

# CESE4130: Computer Engineering

2024-2025, lecture 8

## Going Parallel (take two)

### Computer Engineering Lab

Faculty of Electrical Engineering, Mathematics & Computer Science

2024-2025

# Announcement

- None

# Course objectives

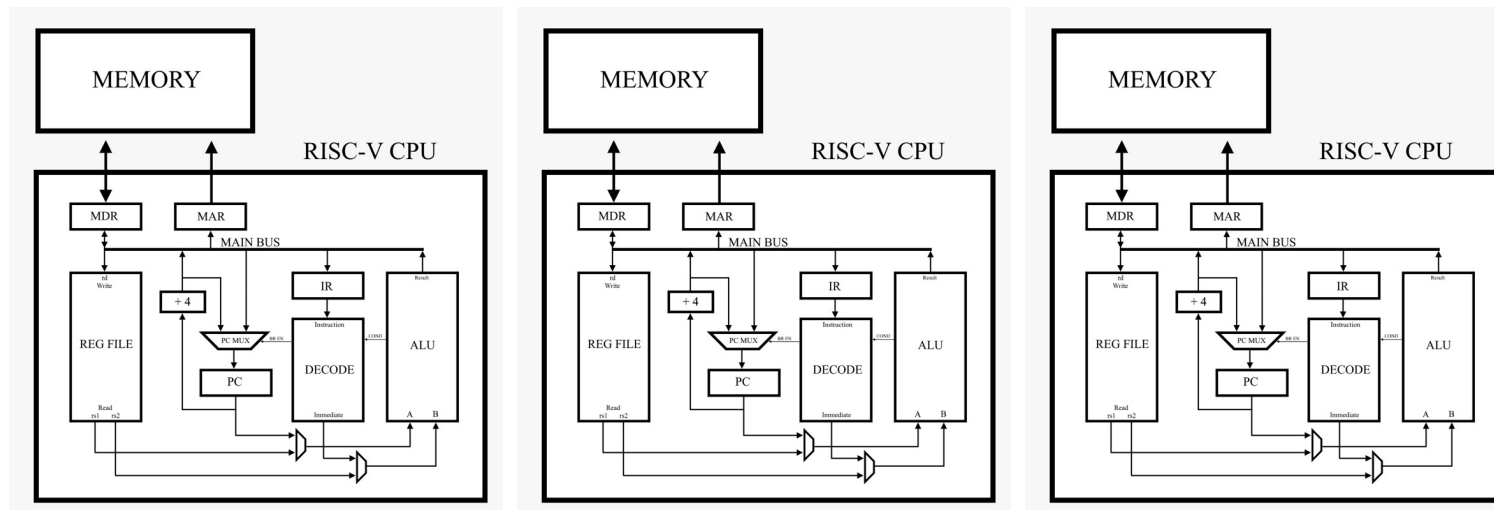
- Describe number representation systems and inter-conversion.
- Perform binary arithmetic operation such as addition and multiplication.
- Explain basic concepts of computer architecture.
- Use logic gates to implement simple combinational circuits.
- Explain system software and operating systems fundamentals, task management, synchronization, compilation, and interpretation.
- Use design and automation tools to perform synthesis and optimization.

# Objectives

- Understand the memory hierarchy of CUDA devices
- Explain efficient and inefficient memory access patterns
- Get the basic of the interconnection networks

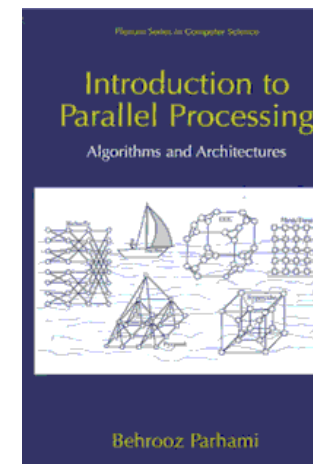
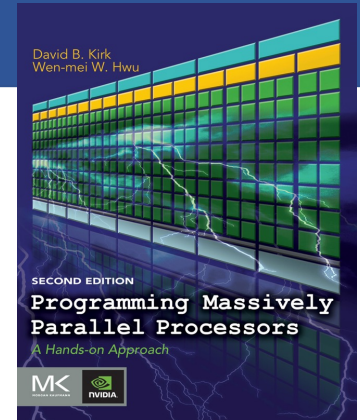
# Recap

- Parallel machines should be understood and efficiently used
- Building a parallel machine is not enough, models, parallel algorithms and || programs are needed
- Fully automated parallelization compilers are still a dream
- Programming Massively Parallel accelerators requires a specific programming model / tools
- More(?)
  - our main goal is **"to remove magic"** as you remember

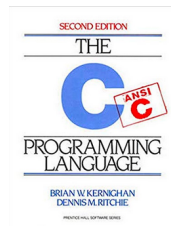
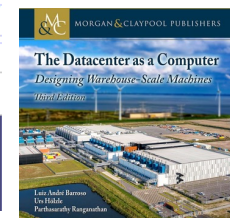


# Overview

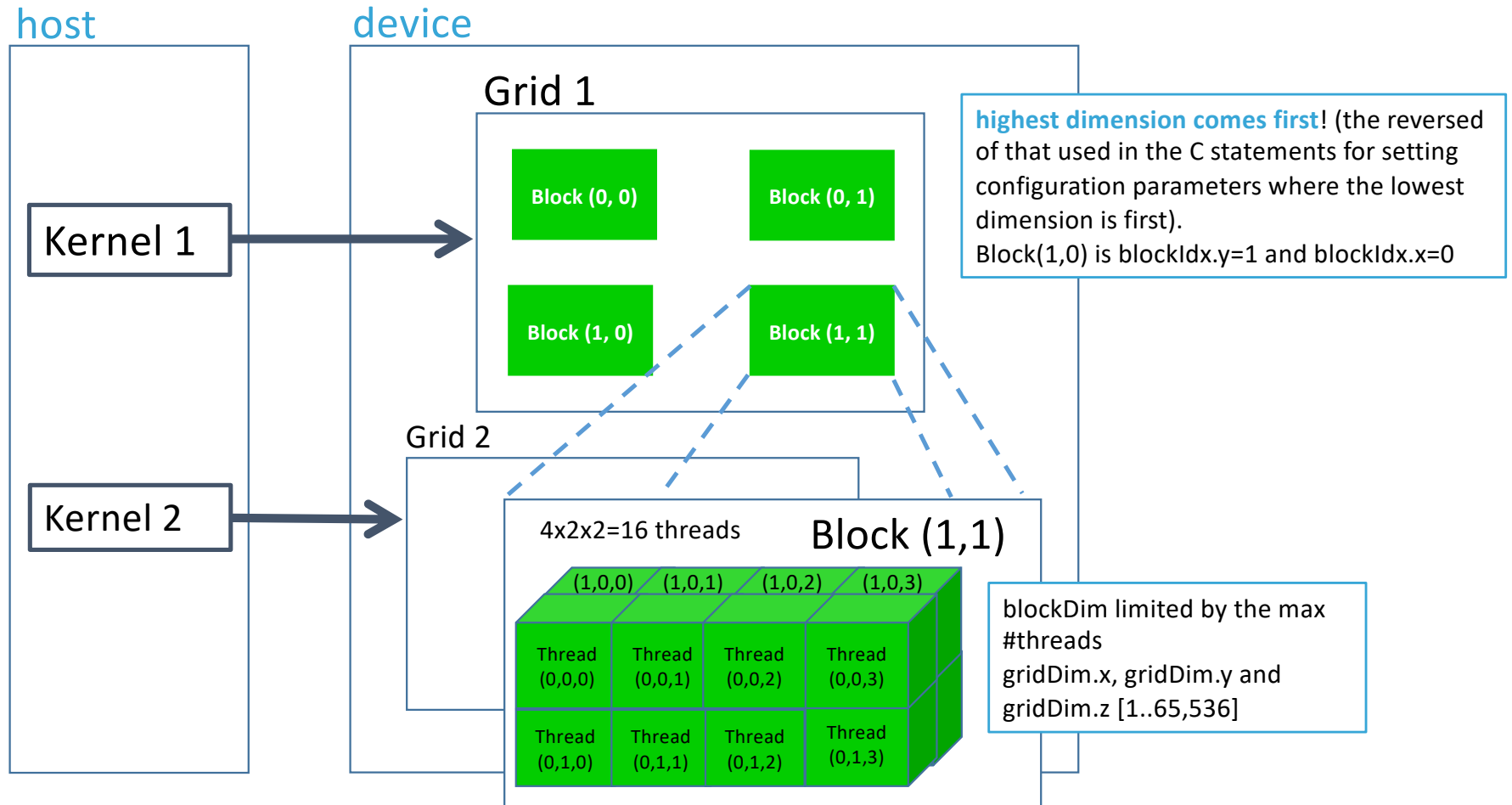
- The lecture material is collected from various sources
- About CUDA, please refer to Wen-Mei and David
  - <https://shop.elsevier.com/books/programming-massively-parallel-processors/hwu/978-0-323-91231-0>
- Also NVIDIA has a lot of tutorials and recorded lectures
  - <https://developer.nvidia.com/educators/existing-courses>
- Parallel Processing course, again Behrooz Parhami
  - [https://web.ece.ucsb.edu/~parhami/text\\_par\\_proc.htm#slides](https://web.ece.ucsb.edu/~parhami/text_par_proc.htm#slides)



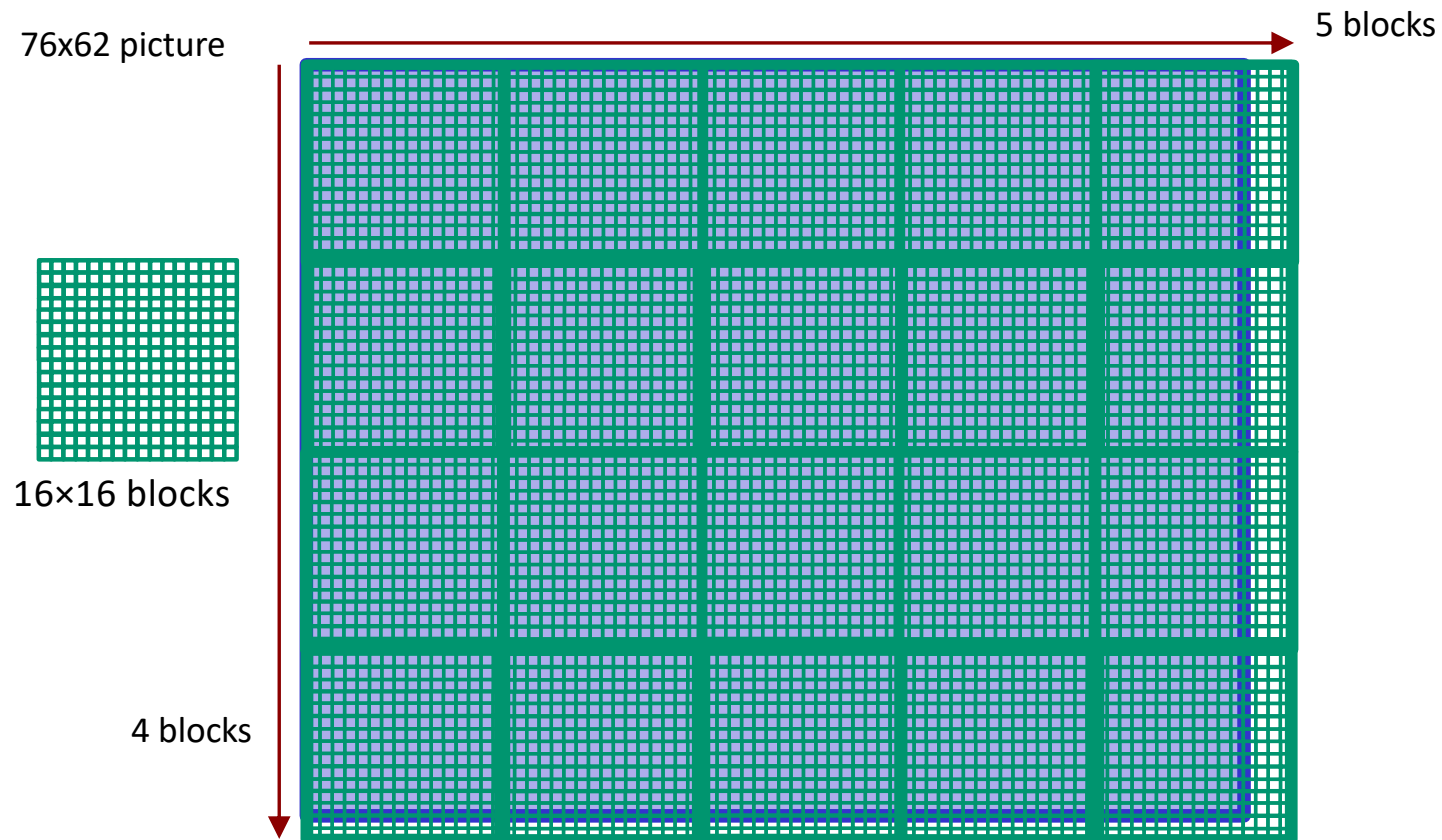
Maybe even



# CUDA: A Multi-Dimensional Grid Example

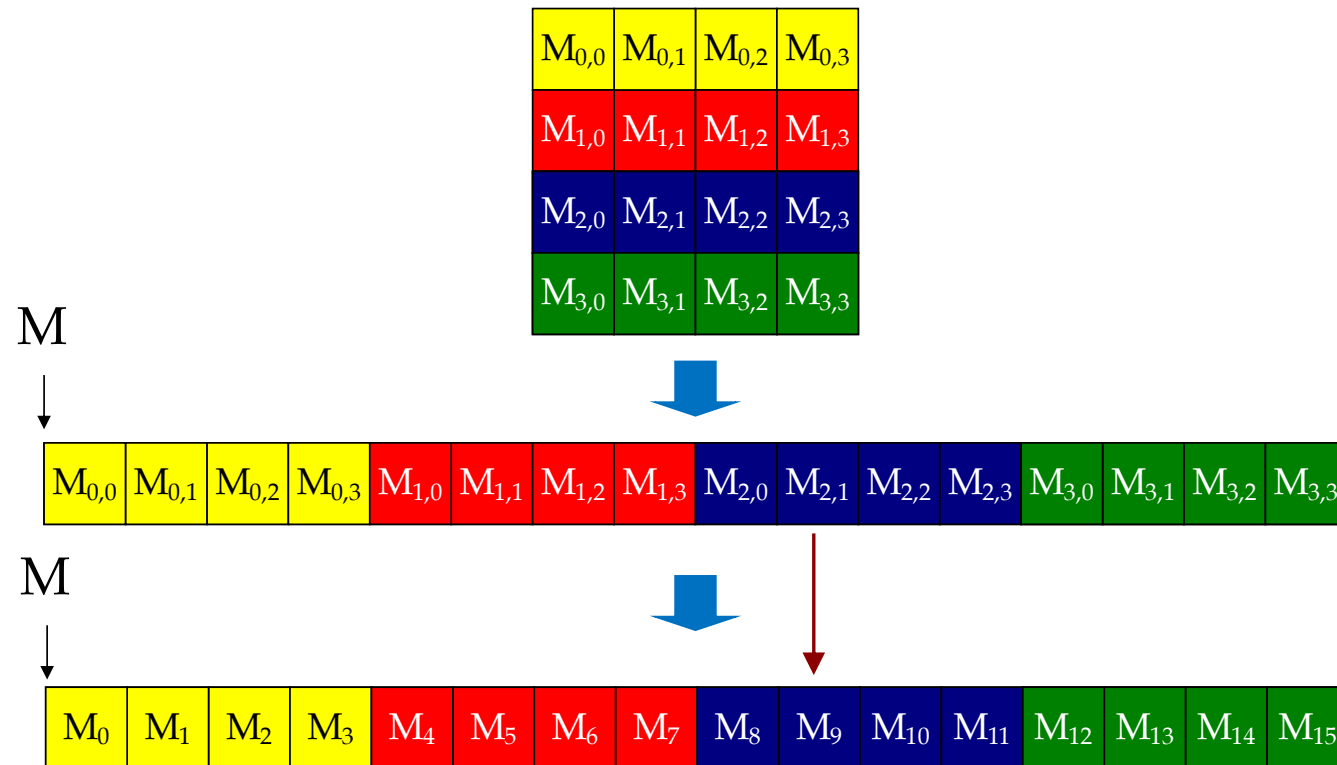


# Processing a Picture with a 2D Grid





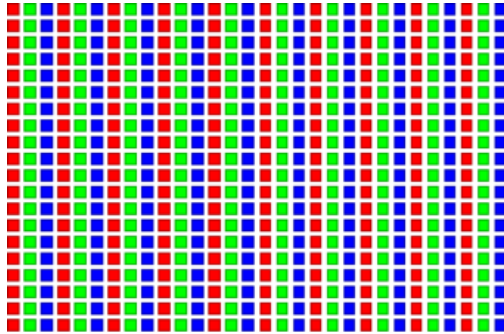
# Row-Major Layout of 2D arrays in C/C++ (reminder)



