GPU Programming #0: Introduction

Wei-Chao Chen

Visiting Professor, National Taiwan University

weichao.chen@gmail.com

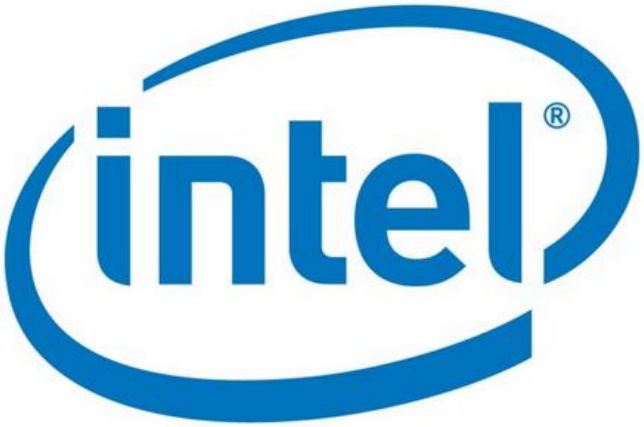
Background

- We offered the “GPU Programming” course from 2010 ~ 2018 at NTU CSIE (sporadically).
- The contents reflect my experiences in building the first CUDA chips, drawing on publicly available information.
- The following are excerpts from the **2018** edition of the course, with minor updates.

