CASE SCHOOL
OF ENGINEERING
CASE WESTERN RESERVE
UNIVERSITY

CSDS 451: Designing High Performant Systems for AI

Lecture 3

9/2/2025

Sanmukh Kuppannagari

sanmukh.kuppannagari@case.edu

https://sanmukh.research.st/

Case Western Reserve University

Outline

- Processor Memory Architectures
  - Memory Optimizations to Improve Performance
  - Processor Memory Architectures with Cache
  - Blocked Matrix Multiplication

Announcements

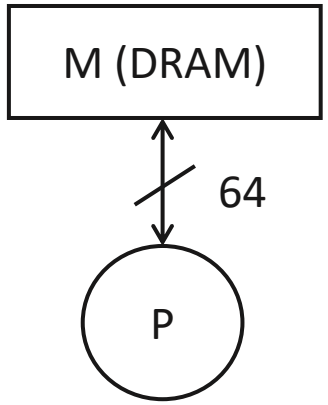
- PA 0 due this Thursday
- WA 1 will be out Thursday
  - Will cover topics till lecture 6
- TA Information and Office hours will be posted by tonight

Outline

- Processor Memory Architectures
  - Memory Optimizations to Improve Performance
  - Processor Memory Architectures with Cache
  - Blocked Matrix Multiplication
- Modeling GPU Architectures

Recall: Memory System

- System performance (execution time) depends on the "rate" at which data can be accessed by the program



Updated Execution Model

Latency: Time to receive the first word since the fetch instruction

Bandwidth: Rate of data transfer = bitwidth of the interconnect x frequency

Execute – 1 cycle
Fetch – Latency + Data size/bandwidth cycles

With DRAM access latency of 100s of cycles,
Need latency hiding mechanisms

Recall: System Performance Metrics

Actual System Performance

Latency Hiding Mechanisms

Efficient Bandwidth Utilization

0.048 Gflops/s

0.4 Gflops/s

2 Gflops/s

Sustained Worst Case
- Latency dominant

Sustained Best Case
- Latency Hidden
- Memory bandwidth dominant

Peak Performance
- Compute Only

System Performance Metrics

- If actual performance less than Sustained Performance (best case)
  - Latency not fully hidden – employ optimizations to hide latency
  - Memory bandwidth not fully utilized – employ optimizations to improve bandwidth

- If actual performance more than Sustained Performance (best case) but less than Peak Performance
  - Latency is hidden, memory bandwidth is fully utilized
  - Need to perform more computations on the fetched data (may or may not be possible depending upon the algorithm)

Techniques to Improve Application Performance

- Compute Parallelization
  - Decompose tasks to enable concurrent processing (we will discuss later in this course)

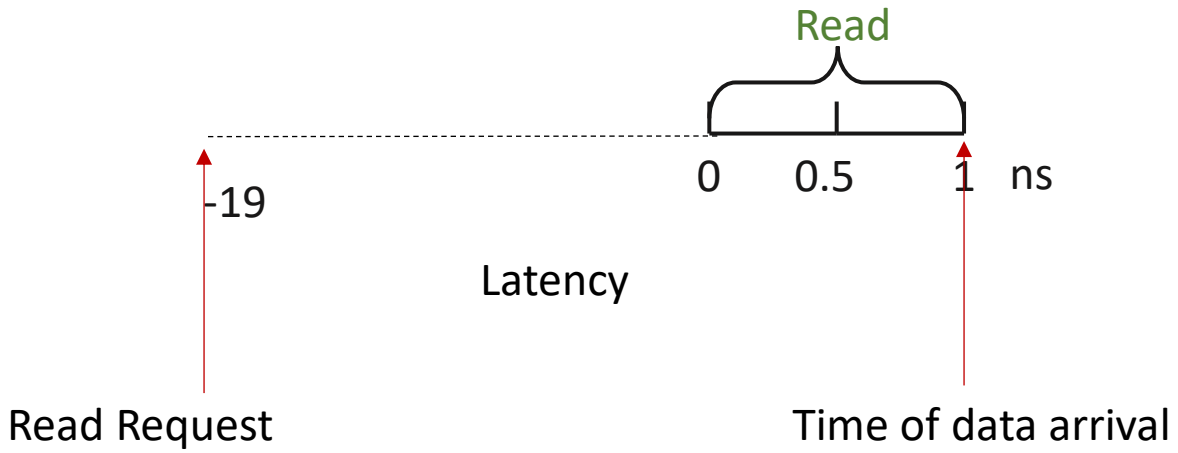
- Memory Optimizations
  - Optimizations addressing latency
  - Optimizations targeted toward maximizing bandwidth utilization
  - Optimizations that maximize computations on the fetched data (in other words, reduce the amount of data transfers that need to occur)

Learning Objectives

- Understand the optimizations addressing latency
- Understand the optimizations targeted toward maximizing bandwidth utilization
- Understand optimizations that reduce the amount of data transfer and maximize computations

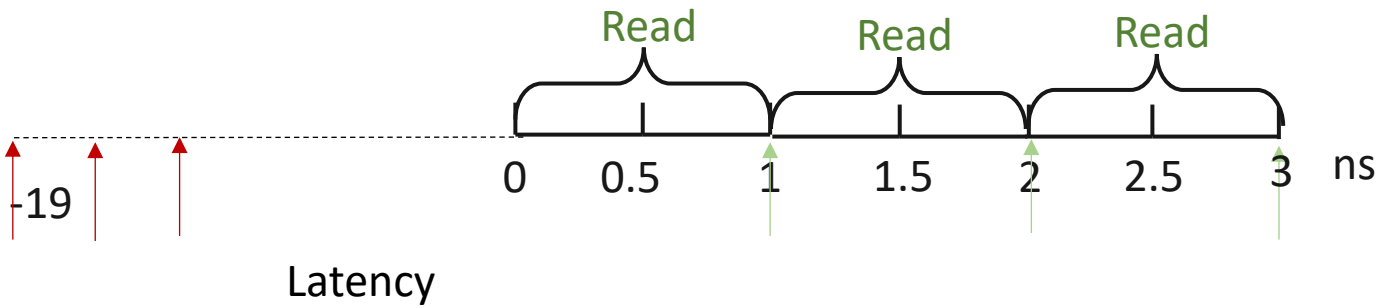
#1 Optimizations Targeting Latency

- Key ideas
  - Latency of a single data item cannot be reduced due to physical constraints
  - Try to hide the latency by doing some independent useful work such as
    - Data transfer
    - computations



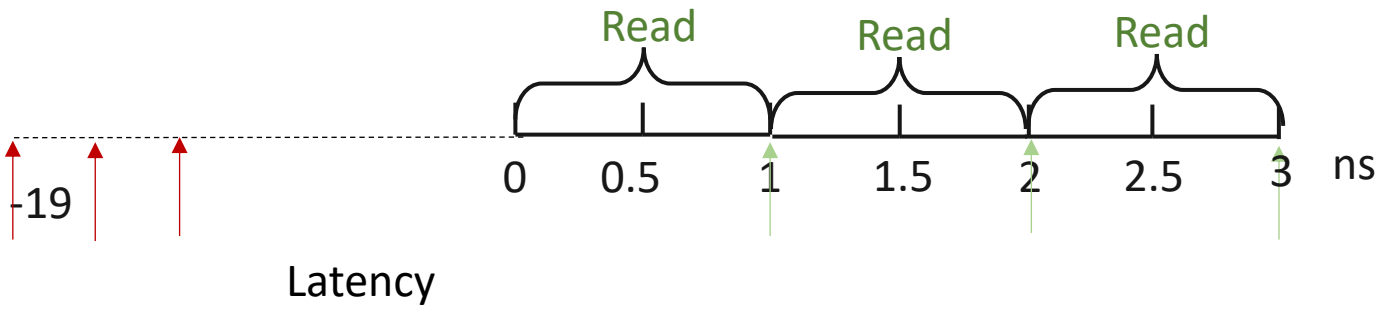
#1 Optimizations Targeting Latency

- Latency Hiding using data transfers
  - Avoid per data latency (Sustained Performance (worst case)) by multiple requests in sequence
  - After an initial latency, data arrives continuously