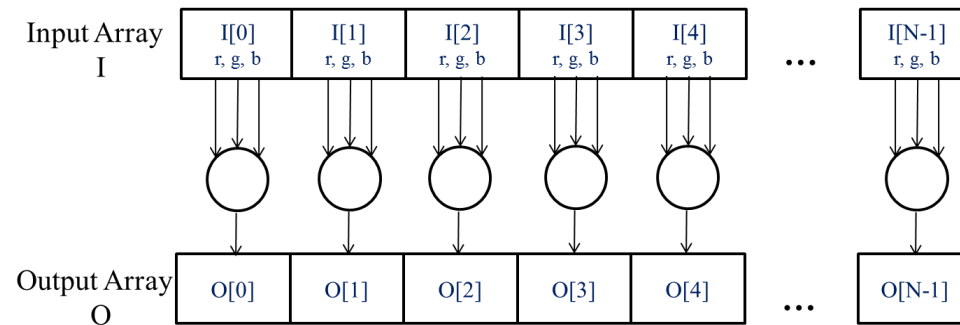
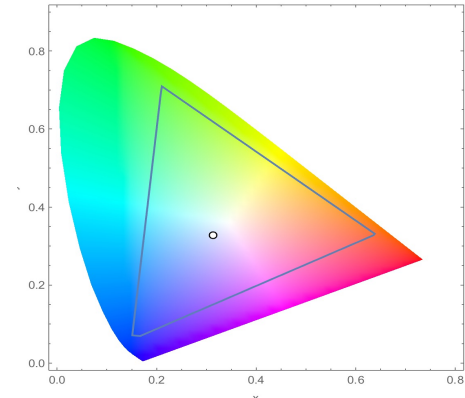
$$M_{2,1} \rightarrow \text{Row} * \text{Width} + \text{Col} = 2 * 4 + 1 = 9$$

FORTRAN compiler layout is column-major

# Conversion of a color image to grey-scale image (review)

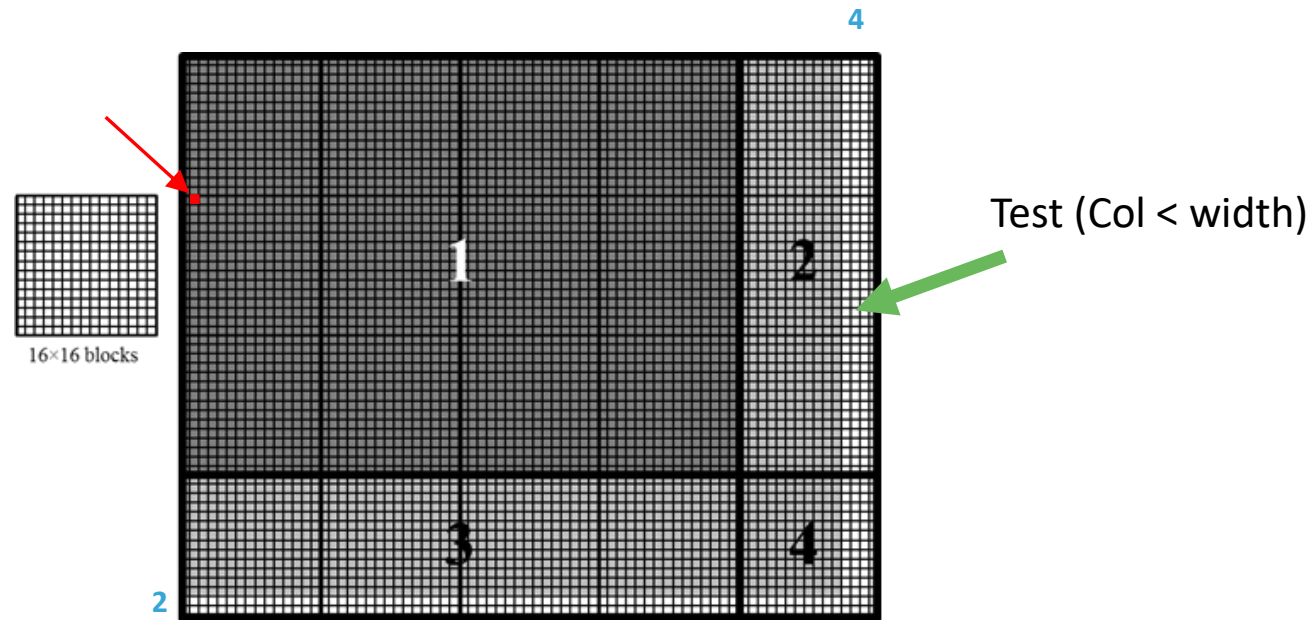


$$L = r * 0.21 + g * 0.71 + b * 0.07$$



*All pixels can be calculated independently of each other*

# Covering a 76×62 picture with 16×16 blocks



$\text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y}, \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}) =$   
 $P(1 * 16 + 0, 0 * 16 + 0) = P(16, 0)$


For the **pixel** handled by thread(0,0) of block(1,0)

## colorToGreyscaleConversion Kernel with 2D thread mapping to data

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(unsigned char * Pout, unsigned char * Pin,
                                int width, int height) {

    int Col = threadIdx.x + blockIdx.x * blockDim.x;
    int Row = threadIdx.y + blockIdx.y * blockDim.y;

    if (Col < width && Row < height) {
        // get 1D coordinate for the grayscale image
        int greyOffset = Row*width + Col;
        // one can think of the RGB image having
        // CHANNEL times columns of the gray scale image
        int rgbOffset = greyOffset*CHANNELS;
        unsigned char r = rgbImage[rgbOffset]; // red value for pixel
        unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
        unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
        // perform the rescaling and store it
        // We multiply by floating point constants
        grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

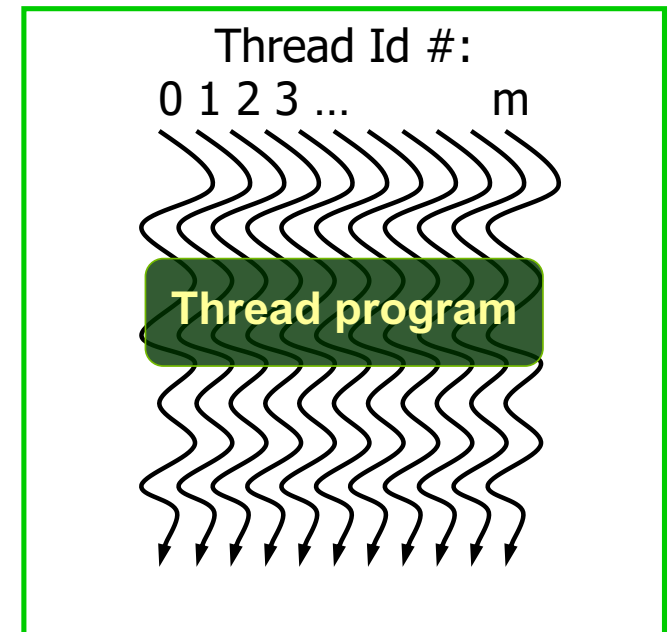


Part of the CUDA C specification  
(same as threadIdx and blockIdx)

# CUDA Thread Block (review)

- All threads in a block execute the same kernel program (***SPMD***)
- Programmer declares block:
  - Block size **1 to 1,024** concurrent threads
  - Block shape 1D, 2D, or 3D
- Threads have **thread index** numbers within block
  - Kernel code uses **thread index and block index** to select work and address shared data
- Threads within the same block share data and synchronize while doing their share of the overall work
- Threads in different blocks **can't** cooperate
  - Each block can execute in any order relative to other blocks!

## CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

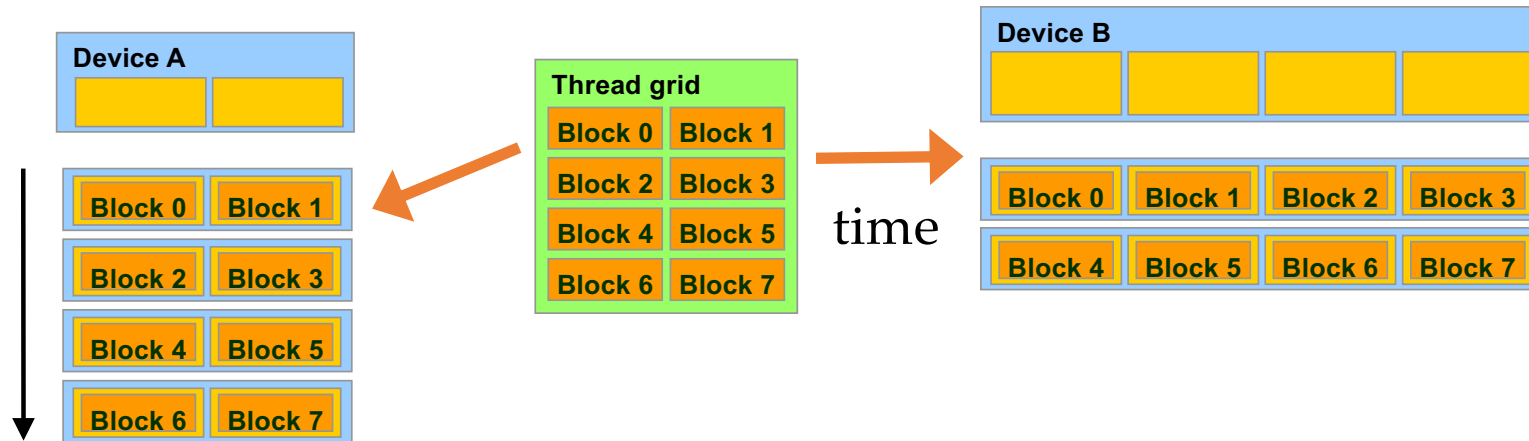
# Compute Capabilities are GPU Dependent

	GRID V100D-32Q	Tesla T10 Processor	GeForce GTX 1080 Ti	
Total amount of global memory:	4,160,749,568	4,294,770,688	3,131,572,224	Bytes
Number of multiprocessors:	80	30	28	
Number of cores:	640	240	224	
Total amount of constant memory:	65,536	65,536	65,536	Bytes
Total amount of shared memory per block:	49,152	16,384	49,152	Bytes
Total number of registers available per block:	65,536	16,384	65,536	Bytes
Warp size:	32	32	32	
Maximum number of threads per block:	1,024	512	1,024	
Maximum sizes of each dimension of a block:	1024 x 1024 x 64	512 x 512 x 64	1024 x 1024 x 64	
Maximum sizes of each dimension of a grid:	2147483647 x 65535 x 65535	65535 x 65535 x 1	2147483647 x 65535 x 65535	
Maximum memory pitch:	2,147,483,647	2,147,483,647	2,147,483,647	Bytes
Texture alignment:	512	256	512	
Clock rate:	1.38 GHz	1.30 GHz	1.58 GHz	
Concurrent copy and execution:	Yes	Yes	Yes	



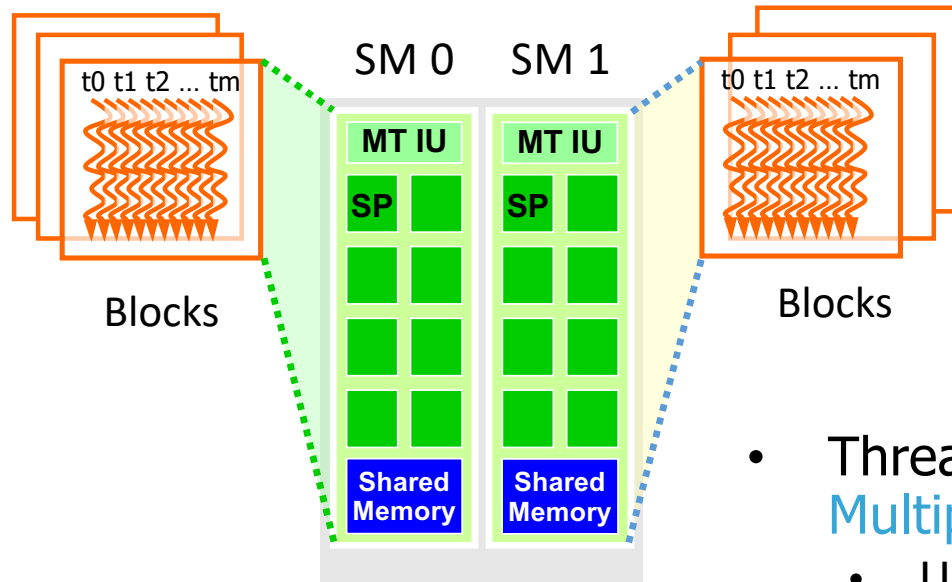
GPU	GK107 (Kepler)	GM107 (Maxwell)
CUDA Cores	384	640
Base Clock	1058 MHz	1020 MHz
GPU Boost Clock	N/A	1085 MHz
GFLOP/s	812.5	1305.6
Compute Capability	3.0	5.0
Shared Memory / SM	16KB / 48 KB	64 KB
Register File Size / SM	256 KB	256 KB
Active Blocks / SM	16	32

# Transparent Scalability (as promised by CUDA)



- Each block can execute in any order relative to others
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of parallel processors

# Executing CUDA Thread Blocks



Mind that **resources are finite**:

Fermi SM architecture supports **up to 8 blocks** and 1,536 threads  
so 6 blocks of 256 threads, 3 of 512,  
etc are valid assignments

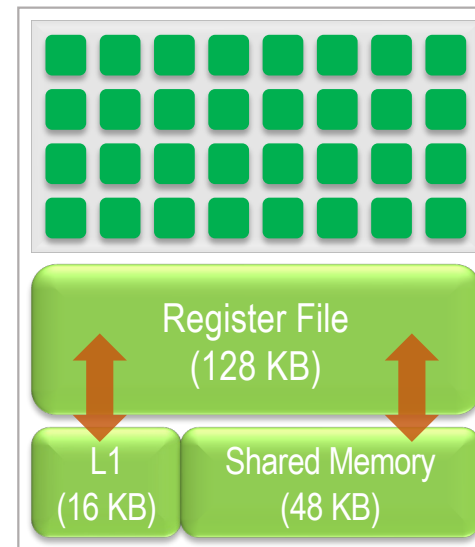
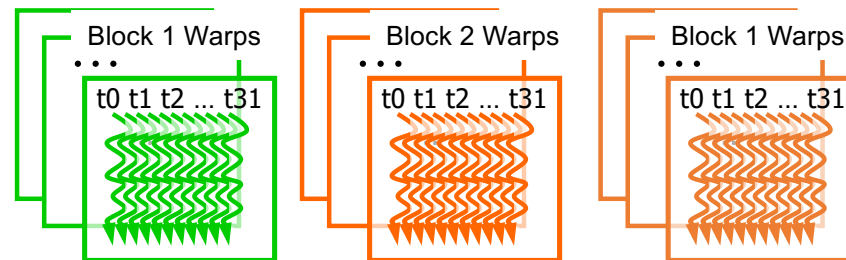
**12 blocks of 128 threads is not!**

- Threads are assigned to **Streaming Multiprocessors** in block granularity
  - Up to **32** blocks to each SM as resource allows
  - Maxwell SM can take up to **2,048** threads
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution



# Thread Scheduling (1/2)

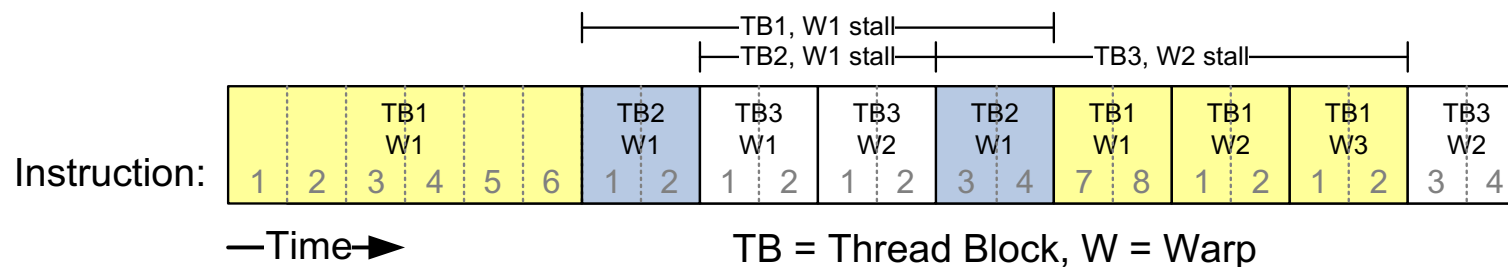
- Each block is executed as **32-thread warps**
  - An implementation decision, **not part of the CUDA programming model**
  - Warps are scheduling units in **SM**
- If three blocks are assigned to an SM and each block has 256 threads, how many warps are there in an SM?
  - Each block is divided into  $256/32 = 8$  warps
  - $8 \text{ warps/blk} * 3 \text{ blks} = 24 \text{ warps}$



#SPs << #threads in wraps – this brings latency tolerance

## Thread Scheduling (2/2)

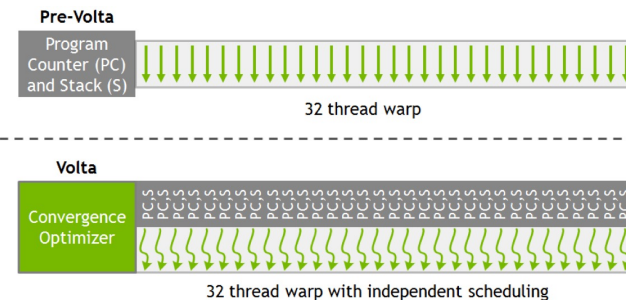
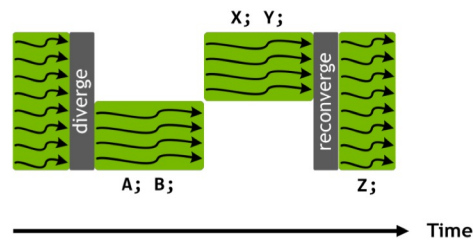
- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible warps are selected for execution on a prioritized scheduling policy
  - **All threads in a warp execute the same instruction when selected** (the classic SIMD)