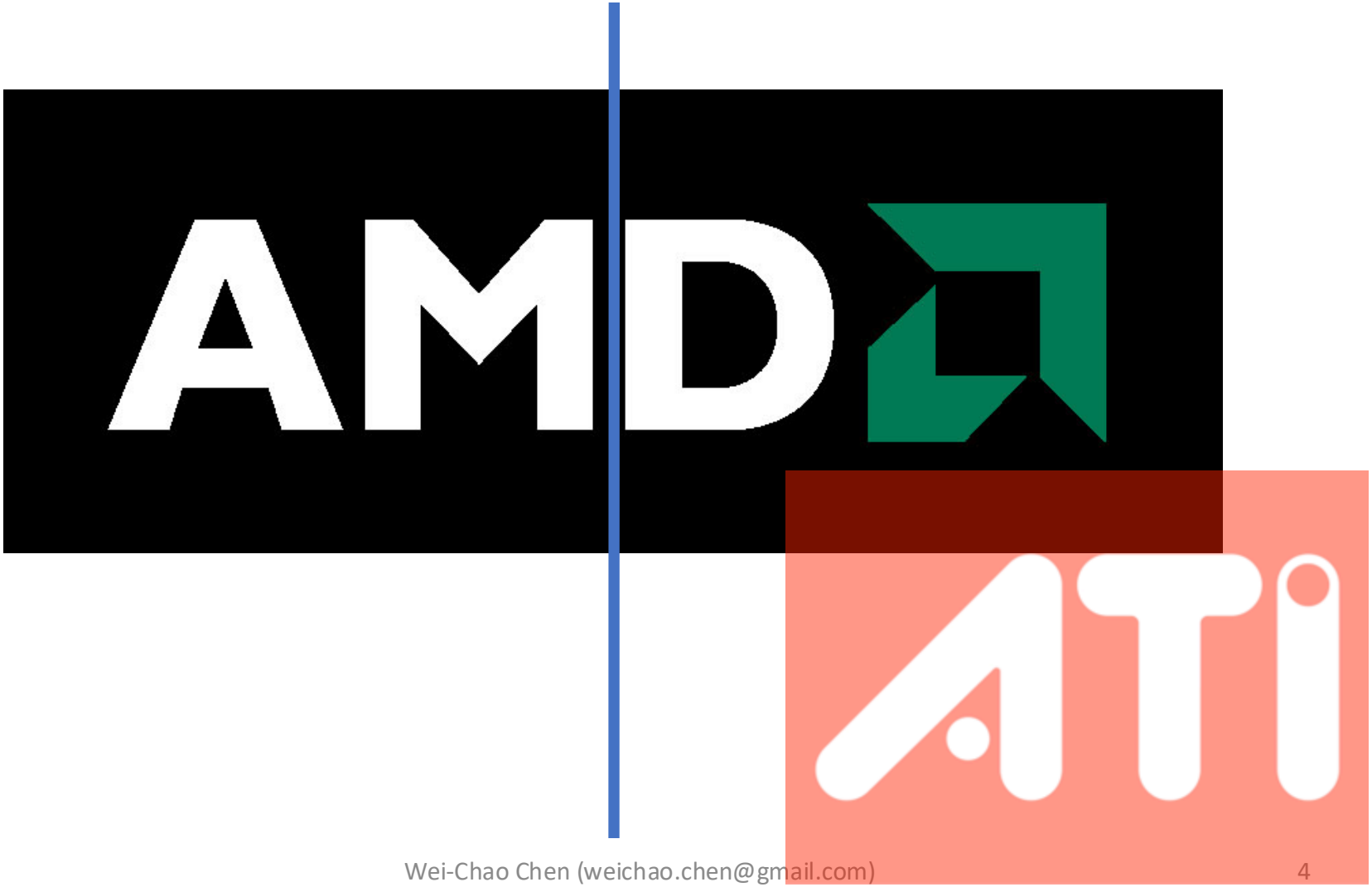
時代感？
經典？

- More about myself: https://www.linkedin.com/in/weichao-chen-tw/

CPU v.s. GPU



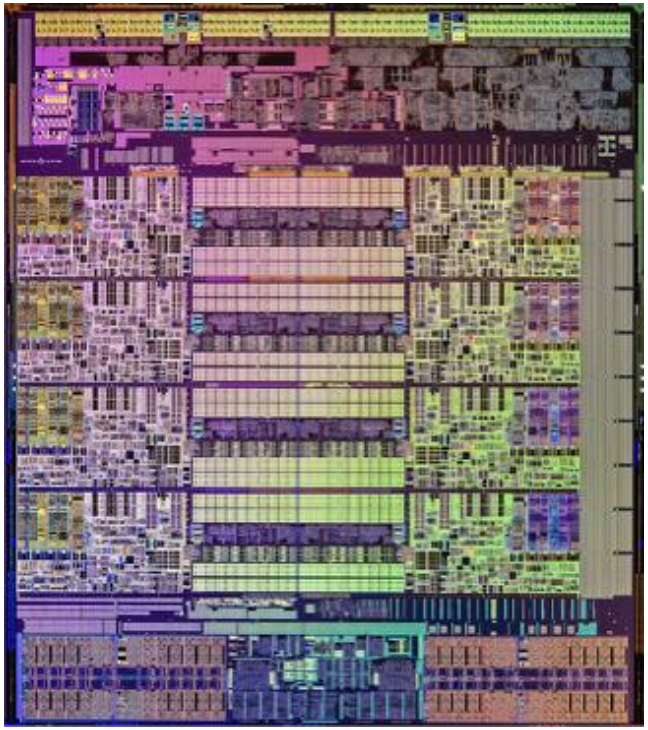
CPU v.s. GPU



CPU v.s. GPU

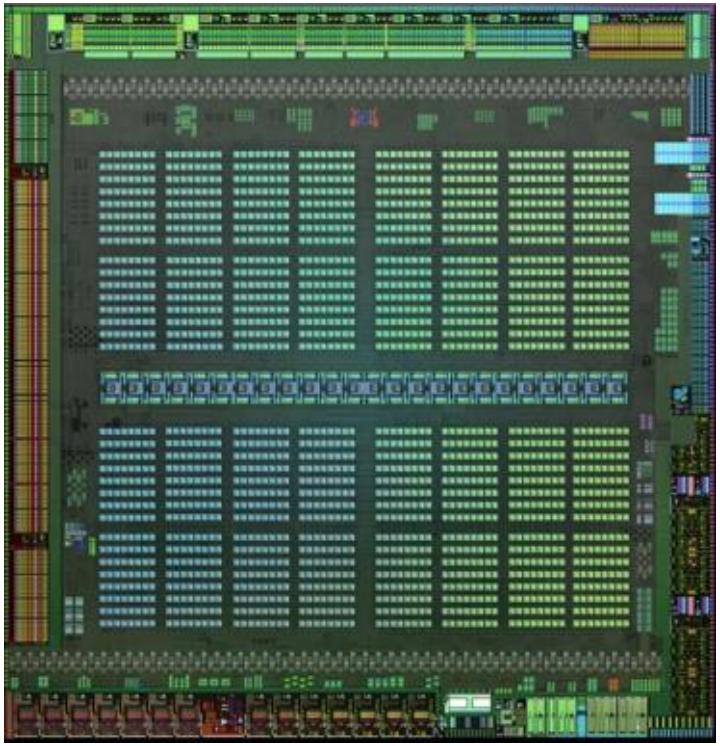
CPU v.s. GPU

Intel Core i7 5960X (2014)
355mm^2 @ 22nm, 2.6B trans.
8 cores



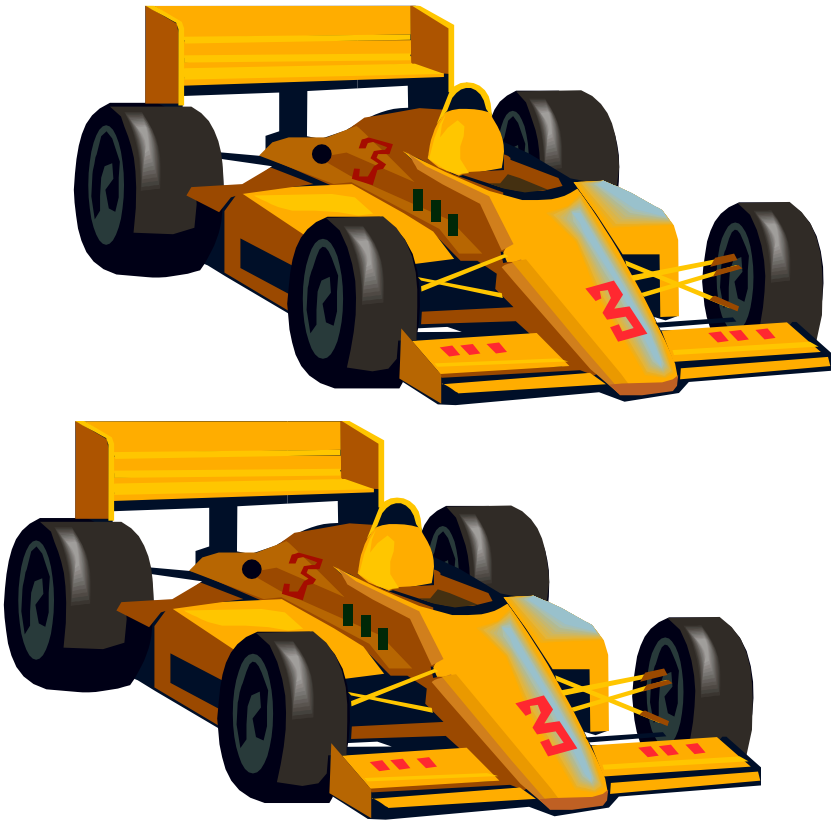
(source: Intel)