#1 Optimizations Targeting Latency

- Latency hiding using data transfers
  - In practice, consecutive requests to random locations leads to addition memory device specific latencies
  - Accessing sequential data leads to higher performance

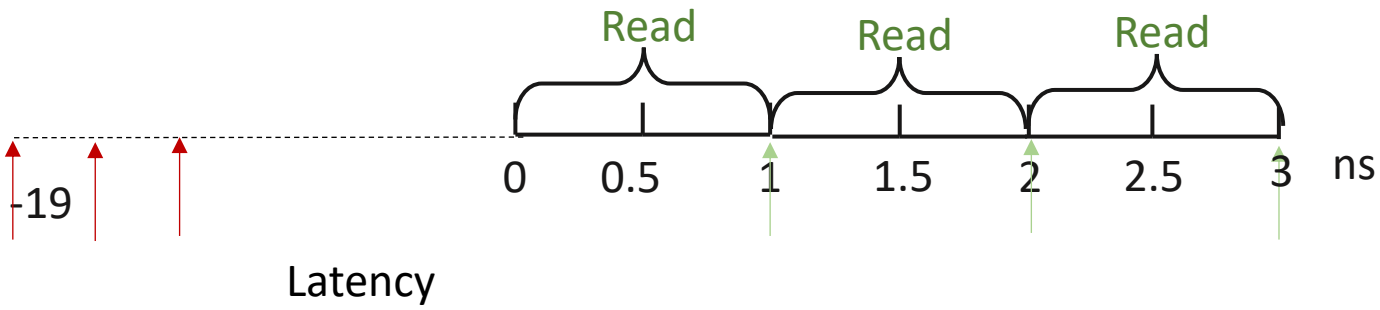


Read Request

Times of data arrival

#1 Optimizations Targeting Latency

- Latency hiding using data transfers
  - **Key takeaway for this course:** data transfers of sequential datastructures (such as arrays) leads to streaming data transfers, i.e., data transfers where latencies for the subsequent data items is hidden.



Recall last class: Fetch request format: read 100 elements starting at location 0.

Read Request

Times of data arrival

#1 Optimizations Targeting Latency

- Latency hiding using computation
- Several mechanisms exist
  - Prefetching
  - Thread switching
  - …
- Mostly provided by the hardware/operating system
- Prefetching is provided as an option to algorithm developer – we will revisit later in the class

#1 Optimizations Targeting Latency

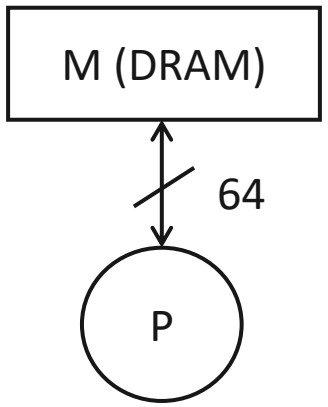
- Optimizations that you can use in your algorithms

1. Store the data in arrays in a sequential manner, in the order in which the algorithm is expected to access it (**Data layout optimizations that we will discuss later in today's lecture**)
2. Prefetch the data into the local memory before the computations begin (**Will discuss later today and in GPU programming lectures**)

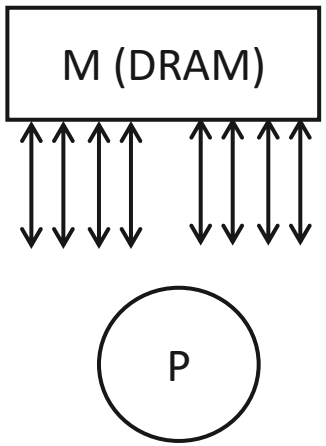
Requires Cache → a local on-chip memory where data can be stored till all the computations are performed

#2 Optimizations to Improve Bandwidth

- Key Idea
  - Memory systems have wide “buses” – bunch of wires that transfer data from memory to processors
  - Bandwidth is essentially buswidth X clock frequency
  - Pack the bus as much as possible in each cycle to transfer more data

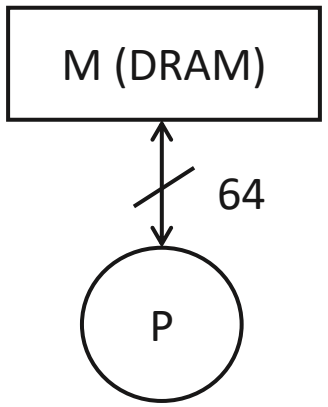


Should pack 64 bits in each cycle to maximize bandwidth

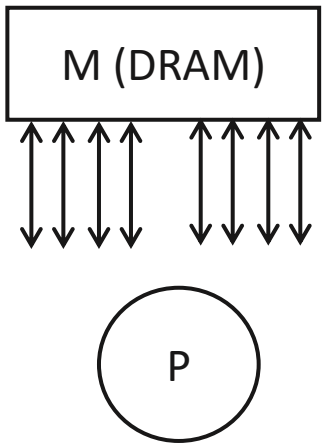


#2 Optimizations to Improve Bandwidth

- In Practice
  - Accessing sequential data leads to better packing
  - Memories usually have much larger bitwidth than 64 bits
    - Nvidia A100's memory has 5120 bit (615 bytes) wide bus.



Should pack 64 bits in each cycle to maximize bandwidth



#2 Optimizations to Improve Bandwidth

- Optimizations that you can use in your algorithms

1. Store the data in arrays in a sequential manner (again, data layout optimizations like we will discuss later in this lecture)
2. When fetching, try to fetch as much data as possible in a single go (again, prefetching that we will discuss in GPU programming lectures)

Requires Cache → a local on-chip memory where data can be stored till all the computations are performed

Memory Optimizations #1 and #2

- Similar strategies
- Should help us achieve Sustained Performance (best case)
  - Theoretically. In practice, getting 0 latency or 100% bandwidth utilization is very difficult, so performance may still be lower.
- Now the next step to improve performance further will be to maximize the computations performed on the fetched data
  - We will call this – optimizations to maximize data reuse (we *reuse* fetched *data* for computations as much as possible)
  - Maximizing data reuse implies reducing data transfers

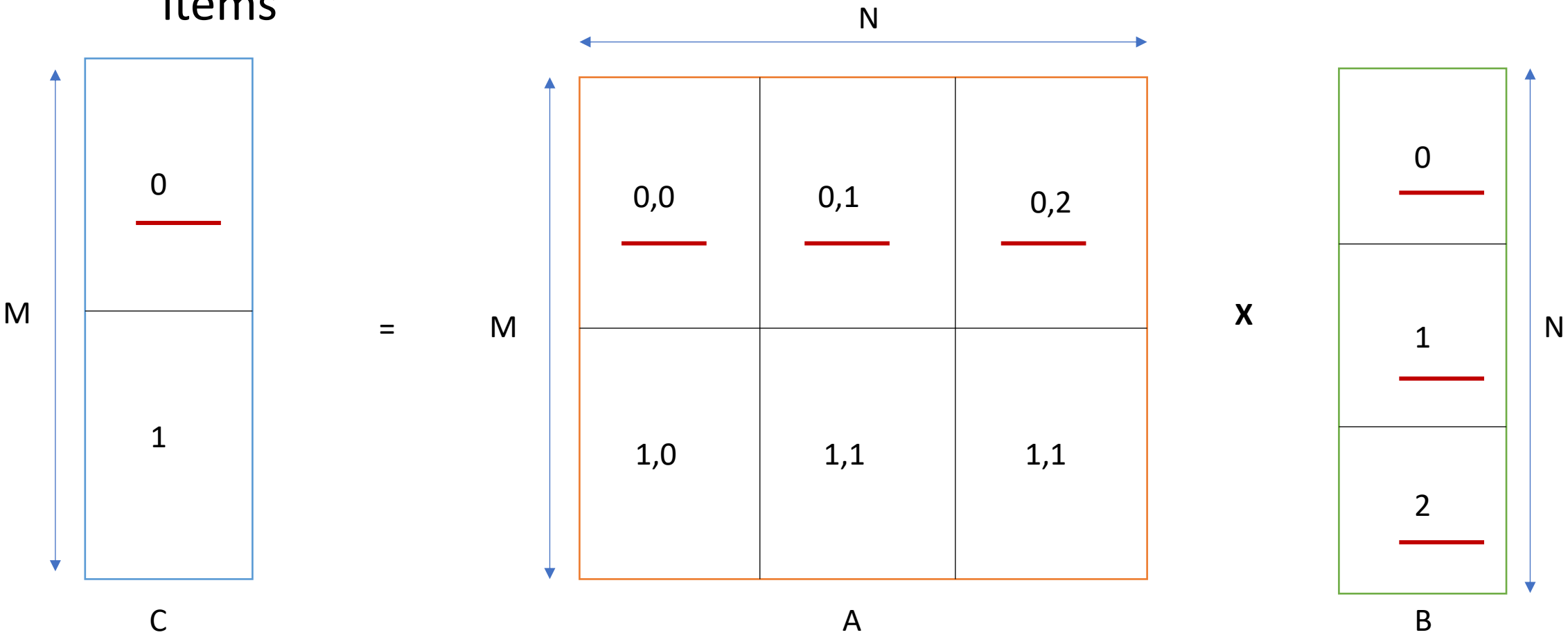
#3 Optimizations to Maximize Data Reuse

- Objective is to maximize the computations that can be performed on the data once fetched

