CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

CSDS 451: Designing High Performant Systems for AI

Lecture 4

9/4/2025

Sanmukh Kuppannagari

sanmukh.kuppannagari@case.edu

https://sanmukh.research.st/

Case Western Reserve University

Outline

- Processor Memory Architectures
    - Blocked Matrix Multiplication
- Modeling GPU Architectures
- Data Parallel Programming

Outline

- Processor Memory Architectures
    - Blocked Matrix Multiplication
- Modeling GPU Architectures
- Data Parallel Programming

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 look at it later in today's lecture and in the next lecture)

- 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)

#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**)
2. Prefetch the data into the local memory before the computations begin

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

#2 Optimizations to Improve Bandwidth

- Optimizations that you can use in your algorithms

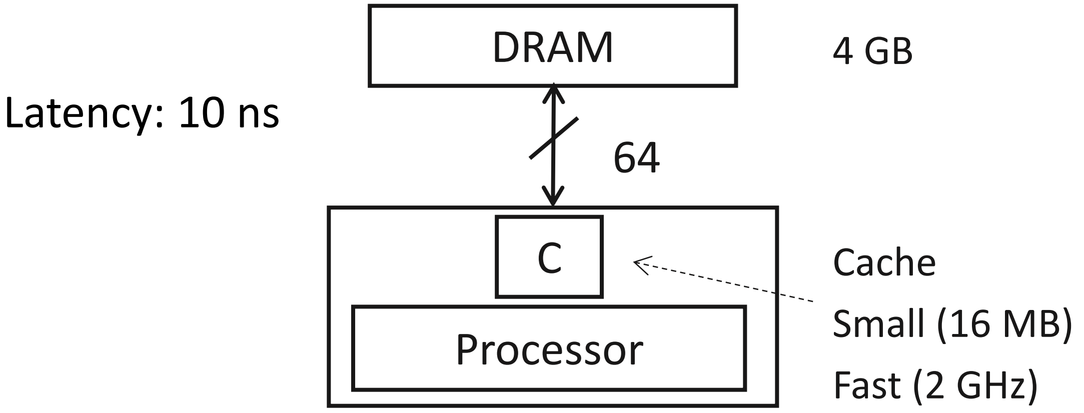
1. Store the data in arrays in a sequential manner
2. When fetching, try to fetch as much data as possible in a single go

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

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

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: Summary

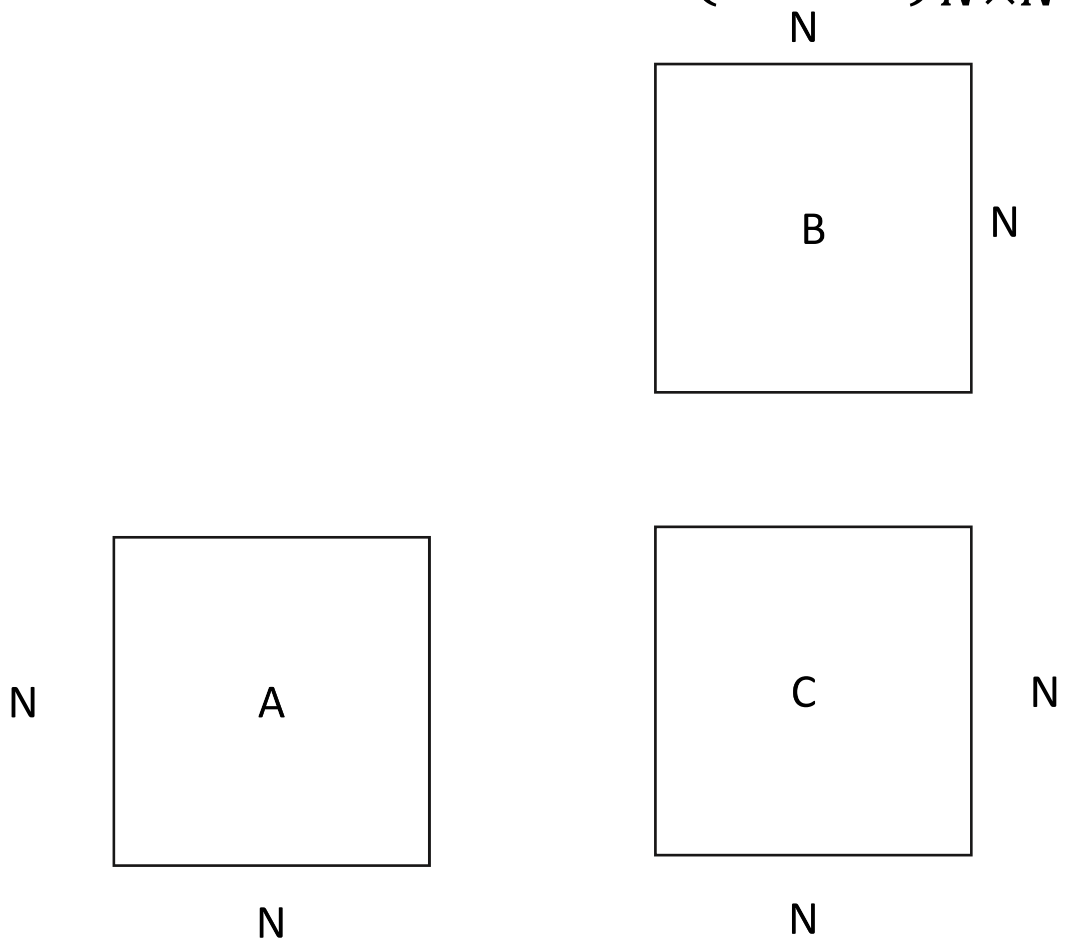
- We should align the layout of the data in the memory to match with the access pattern of the algorithm
  - Simple N-D row/column major layout should be sufficient for most purposes
- Access patterns may change over iterations of the algorithm
  - For example, 2D FFT
  - Algorithm designers sometimes 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
- Blocked Matrix Multiplication

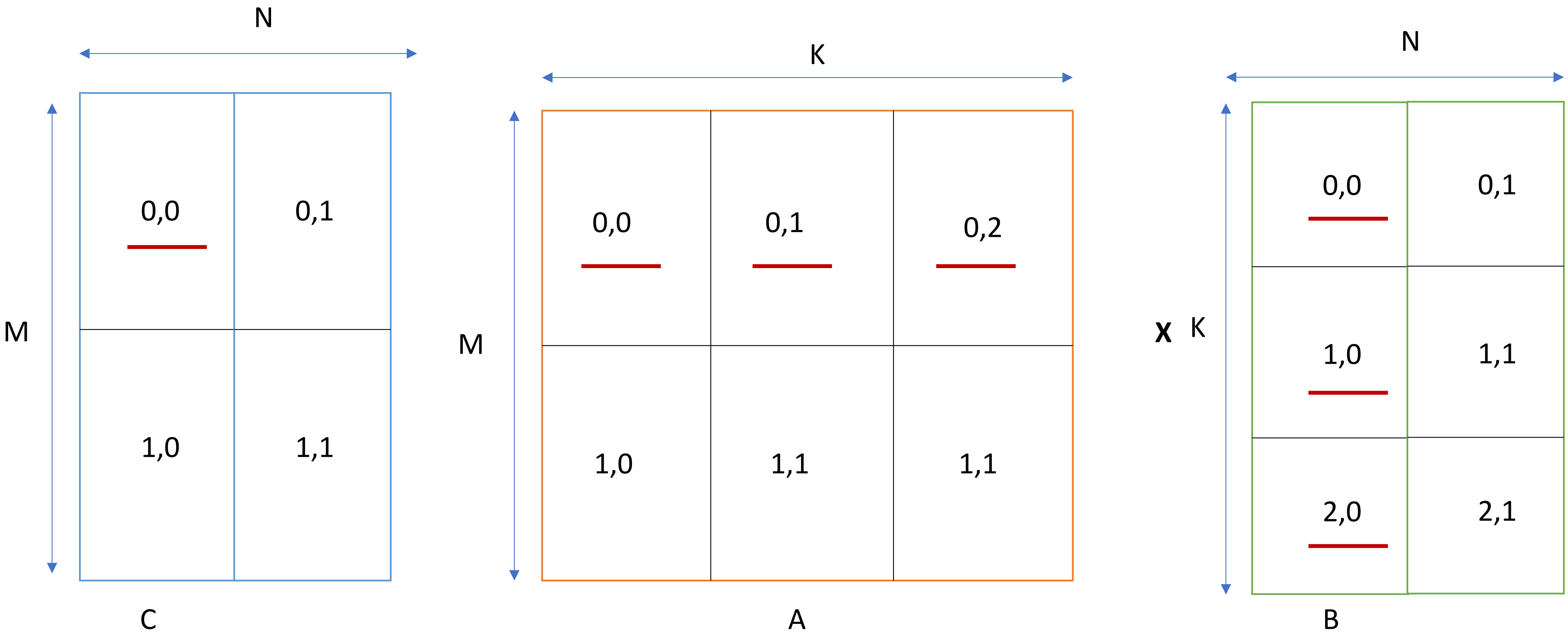
Matrix Multiplication

$$C = (A \times B)_{N \times N}$$



Matrix Matrix Multiplication

$$C[i][j] = \sum A[i][k] * B[k][j]$$



Matrix Multiplication

$$\boldsymbol{C} = (\boldsymbol{A} \times \boldsymbol{B})_{\boldsymbol{N} \times \boldsymbol{N}}$$

Cache Size - S

Definition:

Data Reuse Factor (for an algorithm): Number of Computations/Number of data transfers

Can be used to determine the effectiveness of an algorithm

Matrix Multiplication

- Data Reuse Factor, with infinite cache

- Fetch matrix A - N^2 data transfers
- Fetch matrix B - N^2 data transfers
- Fetch matrix C - N^2 data transfers
- Compute C = $2N^3$ computation operations
- Save C back to memory - N^2 data transfers

- Data Reuse Factor: $\frac{2N^3}{4N^2} = ??$

Matrix Multiplication

- Data Reuse Factor, with infinite cache

- Fetch matrix A - N^2 data transfers
- Fetch matrix B - N^2 data transfers
- Fetch matrix C - N^2 data transfers
- Compute C = $2N^3$ computation operations
- Save C back to memory - N^2 data transfers

- Data Reuse Factor: $\frac{2N^3}{4N^2} = O(N)$

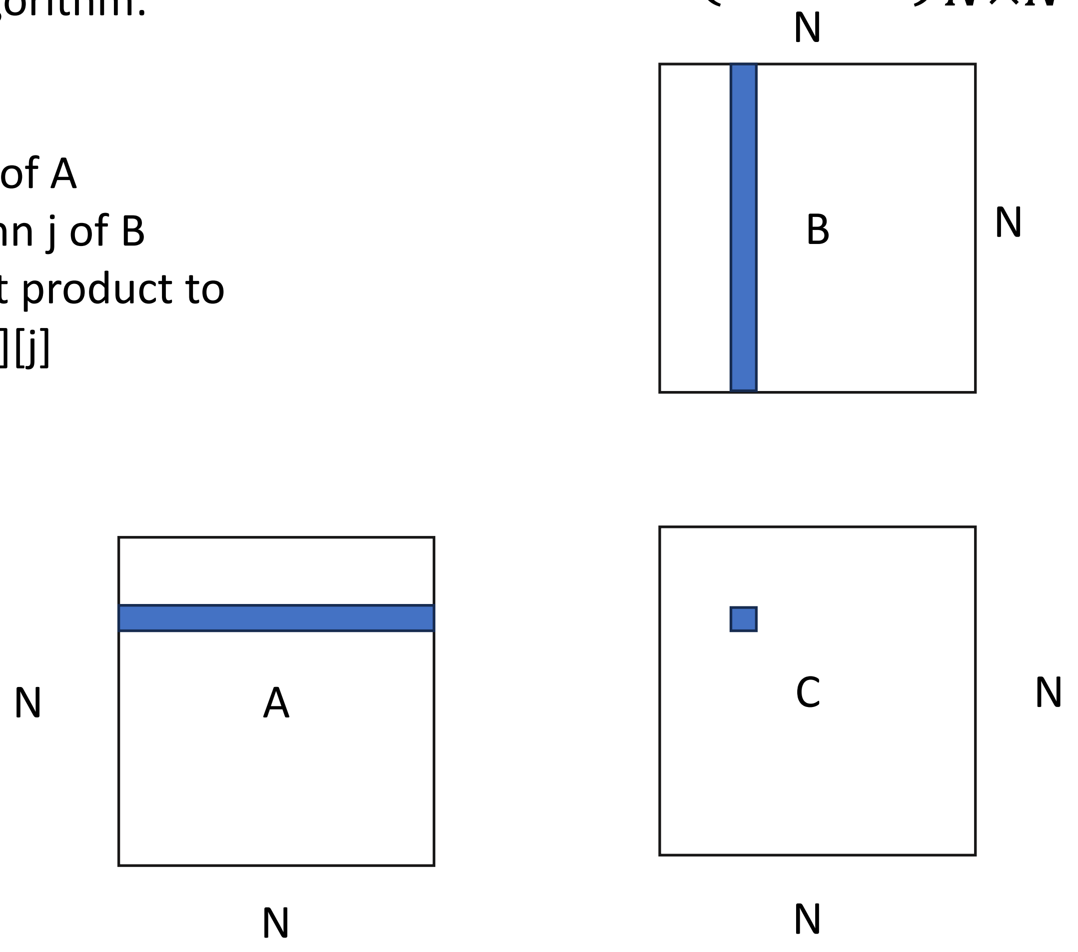
Naïve Matrix Multiplication

$$\boldsymbol{C = (A \times B)_{N \times N}}$$

Basic three loop Algorithm:

For each i,j of C

- Fetch row i of A
- Fetch column j of B
- Perform dot product to produce C[i][j]



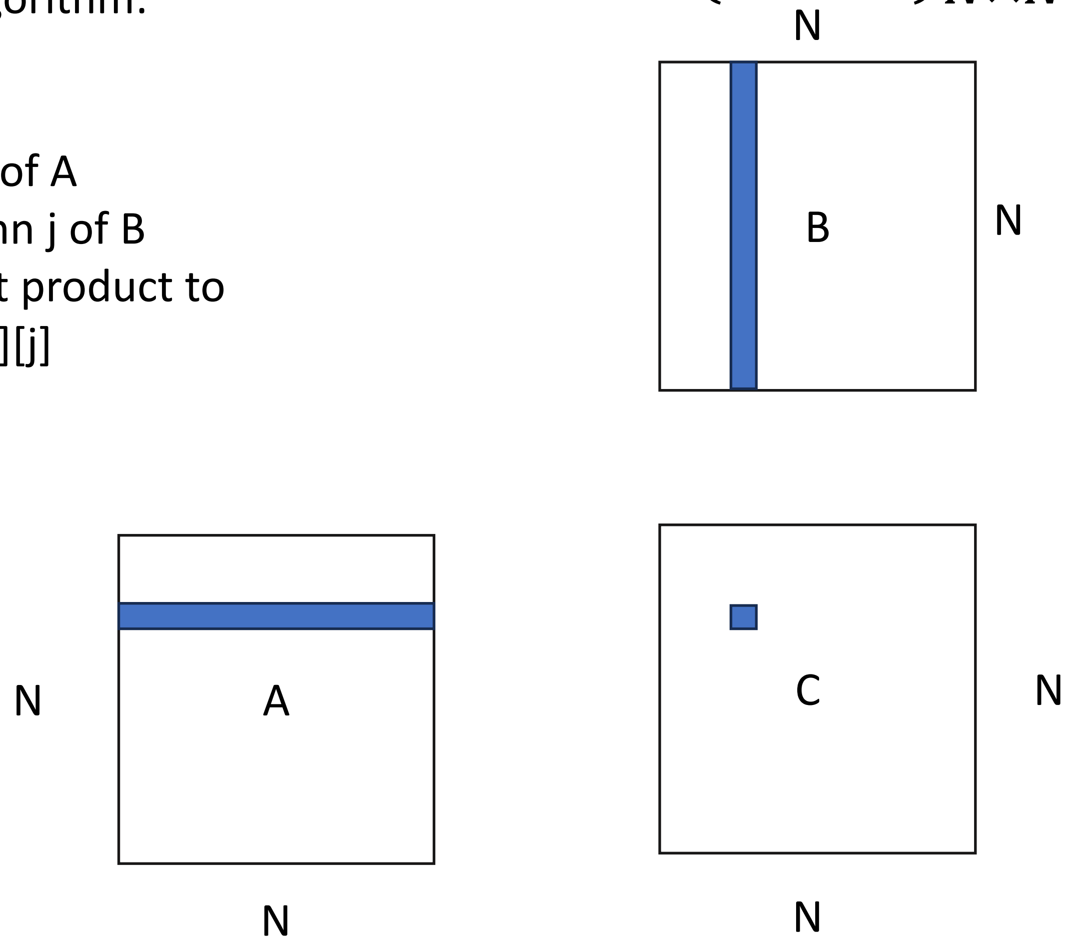
Assume Cache size $S = 3N$

Naïve Matrix Multiplication

$$C = (A \times B)_{N \times N}$$

Basic three loop Algorithm:

For each i,j of C
- Fetch row i of A
- Fetch column j of B
- Perform dot product to produce C[i][j]



Assume Cache size $S = 3N$
Data reuse factor = ??

Naïve Matrix Multiplication

$$\boldsymbol{C = (A \times B)_{N \times N}}$$

Basic three loop Algorithm:

For each i,j of C

- Fetch row i of A
- Fetch column j of B
- Perform dot product to produce C[i][j]

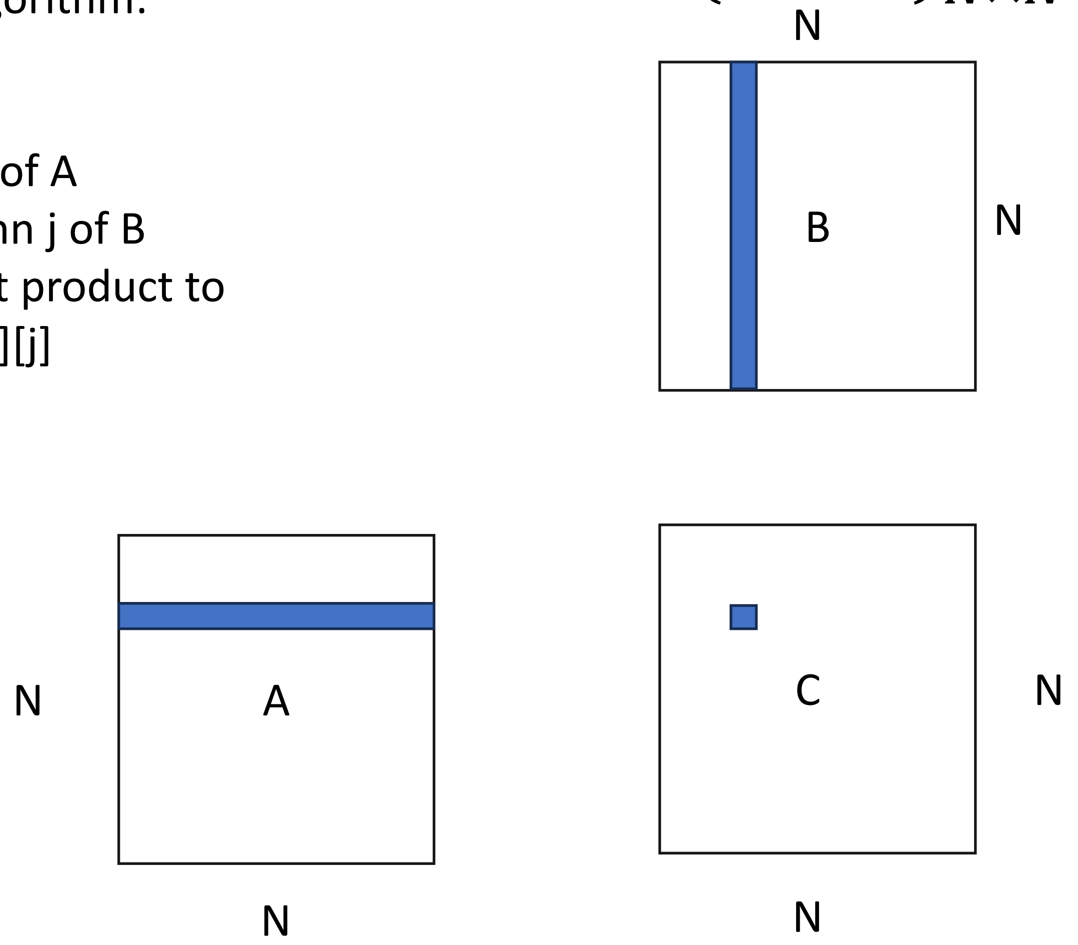
Assume Cache size $S = 3N$

Data reuse factor = ??