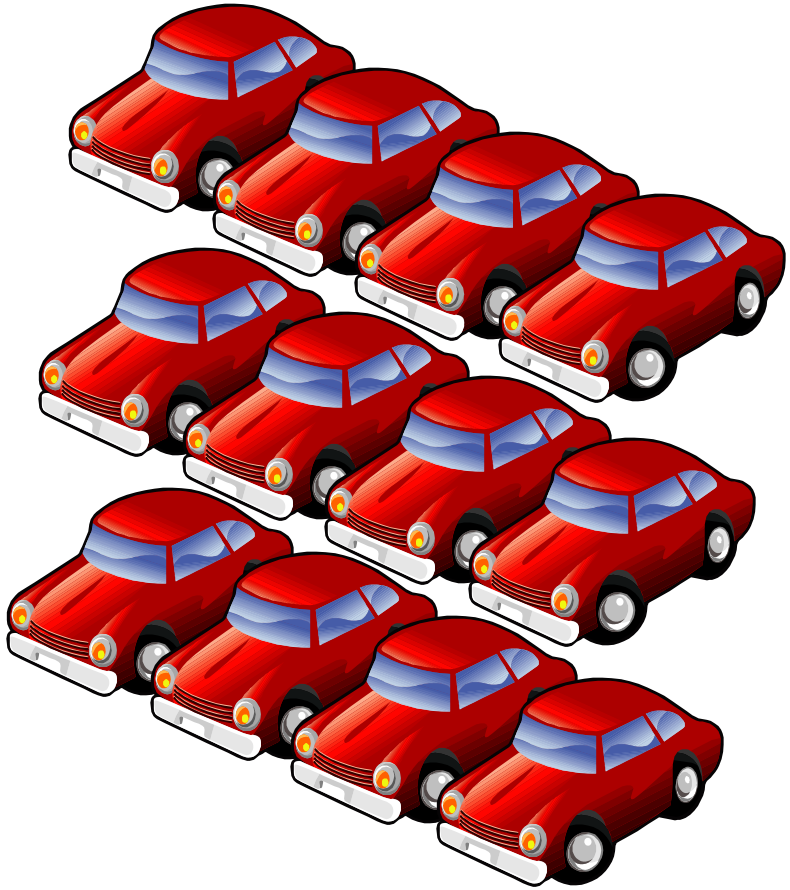
NVIDIA GM204 Maxwell (2014)
398mm^2 @ 28nm, 5.2B trans.
2048 cores