- Most of the memory optimizations employed in application acceleration research fall under this category

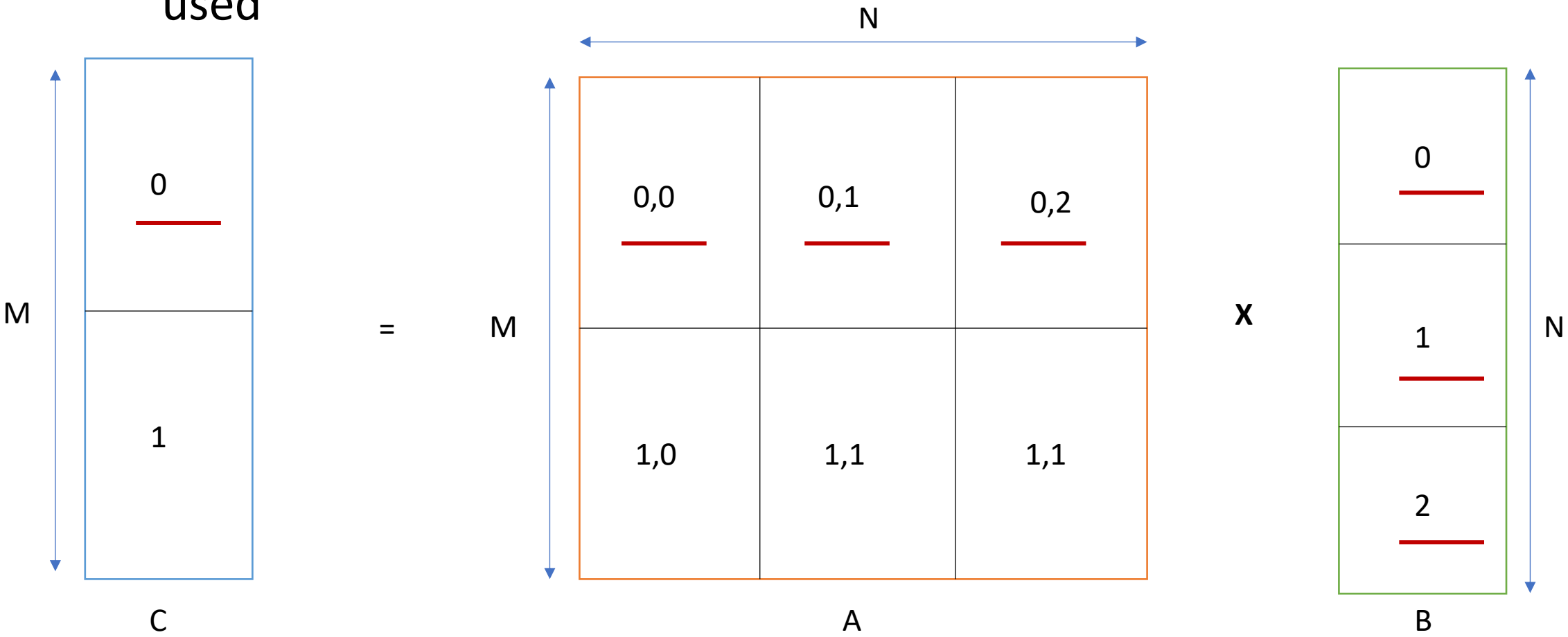
#3 Optimizations to Maximize Data Reuse

- Algorithms typically perform multiple operations on various data items

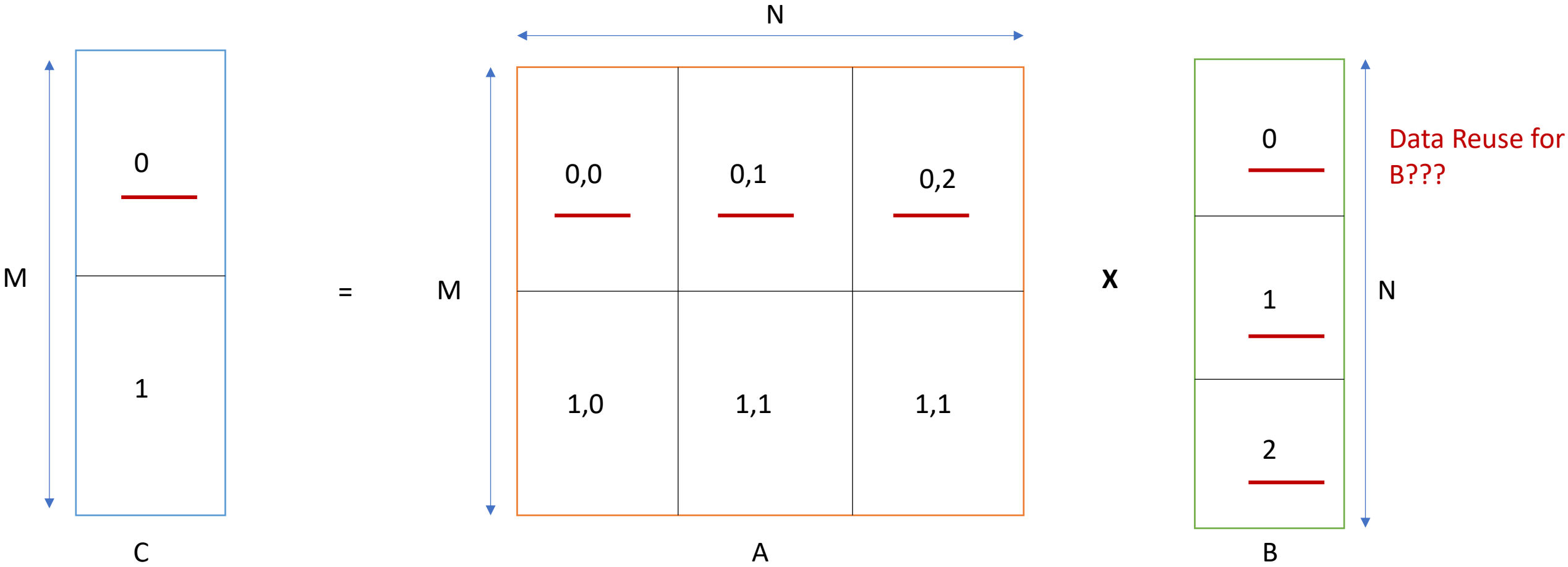


#3 Optimizations to Maximize Data Reuse