Matrix A – Fetch 1 time (N^2 data transfers)

Matrix B – Fetch N times (N^3 data transfers)

Matrix C – Fetch 1 times (N^2 data transfers)

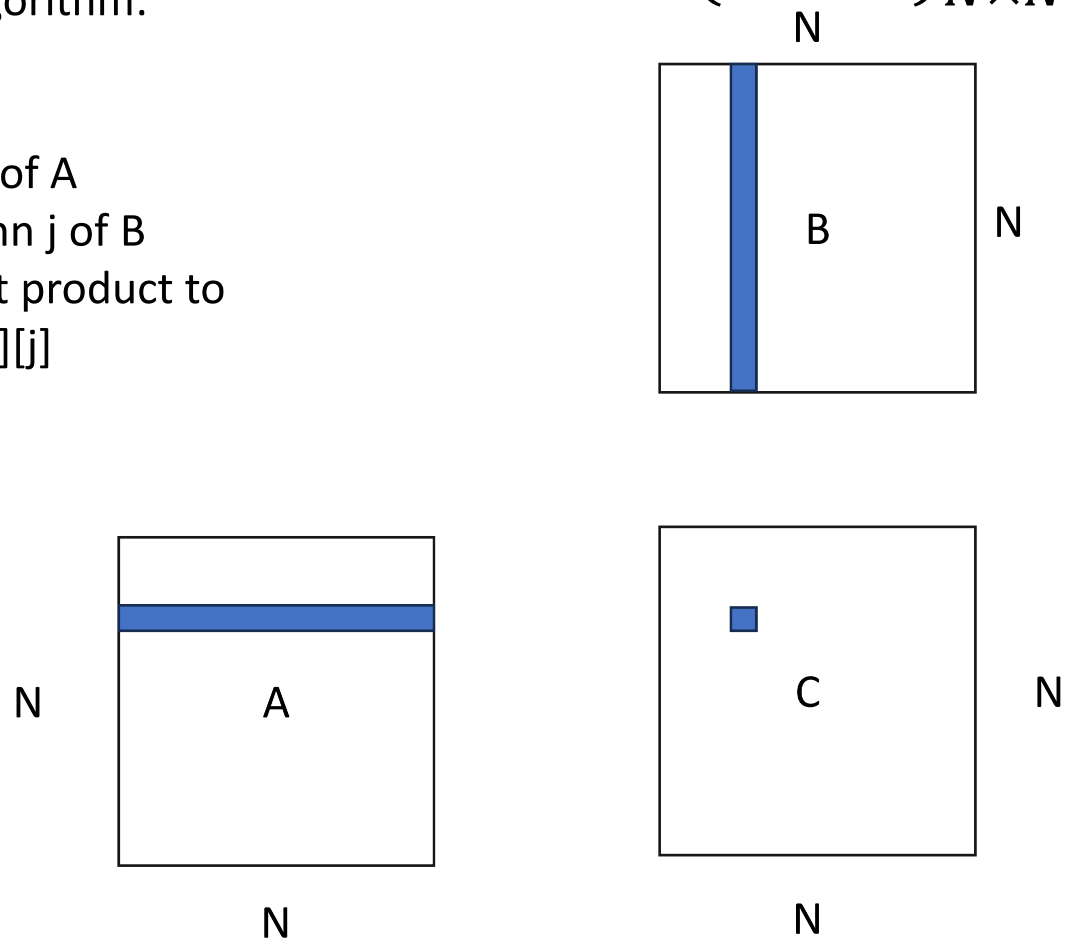


Naïve Matrix Multiplication

Basic three loop Algorithm:

For each i,j of C
- Fetch row i of A
- Fetch column j of B
- Perform dot product to produce C[i][j]

$$\boldsymbol{C} = (\boldsymbol{A} \times \boldsymbol{B})_{\boldsymbol{N \times N}}$$



Assume Cache size $S = 3N$

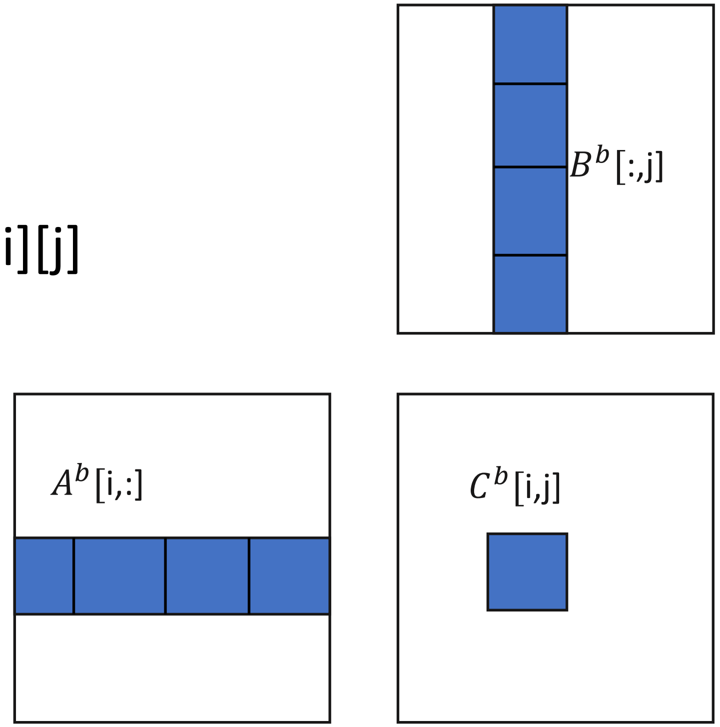
Data reuse factor = $\frac{2N^3}{2N^2+N^3} \sim O(1)$

Matrix A – Fetch 1 time (N^2 data transfers)
Matrix B – Fetch N times (N^3 data transfers)
Matrix C – Fetch 1 times (N^2 data transfers)

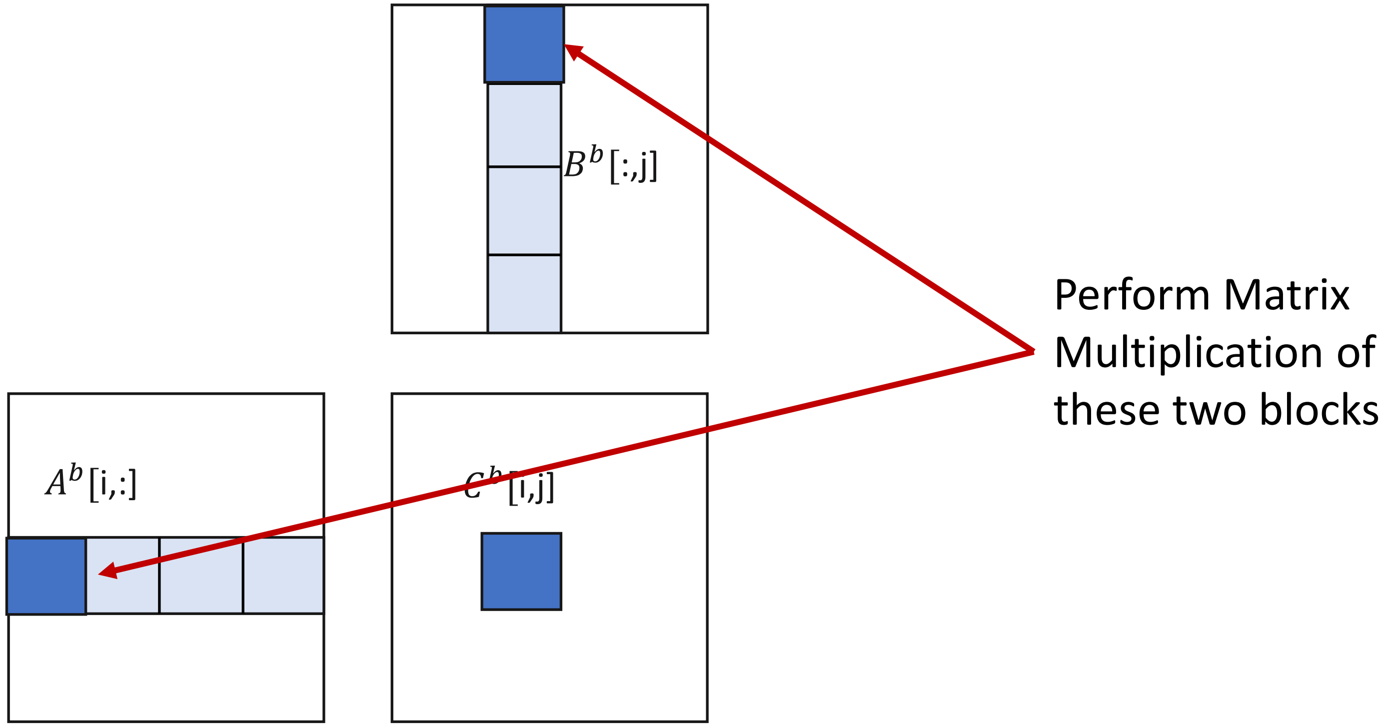
Block Matrix Multiplication

- Key Idea: Partition matrices A/B/C into $b \times b$ size blocks
- Blocks denoted as $A^b[1{:}\frac{N}{b}][1{:}\frac{N}{b}]$/ $B^b[1{:}\frac{N}{b}][1{:}\frac{N}{b}]$/ $C^b[1{:}\frac{N}{b}][1{:}\frac{N}{b}]$

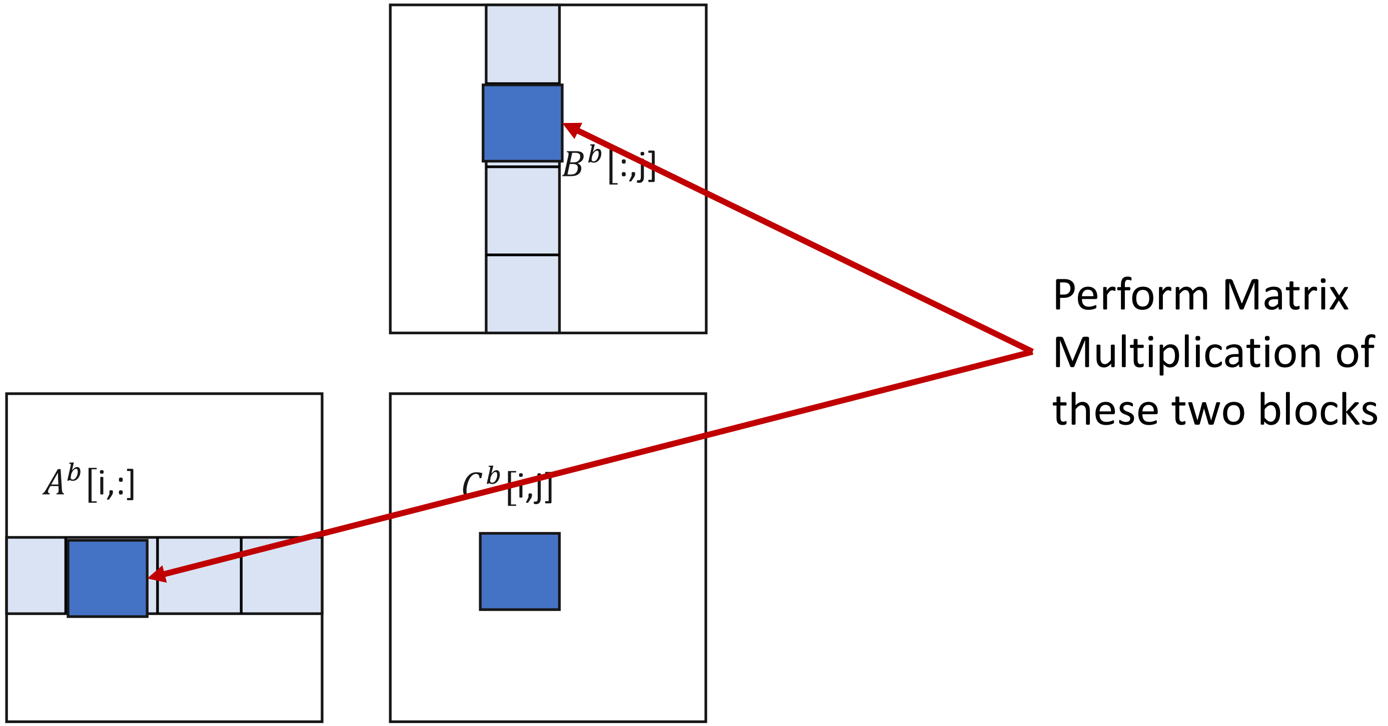
- $C^b[i][j]$ = BlockedMM($A^b[i][:]$, $B^b[:][j]$)
- For $k = 0$ to N/b
  - Fetch $A^b[i][k]$, $B^b[k][j]$
  - Matrix Multiply $A^b[i][k]$, $B^b[k][j]$ and Accumulate $C^b[i][j]$



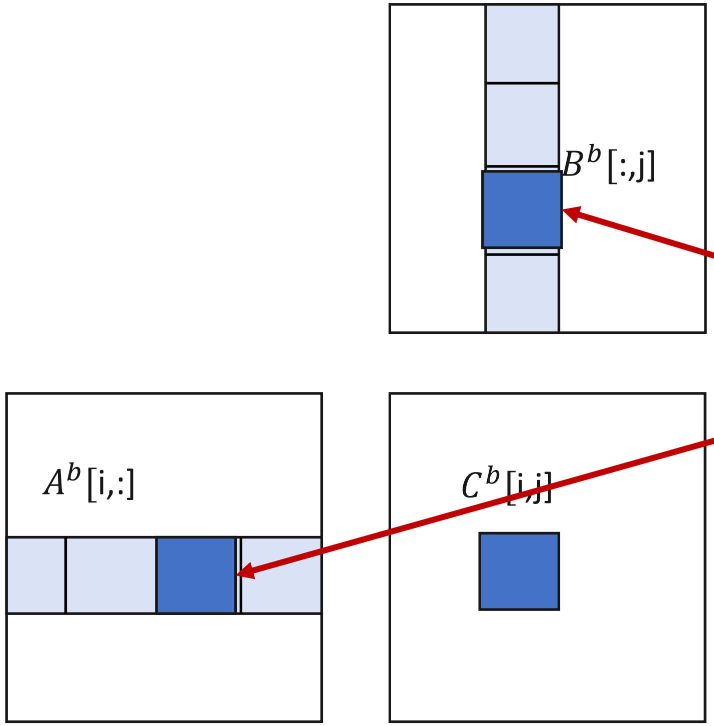
Block Matrix Multiplication



Block Matrix Multiplication

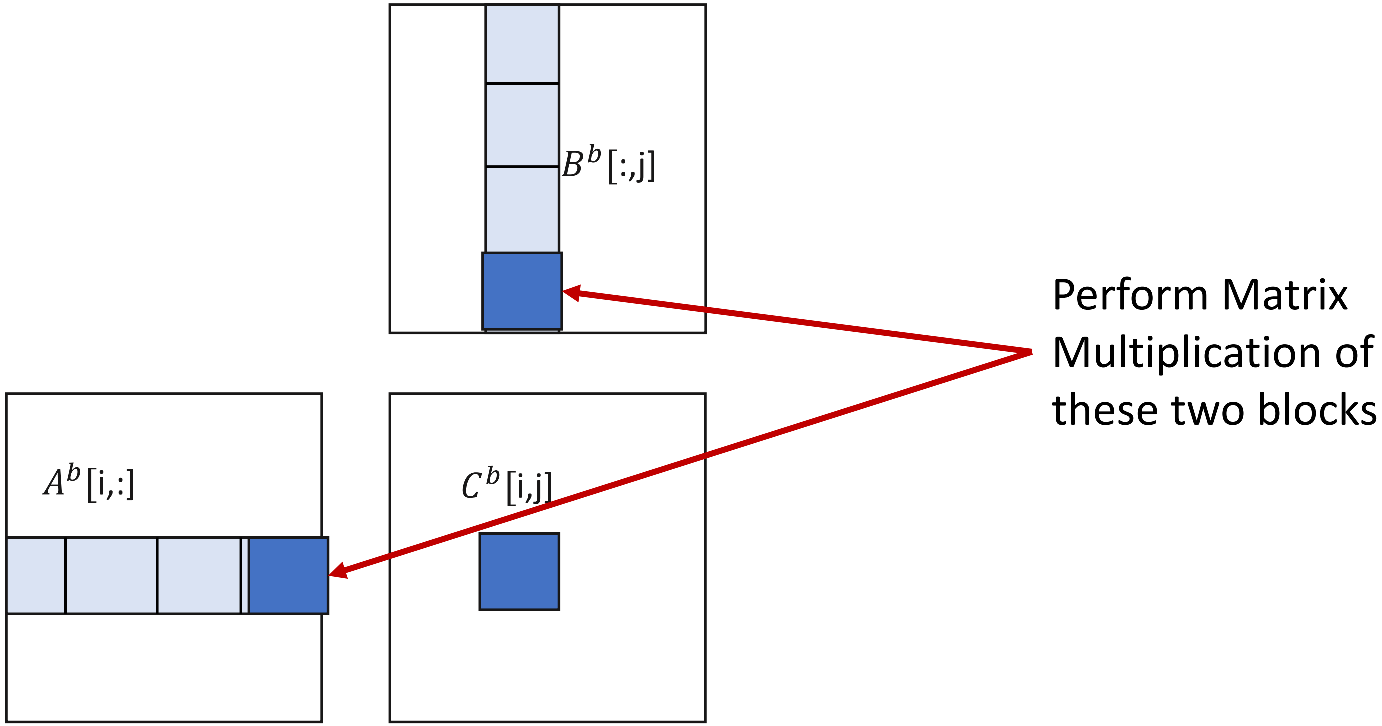


Block Matrix Multiplication



Perform Matrix Multiplication of these two blocks

Block Matrix Multiplication



Block Matrix Multiplication

- $S = 3b^2, b = \frac{\sqrt{S}}{\sqrt{3}} \approx O(\sqrt{S})$

- Data Reuse Factor = ? ? ?