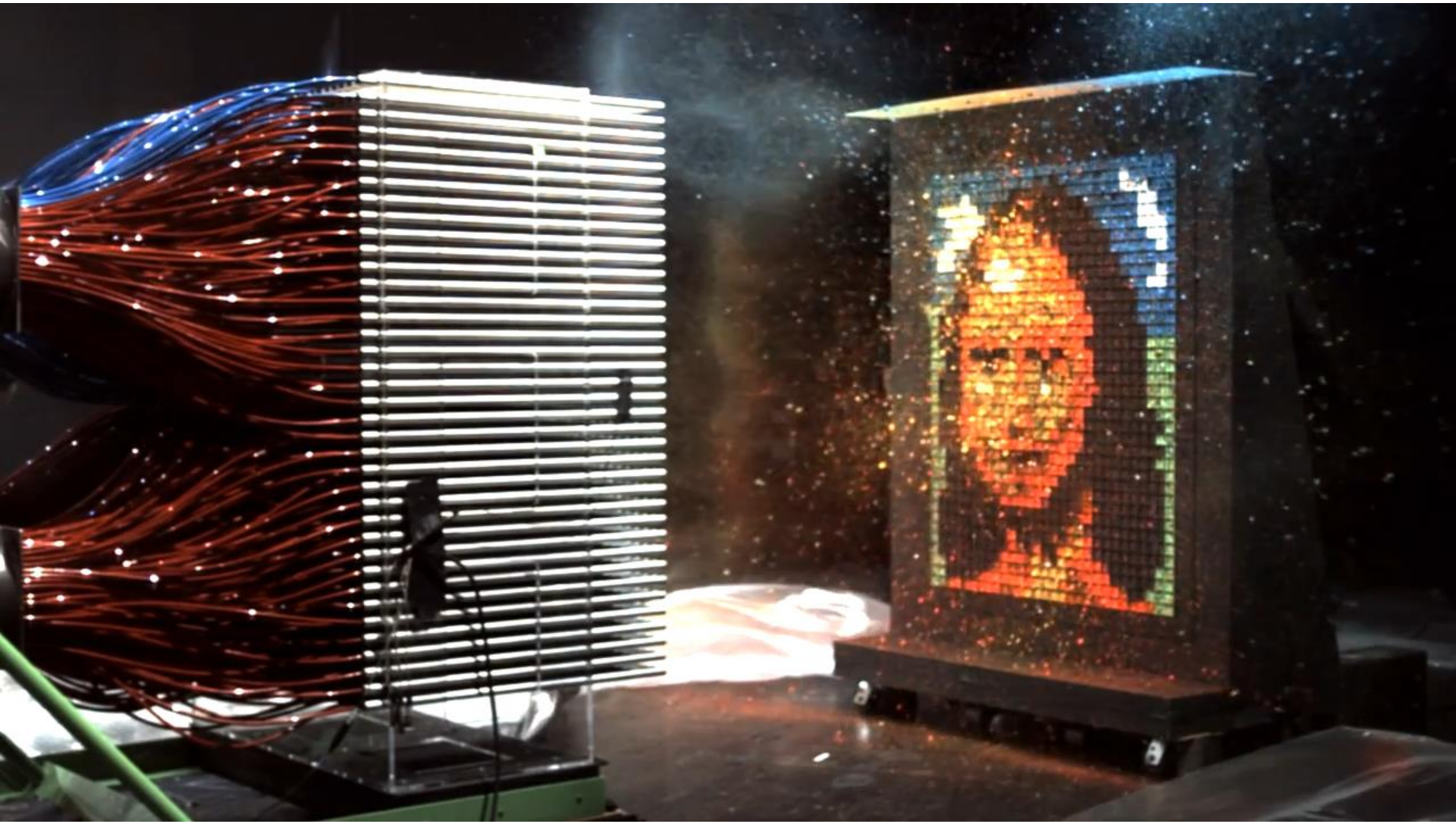


(source: NVIDIA)

