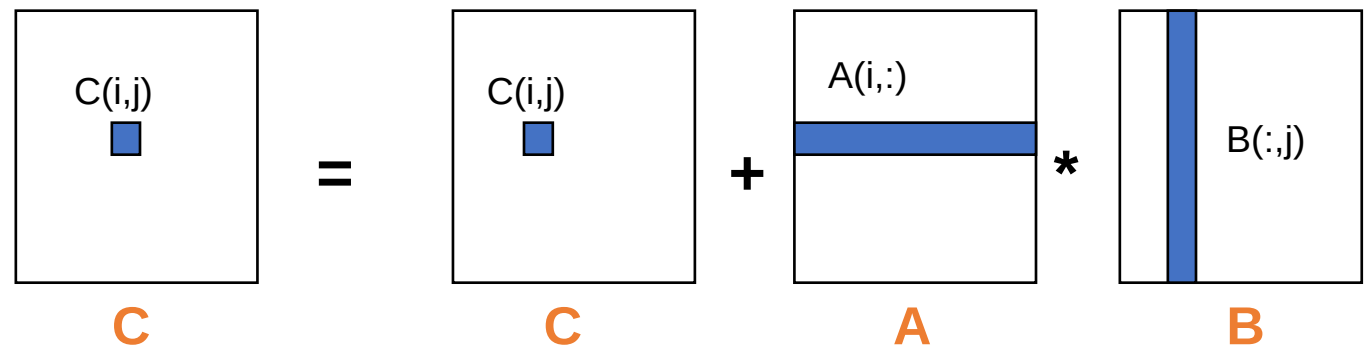
```
{implements  $C = C + A * B$ }  
{read A into fast memory}  
{read B into fast memory}  
{read C into fast memory}  
for i = 1 to n  
  for j = 1 to n  
    for k = 1 to n  
       $C(i,j) = C(i,j) + A(i,k) * B(k,j)$ 
```



Naive Matrix Multiply

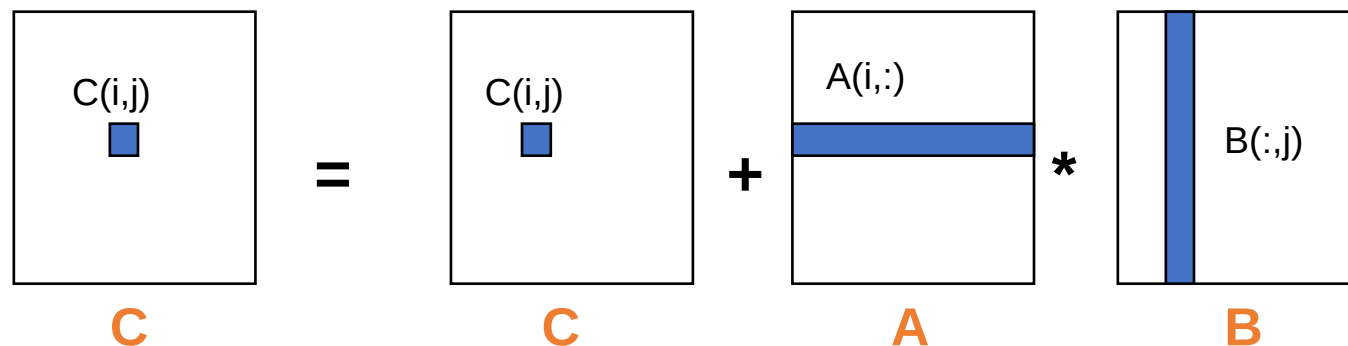
- Number of slow memory references on unblocked matrix multiply
- Algorithm has $2*n^3 = O(n^3)$ Flops and operates on $3*n^2$ words of memory
- Reuse quotient ($q = \text{flops/word}$) in the algorithm is *potentially* as large as

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



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

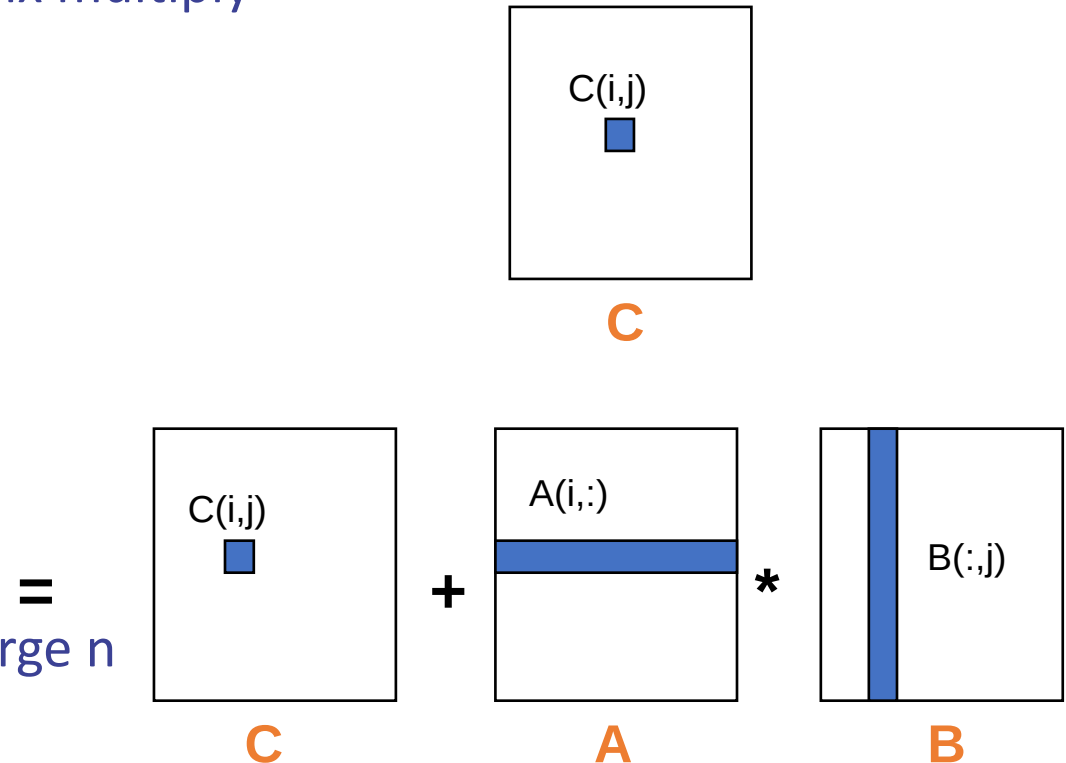
- Number of slow memory references on unblocked matrix multiply