# Be aware of Divergence

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths were serialized in older GPUs
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more
- A common case: divergence could occur when branch condition is a function of thread ID
  - Example **with divergence**:
    - `If (threadIdx.x > 2) { }`
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
  - Example **without divergence**:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;
```



Volta++ maintain per-thread scheduling resources, e.g., PC and Stack. Pre-Volta devices maintained these resources per warp

Consider, understand and benefit from implementation (u-architectural) details

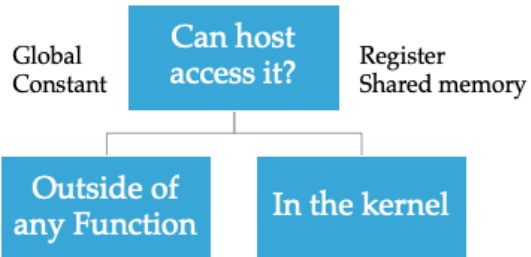
# Block Granularity Considerations Example

- For Matrix Multiplication using multiple blocks, should one use 8X8, 16X16 or 32X32 blocks? Assume that in the GPU used, each SM can take up to 1,536 threads and up to 8 blocks
  - For 8X8, we have 64 threads per block. Each SM can take up to 1,536 threads, which is 24 blocks. But each SM can only take up to 8 Blocks, **only 512 threads** (16 warps) will go into each SM! (under utilization!)
  - For 16X16, we have 256 threads per block. Since each SM can take up to 1,536 threads (48 warps), which is 6 blocks (within the 8 block limit). Thus we use the **full thread capacity** of an SM
  - For 32X32, we would have 1,024 threads per Block. Only one block can fit into an SM, using **only 2/3** of the thread capacity of an SM

*Recap: consider, understand and benefit from implementation (microarchitectural) details*

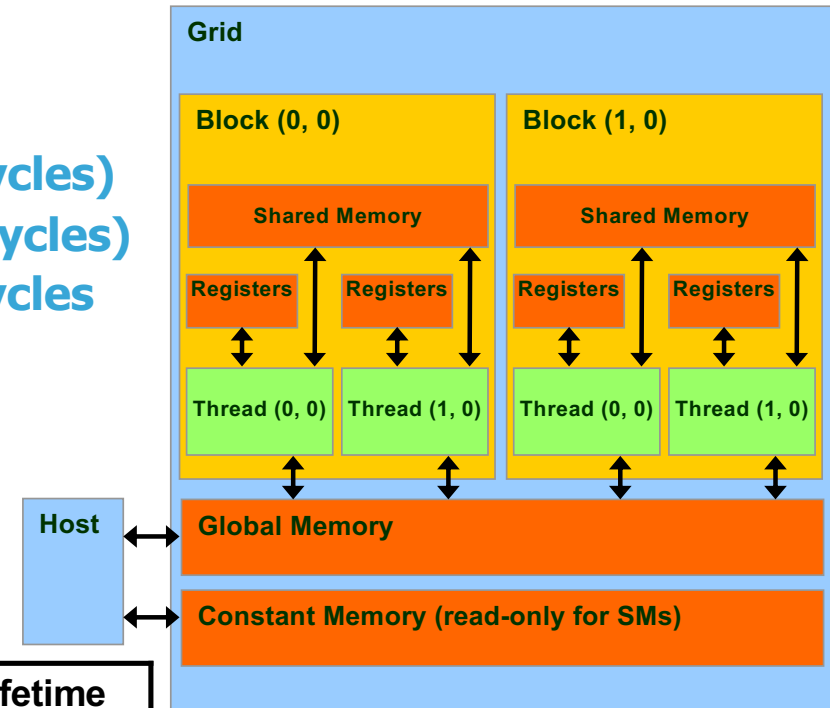
# Programmer View of CUDA Memories

- Each thread can:
  - Read/write per-thread **registers (~1 cycle)**
  - Read/write per-block **shared memory (~5 cycles)**
  - Read/write per-grid **global memory (~500 cycles)**
  - Read/only per-grid **constant memory (~5 cycles with caching)**



CUDA Variable Type Qualifiers

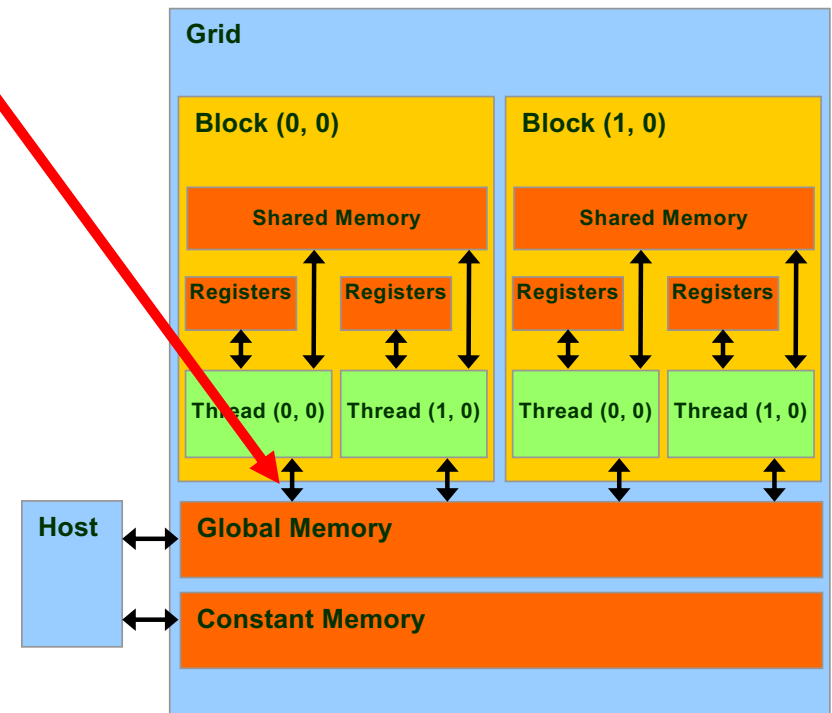
Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application



- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables** without any qualifier reside in a **register**
  - Except per-thread arrays that reside in global memory

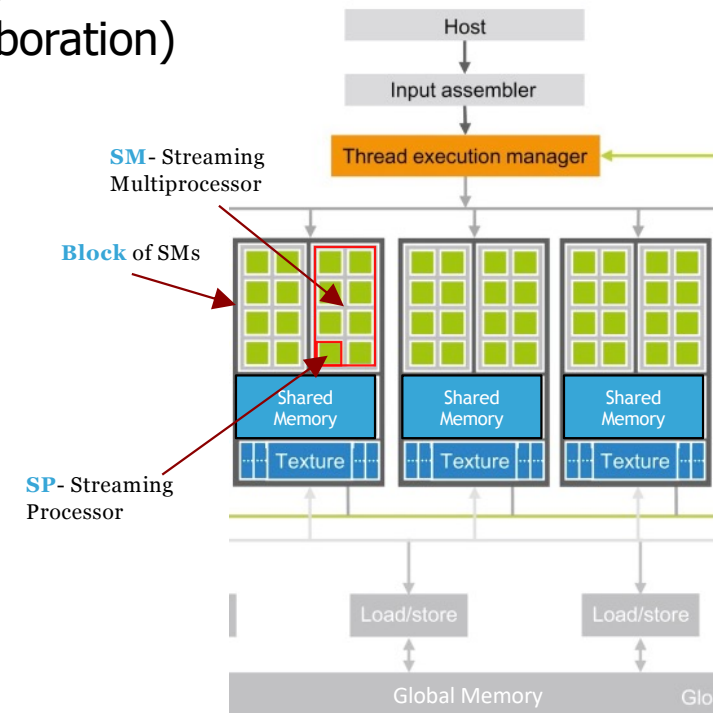
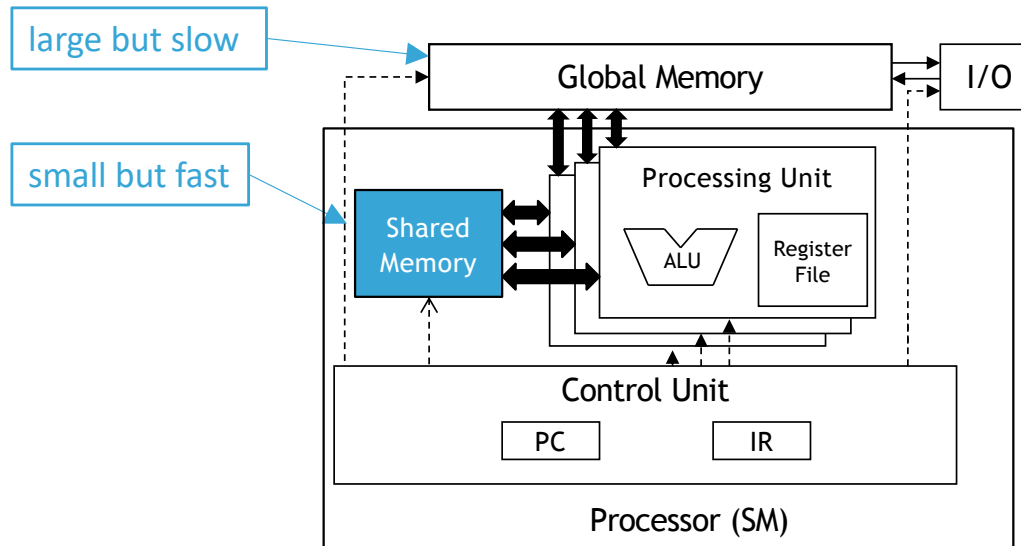
# How about GPU performance? (it is the memory ...)

- All threads access global memory for their input matrix elements
  - One memory accesses per SP floating-point addition (4 bytes)
  - 4B/s of memory bandwidth/FLOPS
- Consider a GPU with
  - Peak floating-point rate 12 TFLOPS with 1 TB/s DRAM bandwidth
  - $4 \times 12 = 48$  TB/s required to achieve peak FLOPS rating
  - The 1 TB/s memory bandwidth limits the execution at 250 GFLOPS
- This limits the execution rate to **2%** ( $.25/12$ ) of the peak floating-point execution rate of the device!
- Need to **drastically cut down memory accesses** to get close to the 12 TFLOPS device capability



# CUDA Shared Memory (SM)

- Shared memory is on-chip (similar to Registers)
  - Access costs and functionality are quite different
  - Part of the memory space (load/store access)
  - Visible to all threads in a block (enables collaboration)



# Common Programming Strategy (reminds of something?)

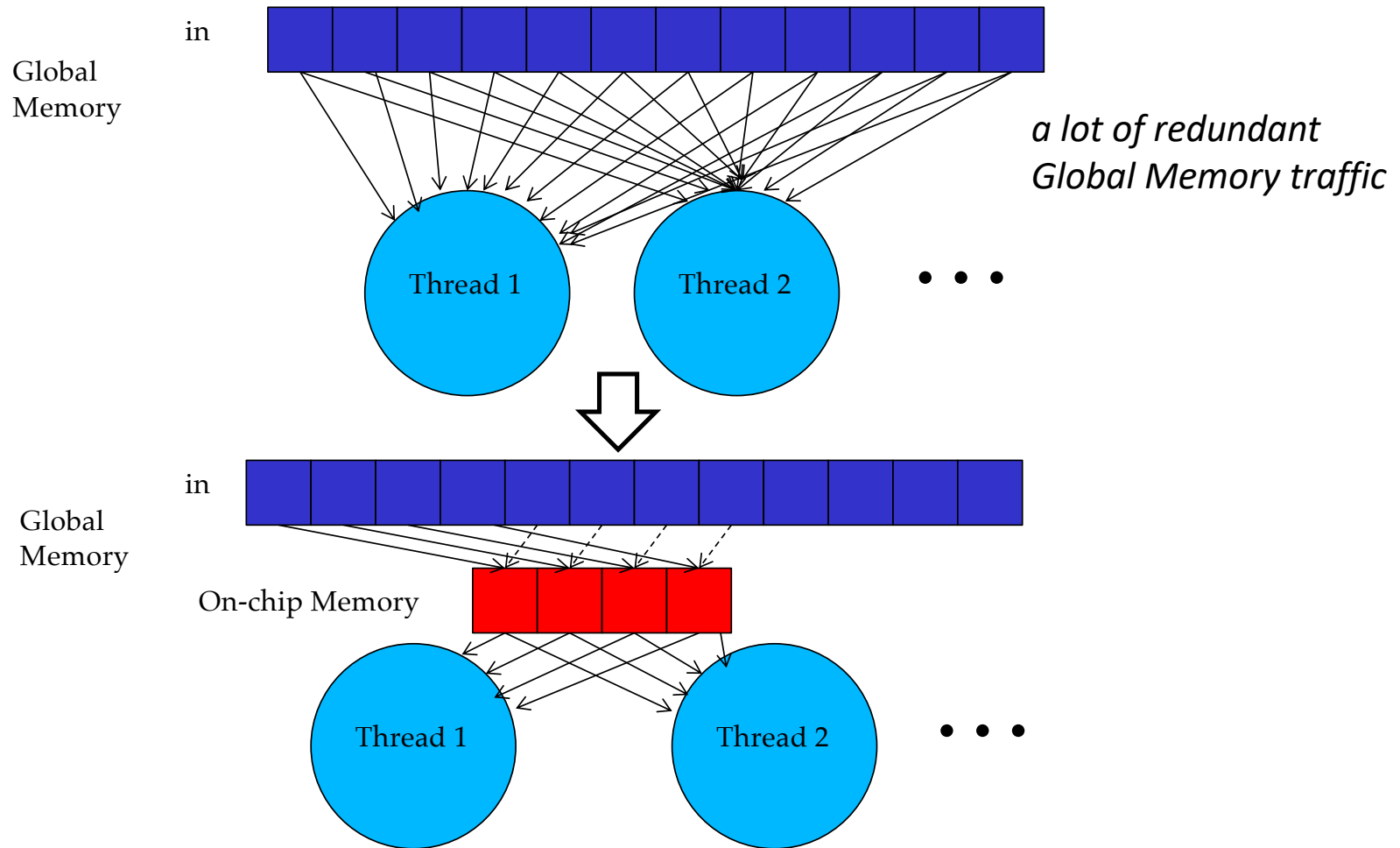
- Global memory resides in device memory (DRAM)
- A profitable way of performing computation on the device is to **tile the input data** to take advantage of fast shared memory:
  - Partition data into subsets (tiles) that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory



*(!) “Blocked Matrix Operations” are widely used in the literature, CUDA reserved the word “blocks”*



# Shared Memory Blocking Basic Idea



# Basic Concept of Tiling

In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles

- Carpooling for commuters
- Tiling for global memory accesses
  - drivers = threads accessing their memory data operands
  - cars = memory access requests

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width){  
    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];  
    ...  
}
```

Access order →

thread <sub>0,0</sub>	<span style="border: 1px solid red; padding: 2px;"><math>M_{0,0} * N_{0,0}</math></span>	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
thread <sub>0,1</sub>	<span style="border: 1px solid red; padding: 2px;"><math>M_{0,0} * N_{0,1}</math></span>	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
thread <sub>1,0</sub>	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
thread <sub>1,1</sub>	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

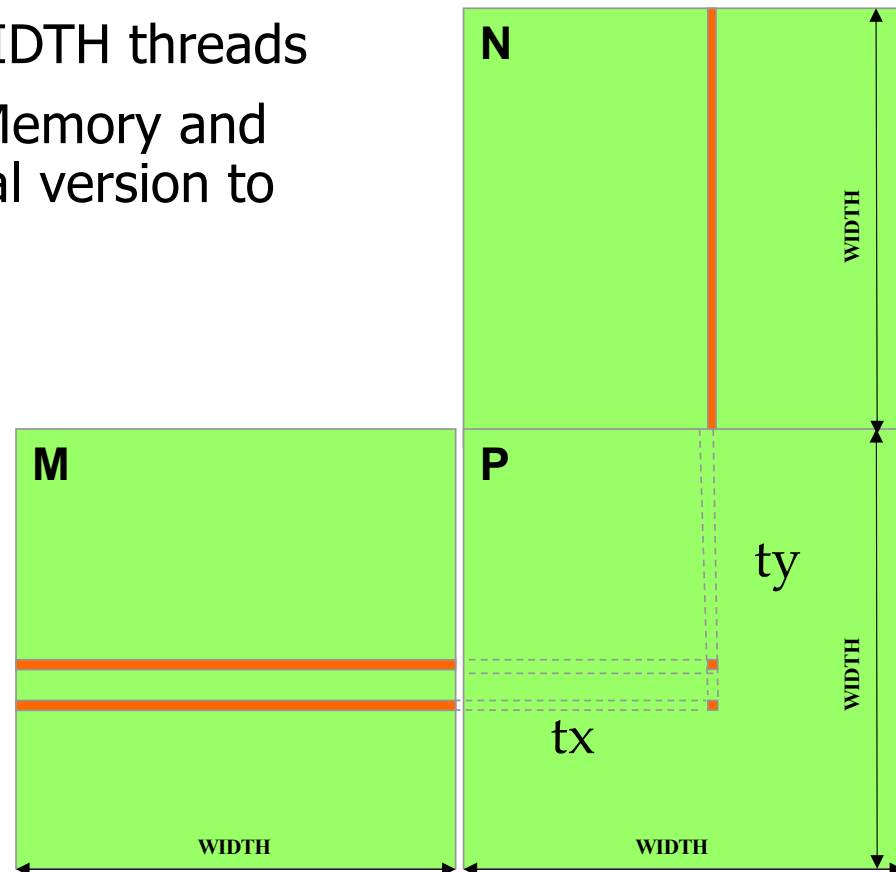


# Outline of the Tiling Technique

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Have the multiple threads to access their data from the on-chip memory
- Move on to the next block/tile
- *(threads timing is still missing above, more later)*