CPU v.s. GPU

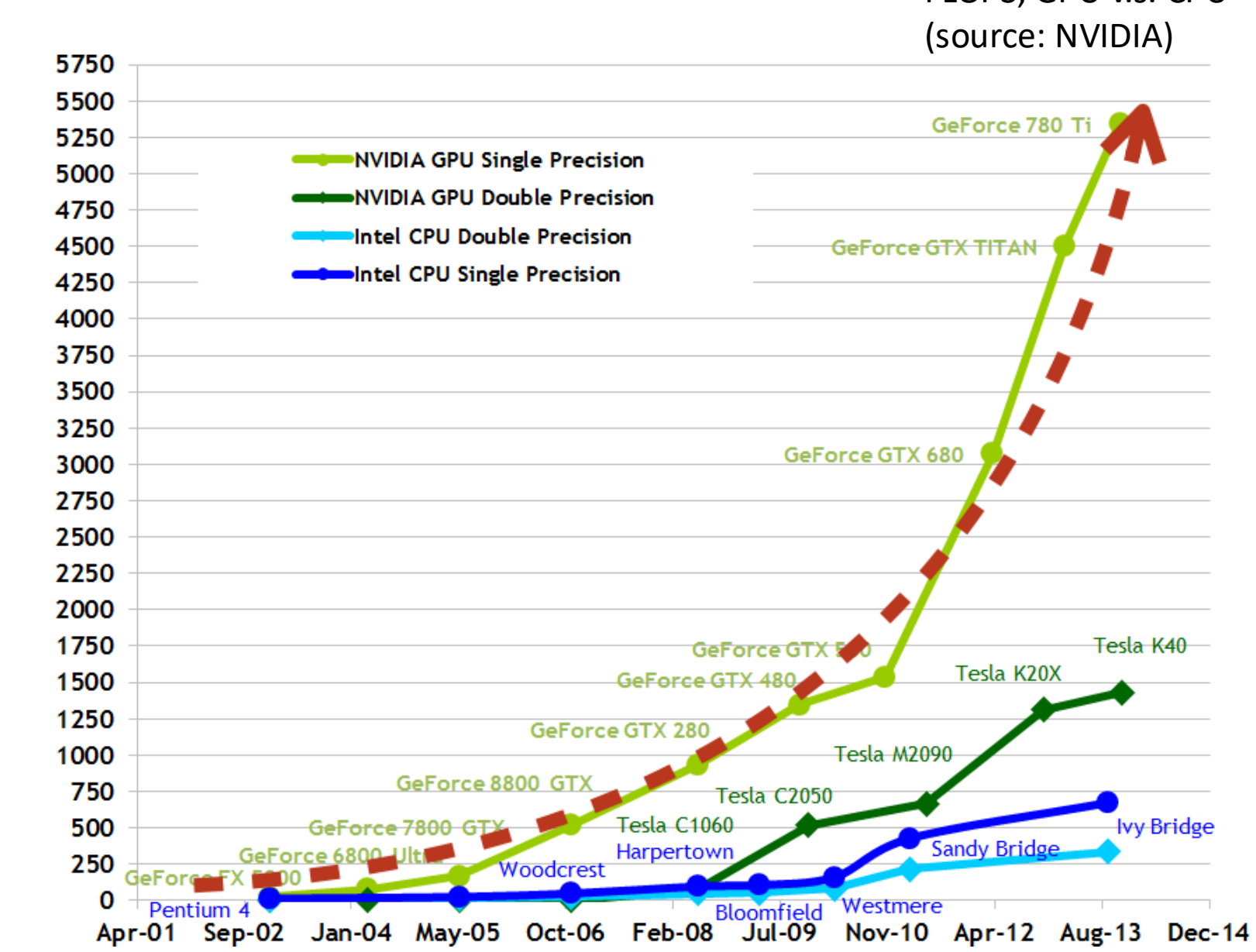




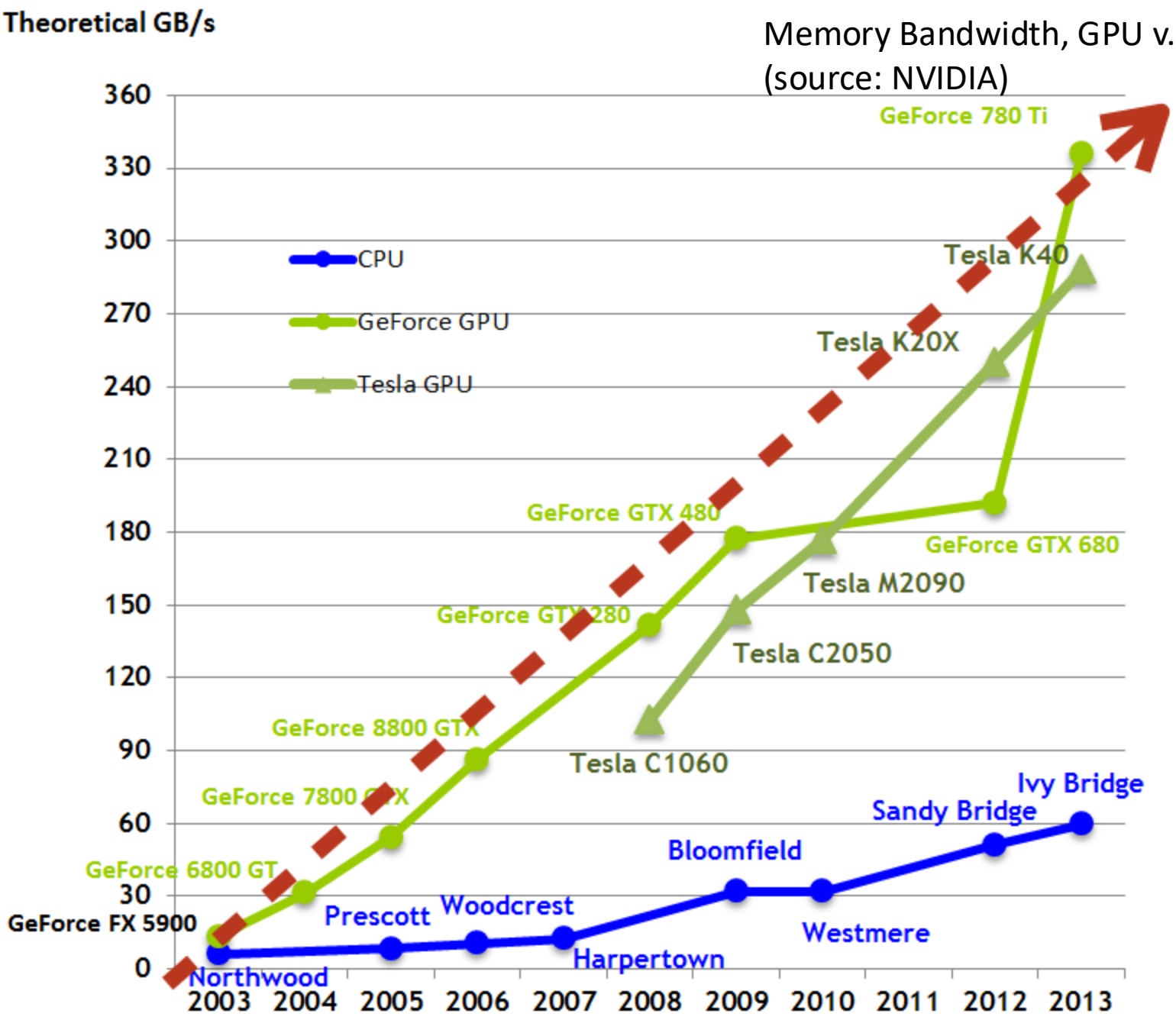


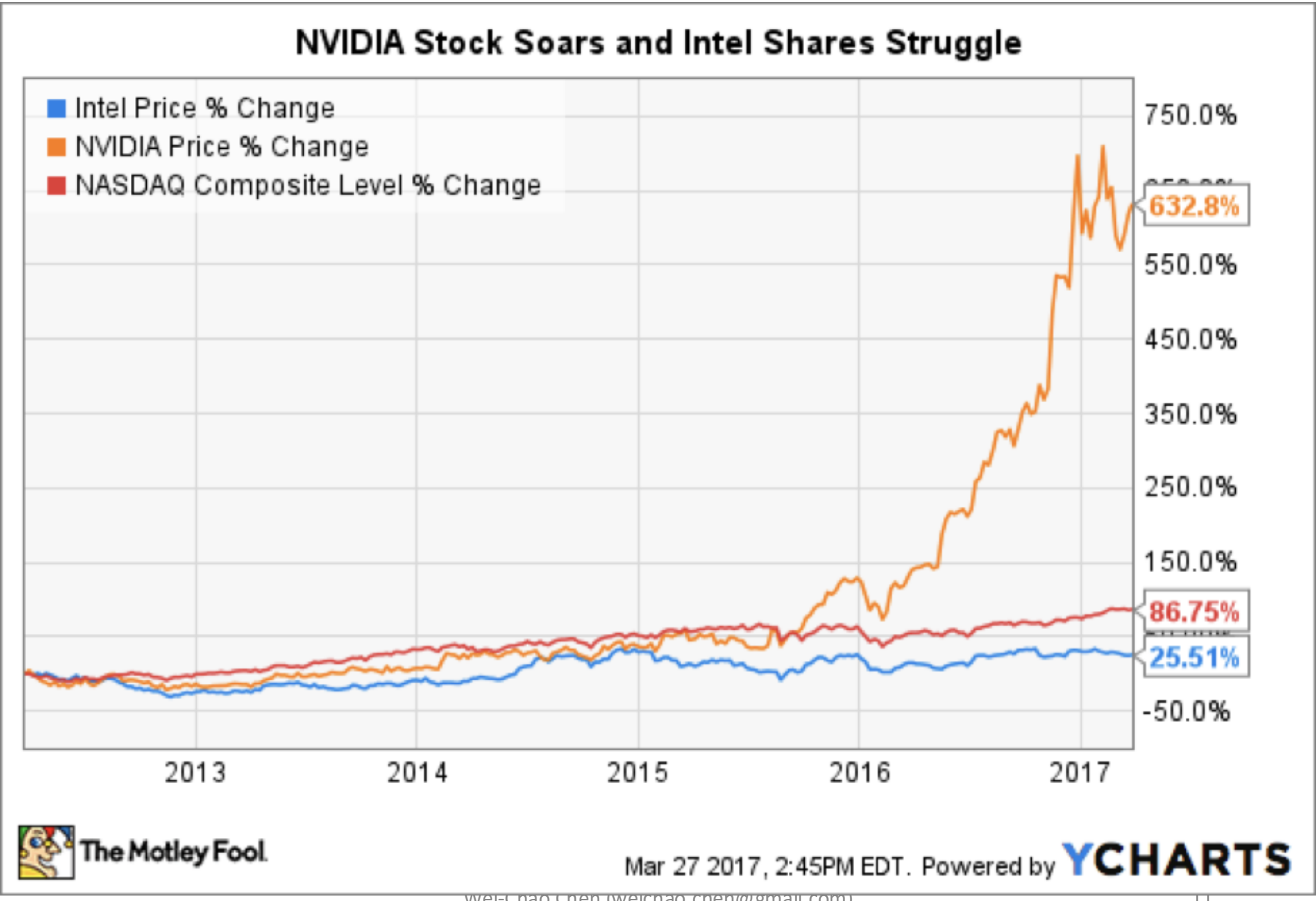
https://www.youtube.com/watch?v=-P28LKWTzrI

FLOPS, GPU v.s. CPU
(source: NVIDIA)



Memory Bandwidth, GPU v.s. CPU
(source: NVIDIA)




NVIDIA Stock Soars and Intel Shares Struggle
Intel Price % Change
NVIDIA Price % Change
NASDAQ Composite Level % Change
750.0%
632.8%
550.0%
450.0%
350.0%
250.0%
150.0%
86.75%
25.51%
-50.0%
2013
2014
2015
2016
2017
The Motley Fool
Mar 27 2017, 2:45PM EDT. Powered by YCHARTS