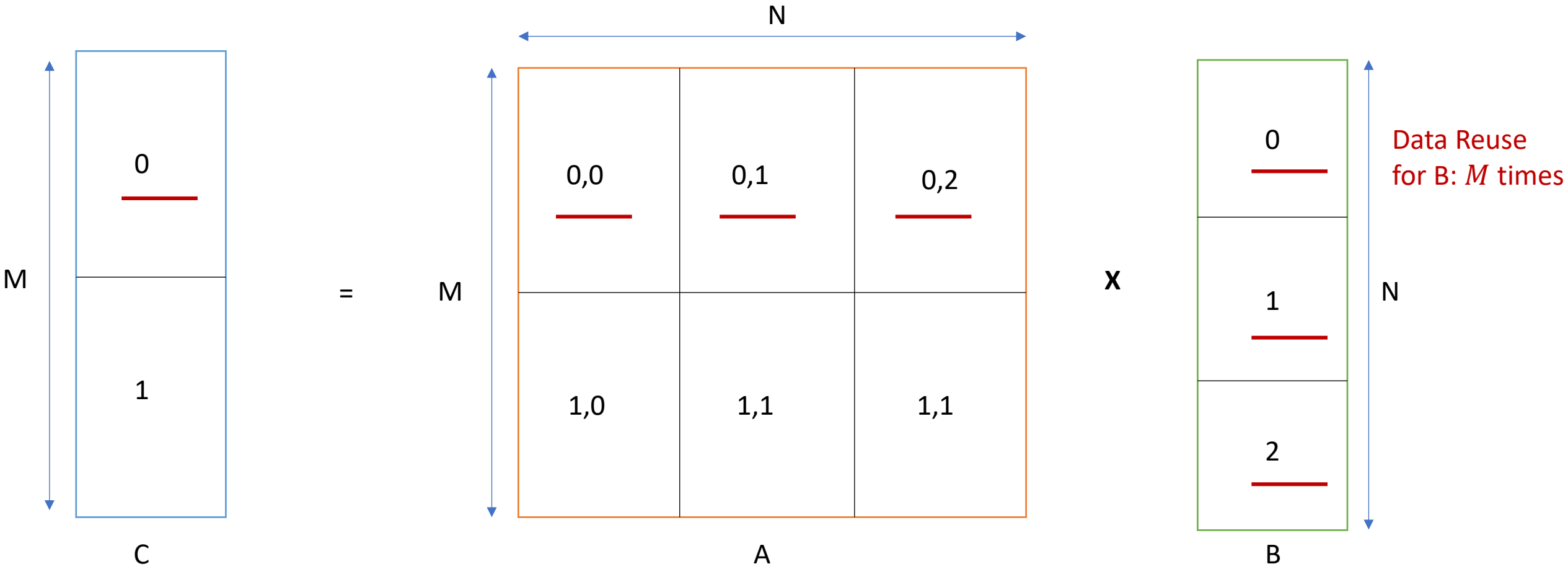
- Data Reuse: How many times (on average) an input/output location is used



#3 Optimizations to Maximize Data Reuse

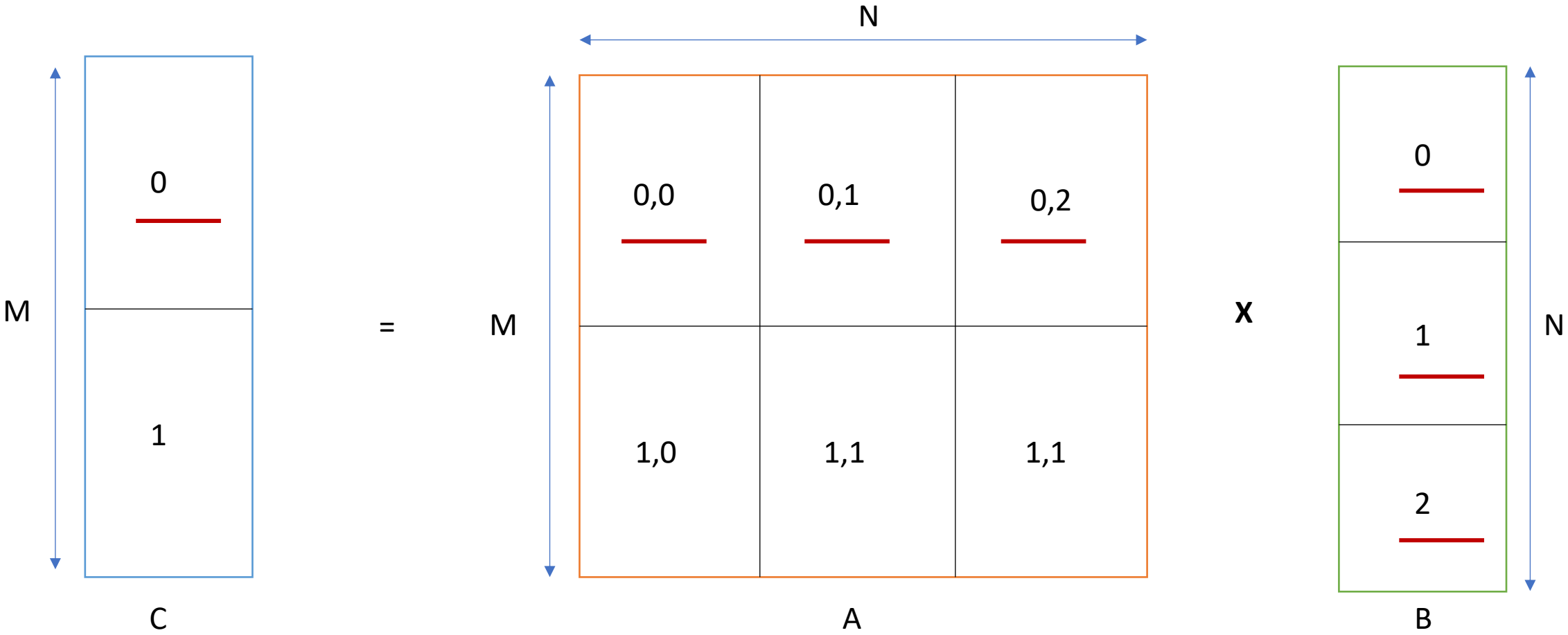


#3 Optimizations to Maximize Data Reuse



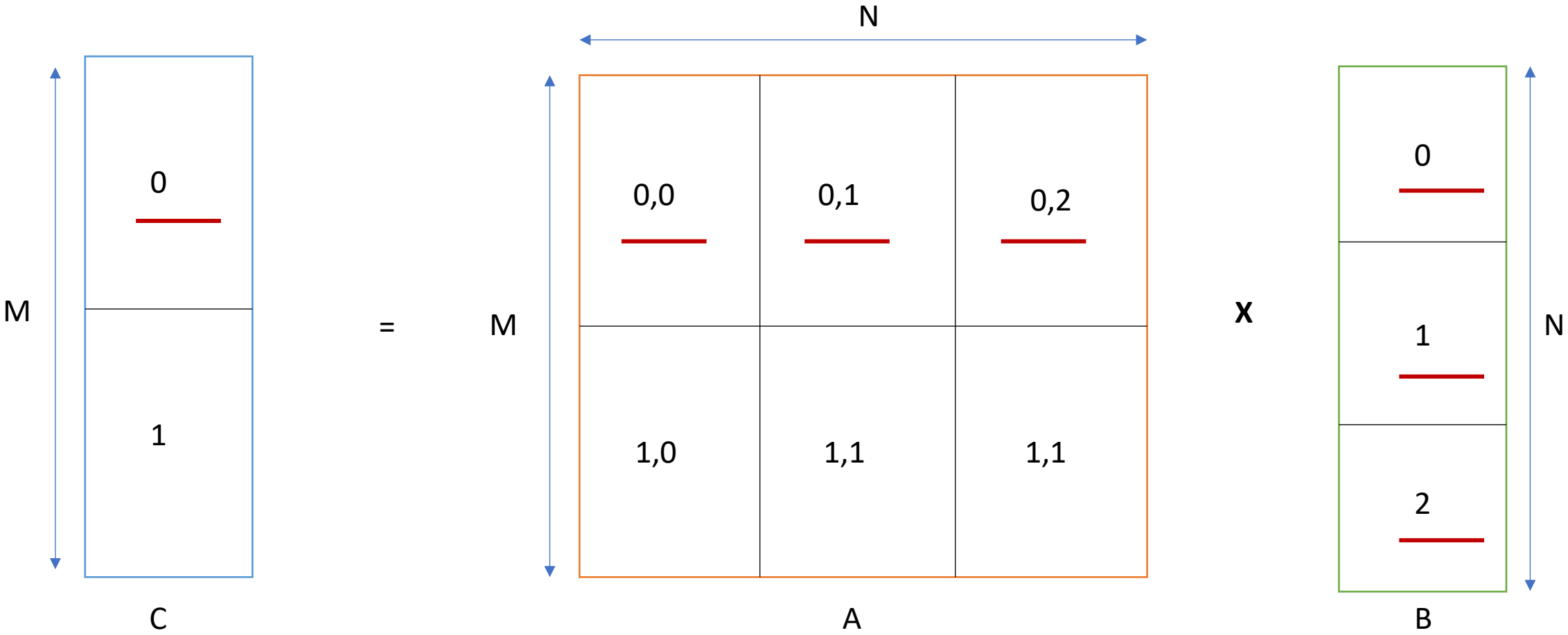
#3 Optimizations to Maximize Data Reuse

Data Reuse for A???