# Idea: Place global memory data into Shared Memory for reuse

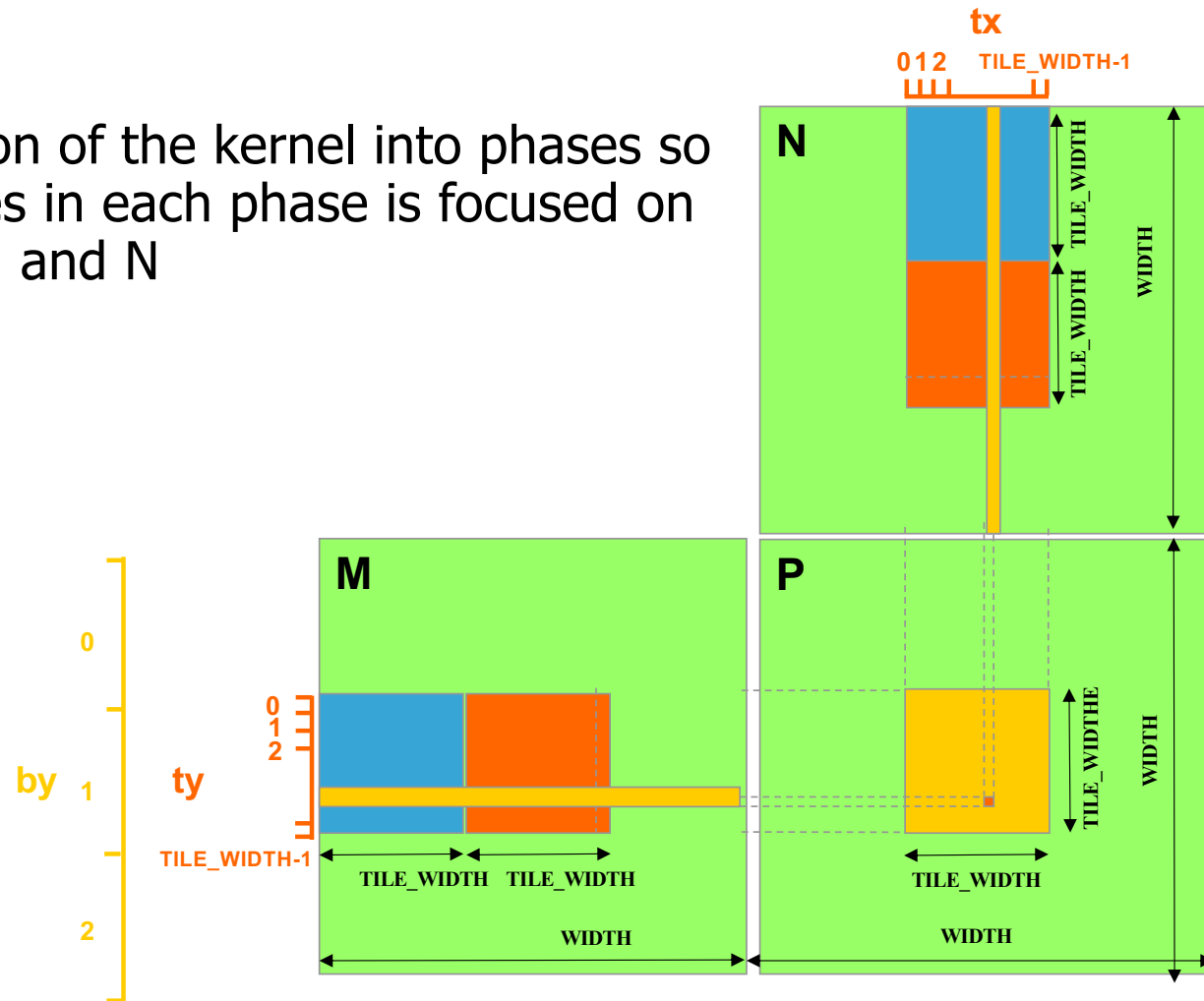
- Each input element is read by WIDTH threads
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth



# Tiled Multiply

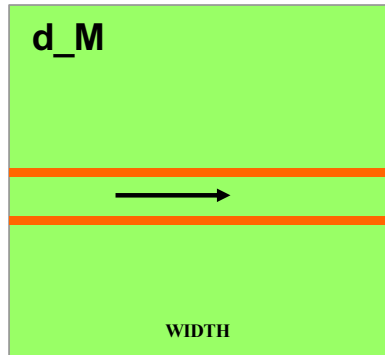


- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of M and N



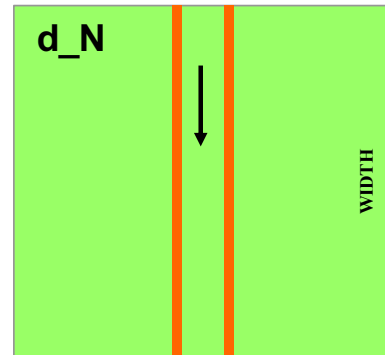
# Two Access Patterns

not coalesced



$M[\text{Row} \times \text{Width} + k]$

coalesced

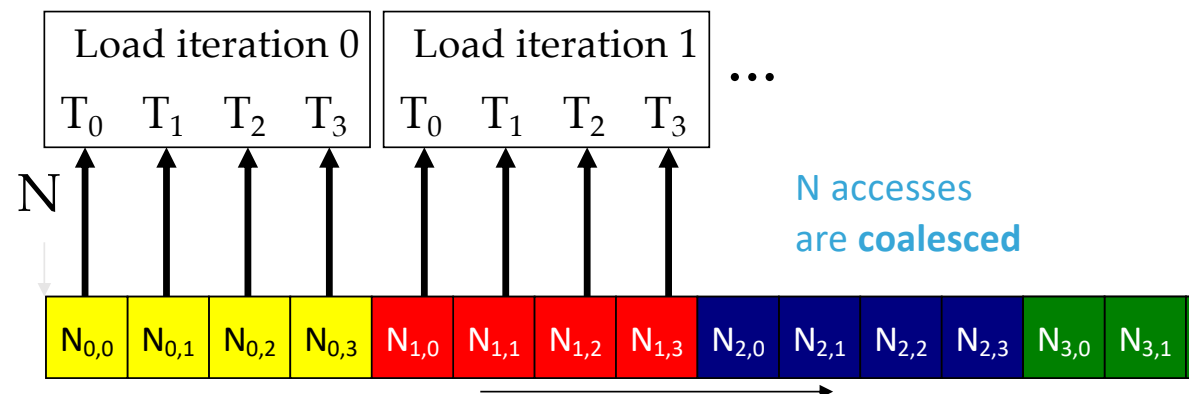


$N[k \times \text{Width} + \text{Col}]$

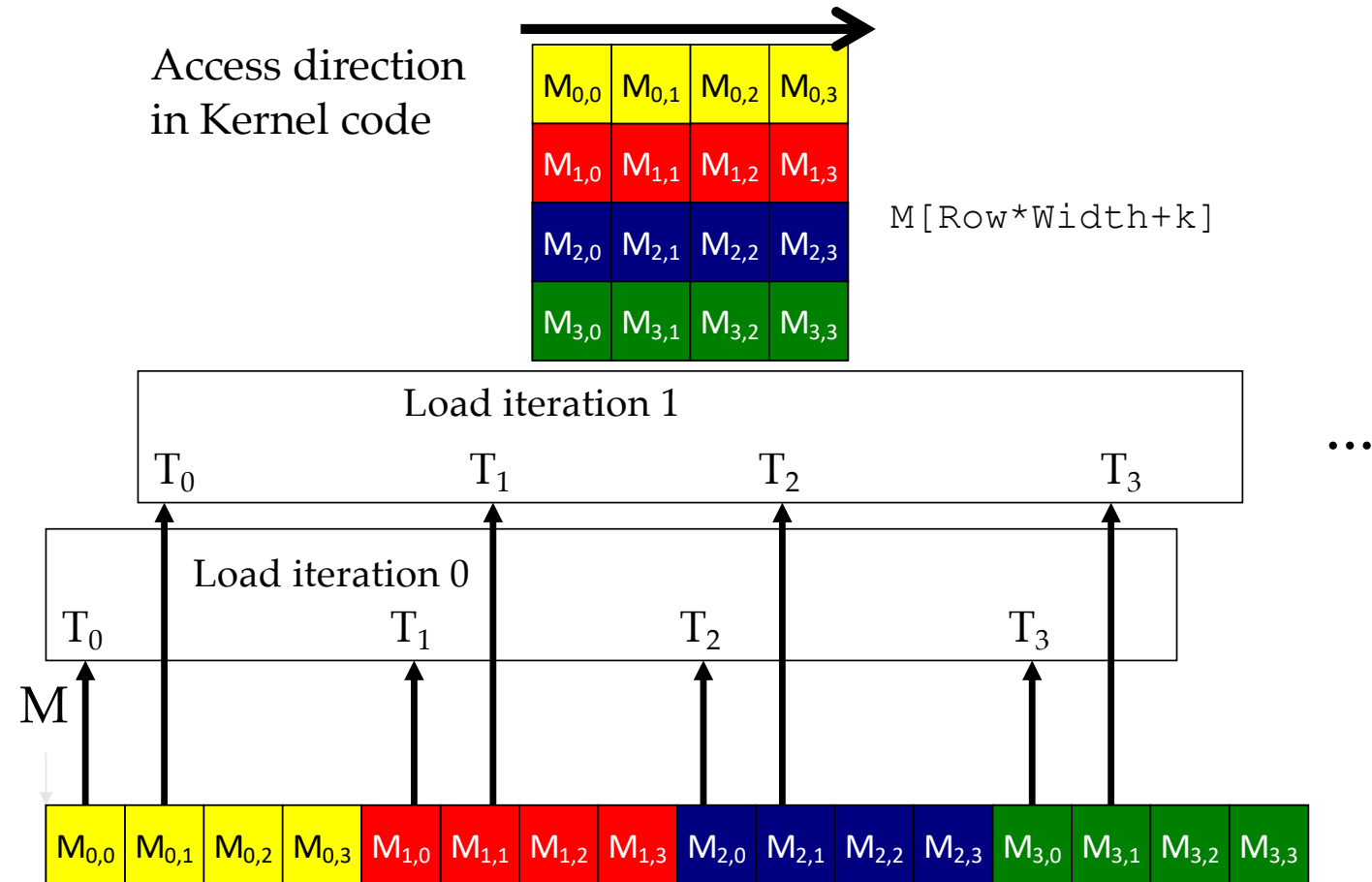
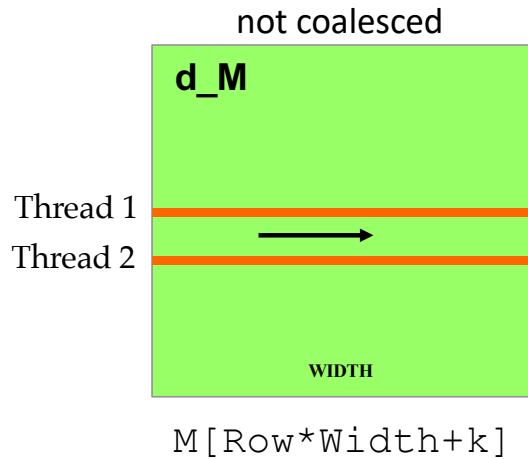
$k$  is loop counter in the inner product loop of the kernel code

Access  
direction in  
Kernel code

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$



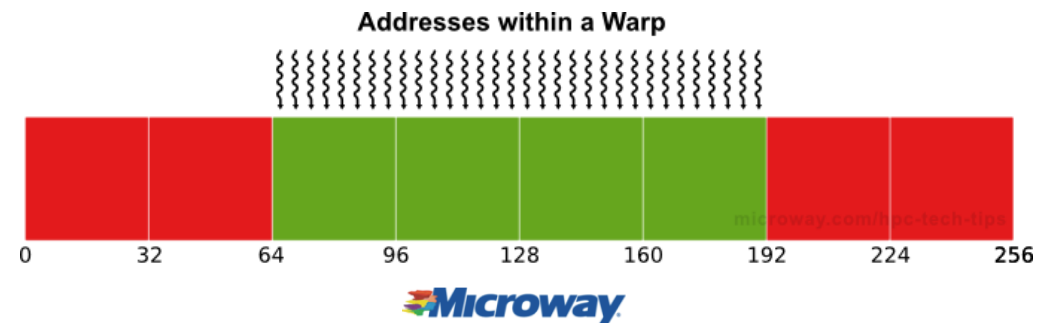
# M accesses are not coalesced



## Determine if access is coalesced

› Accesses in a warp are to consecutive locations **if** the index in an array access is in the form of

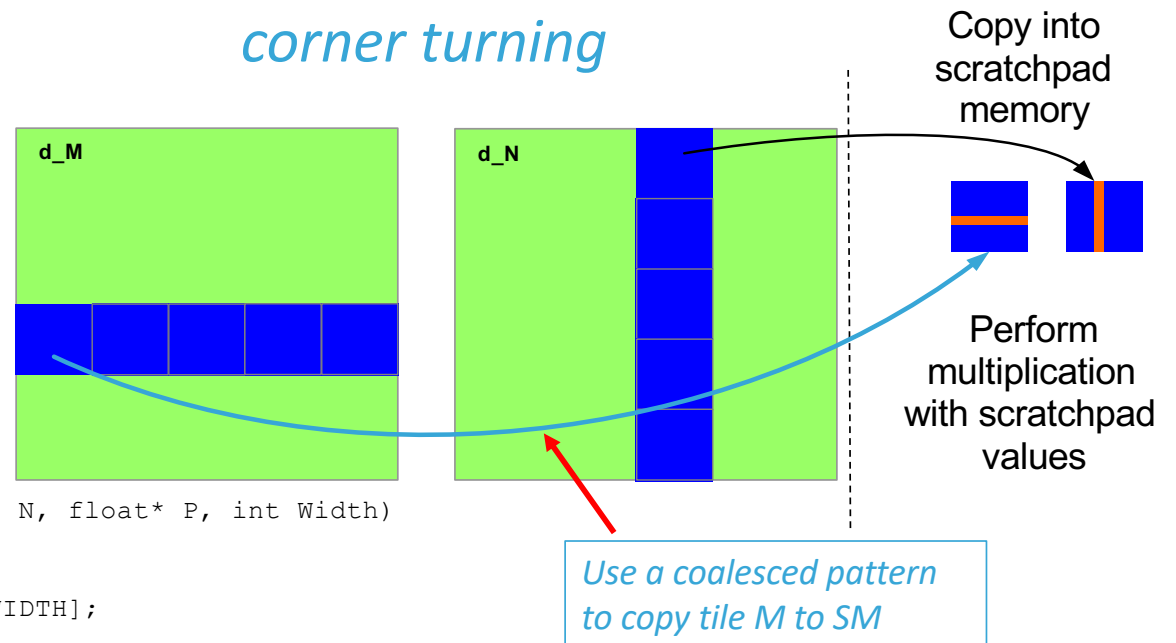
- $A[(\text{expression with terms independent of threadIdx.x}) + \text{threadIdx.x}]$ ;
- Then  $(\text{expression with terms independent of threadIdx.x})$  is also multiple of the burst size we speak of **fully coalesced** access