GPU Applications

GPGPU Applications

https://renderman.pixar.com/

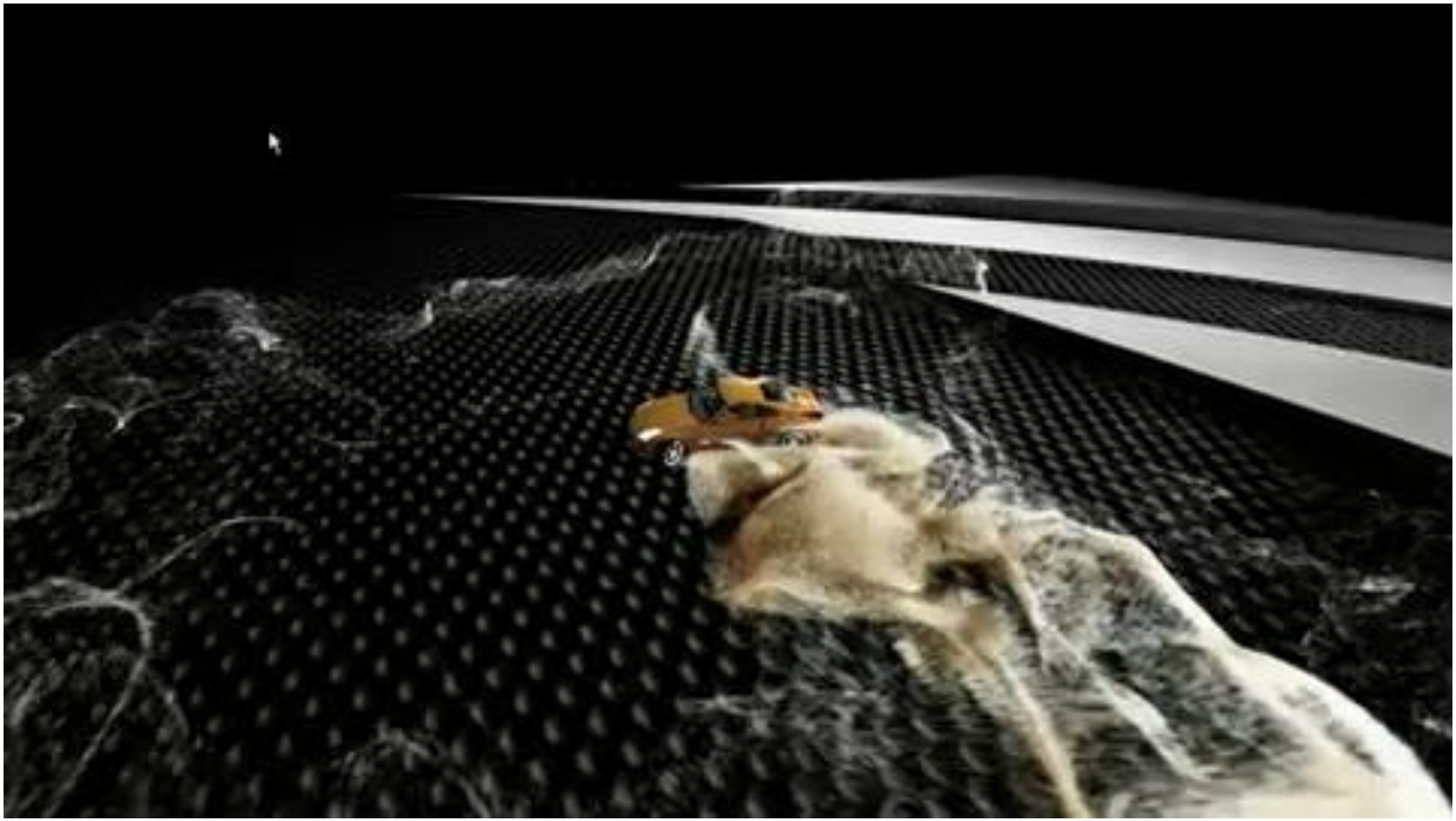
GPUs Today

- GPUs are very programmable
  - Unified & programmable shaders
- GPUs support 32 & 64-bit floating-point numbers
  - *Almost* IEEE FP compliant except for some specials
- GPUs have higher memory bandwidth than CPUs
  - Multiple memory banks driven by the need for high-performance graphics

GPGPU

- General-Purpose GPU Programming
  - Use GPUs as massively data-parallel processor
- Graphics APIs for general computational tasks
  - OpenGL, DirectX
- Moving toward high-level language support
  - CUDA, OpenCL
- Driving force behind modern computing innovations
  - Supercomputing
  - Machine learning
  - Artificial intelligence