Block Matrix Multiplication

- For each $C^b[i][j]$

Block Matrix Multiplication

- For each $C^b[i][j]$
- Fetch $A^b[i][k]$, $B^b[k][j]$, $\forall k$ – how many data transfers?

Block Matrix Multiplication

- For each $C^b[i][j]$
- Fetch $A^b[i][k]$, $B^b[k][j]$, $\forall k - O(\overbrace{\sqrt{S} \times \sqrt{S}}^{\text{Block Size}} \times \frac{N}{\sqrt{S}})$

Block Matrix Multiplication

- For each $C^b[i][j]$

- Fetch $A^b[i][k]$, $B^b[k][j]$, $\forall k - O(\overbrace{\sqrt{S} \times \sqrt{S}}^{\text{Block Size}} \times \overbrace{\frac{N}{\sqrt{S}}}^{\text{Range of k}})$

- Computations for $A^b[i][k]$, $B^b[k][j]$, $\forall k$ - ?? computations

Block Matrix Multiplication

- For each $C^b[i][j]$

- Fetch $A^b[i][k]$, $B^b[k][j]$, $\forall k - O(\sqrt{S} \times \sqrt{S} \times \frac{N}{\sqrt{S}})$

- Computations for $A^b[i][k]$, $B^b[k][j]$, $\forall k$ - $O(\overbrace{\sqrt{S}^3}^{\text{Computations per block}} * \overbrace{\frac{N}{\sqrt{S}}}^{\text{Range of k}})$ computations

Block Matrix Multiplication

- For each $C^b[i][j]$
- Fetch $A^b[i][k]$, $B^b[k][j]$, $\forall k - O(\sqrt{S} \times \sqrt{S} \times \frac{N}{\sqrt{S}})$
- Computations for $A^b[i][k]$, $B^b[k][j]$, $\forall k$ - $O(\sqrt{S}^3 * \frac{N}{\sqrt{S}})$ computations
- Store $C^b[i][j]$ - ? ? data transfers

Block Matrix Multiplication

- For each $C^b[i][j]$

- Fetch $A^b[i][k]$, $B^b[k][j]$, $\forall k - O(\sqrt{S} \times \sqrt{S} \times \frac{N}{\sqrt{S}})$

- Computations for $A^b[i][k]$, $B^b[k][j]$, $\forall k$ - $O(\sqrt{S}^3 * \frac{N}{\sqrt{S}})$ computations

- Store $C^b[i][j]$ - $O(\sqrt{S} \times \sqrt{S})$ data transfers

Block Matrix Multiplication

- For each $C^b[i][j]$
- Fetch $A^b[i][k]$, $B^b[k][j]$, $\forall k - O(\sqrt{S} \times \sqrt{S} \times \frac{N}{\sqrt{S}})$
- Computations for $A^b[i][k]$, $B^b[k][j]$, $\forall k$ - $O(\sqrt{S}^3 * \frac{N}{\sqrt{S}})$ computations
- Store $C^b[i][j]$ - $O(\sqrt{S} \times \sqrt{S})$ data transfers
- Number of $C^b[i][j]$ blocks - ??

Block Matrix Multiplication

- For each $C^b[i][j]$
- Fetch $A^b[i][k]$, $B^b[k][j]$, $\forall k - O(\sqrt{S} \times \sqrt{S} \times \frac{N}{\sqrt{S}})$
- Computations for $A^b[i][k]$, $B^b[k][j]$, $\forall k$ - $O(\sqrt{S}^3 * \frac{N}{\sqrt{S}})$ computations
- Store $C^b[i][j]$ - $O(\sqrt{S} \times \sqrt{S})$ data transfers
- Number of $C^b[i][j]$ blocks - $\frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}}$

Block Matrix Multiplication

- Data Reuse Factor:

- Total Computations - ? ?
- Total data transfers – ? ? transfers

- Data Reuse Factor: ? ?

Block Matrix Multiplication

- Data Reuse Factor:

- Total Computations - $O(\sqrt{S}^3 \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}})$
- Total data transfers – $O(\sqrt{S} \times \sqrt{S} \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}})$ transfers

- Data Reuse Factor: $O(\sqrt{S})$

Block Matrix Multiplication

- Data Reuse Factor:
- Total Computations - $O(\sqrt{S}^3 \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}})$
- Total data transfers – $O(\sqrt{S} \times \sqrt{S} \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}})$ transfers
- Data Reuse Factor: $O(\sqrt{S})$
- Data Reuse Factor: $O\left(\sqrt{S}\right)$ = $O(\sqrt{N})$ for $S = 3N$
- For naïve case: $O(1)$
- For infinite cache: $O(N)$

Block Matrix Multiplication – Things to Notice

- No change in total computations – still $O(N^3)$
- In naïve MM – A fetched once, B fetched N times
- In Blocked MM – A fetched $\sqrt{N}$ times, B fetched $\sqrt{N}$ times
- We balanced the data reuse between A and B to achieve a better overall reuse

Block Matrix Multiplication – Things to Notice

- In practice, Matrices A and B can be of different sizes,
- Optimal block sizes may be different for A and B.
  - May not even be square blocks – tiles
- Tiled Matrix Multiplication – Use rectangular blocks instead of square
- Automatic Tuning of Tile sizes – An important way to optimize matrix operations in practice
  - ATLAS - https://math-atlas.sourceforge.net/
  - Jack Dongarra – Turing Award, 2021
  - "For his pioneering contributions to numerical algorithms and libraries that enabled high performance computational software to keep pace with exponential hardware improvements for over four decades"

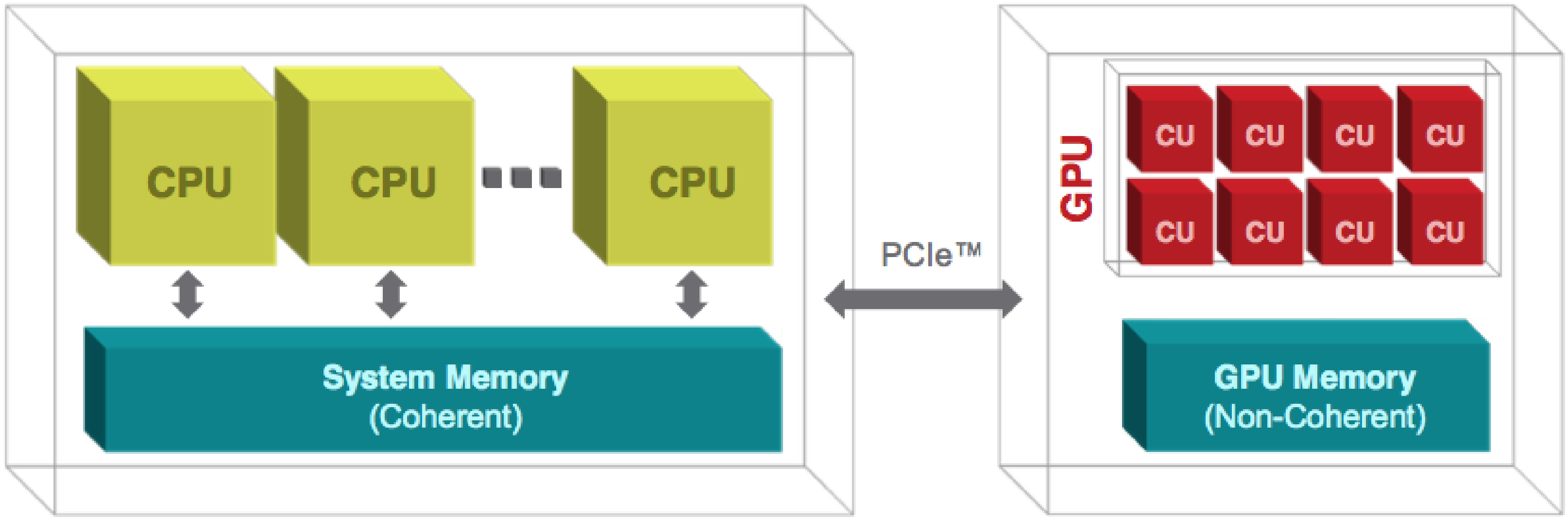
Ungraded HW Assignment

- Consider our memory system with cache model
- Consider the naïve and blocked matrix multiplication algorithms
- Model them on the platform and compare performance (Sustained Performance (with cache))
- Change or assume any input or cache sizes as needed to simplify calculations and improve understanding

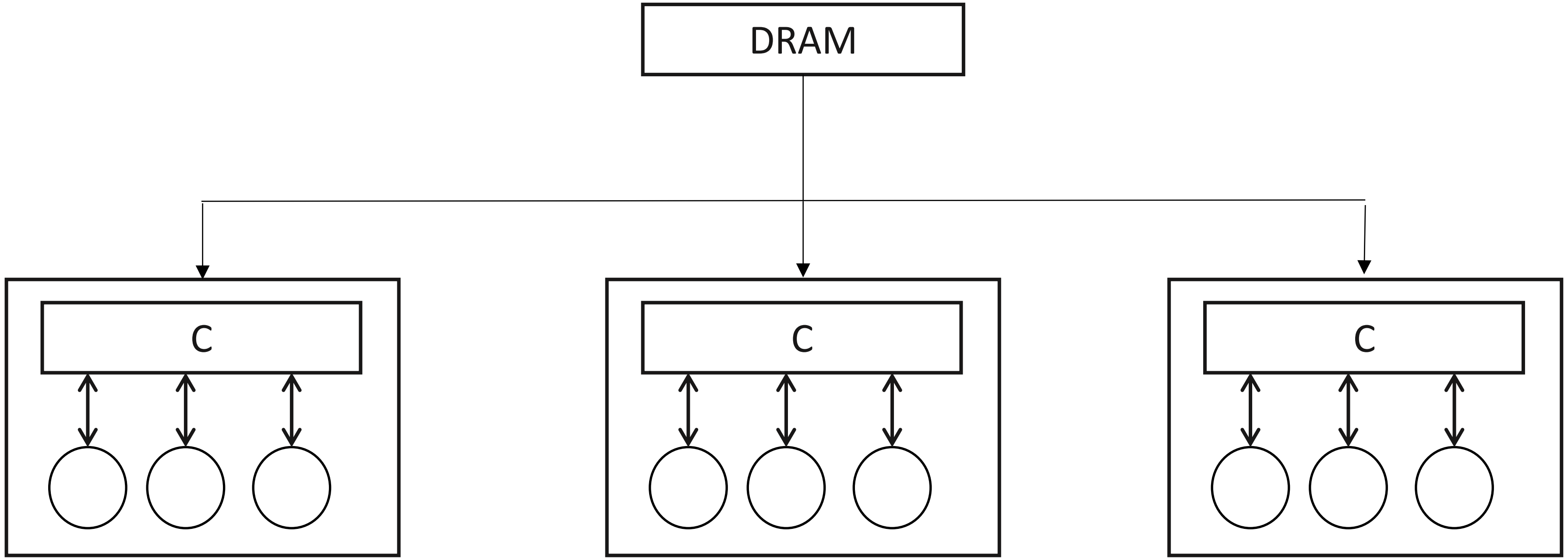
Outline

- Processor Memory Architectures
    - Blocked Matrix Multiplication
- Modeling GPU Architectures
- Data Parallel Programming

CPU + GPU Heterogeneity



Modeling GPU Architectures

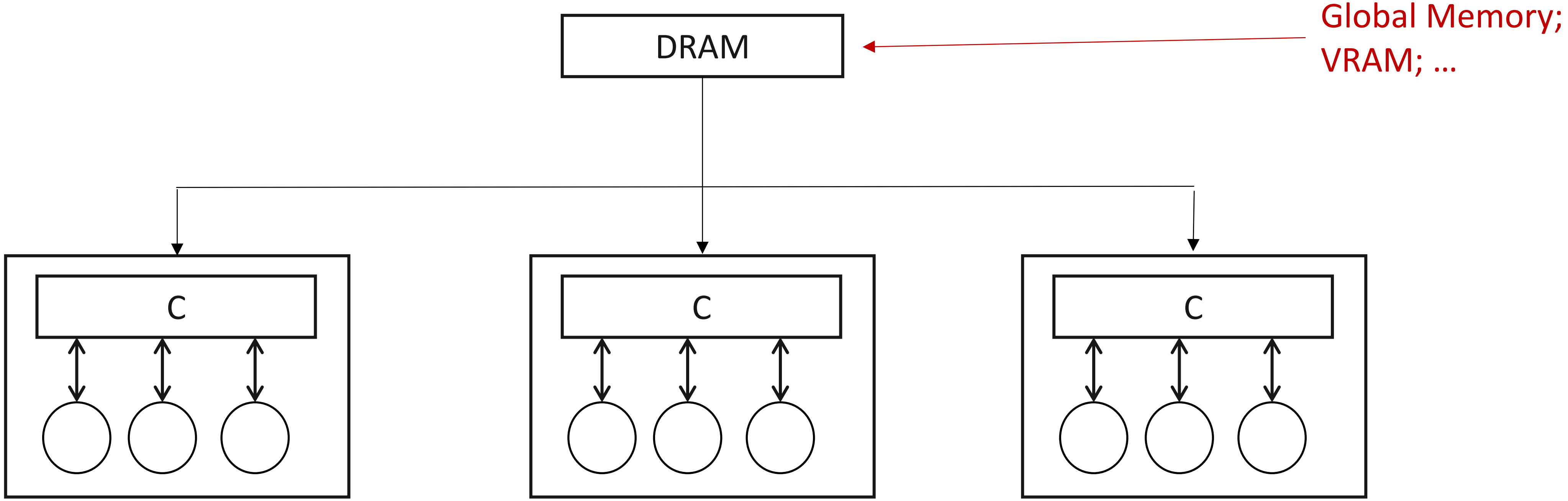


GPU Architecture

- A collection of Symmetric Multi-Processor (SMP)
- Each SMP – Processor/Memory Architecture with Cache with multiple processors running in lock step

- SPMD – Single Program Multiple Data Paradigm

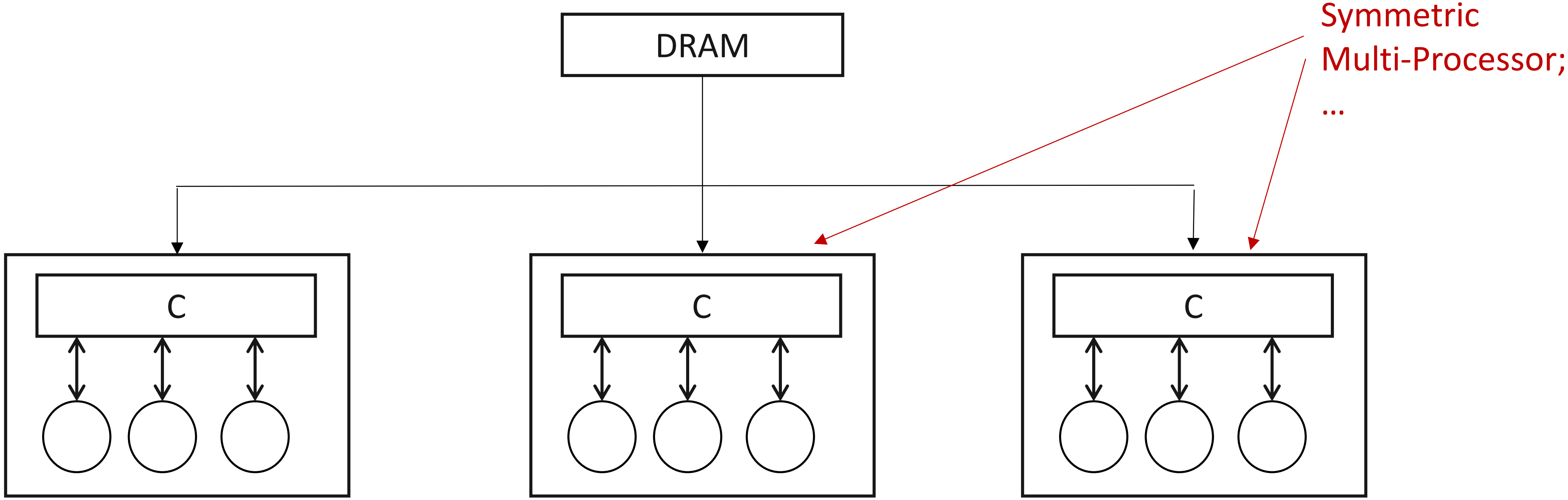
Modeling GPU Architectures



Global Memory

- This is where we transfer data from the CPU
- Nvidia A100 – 80 GB; Nvidia H100 – 96 GB

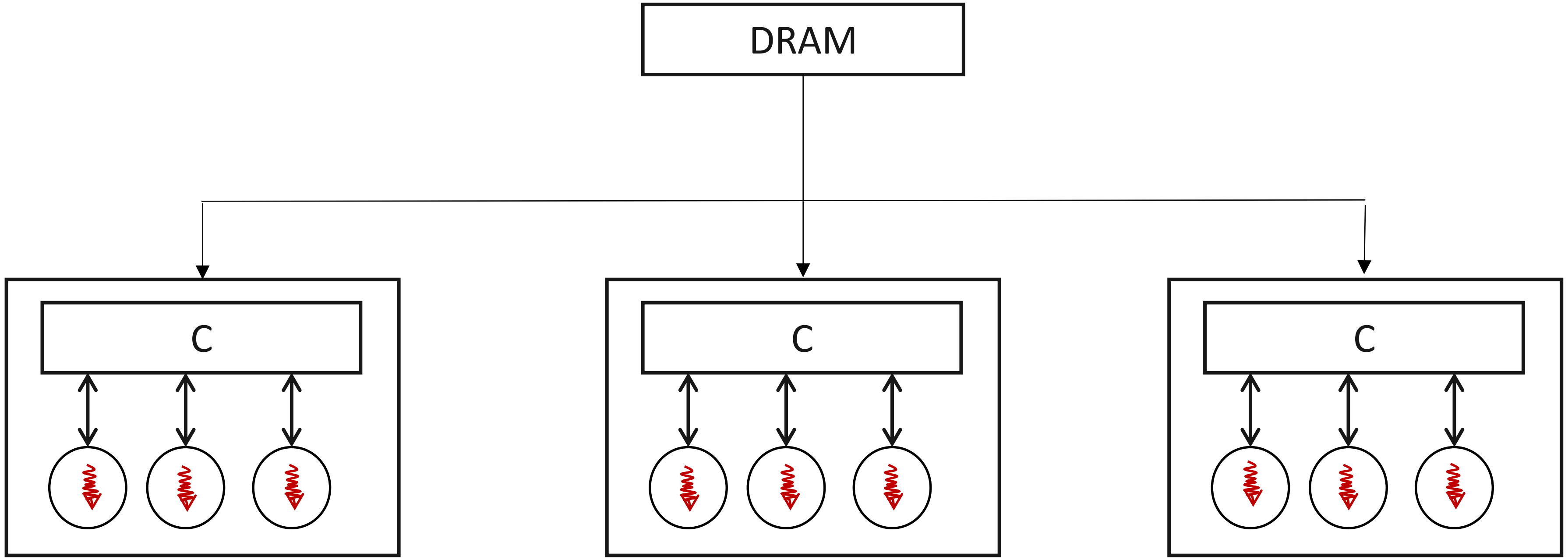
Modeling GPU Architectures



Symmetric Multi Processor

- An array of compute cores working in lockstep manner
- Have access to a shared cache
- Each compute core can access any location in the shared cache, even if the location was written by a different compute core