# Use shared memory to enable coalescing in tiled mxMUL

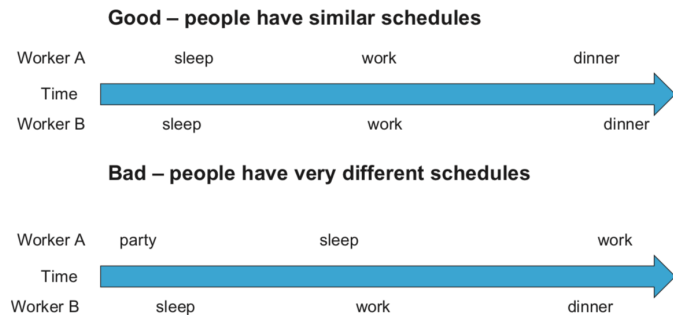
*corner turning*



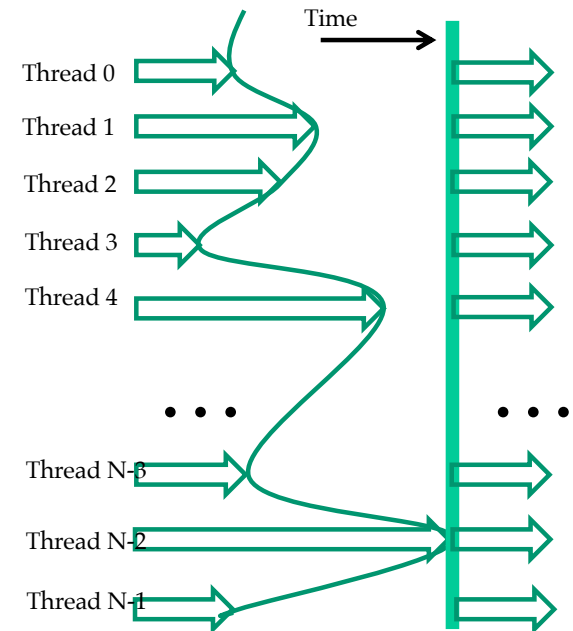
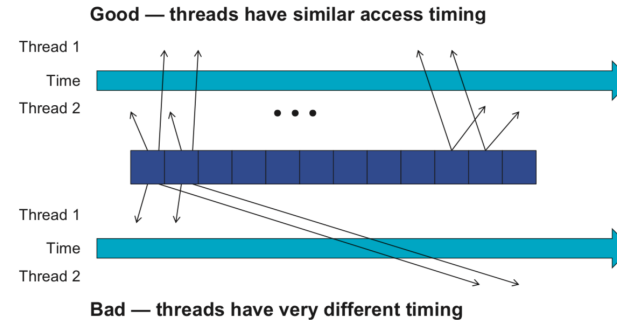
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
1.  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
    . . .
    // Loop over the M and N tiles required to compute the P element
    8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        // Collaborative loading of M and N tiles into shared memory
        9.      subTileM[ ? ][ ? ] = M[          ?          ];
        10.     subTileN[ ? ][ ? ] = N[          ?          ];
        . . .
    }
```

# Barrier Synchronization

## carpooling



## tiling



- An API function call in CUDA
  - `__syncthreads()`
- All threads in the same block must reach the `__syncthreads()` before any can move on
- Best used to coordinate tiled algorithms phased execution
  - To ensure that all elements of a tile are loaded (at begin)
  - To ensure that all elements of a tile are consumed (at end)

Each of the  $\text{TILE\_WIDTH}^2$  threads loads one element followed by a `__syncthreads()`

# Shared Memory and Threading

- SMs in Maxwell have 64KB shared mem. (max 48KB/block)
  - Shared memory size is implementation dependent!
  - For `TILE_WIDTH = 16`, each thread block uses  $2 \times 256 \times 4B = 2KB$  of shared memory
    - Shared memory can potentially support up to 32 thread blocks actively executing
    - This allows up to  $8 \times 512 = 4,096$  pending loads (2 per thread, 256 threads/block)
  - The next `TILE_WIDTH 32` would lead to  $2 \times 32 \times 32 \times 4B = 8KB$  shared memory usage per thread block, allowing 8 thread blocks active at the same time ( $2 \times 1,024 = 2,048$  loads /  $1,024 \times (2 \times 32) = 65,536$  mul/add), however, max #threads (1,536) will reduce #thread blocks to just 1!
    - Each `__syncthread()` can reduce the number of active threads for a block, hence more thread blocks can be advantageous
- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - Device with 150GB/s mem BW supports  $(150/4) \times 16 = 600$  GFLOPS!

# Register file capacity and parallelism

- Assume a current-generation device D, with SM of up to 1,536 threads and 16,384 registers
- With 16,384 registers to support 1,536 threads, there are only 10 registers ( $16,384/1,536$ ) for each thread!
  - using 11 registers, will limit the number of concurrent threads in each SM
  - Such reduction is at block granularity; e.g., with blocks of 512 threads, the reduction of threads will be done with 512 threads at a time
  - Next smaller #threads from 1,536 is 1,024, a 1/3 reduction of threads that can simultaneously reside in each SM
  - This can substantially reduce the #warps available for scheduling, thereby decreasing the ability of the processor to find useful work in the presence of long-latency operations
- The number of registers is device dependent

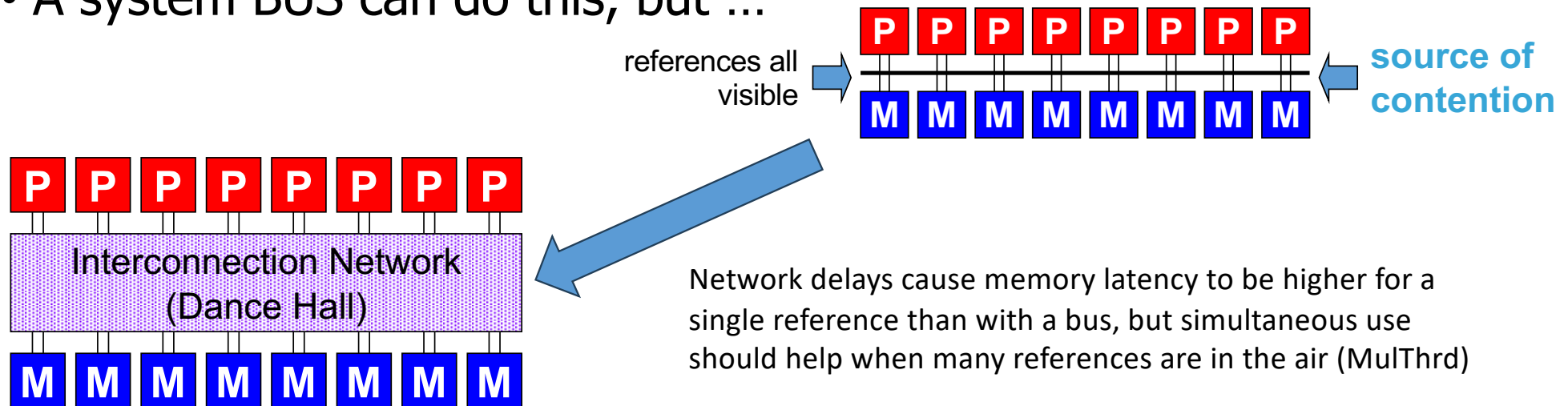
*Recap: consider, understand and benefit from implementation (microarchitectural) details*

## Back 2 || Computers

- How the different sub-systems are connected together
- Some fundamentals of Interconnection Networks

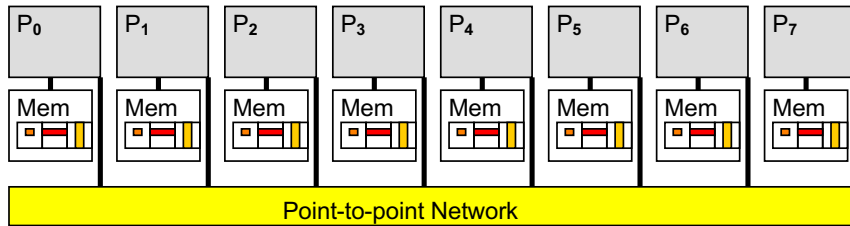
# Multiple processors and multiple memories

- Global memory shared among ||processors is the natural generalization of the sequential memory model (PRAM)
  - Thinking about it, programmers assume **sequential consistency** (SC) when they think about ||ism
- SC difficult to achieve under all circumstances (and is costly)
- A system BUS can do this, but ...

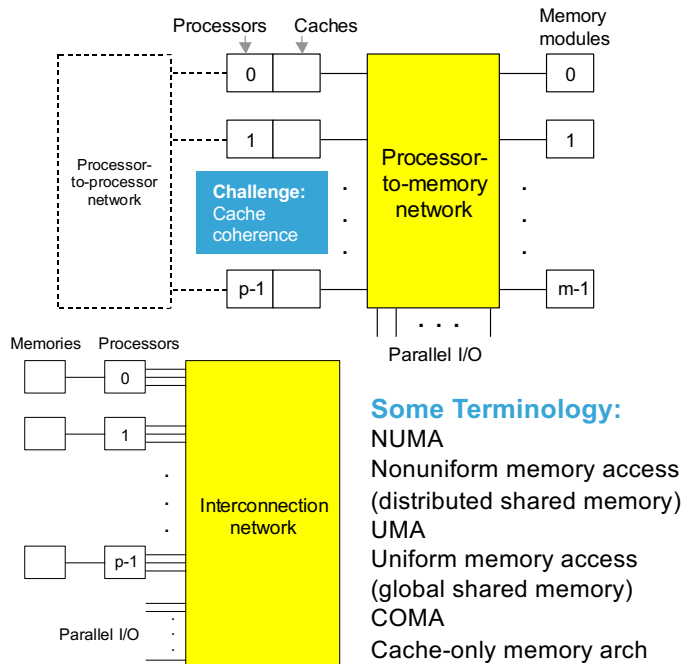


# Interconnect Networks

Architecture common for servers



Removing the Processor-to-Memory Bottleneck

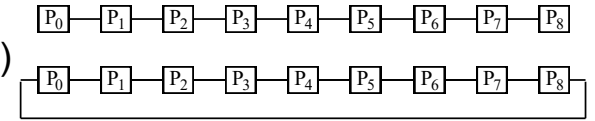


Max node degree  
Network diameter  
Bisection width

$$d = 2$$

$$D = p - 1 \quad (\lfloor p/2 \rfloor)$$

$$B = 1 \quad (2)$$

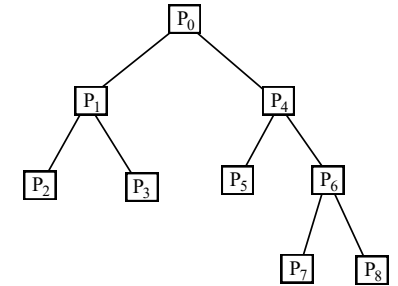


Max node degree  
Network diameter  
Bisection width

$$d = 3$$

$$D = 2 \lfloor \log_2 p \rfloor \quad (-1)$$

$$B = 1$$

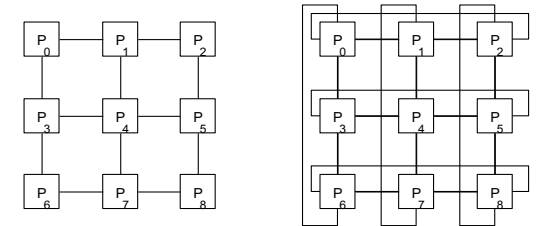


Max node degree  
Network diameter  
Bisection width

$$d = 4$$

$$D = 2\sqrt{p} - 2 \quad (\sqrt{p})$$

$$B \cong \sqrt{p} \quad (2\sqrt{p})$$

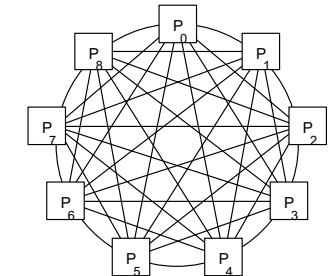


Max node degree  
Network diameter  
Bisection width

$$d = p - 1$$

$$D = 1$$

$$B = \lfloor p/2 \rfloor \lceil p/2 \rceil$$



# Some Interconnection Networks in use

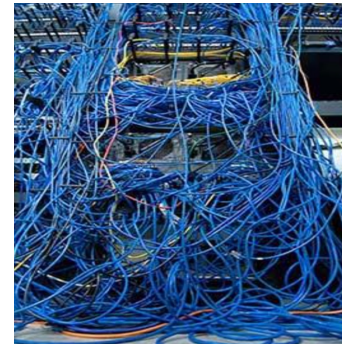
Network name(s)	Number of nodes	Network diameter	Bisection width	Node degree	Local links?
1D mesh (linear array)	$k$	$k - 1$	1	2	Yes
1D torus (ring, loop)	$k$	$k/2$	2	2	Yes
2D Mesh	$k^2$	$2k - 2$	$k$	4	Yes
2D torus ( $k$ -ary 2-cube)	$k^2$	$k$	$2k$	4	Yes <sup>1</sup>
3D mesh	$k^3$	$3k - 3$	$k^2$	6	Yes
3D torus ( $k$ -ary 3-cube)	$k^3$	$3k/2$	$2k^2$	6	Yes <sup>1</sup>
Pyramid	$(4k^2 - 1)/3$	$2 \log_2 k$	$2k$	9	No