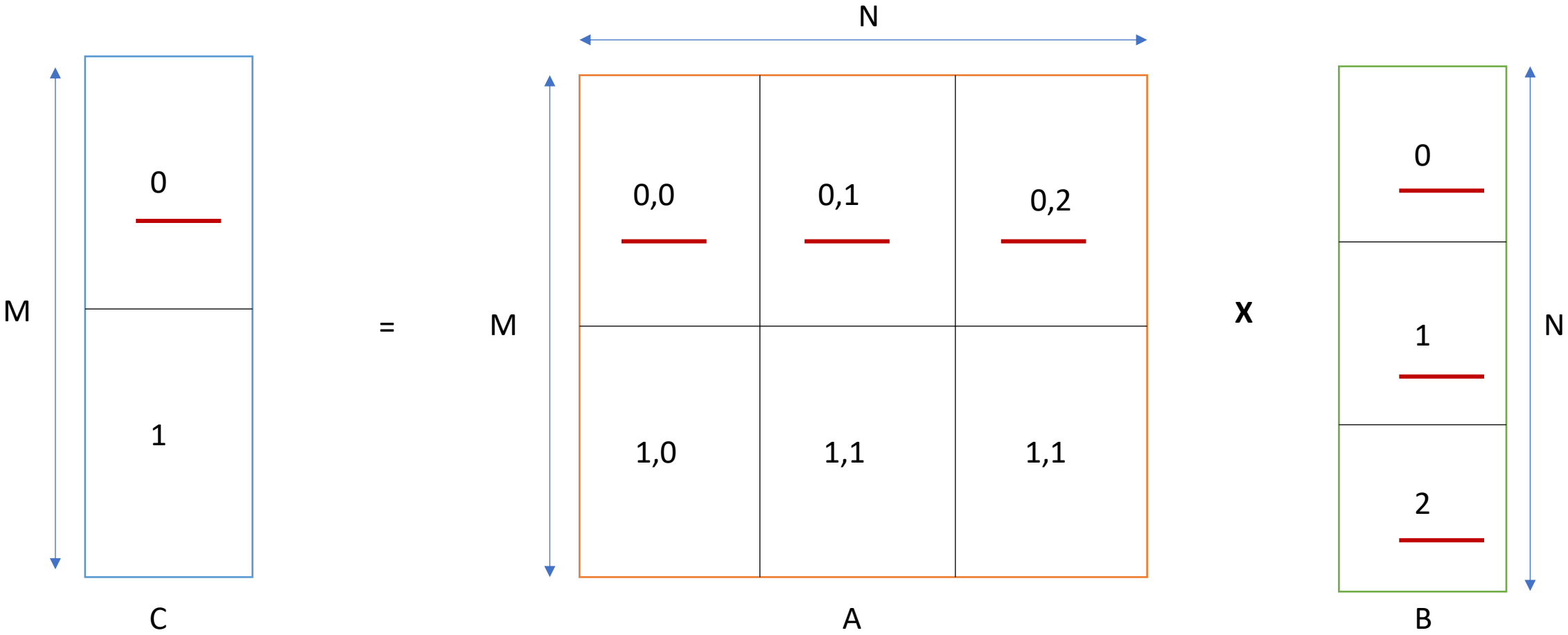
#3 Optimizations to Maximize Data Reuse

Data Reuse
for A: 1 time



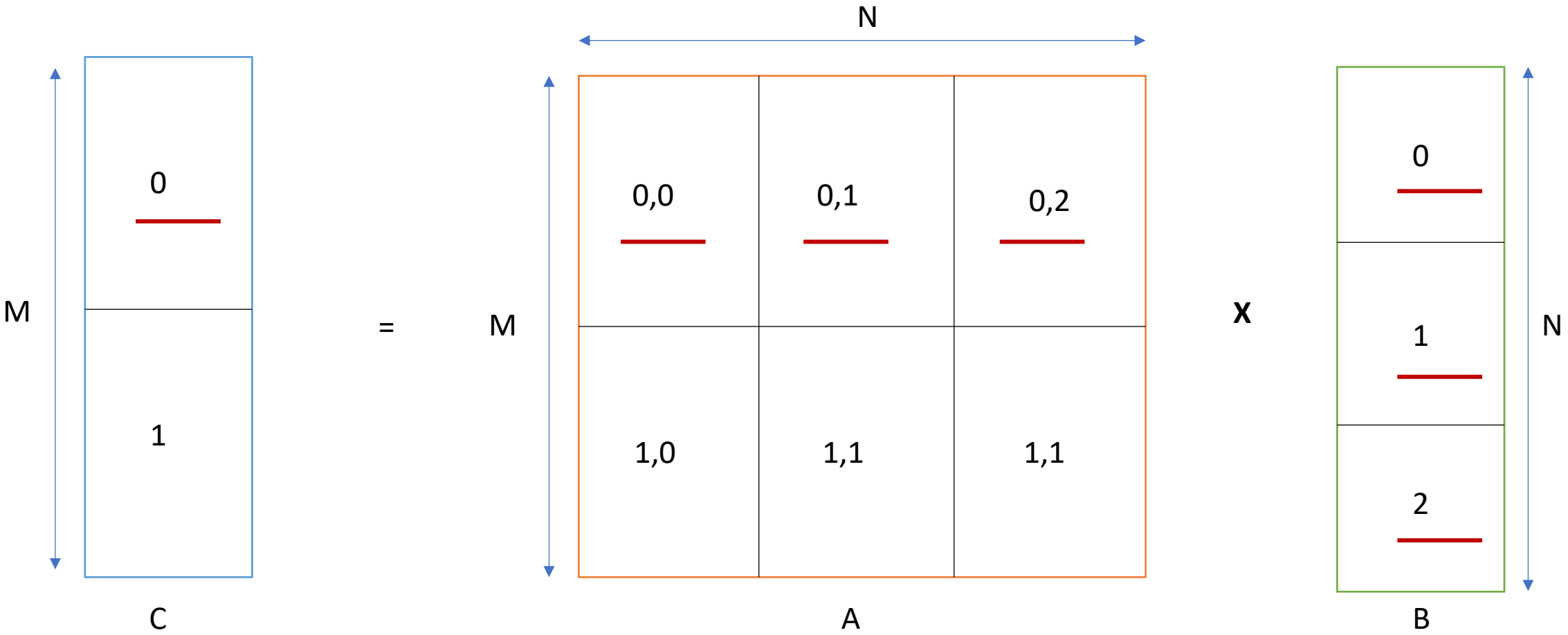
#3 Optimizations to Maximize Data Reuse

Data Reuse for
A+B+C???



#3 Optimizations to Maximize Data Reuse

Data Reuse for A+B+C : $\frac{MN\times1+N\times M+M\times N}{MN+M+N}=\frac{3MN}{MN+M+N}\sim O(1);\ M,N\ll MN$



#3 Optimizations to Maximize Data Reuse

- How do we make sure that each data item of B is fetched just once and used M times?