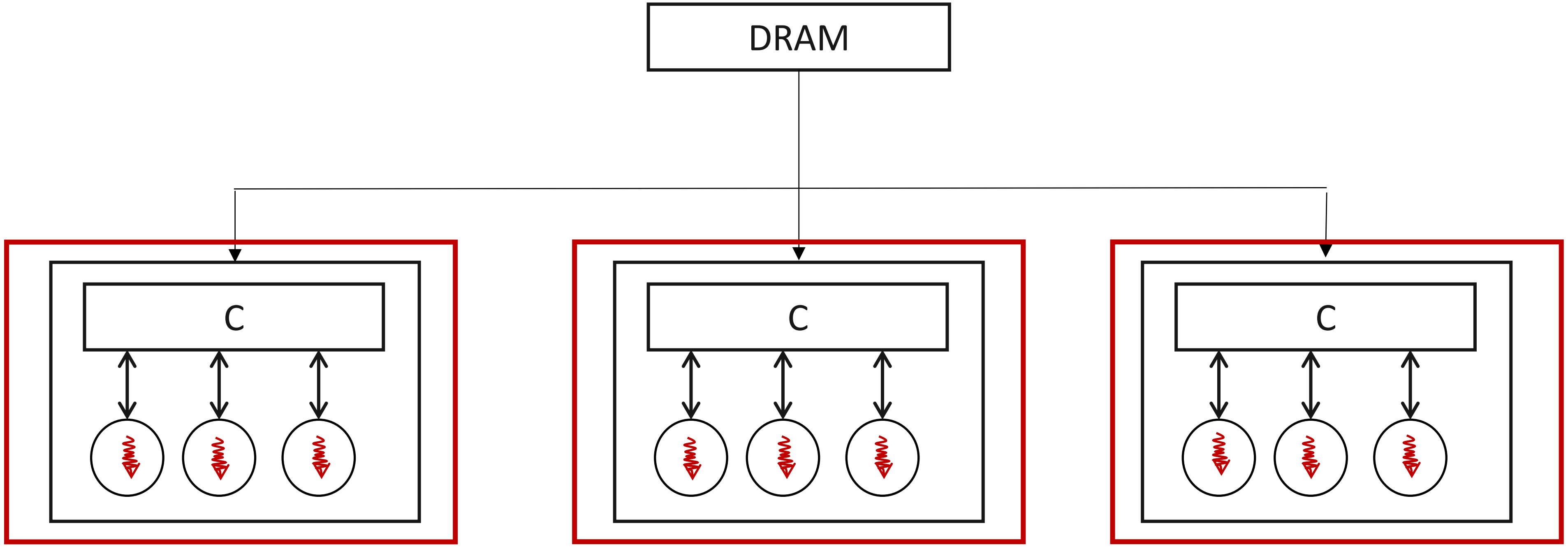
Modeling GPU Architectures



Threads and Blocks

- Threads run on compute cores

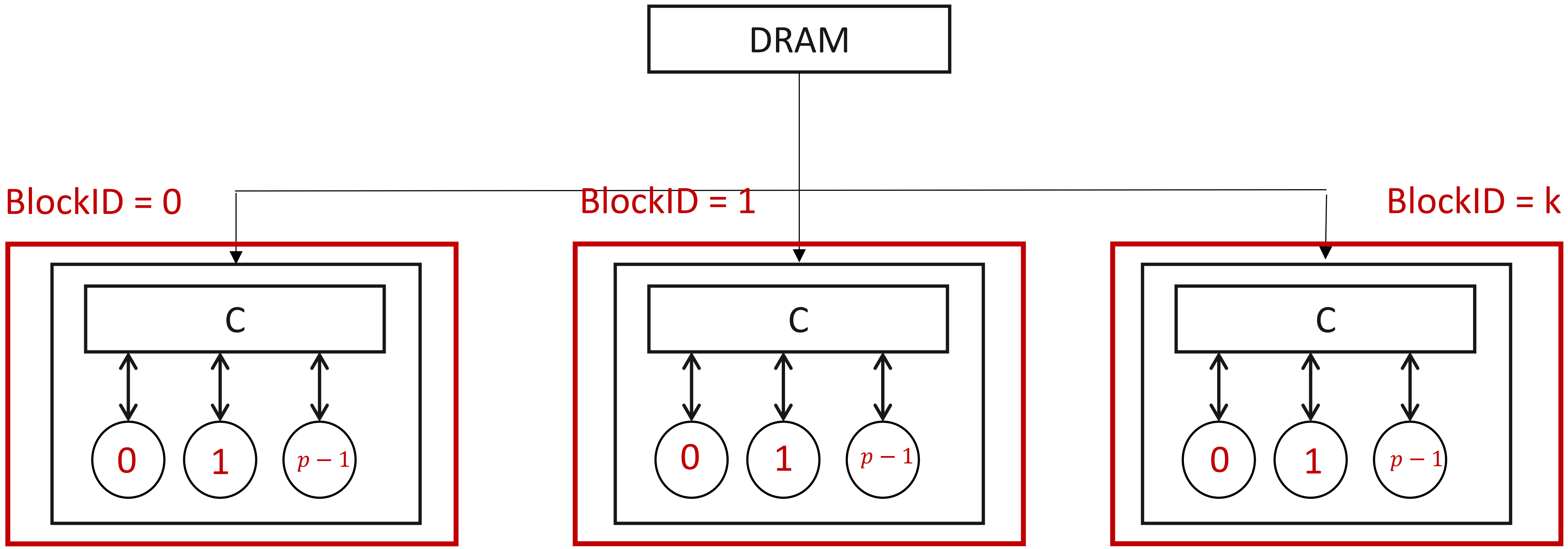
Modeling GPU Architectures



Threads and Blocks

- Threads run on compute cores
- Collection of threads is called a block
- Blocks run on Symmetric Multi-Processor (SMP)

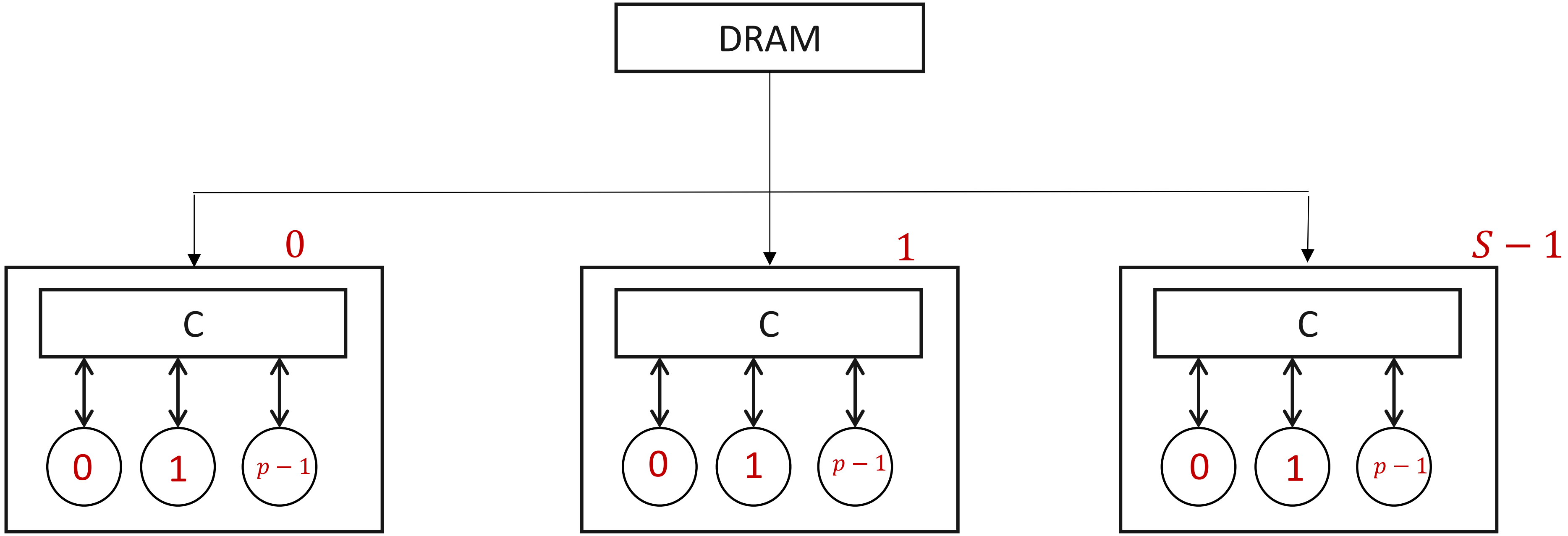
Modeling GPU Architectures



Threads and Blocks

- Threads run on compute cores
- Collection of threads is called a block
- Blocks run on Symmetric Multi-Processor (SMP)
- Each block associated with a BlockID
  - Each thread associated with a ThreadID
  - Unique within a block, but threads across different blocks can have same ThreadID

Modeling GPU Architectures



- Number of Blocks can be (in fact should be) greater than the number of SMPs
- Number of threads per block need not be equal to the number of compute cores per SMP
  - It is good to have it as a multiple though
- We will discuss these considerations when we discuss GPU programming later

- For now, assume the simple model above.
  - S Blocks
  - p Threads per block

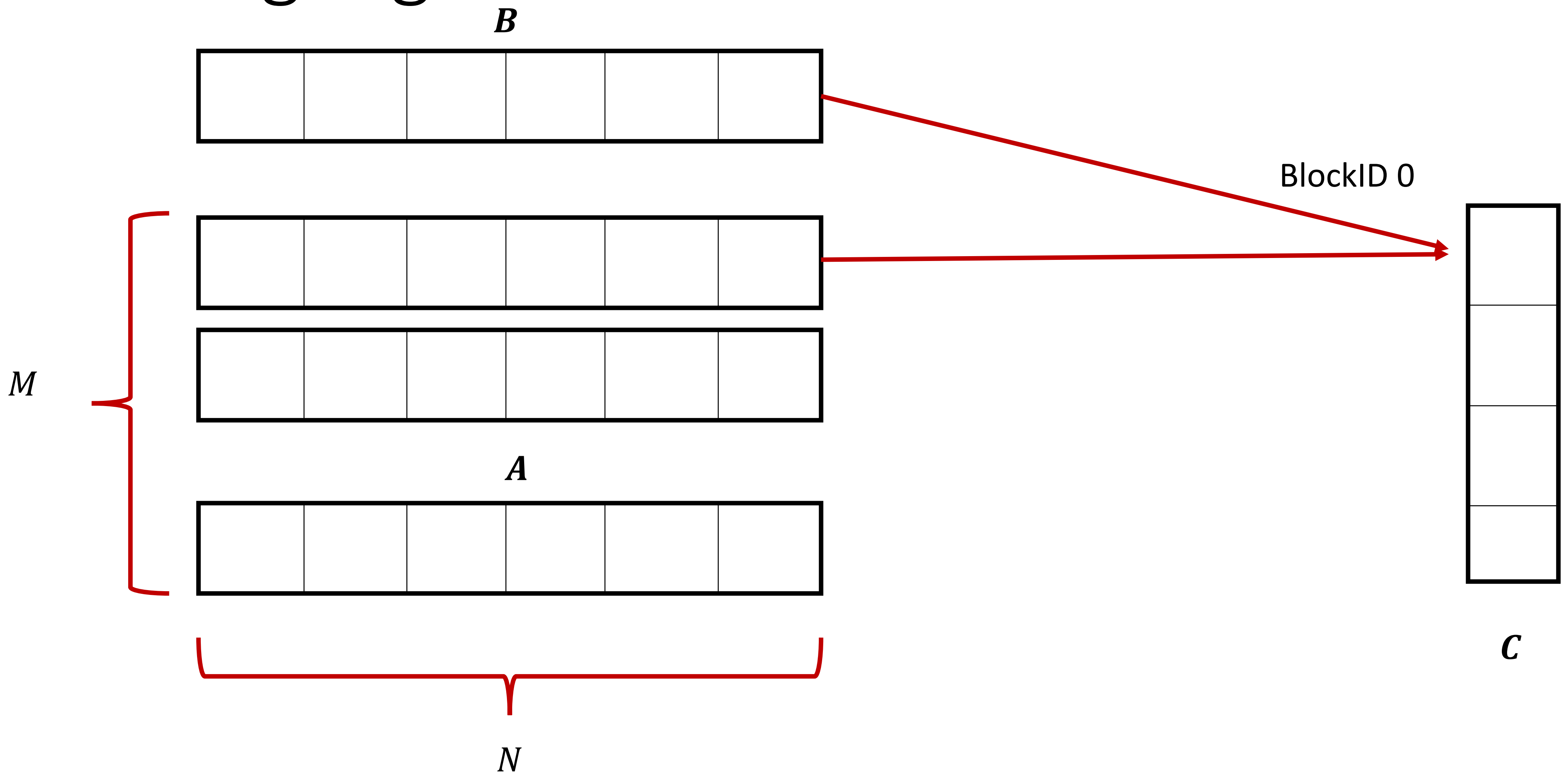
GPU Model
Calculating Algorithm Performance

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

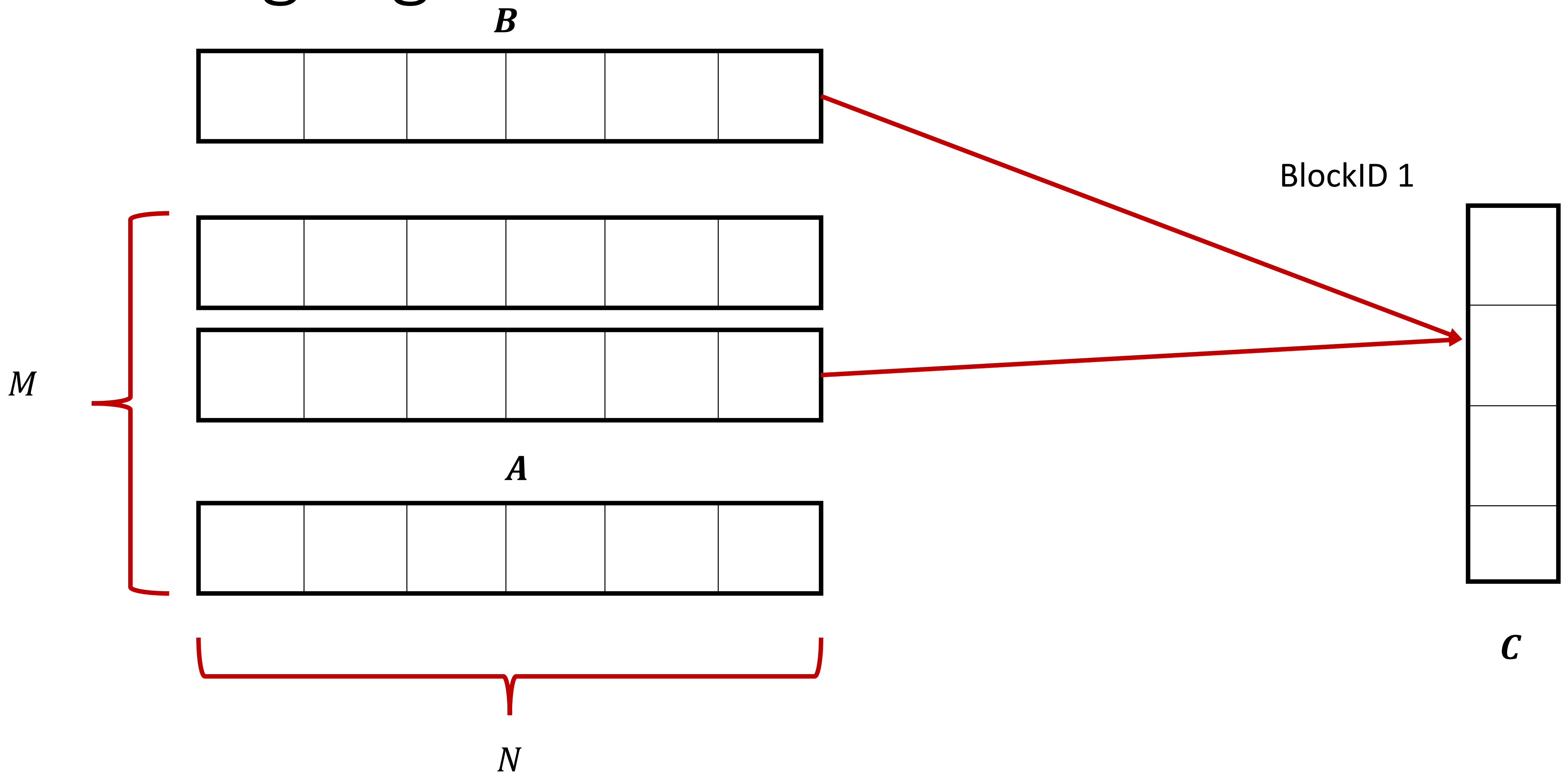
- $S \geq M, p \geq N$

- A not so good algorithm
  - BlockID i responsible for computing $C[i]$
  - ThreadID k computes the product of $A[i][k] * B[k]$ and store into Temp[k]
  - ThreadID 0 computes the sum $\sum_k Temp[k]$ and stores into C[i]