Binary tree	$2^l - 1$	$2l - 2$	1	3	No
4-ary hypertree	$2^l(2^{l+1} - 1)$	$2l$	$2^{l+1}$	6	No
Butterfly	$2^l(l + 1)$	$2l$	$2^l$	4	No
Hypercube	$2^l$	$l$	$2^{l-1}$	$l$	No
Cube-connected cycles	$2^l l$	$2l$	$2^{l-1}$	3	No
Shuffle-exchange	$2^l$	$2l - 1$	$\geq 2^{l-1}/l$	4 unidir.	No
De Bruijn	$2^l$	$l$	$2^l/l$	4 unidir.	No

<sup>1</sup> With folded layout



# Interconnection Network Topics

- Interconnection networks for parallel computers
  - components
  - characteristics
  - network models
- Analysis of networks
  - diameter
  - bisection bandwidth
  - degree
  - cost
  - example networks
- Simple cost measures for communication
  - store-and-forward model
  - cut-through model

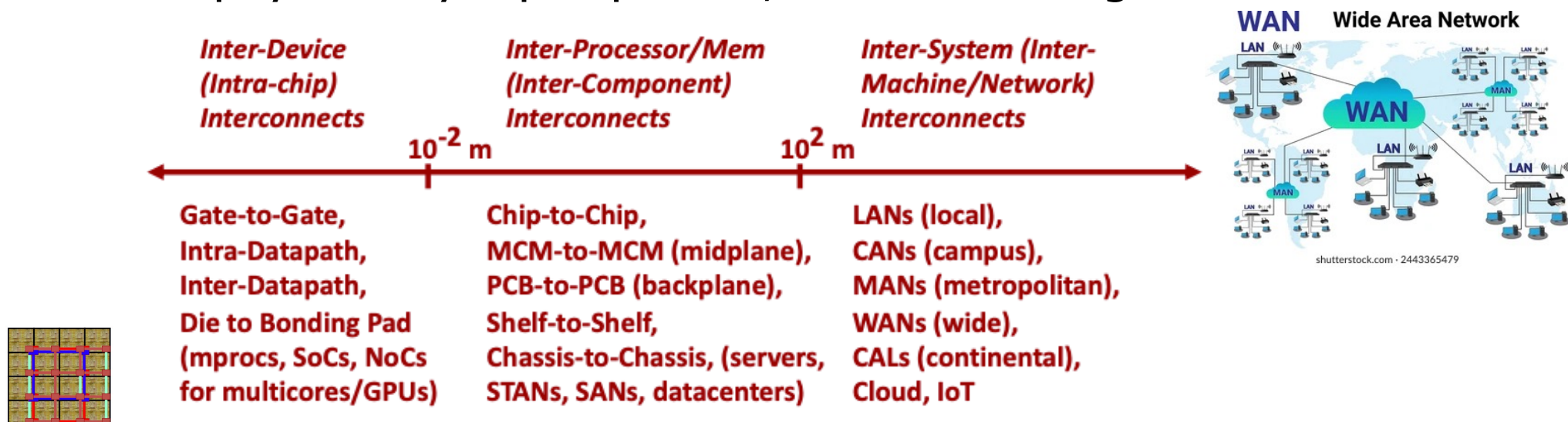


# Kinds of Networks

- Wide-area networks (WAN)
  - telephone, internet
- Local-area networks (LAN)
  - ethernet, wireless 802.11x
- System-level networks
  - processor to processor
  - (processor to memory)
- These networks differ in scalability, assumptions, cost
  - Primary focus of our discussion is system-level networks

# Interconnection Network Domains Wider Scale

- Communication and computation occur at many levels
- Which designs make sense for particular technologies, architectures, applications, etc., and at which levels?
- From physical layer perspective, three broad regimes:



*Similar principles apply*

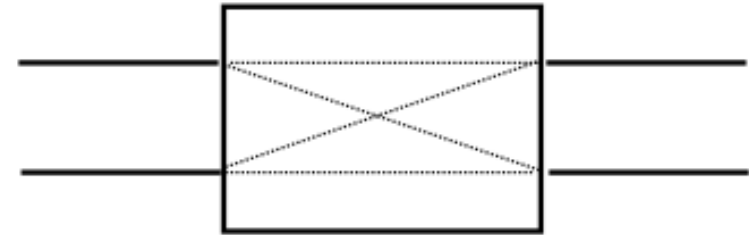
# Components of a network

- **clusters**

- each processor has a dedicated network interface

- **switches**

- k inputs, m outputs,  $m \geq k$ 
  - simplest:  $k = m = 2$



- **links**

- characteristic bandwidth
  - (# parallel bits per link) • (signaling rate)

# Four characteristics of networks

- **Network topology**

- physical interconnection structure of network
  - analogy: Roadmap showing interstates

- **Routing algorithm**

- rules that specify which routes a message may follow
  - analogy: To drive from Delft to Amsterdam, take A13 and then A4

- **Switching Strategy**

- determines how a message traverses a route
  - analogy: Presidential convoy reserves entire route in advance, while a group of travelers in separate cars make individual switching decisions

- **Flow control**

- determines when a message makes progress
  - analogy: Traffic signals and rules: two cars cannot occupy the same location at the same time

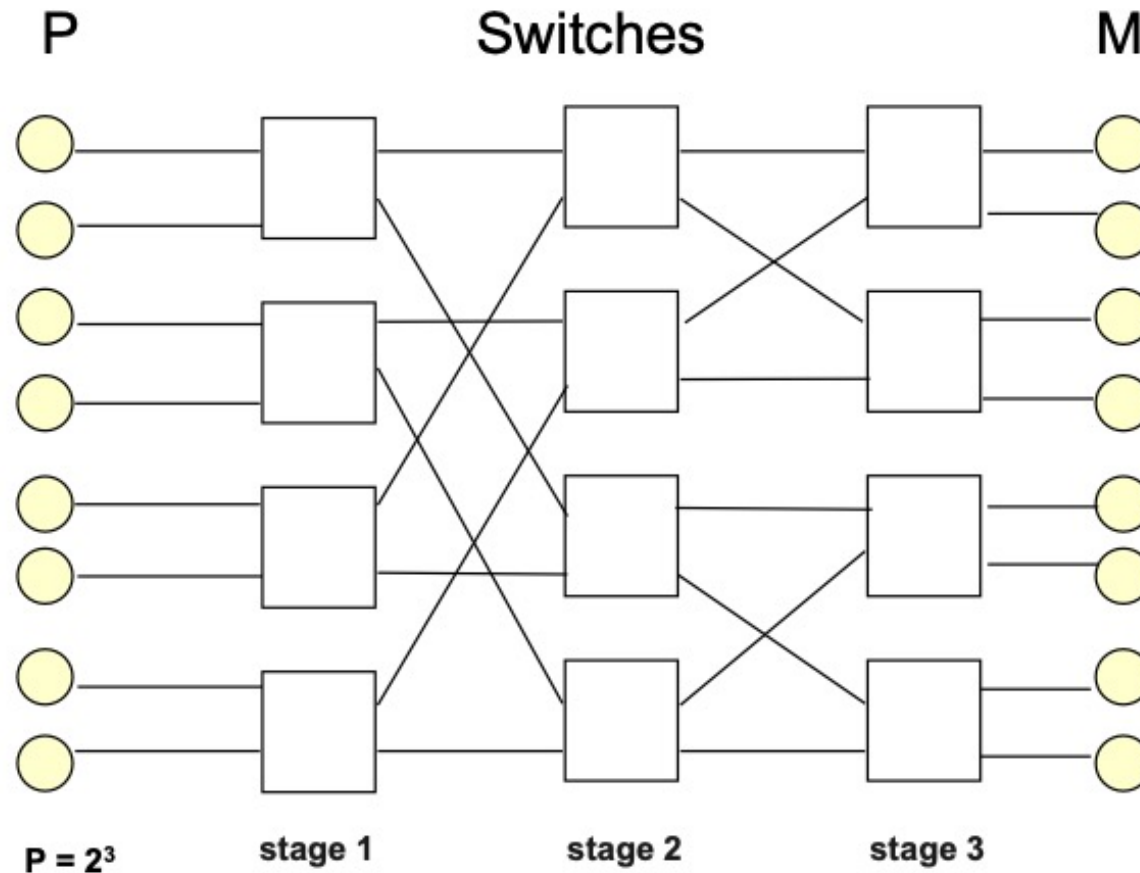
# Network topology

- **Connected undirected graph  $G = (N, C)$** 
  - $N$  = set of nodes
  - $C$  = set of channels (bidirectional links)
- **Indirect network (switching fabric)**
  - contains switch nodes without an attached processor or memory
  - switching nodes do not generate traffic
  - typical case in modern networks
- **Direct network**
  - every node can be a producer and/or consumer of messages
  - no pure switching nodes

# Indirect networks

- **Processor to memory interconnect in shared-memory machines**
- **Connect  $p$  processors to  $p$  memory banks**
  - Example: bus
    - $\Theta(p)$  switches
    - *simultaneous references always serialize*
  - Example: crossbar
    - $\Theta(p^2)$  switches
    - *simultaneous references in disjoint banks serviced in parallel*
  - Example: multistage network
    - $\Theta(p \lg p)$  switches and links
      - $\Theta(\lg p)$  stages of  $\Theta(p)$  switches each
    - *simultaneous reference of disjoint memories may be serialized*
      - contention within the network

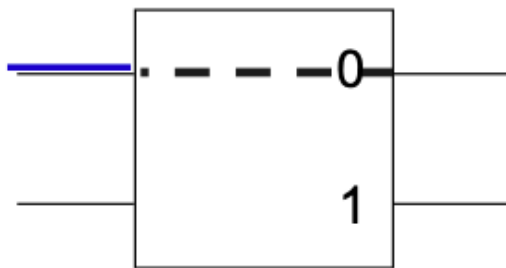
# Multistage Butterfly indirect network ( $p = 8$ )



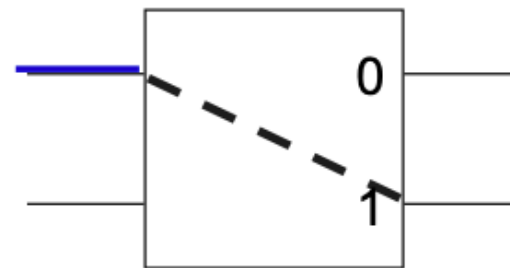


# Routing in butterfly networks

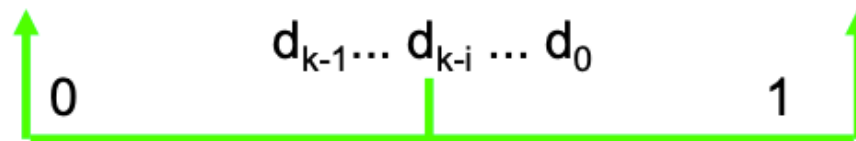
- based on destination address
  - destination address  $d_{k-1} \dots d_0$
  - in stage  $i$ , switch setting is determined by  $d_{k-i}$ 
    - *switch to top or bottom*



*Switch to top*

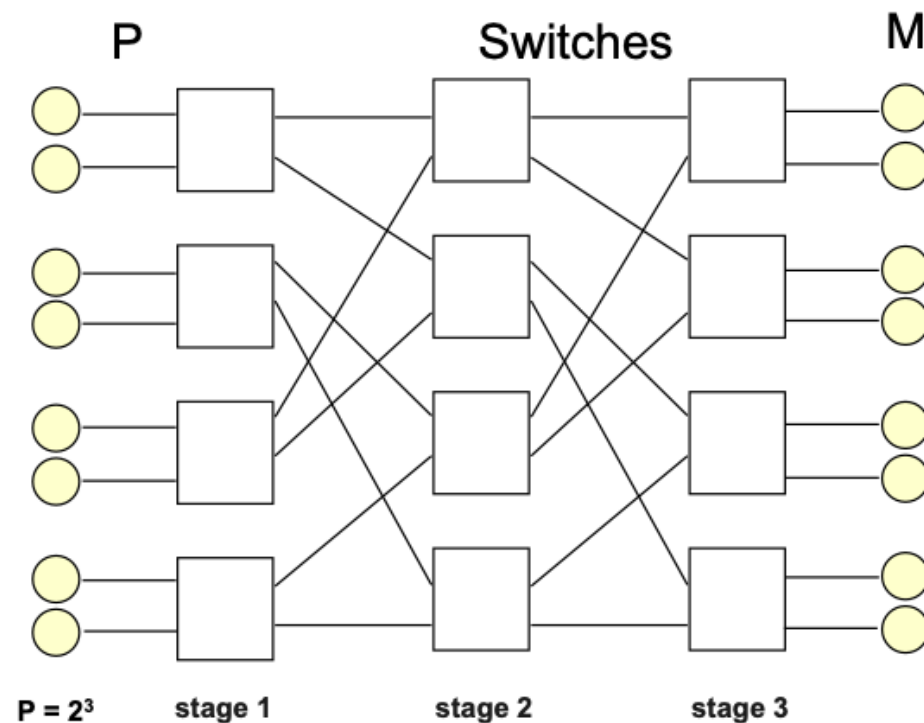


*Switch to bottom*



# Multistage Omega network ( $p = 8$ )

- Isomorphic to butterfly network
  - same “perfect shuffle” connection pattern between successive stages



# Network Topology: Graph-theoretical measures

- **Diameter:** *Maximum length of shortest path between any pair of nodes*

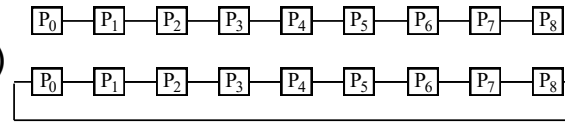
$$\max_{u,v \in N} \left( \min_{u \rightarrow v \in C^*} |u \rightarrow v| \right)$$

- i.e. distance between maximally separated nodes - related to latency
- **Bisection width:** *Minimum number of edges crossing approximately equal bipartition of nodes*
  - related to bandwidth with full applied load
  - a *scalable* network has bisection width  $\Omega(p)$
- **Degree:** *number of edges (links) per node (switch)*
  - related to cost and switch complexity
  - fixed degree is simpler and more scalable
- **Cost:** *number of wires*
  - length of wires and wiring regularity is also an issue

# Linear array, Ring, Binary Tree, Mesh and Crossbar (review)

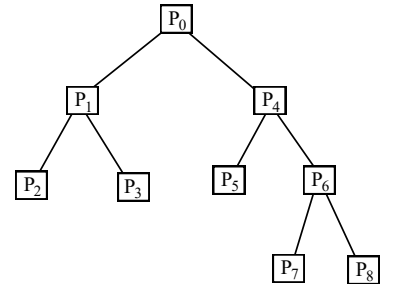
Max node degree  
Network diameter  
Bisection width