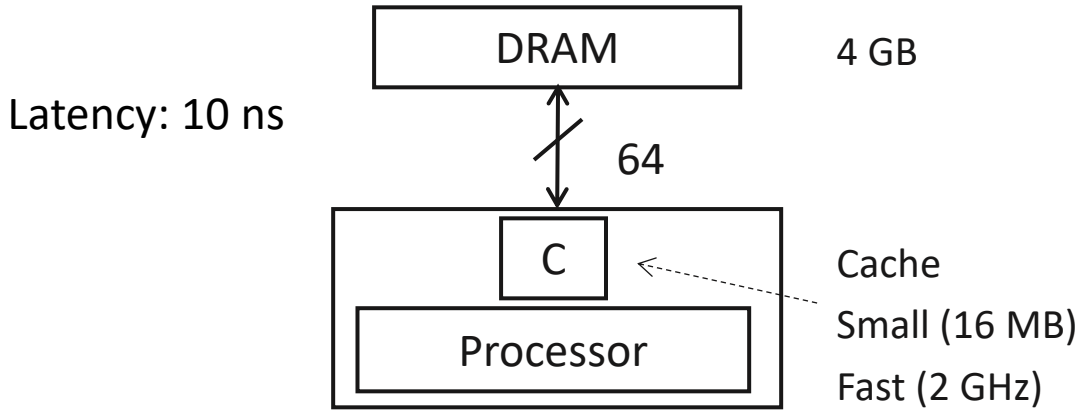
#3 Optimizations to Maximize Data Reuse

- How do we make sure that each data item of B is fetched just once and used M times?

- Effectively use Cache → a local on-chip memory where data can be stored till all the computations are performed

- Before we can “effectively” use cache, lets first understand how to model it

Execution Model with Cache



Bandwidth = 64 bits at 1 GHz
(64 Gbits/sec or 8 GB/sec) or
(64 bits in every 2 processor cycles) or **1 word in 1 ns**

Fetch_Memory – Latency + Data size/bandwidth
Fetch_Cache – 1 cycle
Execute – 1 cycle

Read/Write from DRAM to/from Cache

Basic Execution Model

Execution Model with Cache

- Assumptions
- Fetch_Memory: DRAM <-> Cache
- Processor has a limited number of registers, say 3-4
  - You can decide what “limited” means
  - Enough such that you don’t need complicated code
  - But not so large that you do not require cache anymore

Execution Model with Cache
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$

RMA: Read A into cache

RMB: Read B into cache

{ //repeat M*N times

    RA: Read A[i][k] into processor
    RB: Read B[k] into processor
    M: Res <- Mult A[i][k]*B[k]
    RC: Read C[i] into processor
    A: Add C[i] + Res
    S: Store C[i] into cache

}

SMC: Store C into DRAM

Execution Model with Cache
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$

RMA: Read A into cache
RMB: Read B into cache
{ //repeat M*N times
    RA: Read A[i][k] into processor
    RB: Read B[k] into processor
    M: Res <- Mult A[i][k]*B[k]
    RC: Read C[i] into processor
    A: Add C[i] + Res
    S: Store C[i] into cache
}
SMC: Store C into DRAM

This is prefetching which avoids the latency of fetching each M*N RA, RB, and S operations

Execution Model with Cache
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$

RMA: Read A into cache ?? 
RMB: Read B into cache ?? 
{ //repeat M*N times
RA: Read A[i][k] into processor
RB: Read B[k] into processor
M: Res <- Mult A[i][k]*B[k]
RC: Read C[i] into processor
A: Add C[i] + Res
S: Store C[i] into cache
} ??
SMC: Store C into DRAM ?? 

Execution Model with Cache
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$

RMA: Read A into cache — 10 + MN ns

RMB: Read B into cache — 10 + N ns

{ //repeat M*N times

RA: Read A[i][k] into processor

RB: Read B[k] into processor

M: Res <- Mult A[i][k]*B[k]

RC: Read C[i] into processor

A: Add C[i] + Res

S: Store C[i] into cache

} — 3MN ns

SMC: Store C into DRAM — 10 + N ns

Execution Model with Cache

- System Performance: ??

Execution Model with Cache

- System Performance:
- 2MN useful operations
- Time (ns): 30 + 3MN + MN + 2N
- System Performance (GFLOPs): 2MN/(30 + 4MN + 2N)

Execution Model with Cache

- Recall
- Peak Performance: 2 GFLOPs
- Sustained Performance (best case): 0.4 GFLOPs
- Sustained Performance (worst case): 0.048 GFLOPs
- Sustained Performance (with cache): 2MN/(30 + 4MN + 2N)

Execution Model with Cache

- Recall
- Peak Performance: 2 GFLOPs
- Sustained Performance (best case): 0.4 GFLOPs
- Sustained Performance (worst case): 0.048 GFLOPs
- Sustained Performance (with cache): 2MN/(4MN)
  - For large MN

Execution Model with Cache

- Recall
- Peak Performance: 2 GFLOPs
- Sustained Performance (best case): 0.4 GFLOPs
- Sustained Performance (worst case): 0.048 GFLOPs
- Sustained Performance (with cache): 0.5 GFLOPs
  - For large MN

Execution Model with Cache

- Ungraded HW assignment (may include as a question in WA 1, or even better, in the exam)

- Why is the performance marginally better than Sustained Performance (best case)? Why not lower or significantly better?

Memory Bound Applications

- Why is the performance still significantly lower than peak performance?
- Peak Performance: 2 GFLOPs
- Sustained Performance (best case): 0.4 GFLOPs
- Sustained Performance (with cache): 0.5 GFLOPs

Memory Bound Applications

- Why is the performance still significantly lower than peak performance?
- Time (ns): $30 + 3MN + MN + 2N \approx 4MN$
- Time spent on Computation: MN
- Time spent on Memory Operations: 3MN
- Performance is limited by how fast data can be fed to the processors – **Memory boundedness**
  - We will look at optimizations to maximize data reuse in next class which can be used to optimize these types of applications

Cache Limitations

- In practice, cache size is limited
- Previously fetched data is evicted from cache, if no space.
- We need to carefully design our algorithms to obtain best performance using cache

Enable effective utilization of cache to maximize data reuse and minimize data transfers