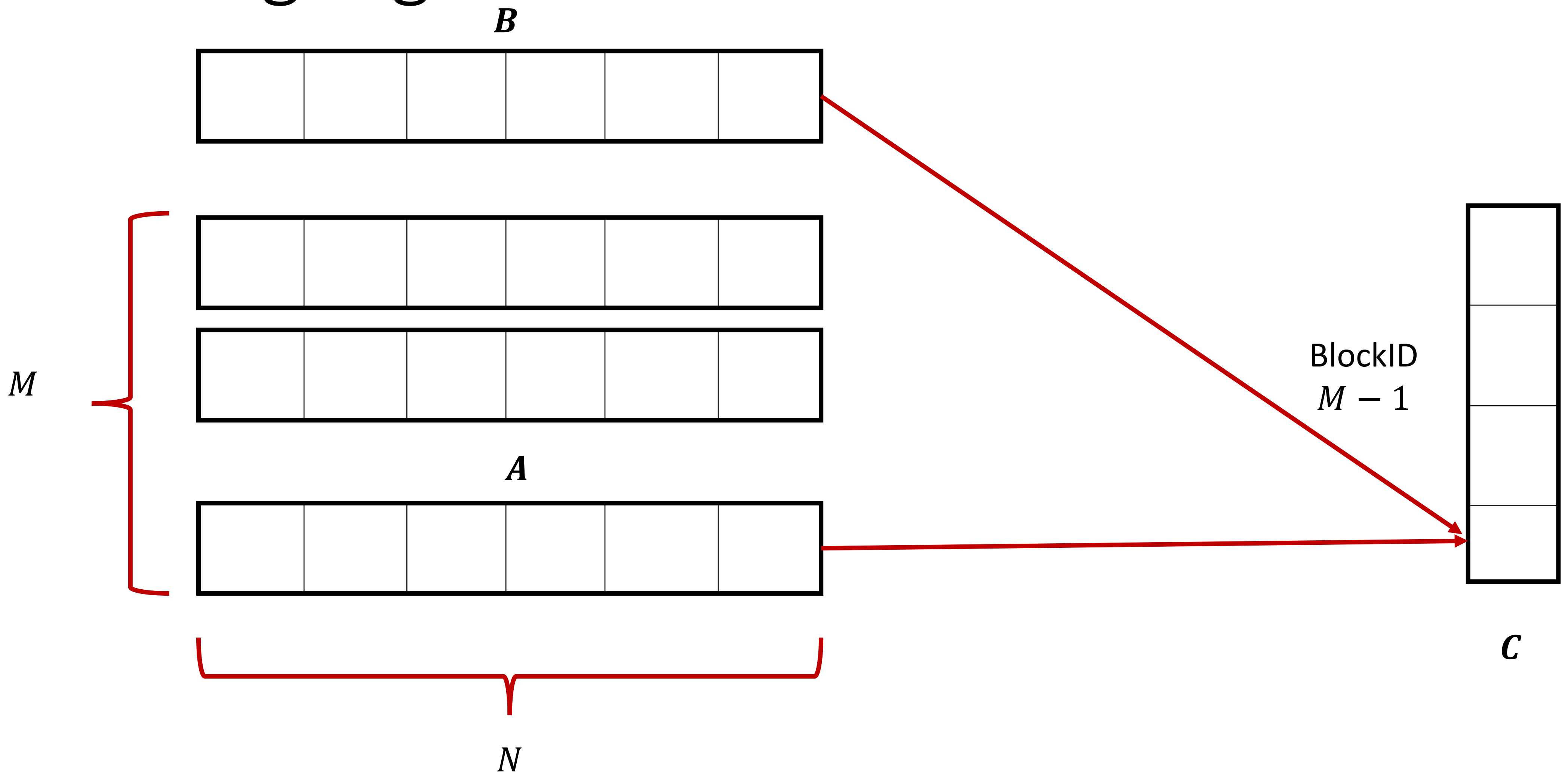
GPU Model
Calculating Algorithm Performance



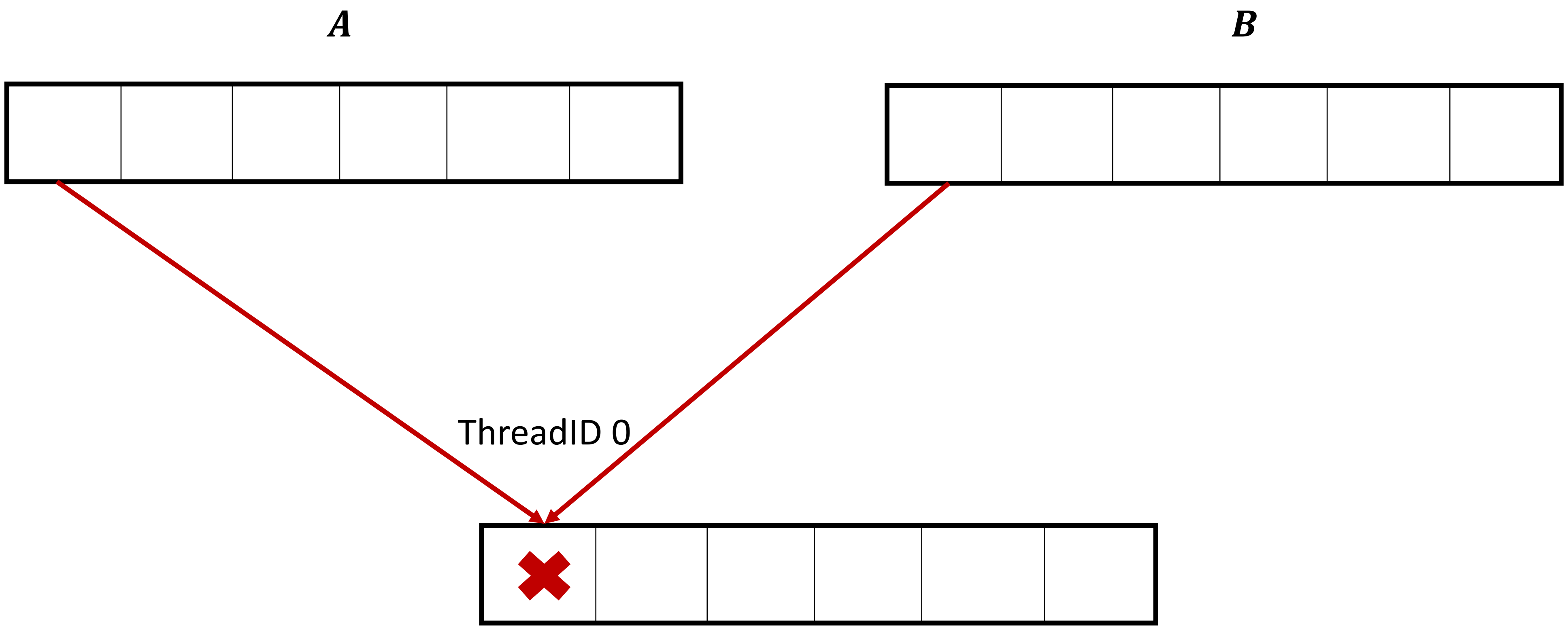
GPU Model
Calculating Algorithm Performance



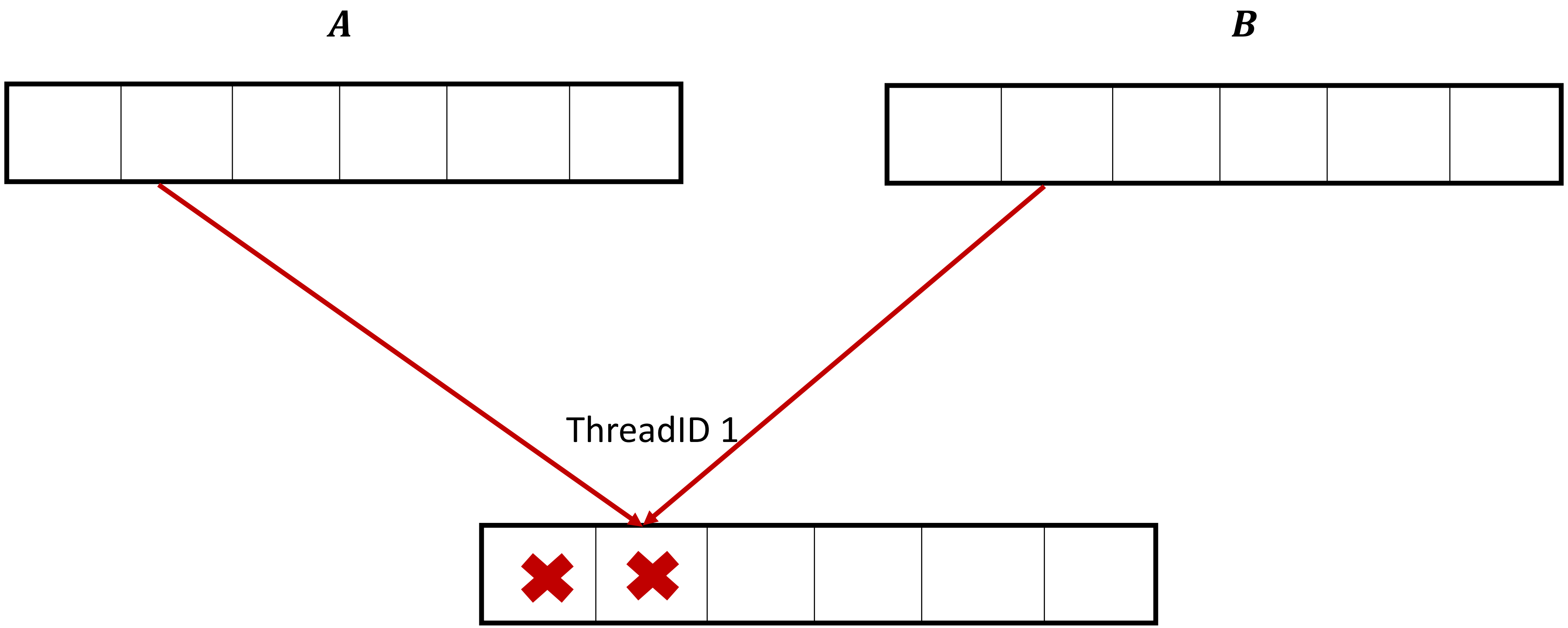
GPU Model
Calculating Algorithm Performance



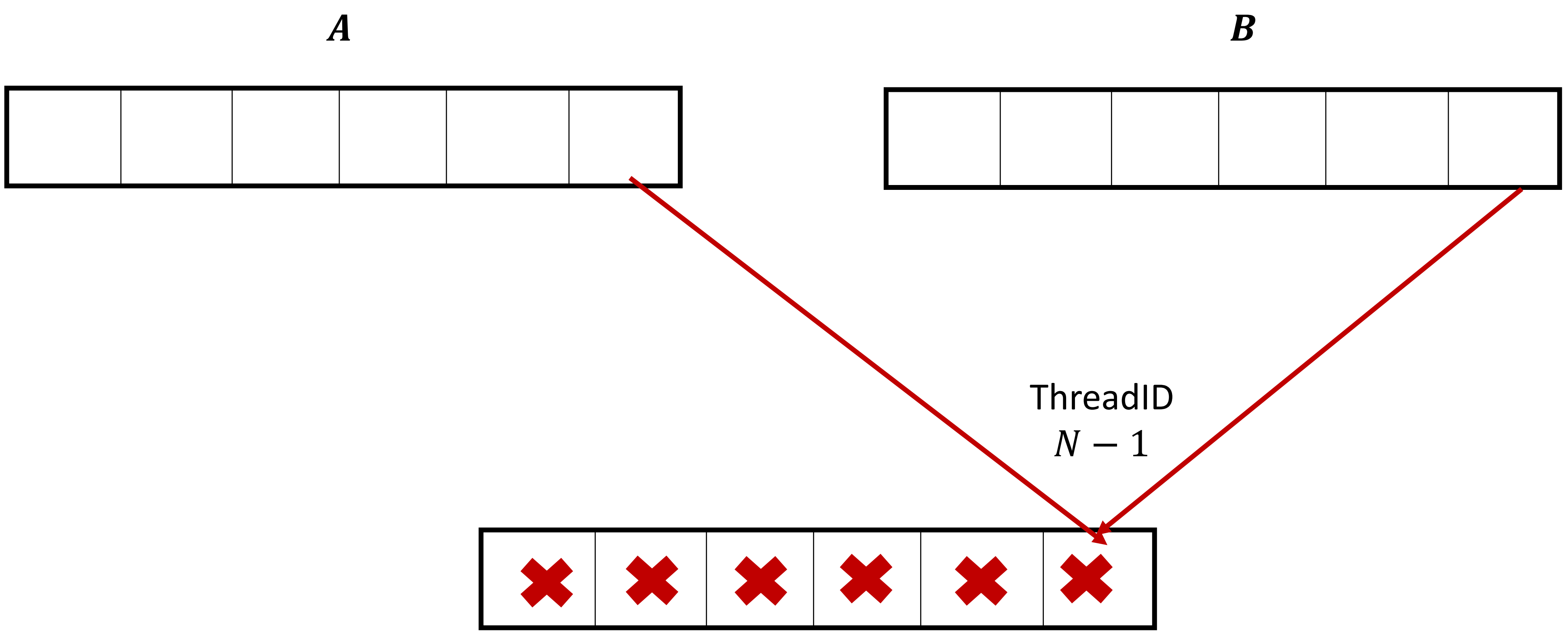
GPU Model
Calculating Algorithm Performance



GPU Model
Calculating Algorithm Performance

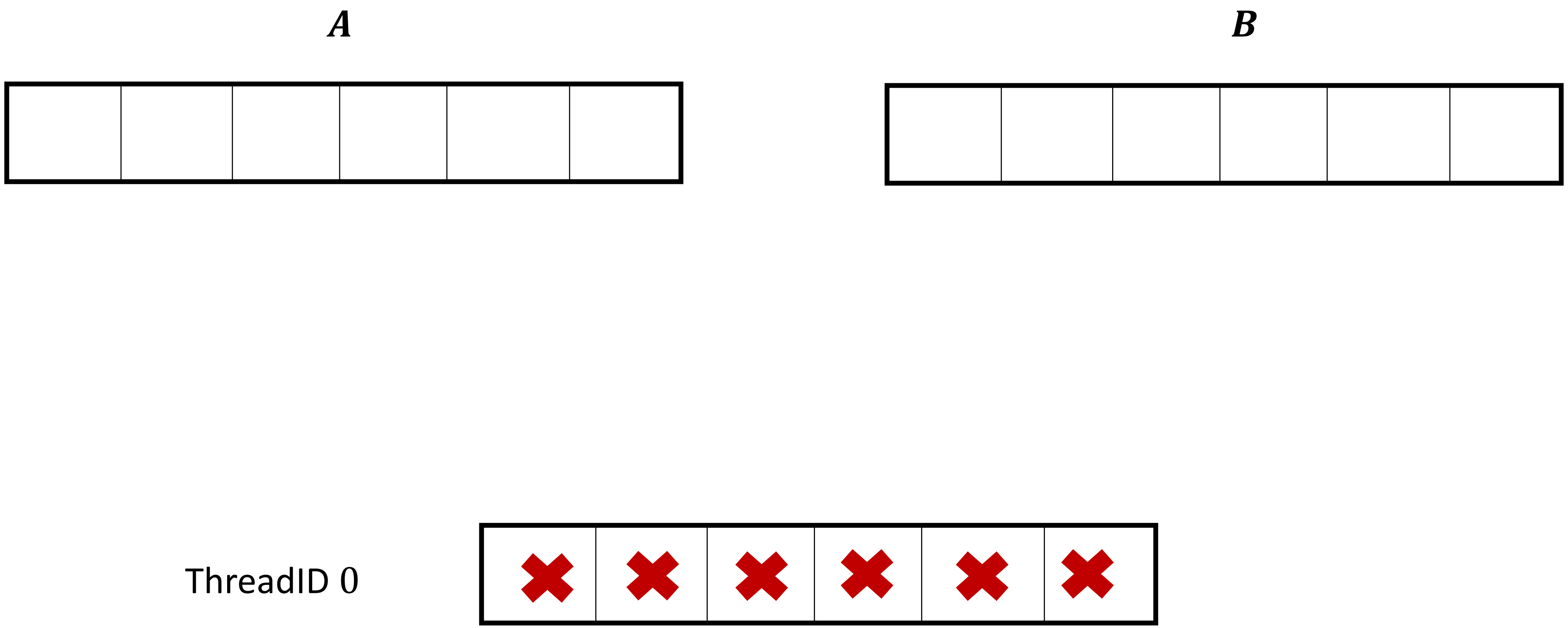


GPU Model
Calculating Algorithm Performance

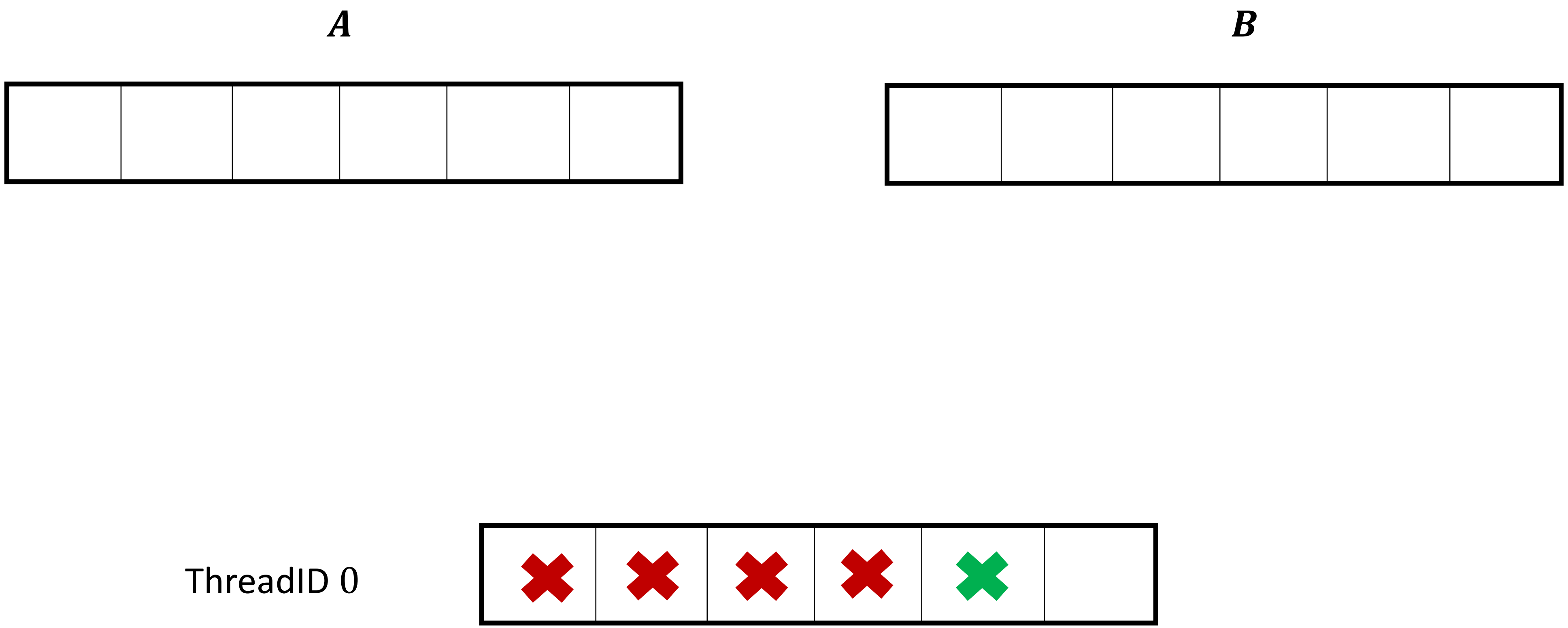


CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

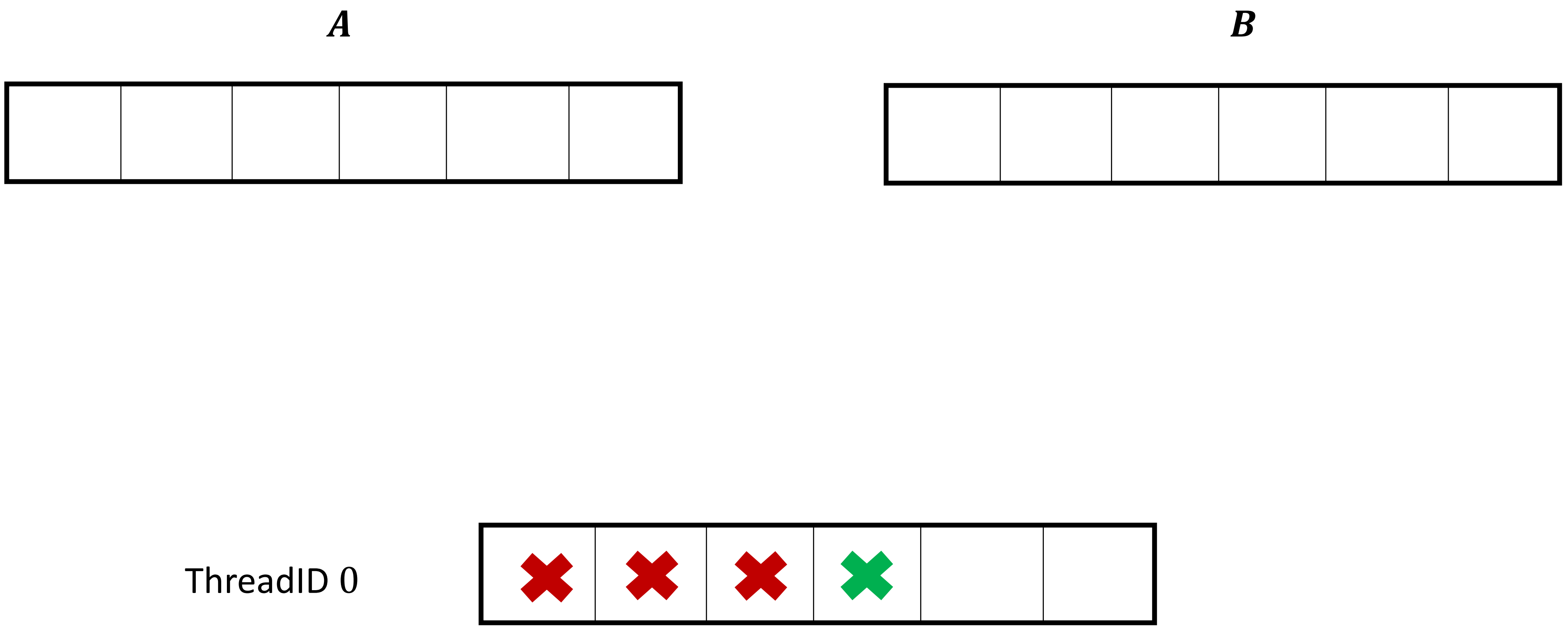
GPU Model
Calculating Algorithm Performance