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

- 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



Impact of Caches

- Repeated references to the same data item correspond to **temporal locality**.
- In our example, we had $O(n^2)$ data accesses and $O(n^3)$ computation.
 - This asymptotic difference makes the above example particularly desirable for caches.
- **Data reuse** is critical for cache performance.

Impact of Memory Bandwidth: Example

- Consider the same setup as before, except in this case, the block size is 4 words instead of 1 word.
 - Matrices are laid out linearly in memory
 - A single memory access fetches four consecutive words in the vector
- Assuming also that the processor has two multiply-add units
 - Capable of executing four instructions in each cycle
- Potentially 4x improvement in execution time

Impact of Memory Bandwidth: Example

Consider the following code fragment:

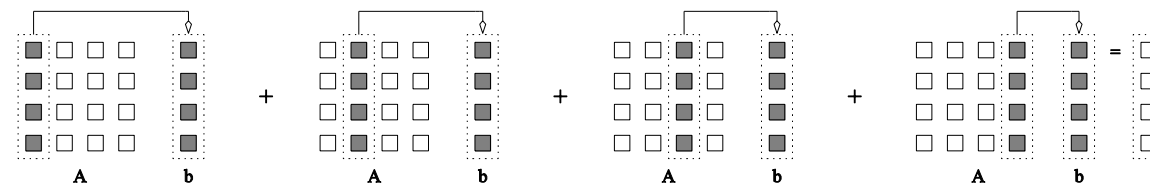
```
for (i = 0; i < 1000; i++)  
    column_sum[i] = 0.0;  
for (j = 0; j < 1000; j++)  
    column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix `b` into a vector `column_sum`.

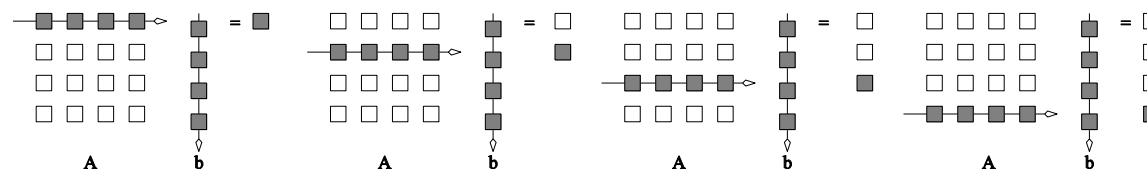
Impact of Memory Bandwidth: Example

- The vector `column_sum` is small and easily fits into the cache
- The matrix `b` is accessed in a column order.
- The strided access results in very poor performance.
- Multiplying a matrix with a vector:

Column major data access → multiplying column-by-column, keeping a running sum



Row major data access → computing the result as a dot product of a row of the matrix with the vector



Impact of Memory Bandwidth: Example

We can fix the above code as follows:

```
for (i = 0; i < 1000; i++)  
    column_sum[i] = 0.0;  
for (j = 0; j < 1000; j++)  
    for (i = 0; i < 1000; i++)  
        column_sum[i] += b[j][i];
```

In this case, the matrix is traversed in a **row-order** and performance can be expected to be significantly better.

Alternate Approaches for Hiding Memory Latency

- Consider the problem of browsing the web on a very slow network connection. We deal with the problem in one of three possible ways:
 1. we anticipate which pages we are going to browse ahead of time and issue requests for them in advance;
 2. we open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others; or
 3. we access a whole bunch of pages in one go - amortizing the latency across various accesses.

Multithreading for Latency Hiding

A thread is a single stream of control in the flow of a program.

We illustrate threads with a simple example:

```
for (i = 0; i < n; i++)  
    c[i] = dot_product(get_row(a, i), b);
```

Each dot-product is independent of the other, and therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

```
for (i = 0; i < n; i++)  
    c[i] = create_thread(dot_product, get_row(a, i), b);
```

Prefetching for Latency Hiding

- Misses on loads cause programs to stall.
- Why not advance the loads so that by the time the data is actually needed, it is already there!
- The only drawback is that you might need more space to store advanced loads.
- However, if the advanced loads are overwritten, we are no worse than before!

Tradeoffs of Multithreading and Prefetching

- Bandwidth requirements of a multithreaded system may increase very significantly because of the smaller cache residency of each thread.
- Multithreaded systems become bandwidth bound instead of latency bound.
- Multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem.
- Multithreading and prefetching also require significantly more hardware resources in the form of storage.

Memory System Performance: Summary

- Exploiting spatial and temporal locality in applications is critical for amortizing memory latency and increasing effective memory bandwidth.
- The ratio of the number of operations to number of memory accesses is a good indicator of anticipated tolerance to memory bandwidth.
- Memory layouts and organizing computation appropriately can make a significant impact on the spatial and temporal locality.

Next Lecture

- Memory Architecture and Coherency

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.