Optimizations to Enable Effective Utilization of Cache

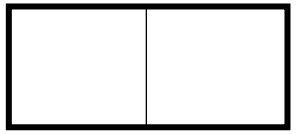
- #1 Data Layout Optimizations

- #2 Algorithm Reordering to Maximize Data Reuse

#1 Data Layout Optimization

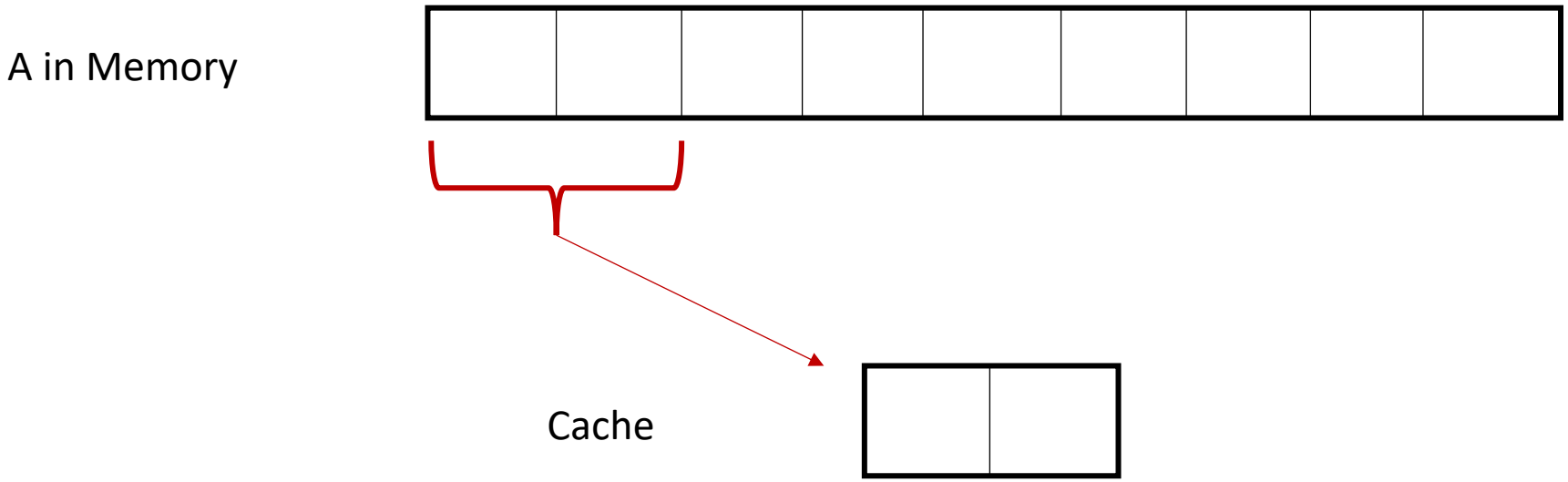
A in Memory

Cache



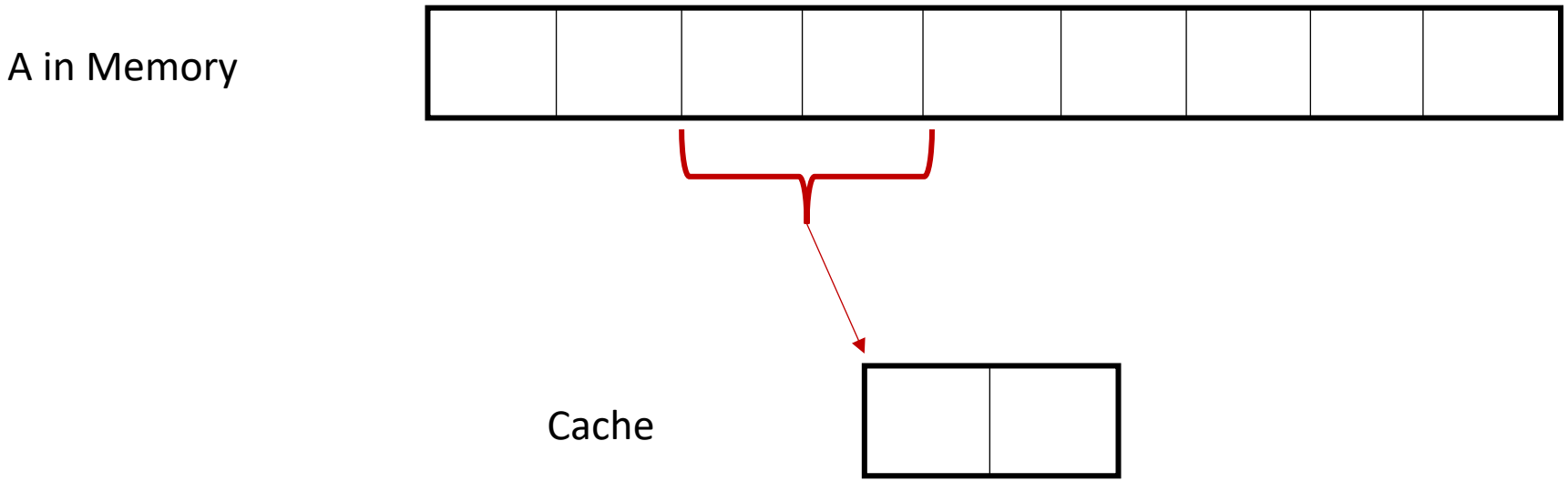
Need to Fetch elements 2 at a time to cache.
Each Fetch_Memory – Latency + data size/bandwidth

#1 Data Layout Optimization



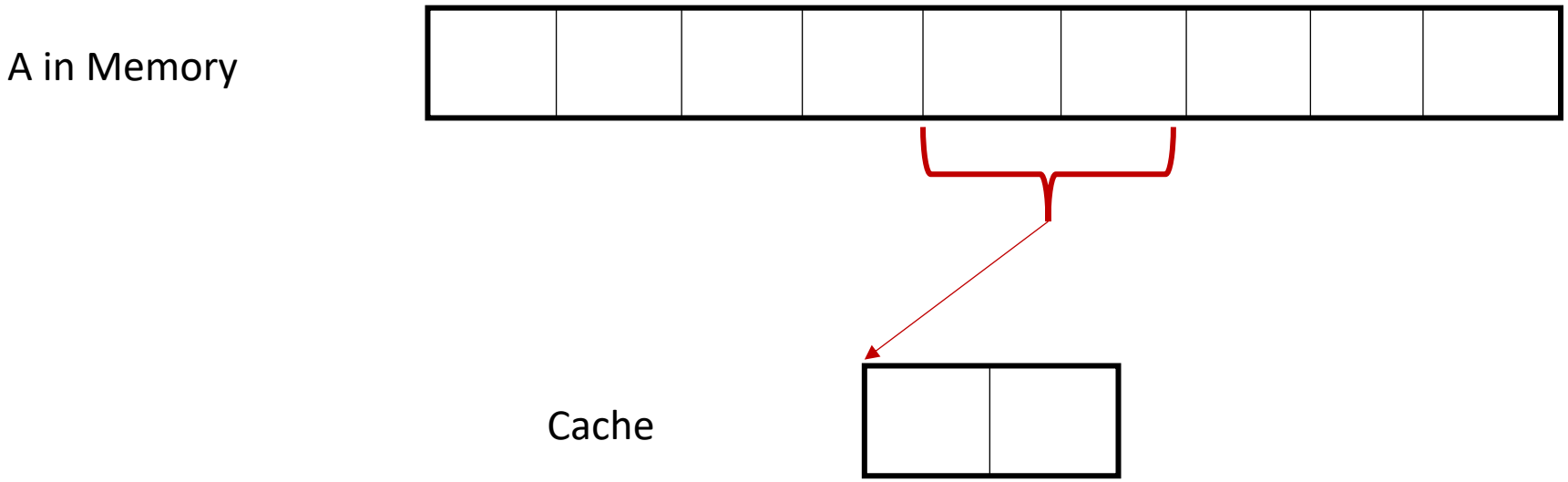
Need to Fetch elements 2 at a time to cache.
Each Fetch_Memory – Latency + data size/bandwidth

#1 Data Layout Optimization



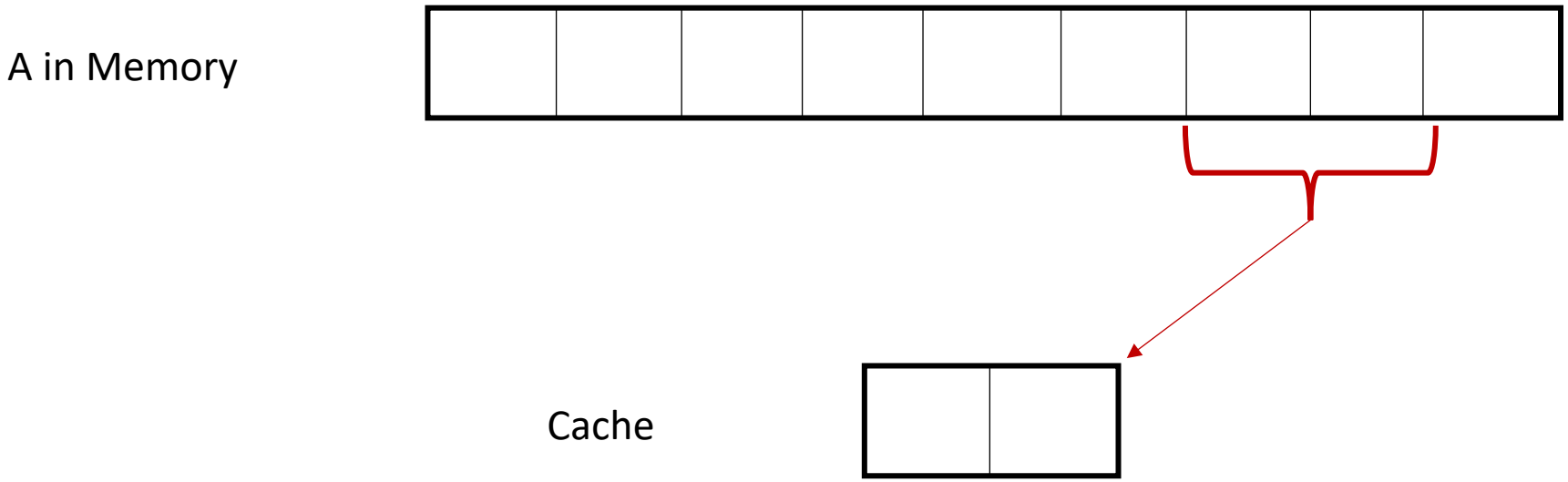
Need to Fetch elements 2 at a time to cache.
Each Fetch _Memory – Latency + data size/bandwidth

#1 Data Layout Optimization



Need to Fetch elements 2 at a time to cache.
Each Fetch _Memory – Latency + data size/bandwidth

#1 Data Layout Optimization

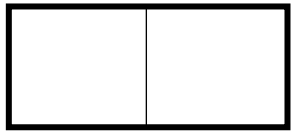


Need to Fetch elements 2 at a time to cache.
Each Fetch _Memory – Latency + data size/bandwidth

#1 Data Layout Optimization

A in Memory

Cache

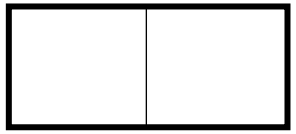


Algorithm access Pattern: 0, 2, 4, 6, 8, 1, 3, 5, 7

#1 Data Layout Optimization

A in Memory

Cache



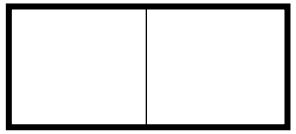
Algorithm access Pattern: 0, 2, 4, 6, 8, 1, 3, 5, 7

Number of Fetch_Memory Needed: ??