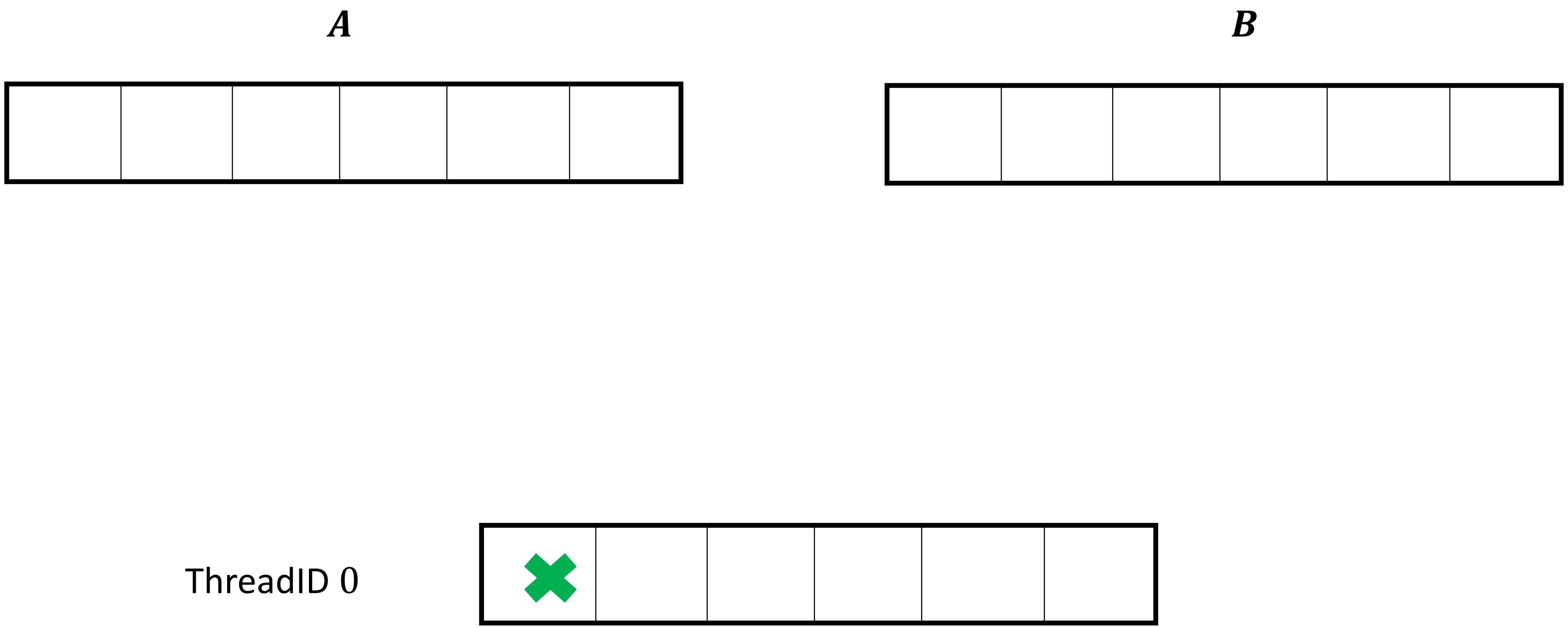
GPU Model
Calculating Algorithm Performance



GPU Model
Calculating Algorithm Performance



GPU Model
Calculating Algorithm Performance



GPU Model
Calculating Algorithm Performance

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

- A not so good algorithm
  - BlockID i responsible for computing $C[i]$
  - ThreadID k computes the product of $A[i][k] * B[k]$ and store into Temp[k]
  - ThreadID 0 computes the sum $\sum_k Temp[k]$ and stores into C[i]

- Why is this not so good?

GPU Model
Calculating Algorithm Performance

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

- A not so good algorithm
  - BlockID i responsible for computing $C[i]$
  - ThreadID k computes the product of $A[i][k] * B[k]$ and store into Temp[k]
  - ThreadID 0 computes the sum $\sum_k Temp[k]$ and stores into C[i]

- Why is this not so good? When adding, all other threads are idling

GPU Model
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$
- Assume $S \geq M, p \geq N$

- In thread: bid, tid

RMA: Read A[bid][tid] into cache

RMB: Read B[tid] into cache

Syncthreads()

RA: Read A[bid][tid] into processor

RB: Read B[tid] into processor

M: Temp[tid] <- Mult A[bid][tid]*B[tid]

syncthreads()

- In thread 0

{ Repeat for iter = 1 to $N - 1$ times

A: Temp[0] = Temp[0] + T[iter]

C[bid] = Temp[0]

}

SMC: Store C[bid] into DRAM

GPU Model
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$
- Assume $S \geq M, p \geq N$

- In thread $\boxed{\text{bid, tid}}$

RMA: Read A[bid][tid] into cache

RMB: Read B[tid] into cache

Syncthreads()

RA: Read A[bid][tid] into processor

RB: Read B[tid] into processor

M: Temp[tid] <- Mult A[bid][tid]*B[tid]

syncthreads()

- In thread 0

{ Repeat for iter = 1 to $N - 1$ times

A: Temp[0] = Temp[0] + T[iter]

C[bid] = Temp[0]

}

SMC: Store C[bid] into DRAM

Notice how the code is bid, tid dependent.

GPU Model
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$
- Assume $S \geq M, p \geq N$

- In thread bid, tid

RMA: Read A[bid][tid] into cache
RMB: Read B[tid] into cache
Syncthreads()
RA: Read A[bid][tid] into processor
RB: Read B[tid] into processor
M: Temp[tid] <- Mult A[bid][tid]*B[tid]
syncthreads()

- In thread 0

{ Repeat for iter = 1 to $N - 1$ times
A: Temp[0] = Temp[0] + T[iter]
C[bid] = Temp[0]
}
SMC: Store C[bid] into DRAM

Also, notice how each thread performs the same computation but on different data (SPMD)

CASE SCHOOL
OF ENGINEERING
CASE WESTERN RESERVE
UNIVERSITY

GPU Model
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$
- Assume $S \geq M, p \geq N$

- In thread: bid, tid

RMA: Read A[bid][tid] into cache
RMB: Read B[tid] into cache
Syncthreads()
RA: Read A[bid][tid] into processor
RB: Read B[tid] into processor
M: Temp[tid] <- Mult A[bid][tid]*B[tid]
syncthreads()

- In thread 0

{ Repeat for iter = 1 to $N - 1$ times
A: Temp[0] = Temp[0] + T[iter]
C[bid] = Temp[0]
}
SMC: Store C[bid] into DRAM

All threads in the block collectively bring data into the shared cache - prefetching

GPU Model
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$
- Assume $S \geq M, p \geq N$

- In thread: bid, tid

RMA: Read A[bid][tid] into cache

RMB: Read B[tid] into cache

$\boxed{\text{Syncthreads()}}$

RA: Read A[bid][tid] into processor

RB: Read B[tid] into processor

M: Temp[tid] <- Mult A[bid][tid]*B[tid]

syncthreads()

- In thread 0

{ Repeat for iter = 1 to $N-1$ times

A: Temp[0] = Temp[0] + T[iter]

C[bid] = Temp[0]

}

SMC: Store C[bid] into DRAM

A synchronization step is needed for all the threads to finish fetching

GPU Model
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$
- Assume $S \geq M, p \geq N$

- In thread: bid, tid

RMA: Read A[bid][tid] into cache

RMB: Read B[tid] into cache

Syncthreads()

RA: Read A[bid][tid] into processor

RB: Read B[tid] into processor

M: Temp[tid] <- Mult A[bid][tid]*B[tid]

syncthreads()

- In thread 0

{ Repeat for iter = 1 to $N - 1$ times

A: Temp[0] = Temp[0] + T[iter]

C[bid] = Temp[0]

}

SMC: Store C[bid] into DRAM

Computation in parallel and synchronization

GPU Model
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$
- Assume $S \geq M, p \geq N$

- In thread: bid, tid

RMA: Read A[bid][tid] into cache

RMB: Read B[tid] into cache

Syncthreads()

RA: Read A[bid][tid] into processor

RB: Read B[tid] into processor

M: Temp[tid] <- Mult A[bid][tid]*B[tid]

syncthreads()

- In thread 0

{ Repeat for iter = 1 to $N - 1$ times

A: Temp[0] = Temp[0] + T[iter]

C[bid] = Temp[0]

}

SMC: Store C[bid] into DRAM

Thread 0 performs the sum.
All other threads idle (in practice they still execute the instructions, but the outputs are invalidated)

GPU Model
Calculating Algorithm Performance

- C[i] = $\sum_k A[i][k] * B[k]$
- Assume $S \geq M, p \geq N$

- In thread: bid, tid

RMA: Read A[bid][tid] into cache

RMB: Read B[tid] into cache

Syncthreads()

RA: Read A[bid][tid] into processor

RB: Read B[tid] into processor

M: Temp[tid] <- Mult A[bid][tid]*B[tid]

syncthreads()

- In thread 0

{ Repeat for iter = 1 to $N-1$ times

A: Temp[0] = Temp[0] + T[iter]

C[bid] = Temp[0]

}

SMC: Store C[bid] into DRAM

Ungraded HW assignment: Calculate the system performance metrics for this code.

Assume hardware parameters for memory system with cache that we used in last class. Assume syncthreads is 0 cycles.

Q: What is the new peak performance?

Q: What is the sustained performance you obtain here?

GPU Model

- Think in terms of what each block computes, and what each thread within a block computes
- Each thread on the GPU executes the *same instructions* on *different data*
- We need to rethink our algorithms in terms of the above
- We will look at a few more GPU algorithms in Lecture 6

Outline

- Processor Memory Architectures
  - Blocked Matrix Multiplication
- Modeling GPU Architectures
- Data Parallel Programming

Data Parallelism

- Key Idea:
  - Partition data into chunks – to ensure load balancing
  - Assign tasks to each partition so that they can proceed concurrently

- Note: data parallelism doesn’t mean each task “need” to perform exact same computations
  - Usually it is true

- SPMD (Single Program Multiple Data) is a special case of data parallelism where same “program” is run on different data

Data Parallelism - Approaches

- #1 Partition Output data
  - Each task is responsible for computing an output partition

Output

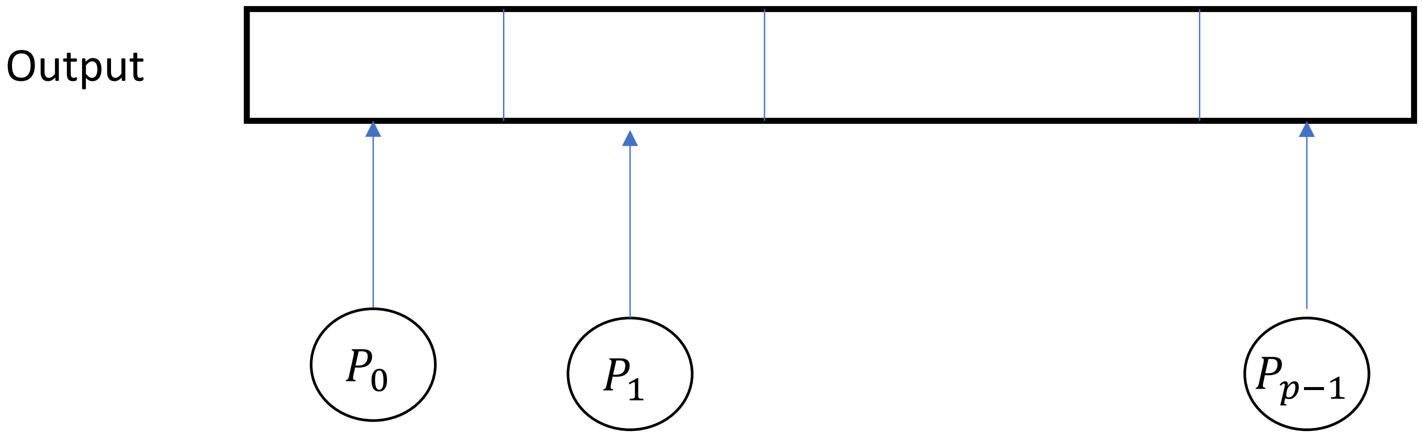
Data Parallelism - Approaches

- #1 Partition Output data
  - Each task is responsible for computing an output partition

Output

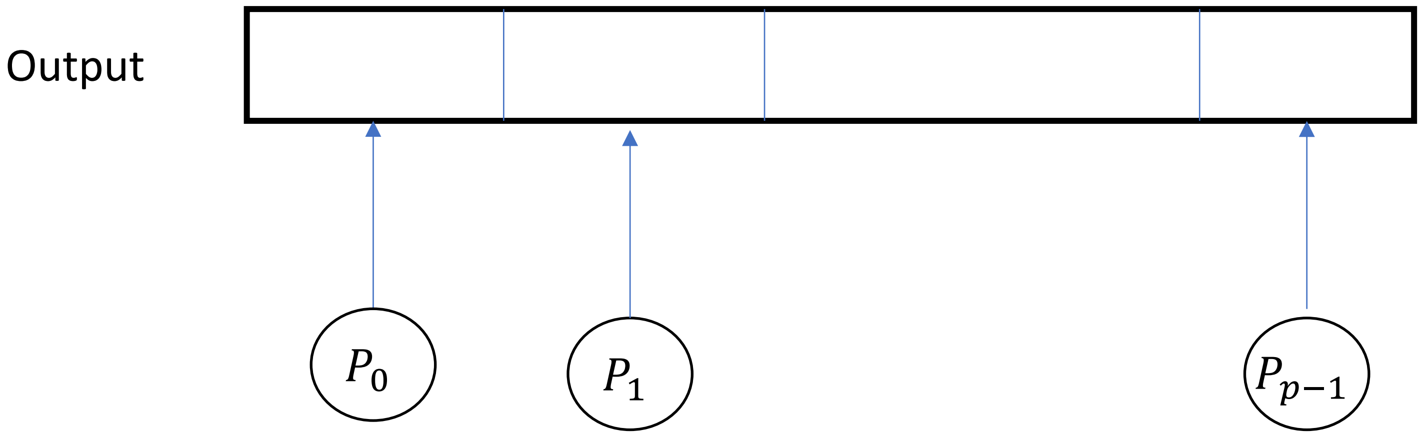
Data Parallelism - Approaches

- #1 Partition Output data
  - Each task is responsible for computing an output partition



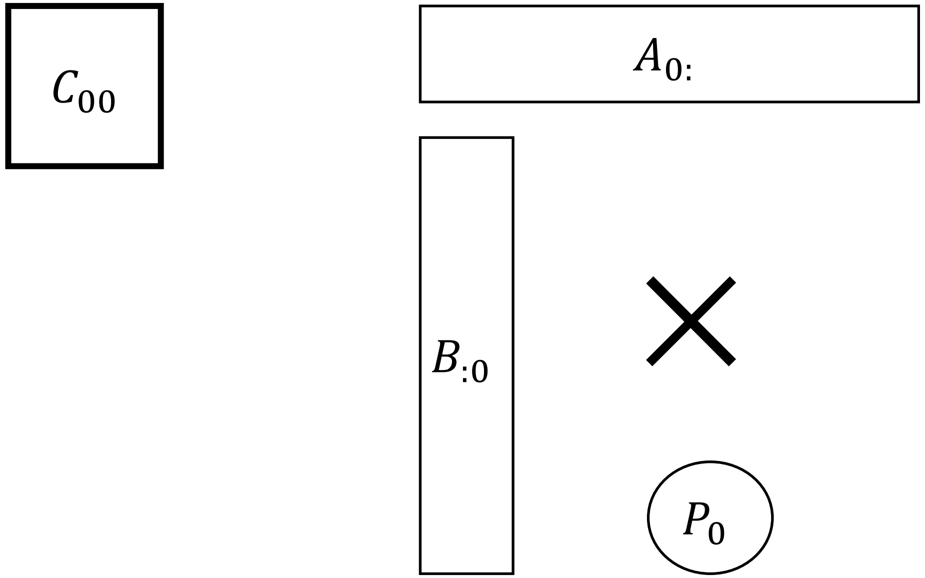
Data Parallelism - Approaches

- #1 Partition Output data
  - Useful when partitions of output can be computed independently of each other
  - Example: Blocked Matrix Multiplication