$$\begin{aligned} d &= 2 \\ D &= p - 1 \quad (\lfloor p/2 \rfloor) \\ B &= 1 \quad (2) \end{aligned}$$

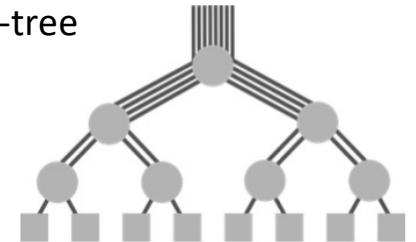


Max node degree  
Network diameter  
Bisection width

$$\begin{aligned} d &= 3 \\ D &= 2 \lfloor \log_2 p \rfloor \quad (-1) \\ B &= 1 \end{aligned}$$

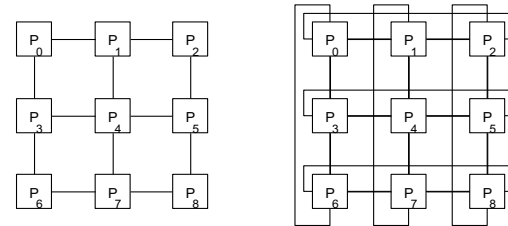


Fat-tree



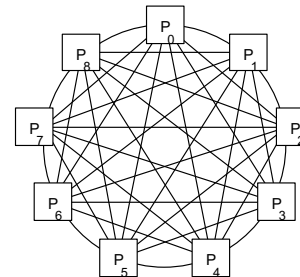
Max node degree  
Network diameter  
Bisection width

$$\begin{aligned} d &= 4 \\ D &= 2\sqrt{p} - 2 \quad (\sqrt{p}) \\ B &\cong \sqrt{p} \quad (2\sqrt{p}) \end{aligned}$$



Max node degree  
Network diameter  
Bisection width

$$\begin{aligned} d &= p - 1 \\ D &= 1 \\ B &= \lfloor p/2 \rfloor \lceil p/2 \rceil \end{aligned}$$



# Networks in current parallel computers

- **Modern interconnects are indirect**
  - Hardware routing between source and destination
- **Indirect networks**
  - Cluster of commodity nodes
    - Fat-tree (assembled using 36 port non-blocking switches)
  - IBM Summit (ORNL)
    - Fat-tree Infiniband [4,608 nodes] (24,000 GPU, 202,752 cores)
  - Fujitsu Fugaku
    - 6D torus [160,000 nodes k-ary d-cube, ?  $k \sim 7$   $d=6$ ] (3M+ cores)
- **Processor – memory interconnects (p procs, m memories)**
  - Tera MTA
    - 3D torus ( $p = 256$ ,  $m = 4,096$ )
  - NEC SX-9
    - crossbar ( $p = 16$  procs \* 16 channels/proc = 256,  $m = 8,192$ )

# Routing and flow control

- **System-level networks**

- Tradeoffs are very different than WAN (TCP)
  - use flow control instead of dropping packets
  - mostly static routing instead of dynamic routing
- Routing algorithm
  - prescribes a unique path from source to destination
    - e.g., dimension ordered routing on hypercube and lower dimensional d-cubes
    - some networks dynamically “misroute” if a needed link is unavailable
  - routing can be *store-and-forward* or *cut-through*
- Flow control
  - contention for output links in a switch can block progress
  - generally low-latency per-link flow control is used
    - delay in access to a link rapidly propagates back to sender

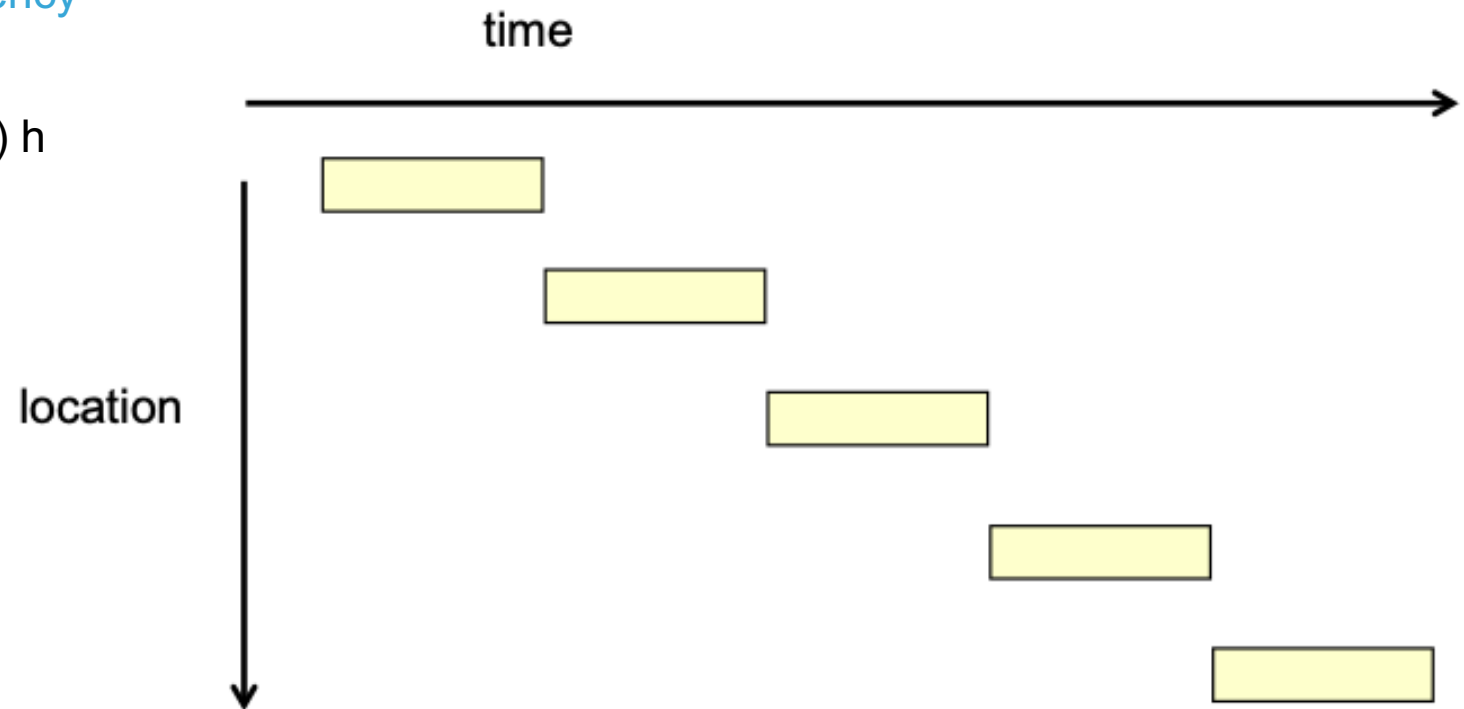
# Communication cost model

- Message size  $m$  bits
- Number of hops (links) to travel  $h$
- Channel width  $W$  and link cycle time  $t_c$ 
  - Per-bit transfer time  $t_w = t_c/W$ 
    - assuming  $m$  is sufficiently large
- Startup time  $t_s$ 
  - overhead to insert message into network
- Node latency or per-hop time  $t_h$ 
  - time taken by message header cross channel and be interpreted at destination

# Store-and-forward routing

- flow-control mechanism at message or packet level
- packets are transferred one link at a time
- large buffers, high latency
- cost

$$t_{SF} = t_s + (t_h + m t_w) h$$

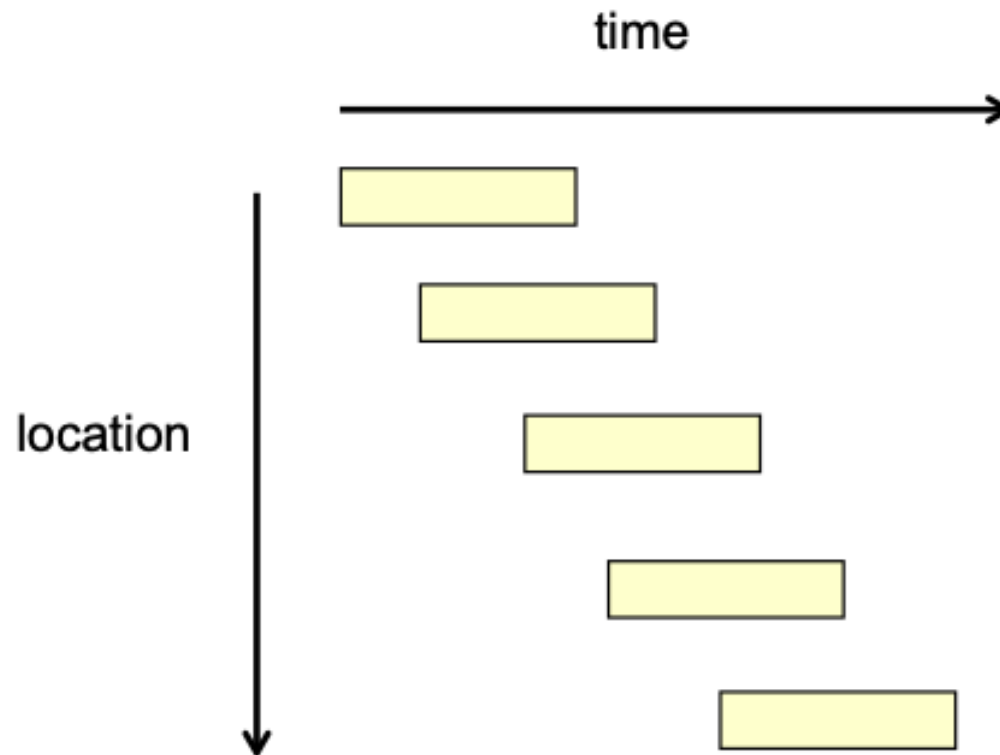




# Cut-through routing

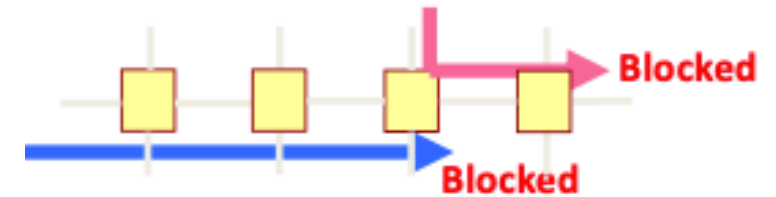
- flow control is per-link and payload transmission is pipelined
- message spread out across multiple links in the network
- small buffers, low latency
- cost

$$t_{CT} = t_s + h t_h + m t_w$$

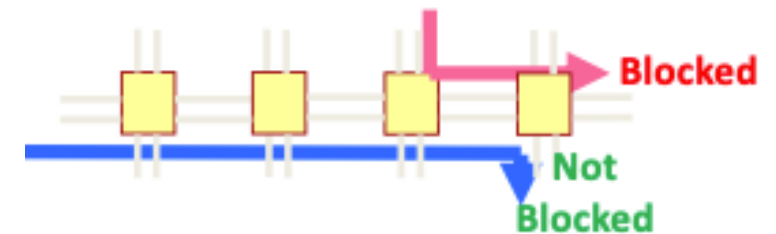


# And much much more

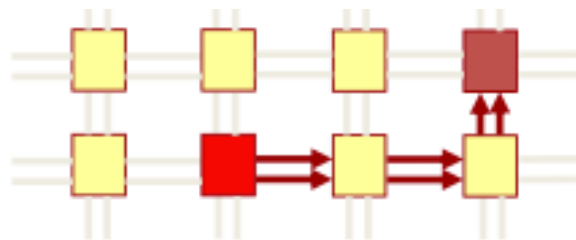
- Virtual Channels to cope with Head-Of-Line (HOL) blocking
- Fully Adaptive Routing to utilize all available network paths
- etc



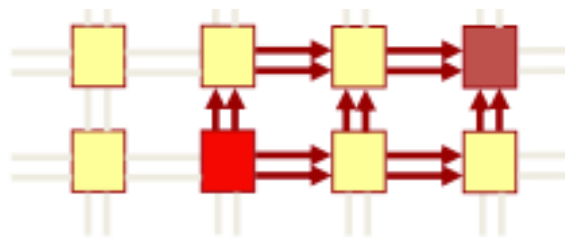
Without Virtual Channels (VCs)



With Virtual Channels (VCs)



Dimension Order Routing (DOR)



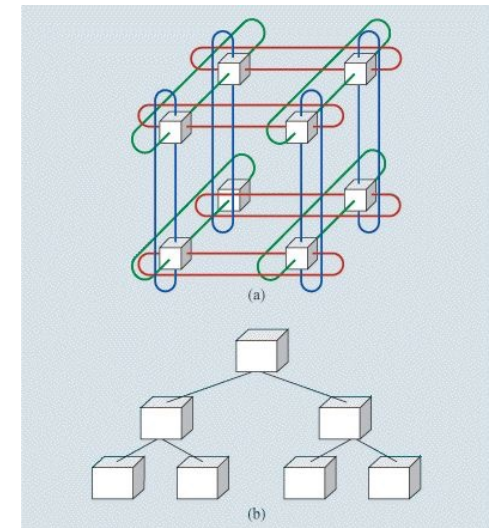
Fully Adaptive Routing (FAR)

# Summary

- CUDA provides programmable Massively Parallel accelerators
- Interconnection Networks play important role
- Memory and Networks kind of merge together



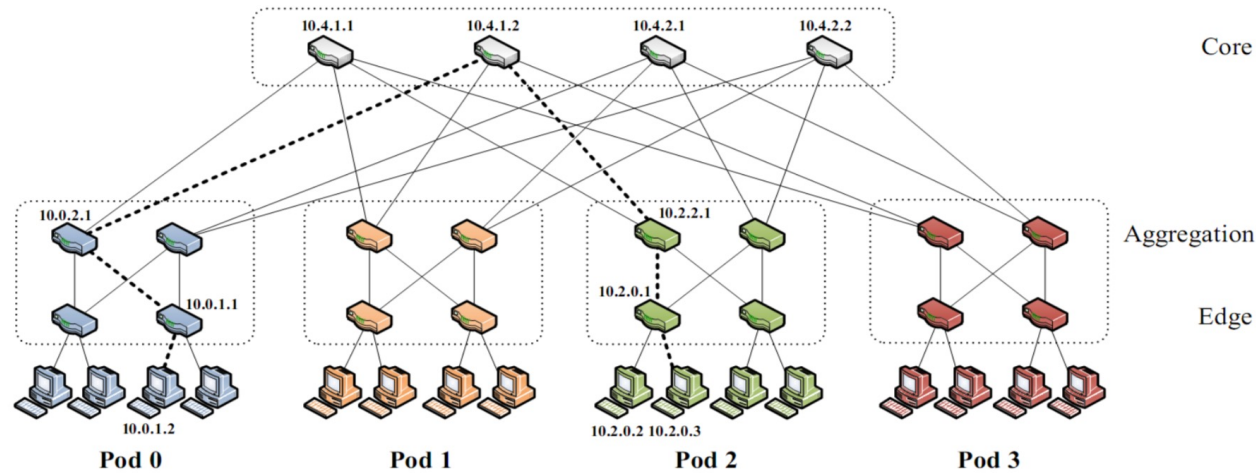
OR





Thank you

# Fat Tree from supercomputers to Data Centers



- K-ary fat tree: three-layer topology (edge, aggregation and core)
- each pod consists of  $(k/2)^2$  servers & 2 layers of  $k/2$  k-port switches
  - each edge switch connects to  $k/2$  servers &  $k/2$  aggregation switches
  - each aggregation switch connects to  $k/2$  edge &  $k/2$  core switches
  - $(k/2)^2$  core switches: each connects to  $k$  pods

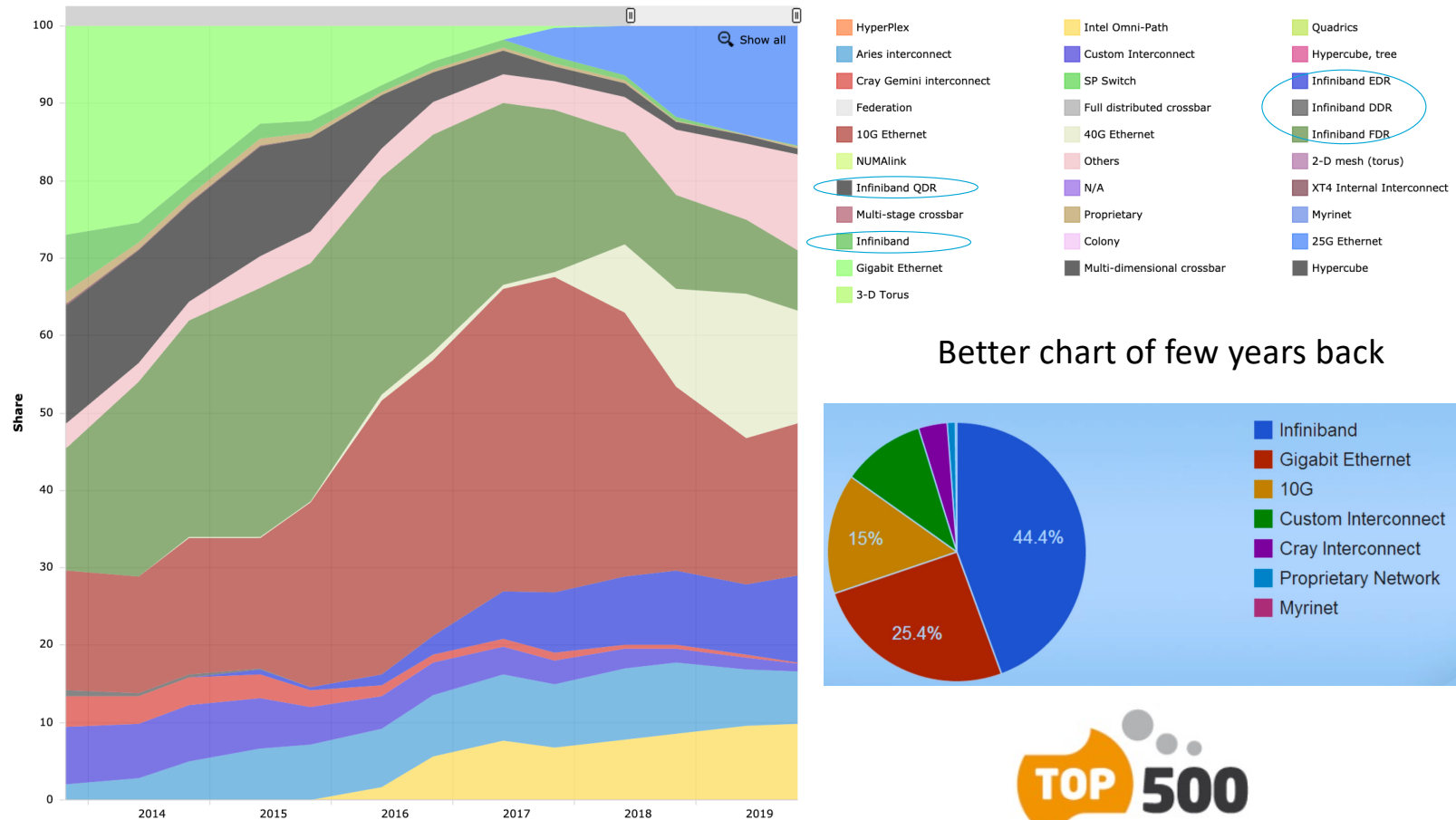
Great **Scalability**, Uniform Bandwidths, Can be built with cheap components

# Interconnects of some TOP-500 Machines

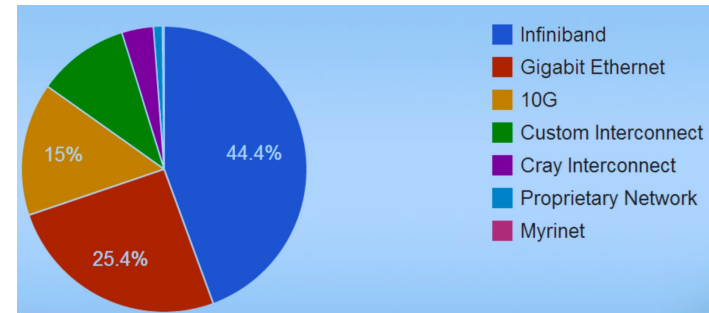
System	Site	Topology	Date
TMC CM-5	Los Alamos National Lab	Fat Tree	6/93~11/93
Fujitsu Numerical Wind Tunnel	National Aerospace Laboratory of Japan	Crossbar	11/93~6/96
Intel XP/S 140 Paragon	Sandia National Labs	2D Mesh	6/94~11/94
Hitachi SR2201	University of Tokyo	3D Crossbar	6/96~11/96
Hitachi CP-PACS	University of Tsukuba	3D Hyper- crossbar	11/96~6/97
Intel ASCI Red	Sandia National Laboratory	Mesh	6/97 ~11/00
IBM ASCI White	Lawrence Livermore National Laboratory	Omega	11/00~6/02
NEC The Earth Simulator	Earth Simulator Center	Crossbar	6/02~11/04
IBM BlueGene/L	Lawrence Livermore National Laboratory	3D Torus	11/04~6/08
IBM Roadrunner	Los Alamos National Laboratory	Fat-Tree hierarchy of crossbars	6/08~11/09