#1 Data Layout Optimization

A in Memory

Cache

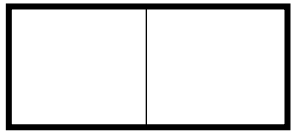


Algorithm access Pattern: 0, 2, 4, 6, 8, 1, 3, 5, 7
Number of Fetch_Memory Needed: 9

#1 Data Layout Optimization

A in Memory

Cache



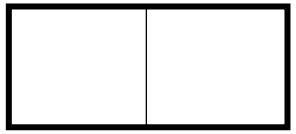
Algorithm access Pattern: 0, 2, 4, 6, 8, 1, 3, 5, 7

Number of Fetch_Memory Needed: ??

#1 Data Layout Optimization

A in Memory

Cache

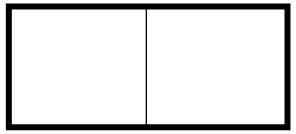


Algorithm access Pattern: 0, 2, 4, 6, 8, 1, 3, 5, 7
Number of Fetch_Memory Needed: 5

#1 Data Layout Optimization

A in Memory

Cache



Or Change Algorithm access Pattern: 0, 1,2,3,4,5,6,7,8

Number of Fetch_Memory Needed: 5

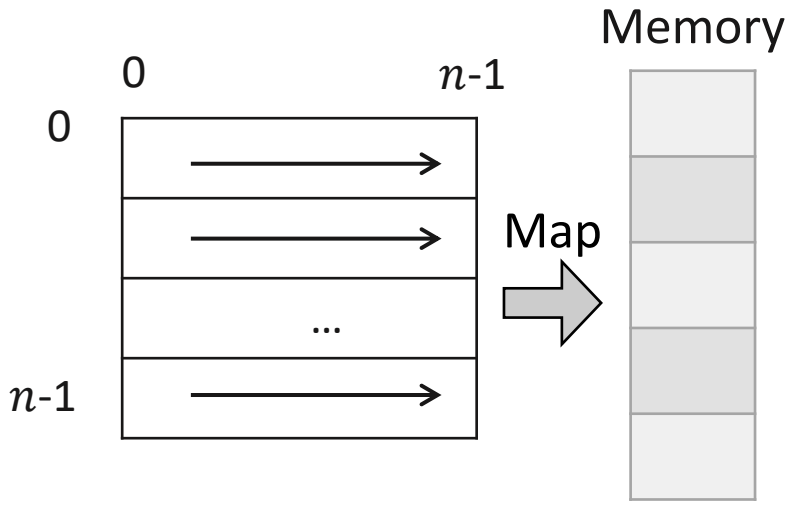
#1 Data Layout Optimization

- Objective: Ensure sequential access to data stored in memory by,
  - Either, storing data in the right format
  - Or, reordering computations (access pattern of program)

#1 Data Layout Optimization

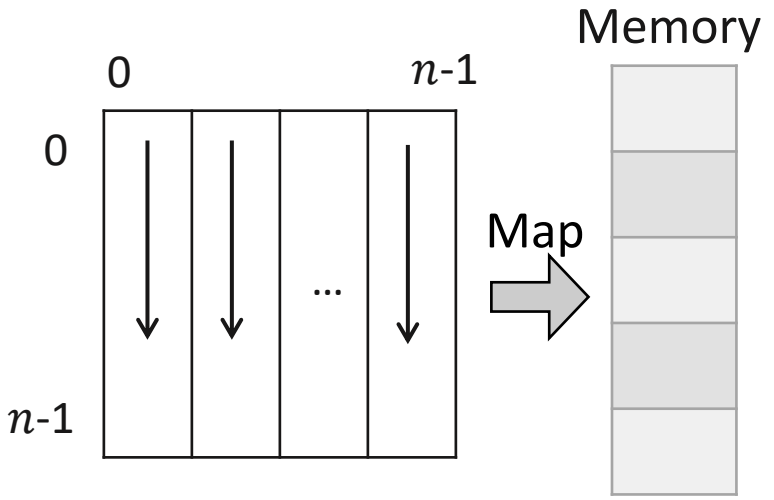
Storing a 2-dimemsional array in memory

Row major order



$$A(i,j) \rightarrow \text{Memory}(i \cdot n + j)$$
$$0 \leq i,j < n$$

Column major order

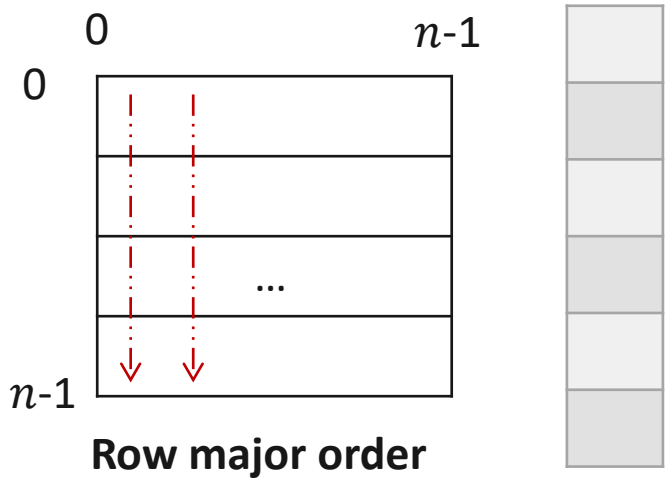


$$A(i,j) \rightarrow \text{Memory}(i + n \cdot j)$$
$$0 \leq i,j < n$$

#1 Data Layout Optimization

- For k = 1 to n
  - For i = 1 to n
    - C[i] <- A[i][k] + B[k] + C[i]

Column major access pattern



- Assume A is stored in row major order

Access pattern of the program

Data Access not sequential
Each read of A[i][k] will incur latency

Data Layout

#1 Data Layout Optimization

- Assume A is stored in row major order
- For i = 1 to m
  - For k = 1 to n
    - C[i] <- A[i][k] + B[k] + C[i]

Row major access pattern

Access pattern of the program

Data Access sequential
Read A[i][k] at the rate of bandwidth

Data Layout

#1 Data Layout Optimization

- Alternate Solution: Store A in Column major order

Access pattern of the program

Data Access sequential
Read A[i][k] at the rate of bandwidth

Data Layout

#1 Data Layout Optimization: Summary

- We should align the layout of the data in the memory to match with the access pattern of the algorithm
  - Simple N-D row/column major layout should be sufficient for most purposes
- Access patterns may change over iterations of the algorithm
  - For example, 2D FFT
  - Algorithm designers sometimes employ dynamic data layout changing mechanisms, if the overheads are justified
- We will not discuss dynamic data layout optimizations in this class

#2 Improving Data Reuse

- Our objective is to reduce the number of data transfers of input and output
- If a data element is used multiple times, try to reorder the algorithm such that all its uses are “temporally” close to each other
- We will discuss this in the next class when we look at how to reorder matrix multiplication algorithm to improve data reuse

Next Class

- 9/4 Lecture 4 – Blocked MM for improving data reuse, Modeling GPU Architectures

CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

Thank You

- Questions?

- Email: sanmukh.kuppannagari@case.edu