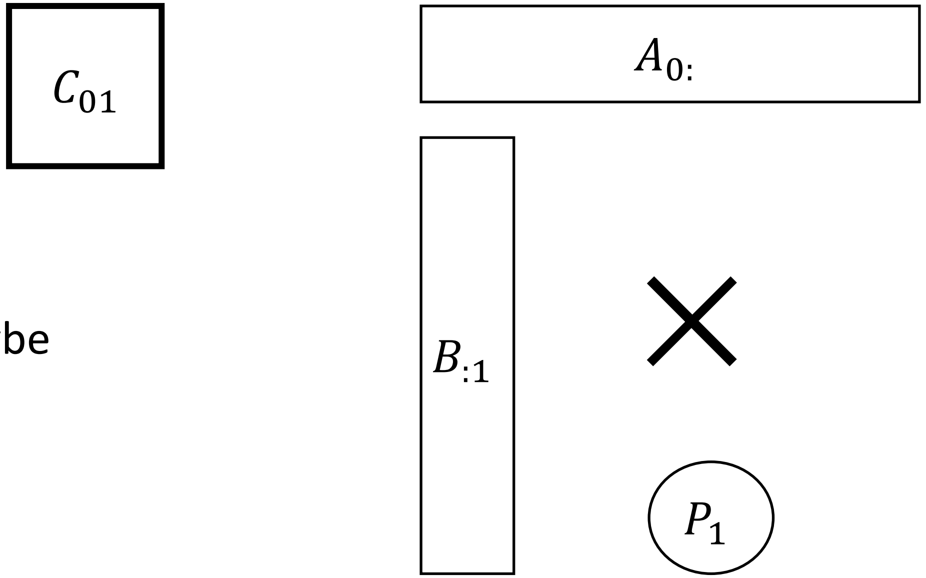


Data Parallelism – Approaches

- #1 Partition Output data
  - Matrix Multiplication



- Input data maybe shared
- Output data distinct



Data Parallelism - Approaches

- #2 Partition Input data
    - Each task is responsible for performing “all” operations on the partition

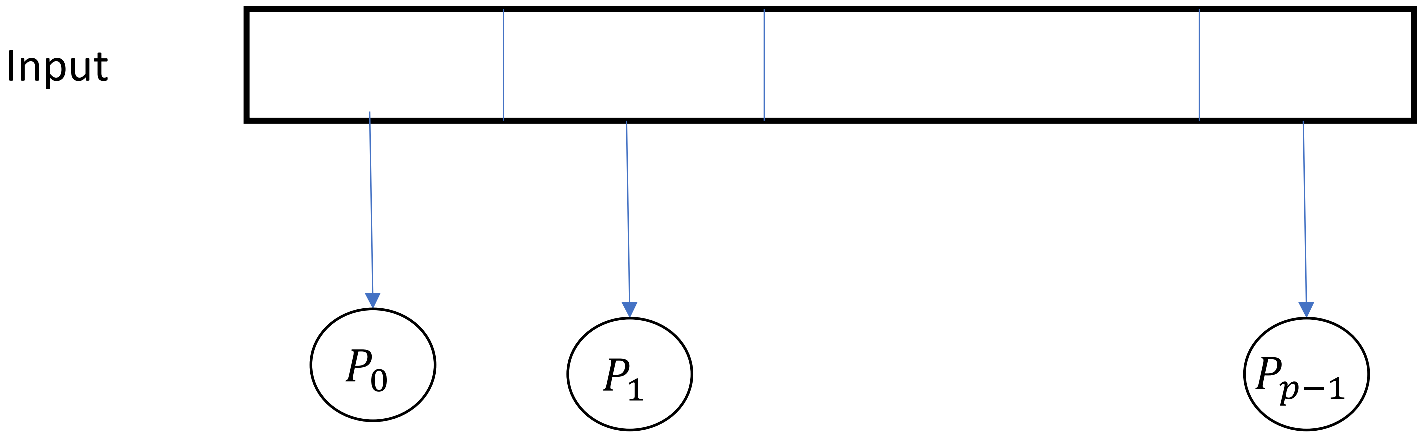
Data Parallelism - Approaches

- #2 Partition Input data
  - Each task is responsible for performing “all” operations on the partition

Input

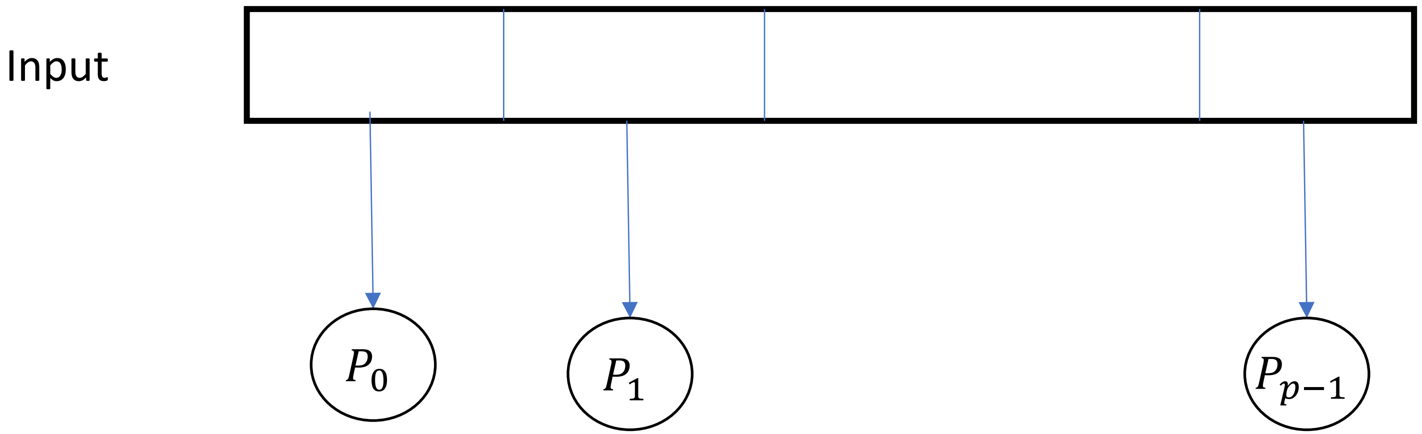
Data Parallelism - Approaches

- #2 Partition Input data
  - Each task is responsible for performing “all” operations on the partition



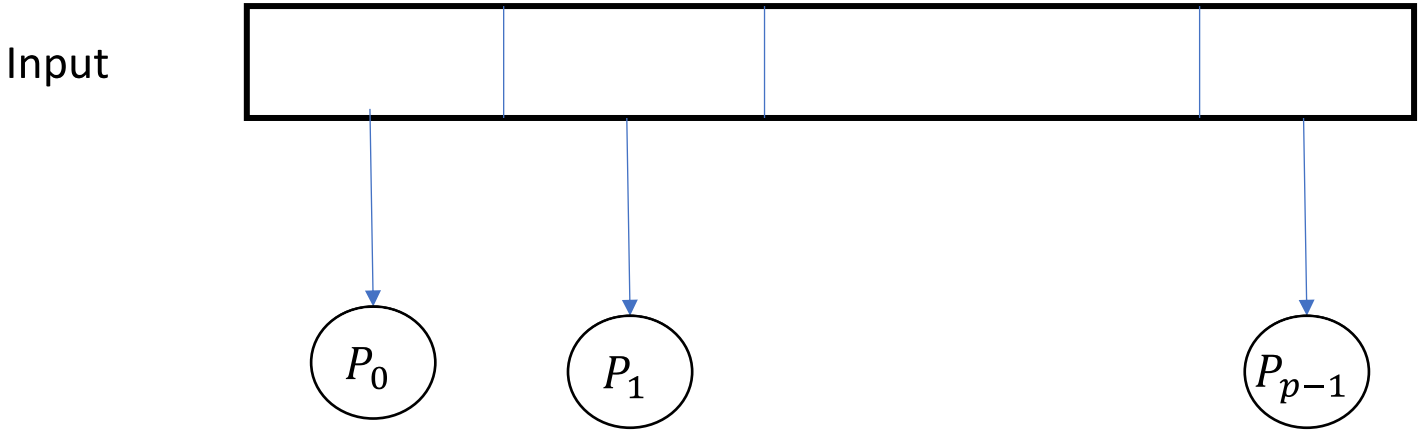
Data Parallelism - Approaches

- #2 Partition Input data
  - Tasks can either directly produce the final output
  - Or produce intermediate output which require another set of tasks to produce the final output



Data Parallelism - Approaches

- #2 Partition Input data
  - Suitable when outputs cannot be independently computed
    - Example: Aggregation functions, Hashing, …



Data Parallelism - Approaches

- #2 Partition Input data
- Problem: Constructing a Hash Table
- Inputs: a set of keys in the Universe U
- Output: mapping of each key to an index in $\{0,1,\ldots,m-1\}$

Data Parallelism - Approaches

- #2 Partition Input data

- Problem: Constructing a Hash Table

- Can we do data parallelism based on output partitioning?

Data Parallelism - Approaches

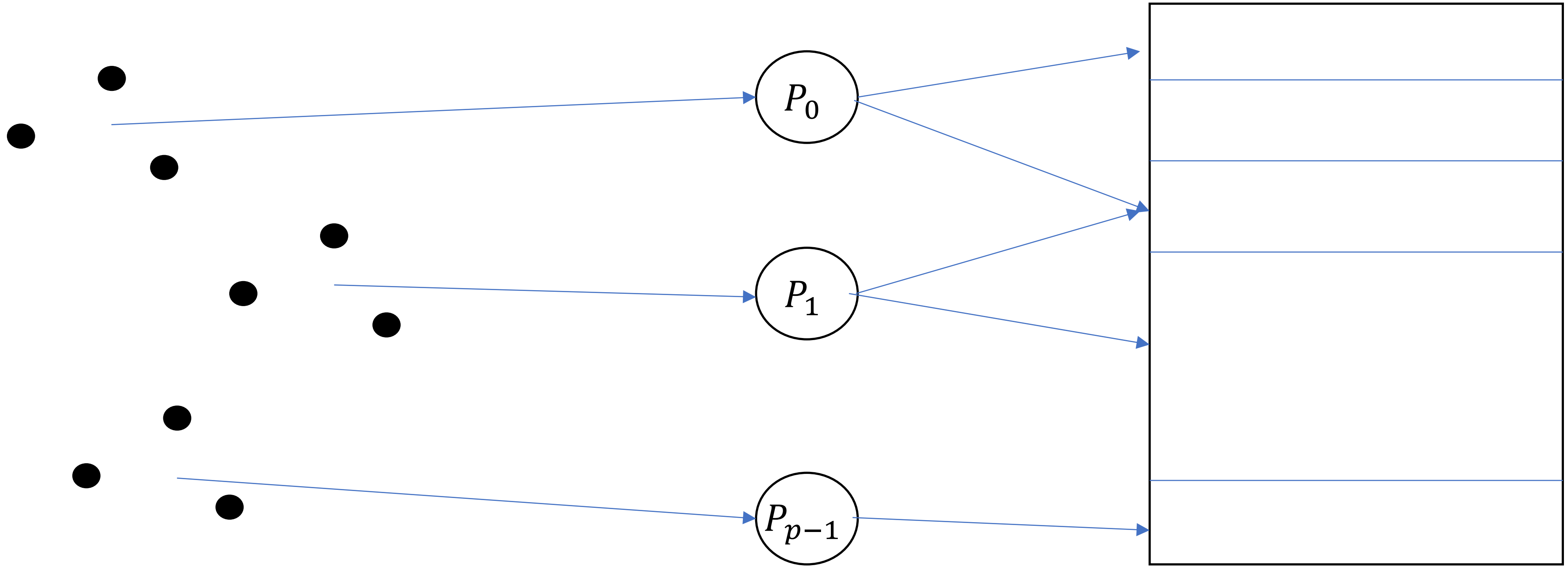
- #2 Partition Input data

- Problem: Constructing a Hash Table

- Can we do data parallelism based on output partitioning?
  - Yes, but
  - What if the partition has no keys mapped to it – idle processor
  - What if the partition has too many keys mapped to it – load balancing issue
  - Does each processor need to look into the entire dataset? – wasteful computations

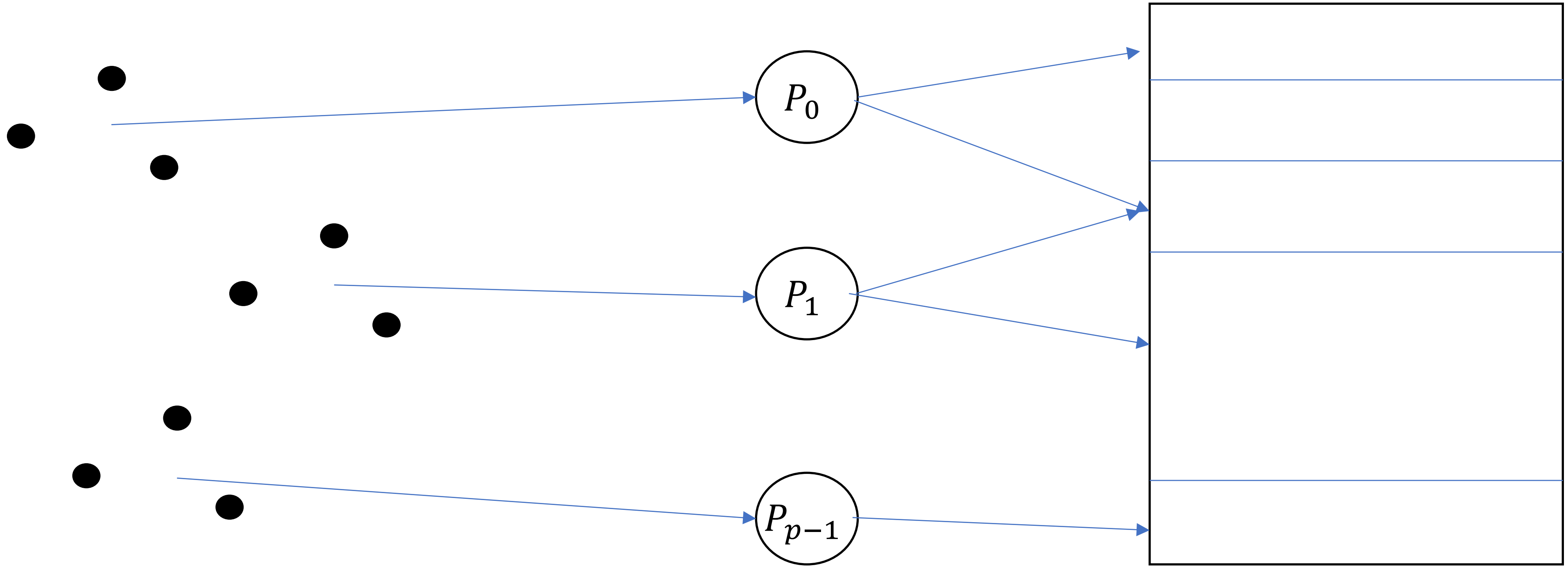
Data Parallelism - Approaches

- #2 Partition Input data
  - Divide the keys equally among processors



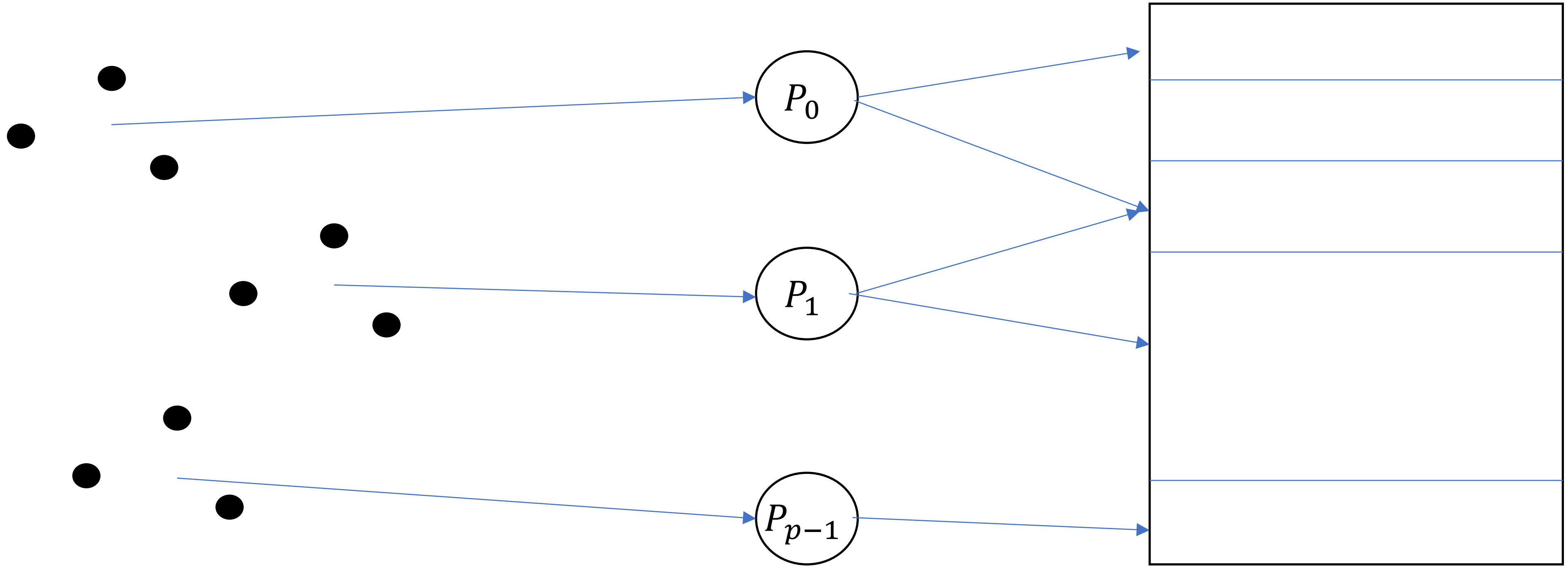
Data Parallelism - Approaches

- #2 Partition Input data
  - Pros: no wasteful computations, no idle processors



Data Parallelism - Approaches

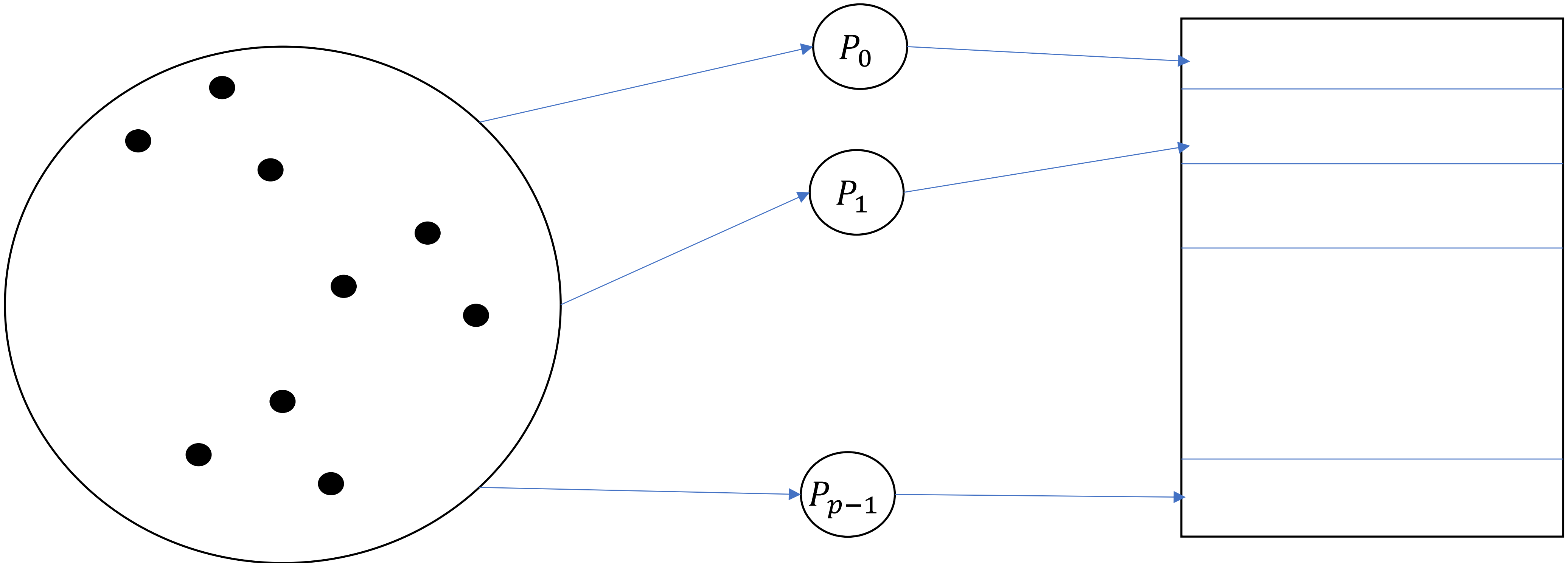
- #2 Partition Input data
  - Cons: May need to do synchronization
    - Load balancing issues – Processors that see more conflicts will complete later than processors that see fewer conflicts