Cray Jaguar	Oak Ridge National Laboratory	3D Torus	11/09~11/10
NUDT Tianhe-1A	National Supercomputing Center in Tianjin	Fat Tree	11/10~6/11
Fujitsu K Computer	RIKEN Advanced Institute for Computational Science	Tofu: 6D Mesh / Torus	6/11~6/12
IBM Sequoia Blue Gene/Q	Lawrence Livermore National Laboratory	5D Torus	6/12~11/12
Cray Titan	Oak Ridge National Laboratory	3D Torus	11/12~6/13
NUDT Tianhe-2	National Super Computer Center in Guangzhou	Fat Tree	6/13~

Must be important

# TOP-500 Interconnect share



Better chart of few years back



# Compute Express Link

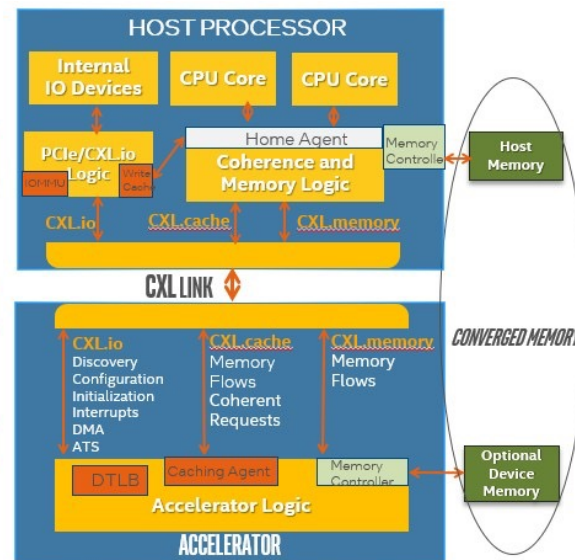
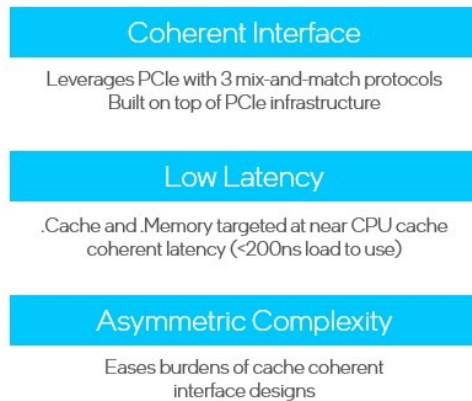


Open standard for CPU-to-Device and CPU-to-Memory in the High Perf Datacenters (2019)  
currently v3.0 (64 GT/s, 7.563 GB/s (x1), 121.0 GB/s (x16); 256-byte FLIT in PAM-4)

Includes *open standards*, e.g., OpenCAPI (IBM), Gen-Z (HPE), and CCIX (Xilinx), and *proprietary protocols*: InfiniBand / RoCE (Mellanox), Infinity Fabric (AMD), Omni-Path and QuickPath/Ultra Path (Intel), and NVLink/NVSwitch (Nvidia)

Supported by the entire Industry (see: <http://www.computeexpresslink.org/>)

## CXL approach

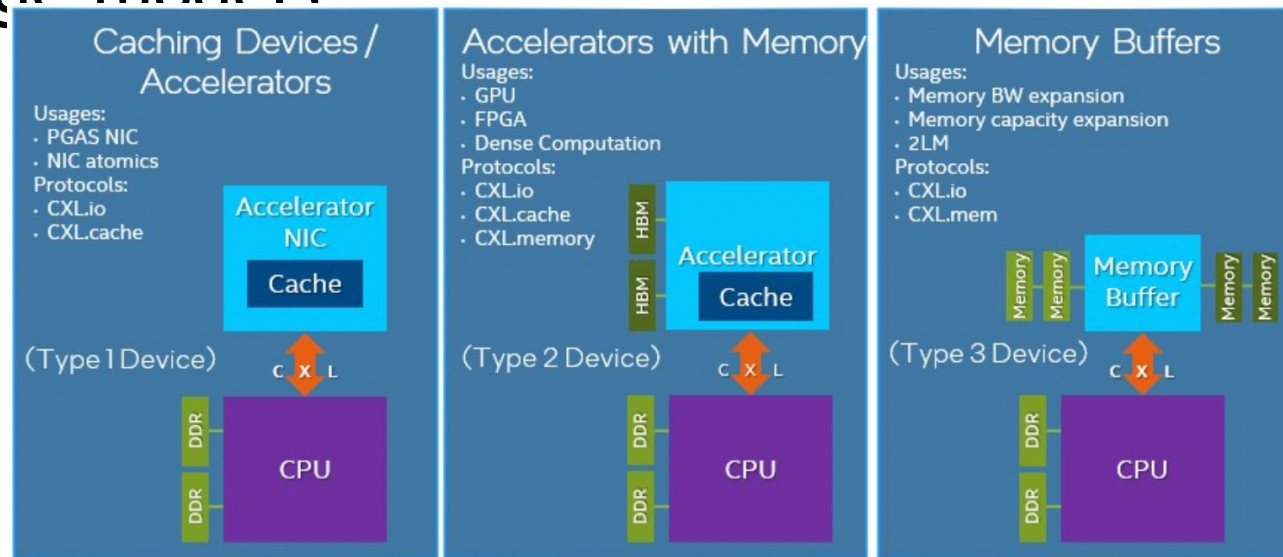


Three distinct protocols:

- **CXL.io** PCIe 5.0 based (configuration, link initialization and management, device discovery and enumeration, interrupts, DMA, and register I/O access using non-coherent loads/stores);
- **CXL.cache** low latency coherent cache access for peripheral devices;
- **CXL.mem** load/store host CPU coherent access to cached device memory (volatile and persistent).



# CXL usage models



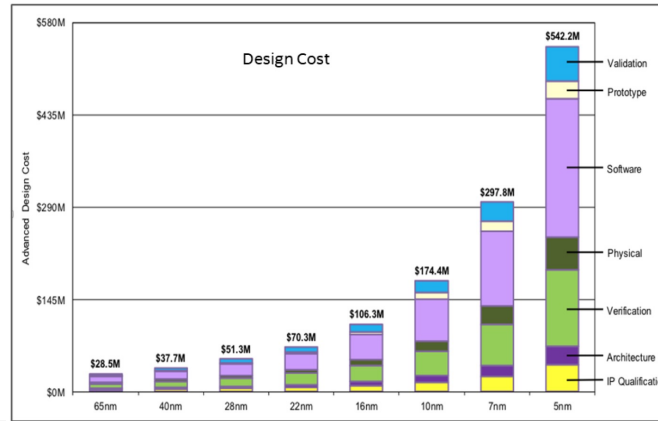
Three primary device types:

**Type 1 (CXL.io and CXL.cache)** – specialised accelerators with no local memory (e.g., smart NIC). Devices rely on coherent access to host CPU memory.

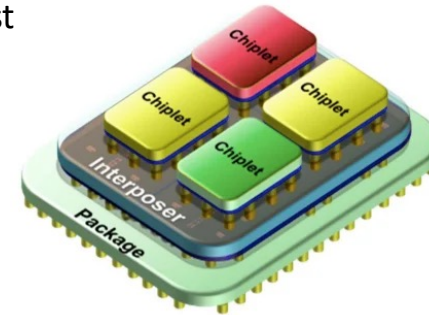
**Type 2 (CXL.io, CXL.cache and CXL.mem)** – General-purpose Accelerators (GPU, ASIC or FPGA) with high-performance GDDR or HBM local memory. Devices can coherently access host CPU's memory and/or provide coherent or non-coherent access to device local memory from the host CPU.

**Type 3 (CXL.io and CXL.mem)** – memory expansion boards and persistent memory. Devices provide host CPU with low-latency access to local DRAM or byte-addressable non-volatile storage.

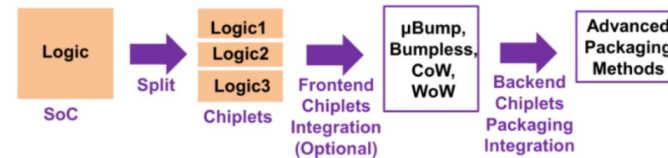
# Chipselets for heterogeneous integration



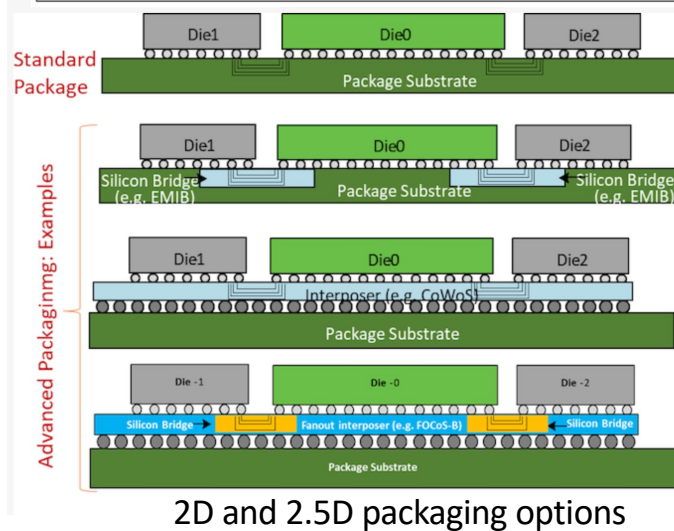
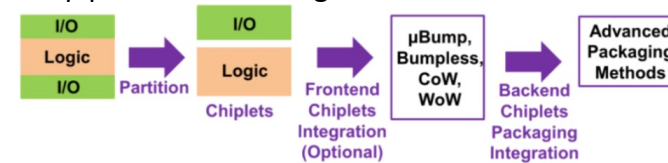
Design costs forecast



## Chip split and integration



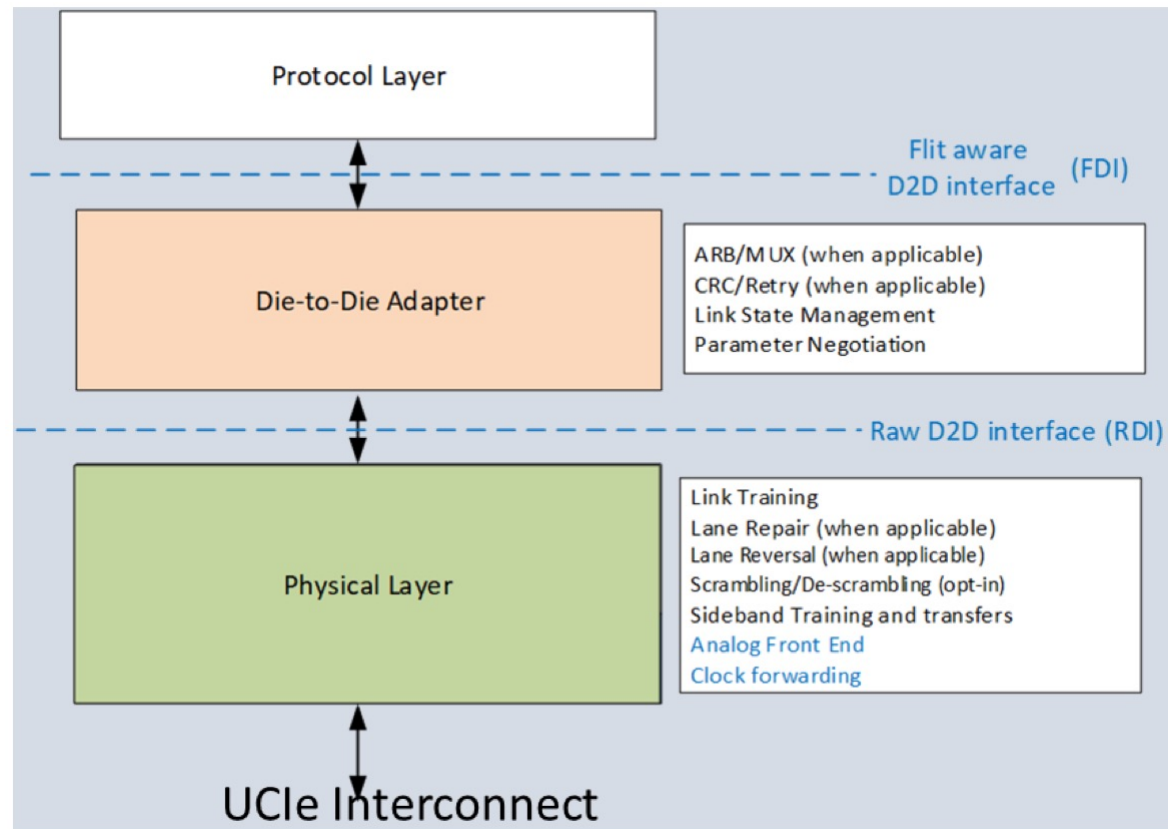
## Chip partition and integration



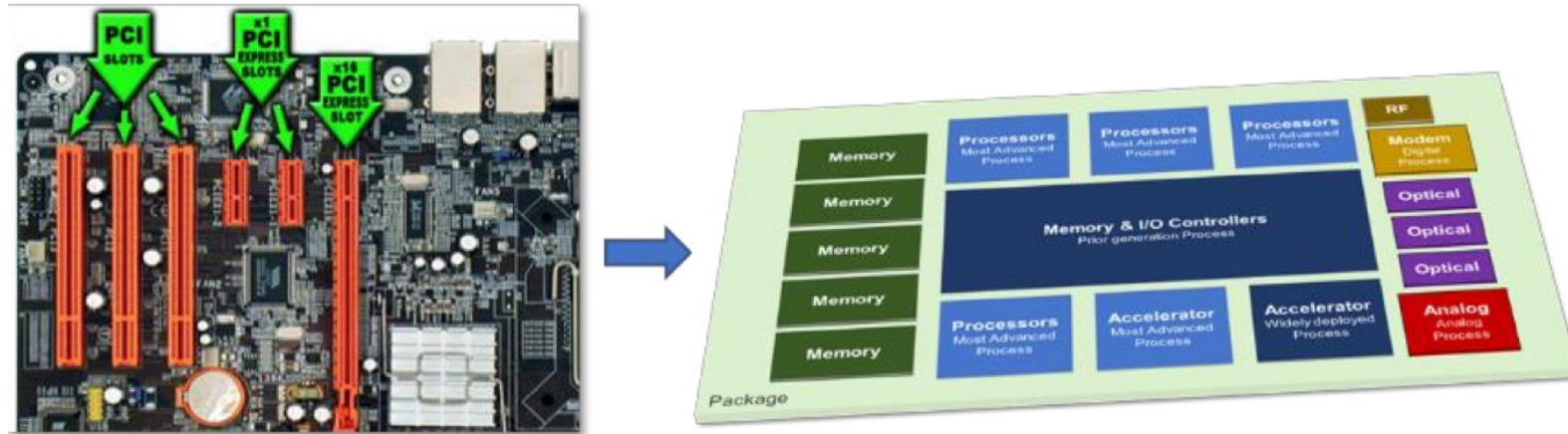
# Universal Chiplet Interconnect Express™ (UCIe)

Characteristics / KPIs	Standard Package	Advanced Package	Comments
<b>Characteristics</b>			
Data Rate (GT/s)	4, 8, 12, 16, 24, 32		Lower speeds must be supported -interop (e.g., 4, 8, 12 for 12G device)
Width (each cluster)	16	64	Width degradation in Standard, spare lanes in Advanced
Bump Pitch (um)	100 – 130	25 - 55	Interoperate across bump pitches in each package type across nodes
Channel Reach (mm)	<= 25	<=2	
<b>Target for Key Metrics</b>			
B/W Shoreline (GB/s/mm)	28 – 224	165 – 1317	Conservatively estimated: AP: 45u for AP; Standard: 110u; Proportionate to data rate (4G – 32G)
B/W Density (GB/s/mm²)	22-125	188-1350	
Power Efficiency target (pJ/b)	0.5	0.25	
Low-power entry/exit	0.5ns <=16G, 0.5-1ns >=24G		Power savings estimated at >= 85%
Latency (Tx + Rx)	< 2ns		Includes D2D Adapter and PHY (FDI to bump and back)
Reliability (FIT)	0 < FIT (Failure In Time) << 1		FIT: #failures in a billion hours (expecting ~1E-10) w/ CXi Flit Mode

# UCIe Layers



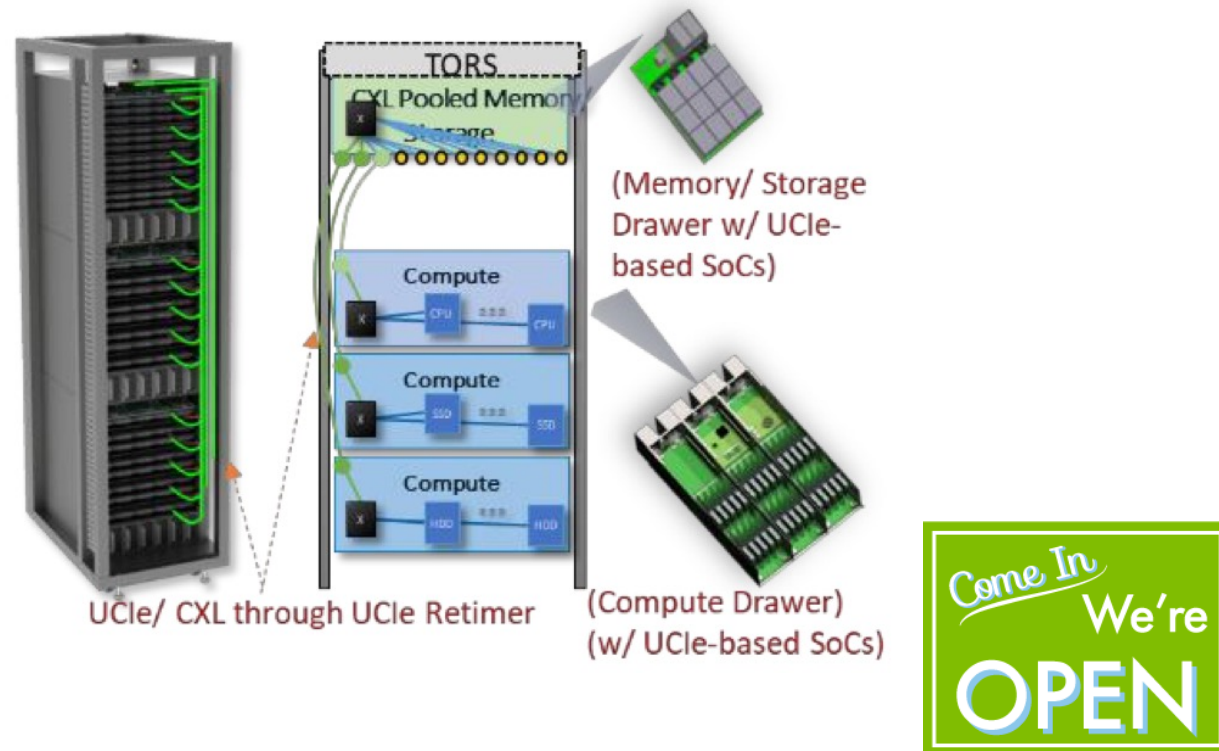
# UCIe Board to Package integration



Die - 1		Die - 2			
x16	<-->	x16	CL-0 x16	<-->	CL-0 x16
x32	<-->	x32	CL-0 x16	<-->	CL-0 x16
			CL-1 x16	<-->	CL-1 x16
x64	<-->	x64	CL-0 x16	<-->	CL-0 x16
			CL-1 x16	<-->	CL-1 x16
			CL-2 x16	<-->	CL-2 x16
			CL-3 x16	<-->	CL-3 x16

(1, 2, or 4 Clusters can be combined in one UCIe Link)

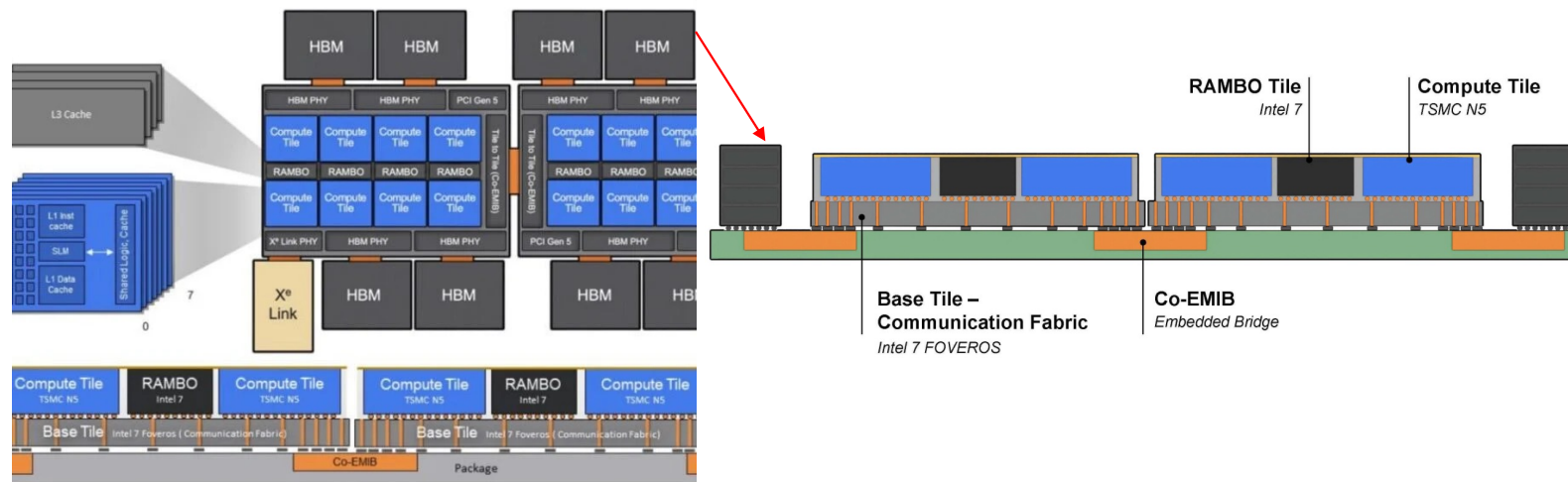
# UCIe and CXL integration



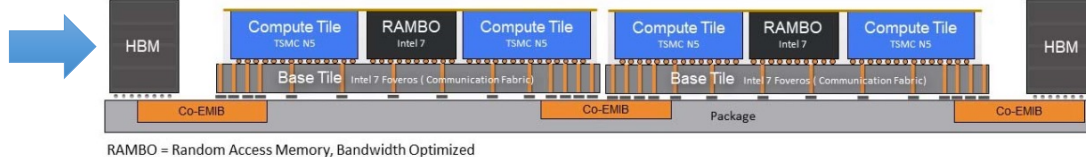
UCIe 1.0 (2022) is open chiplet standard to unleash ecosystem and innovations across the compute continuum with power-efficiency and cost-effectiveness in mind. It envisions a plug-and-play model, modelled after several successful standards, driven by industry leaders for wide-spread adoption.



# Ponte Vecchio



**HBM: 3D stacked**  
High Bandwidth Memory  
(covered in the next topic)



3D processor with 47 functional tiles on five process nodes and connected with two different chiplet technologies (TSMC's N5 5nm, and Intel 7 memory tiles optimized for random access bandwidth-optimized SRAM tiles (RAMBO)). These are stacked on two Foveros base die built with the Intel 7, 17 metal layer process, with each base die measuring 646mm<sup>2</sup>/ 100Bln+ transistors