Data Parallelism - Approaches

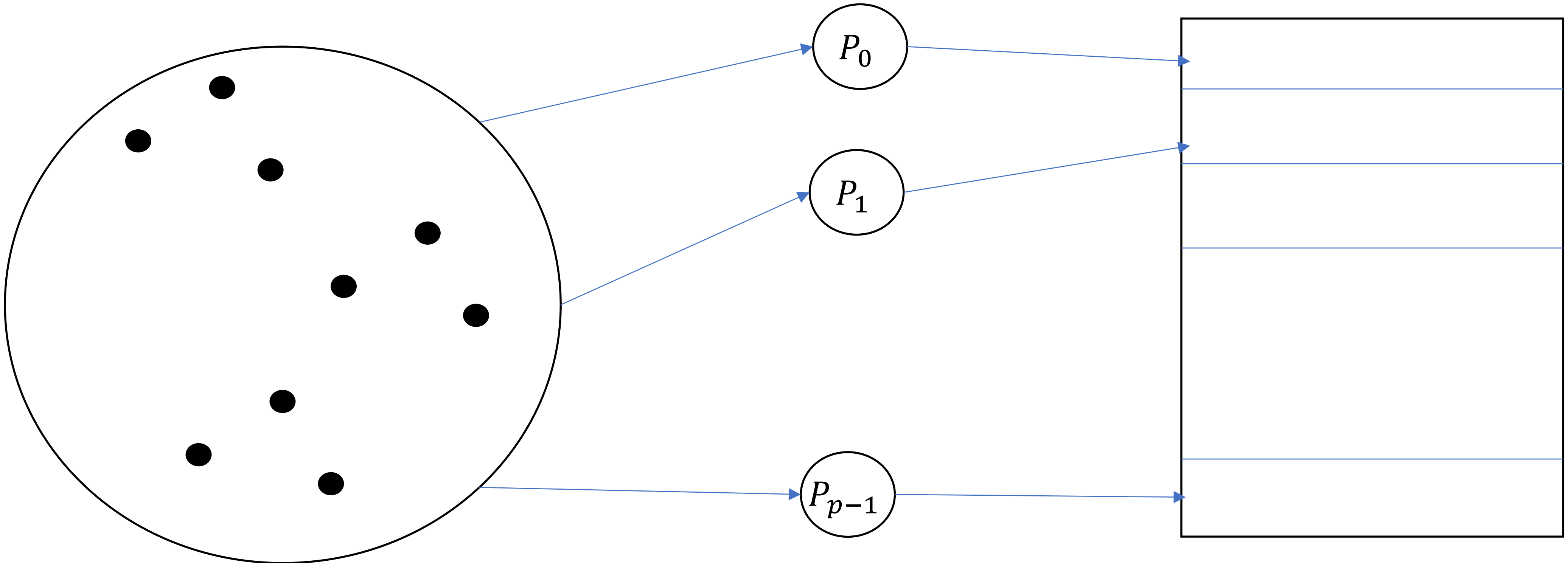
- #3 Partition both Input and Output data

- Lets revisit the output partitioning in hash tables in detail



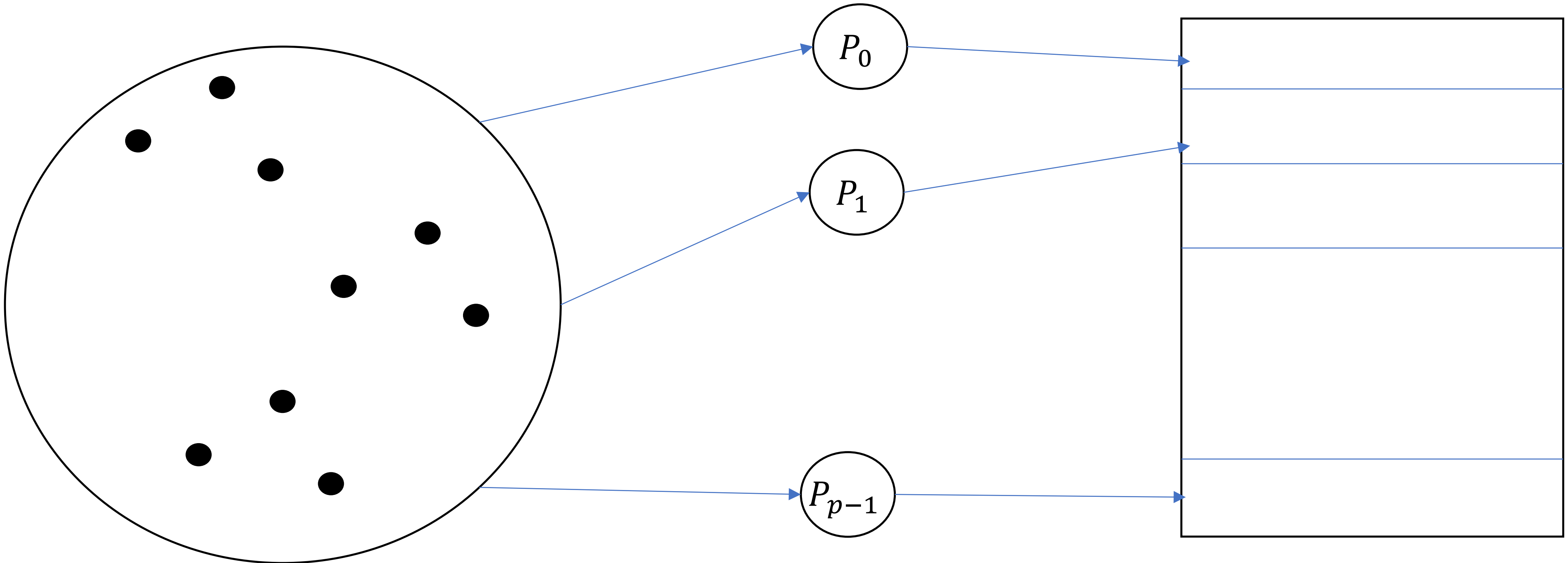
Data Parallelism - Approaches

- Cons: wasteful computations
  - Notice: Hash value of each key computed p times



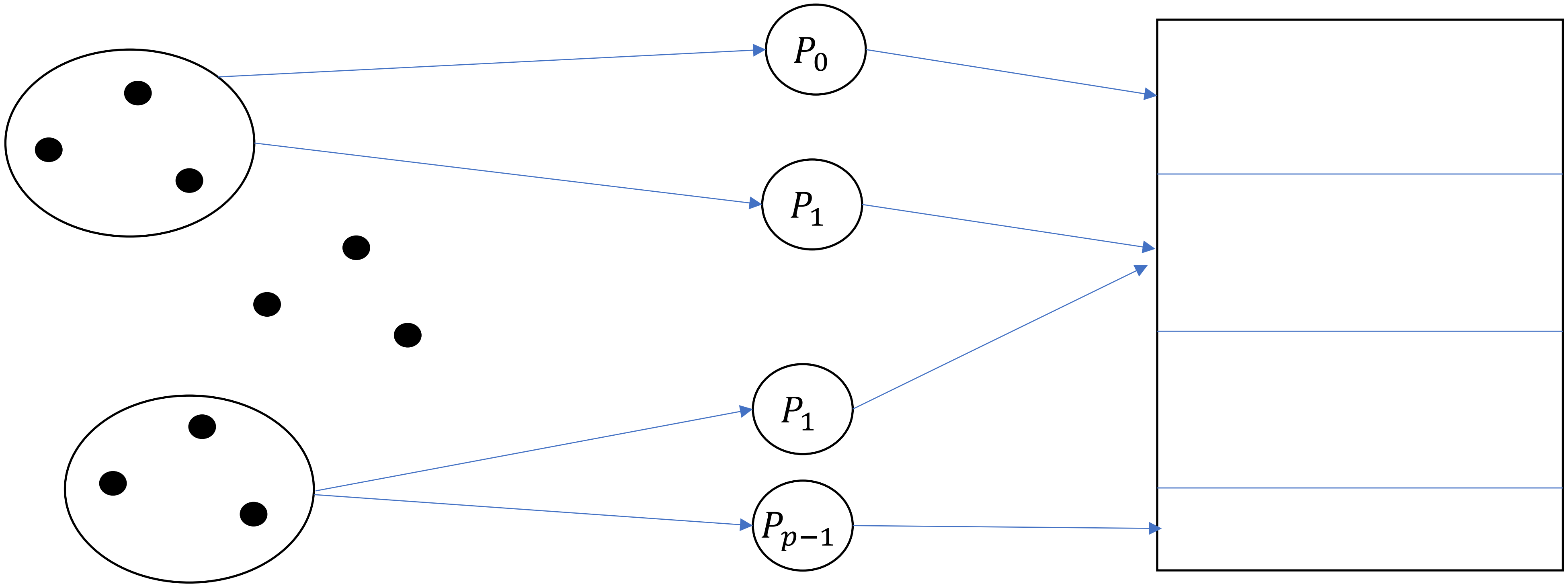
Data Parallelism - Approaches

- Pros: No synchronization



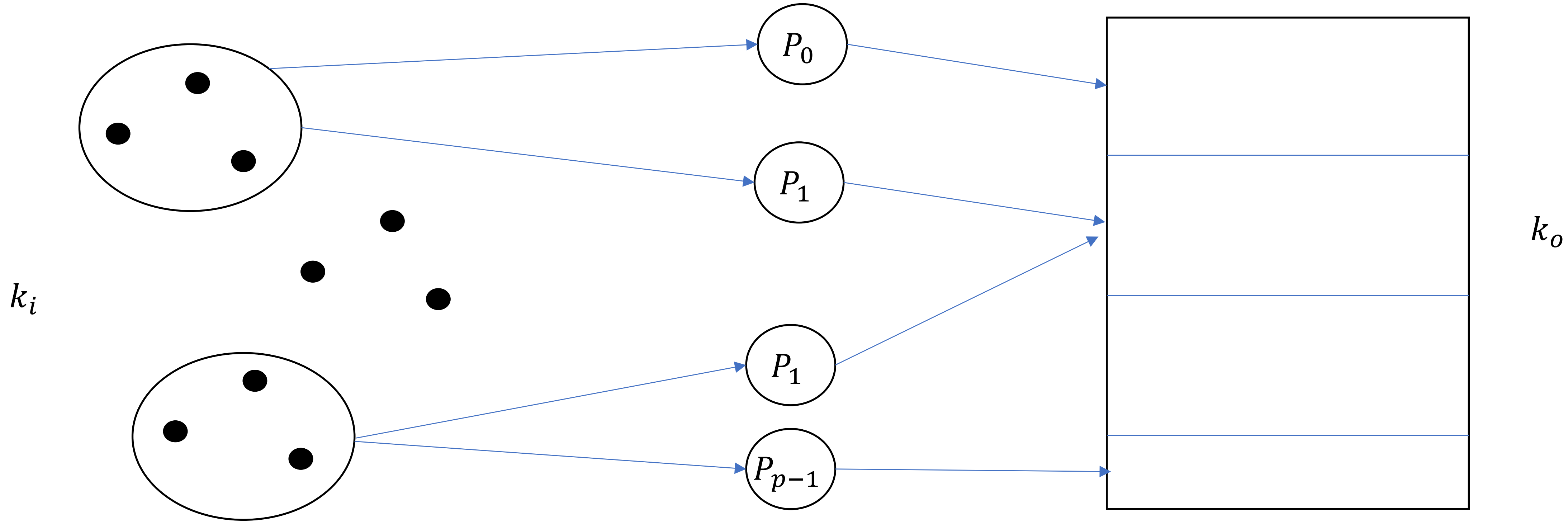
Data Parallelism - Approaches

- #3 Partition both inputs and outputs
  - For each input output pair, assign a processor



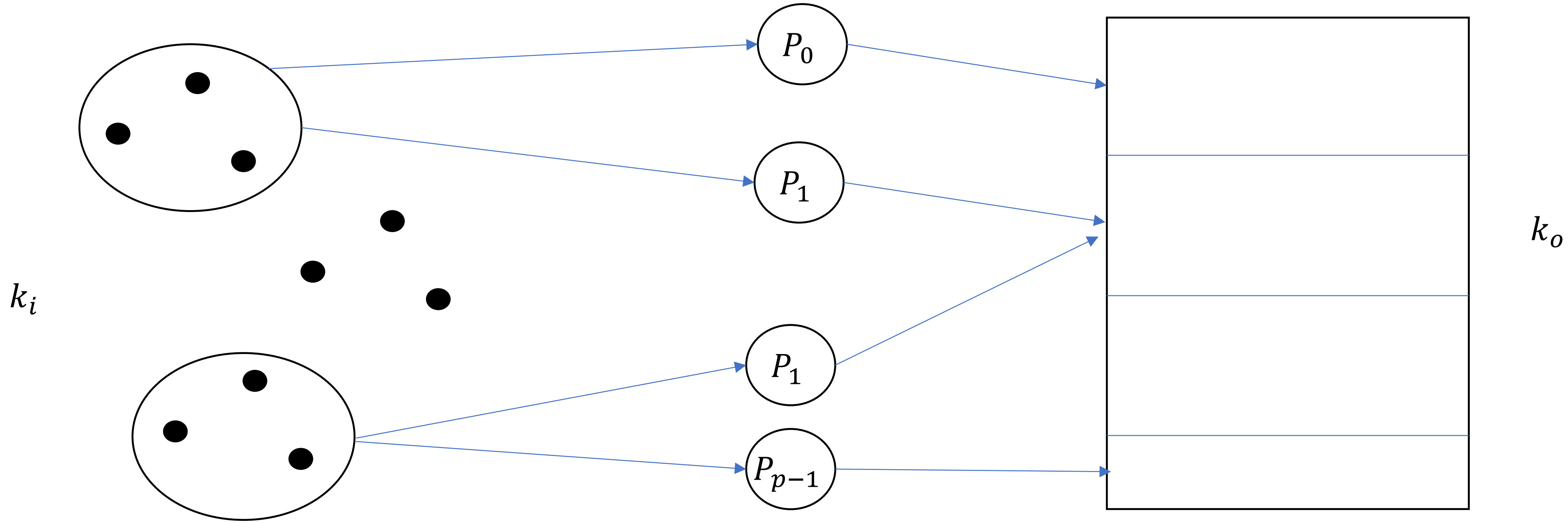
Data Parallelism - Approaches

- Assuming k_i input partitions, k_o output partitions, we have
  - $p = k_i \times k_o$



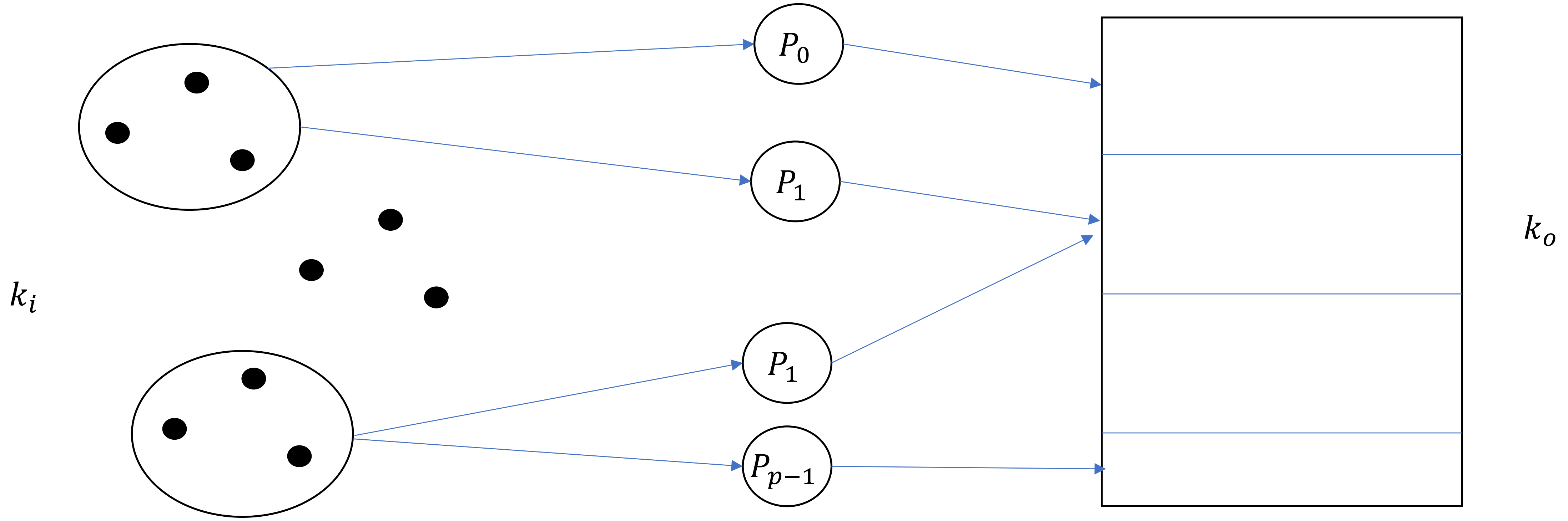
Data Parallelism - Approaches

- # hash computations per key value???
- # Processors that may conflict??



Data Parallelism - Approaches

- # hash computations per key value - $k_o < p$ in output only partitioning
- # Processors that may conflict - $k_i < p$ in input only partitioning



Next Class

- 9/9 Lecture 4 – Access to CWRU HPC; Writing DNN pipelines in pytorch

- 9/11 Lecture 5 – Data Parallel Algorithms on GPUs; Task Parallelism

Thank You

- Questions?

- Email: sanmukh.kuppannagari@case.edu