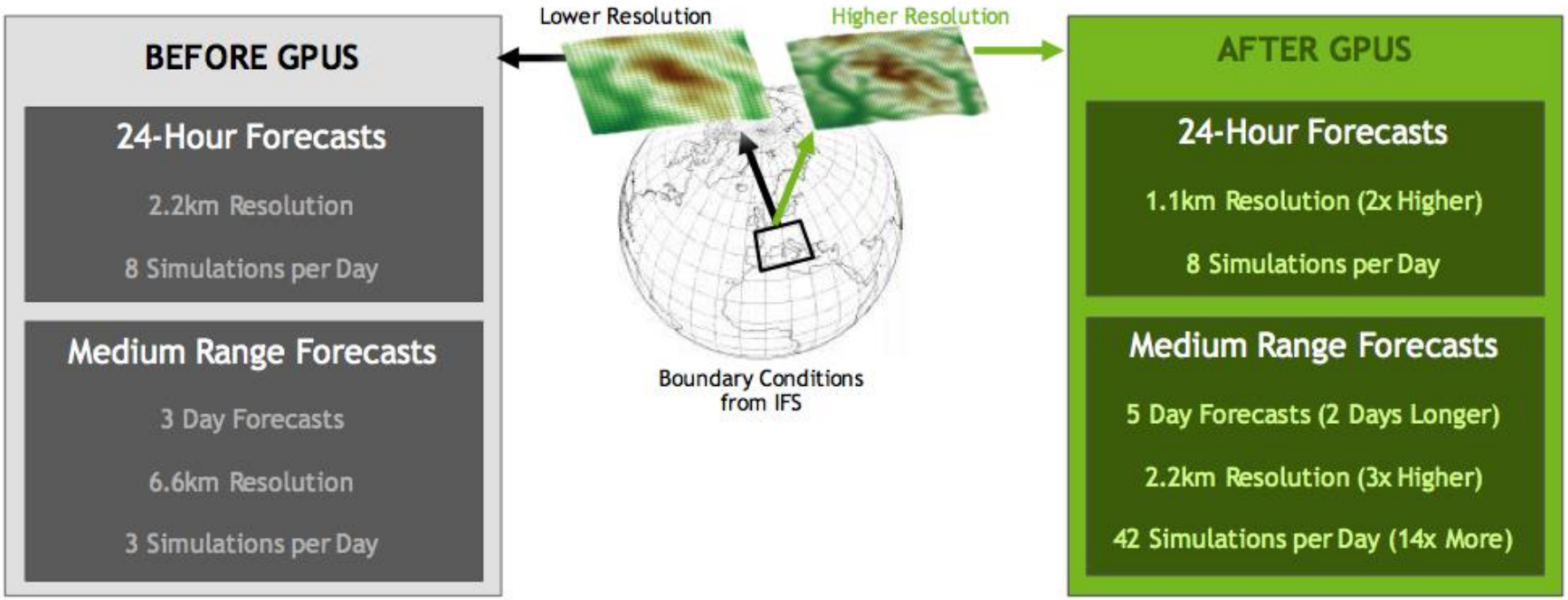
GPGPU Applications



Physics Simulation using PhysX / CUDA
http://www.youtube.com/watch?v=RuZQpWo9Qhs

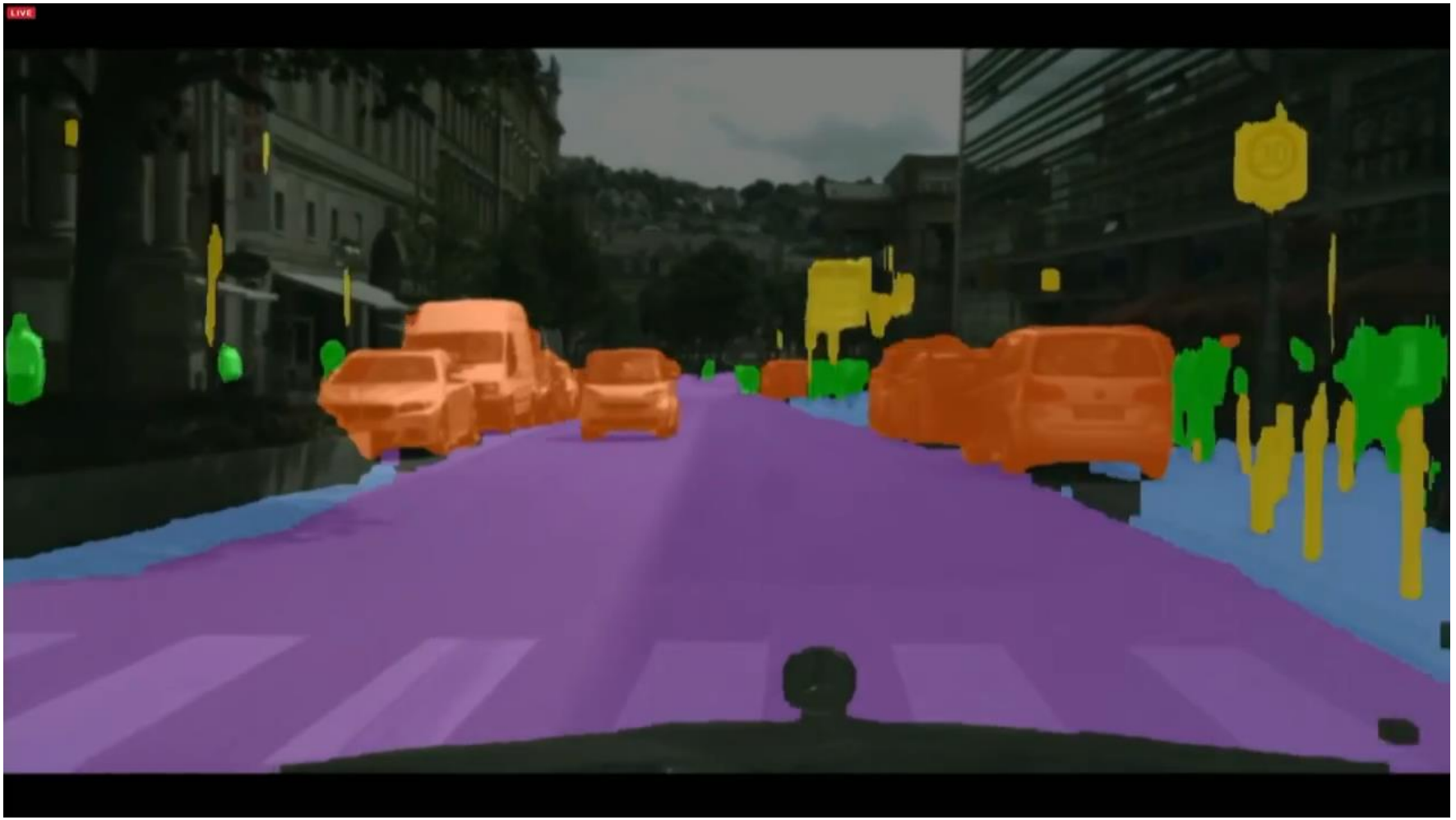
GPGPU Applications

Breakthrough Advance in Swiss Weather Forecasting



http://www.hpcwire.com/2015/09/15/todays-outlook-gpu-accelerated-weather-forecasting/

GPGPU Applications



Self Driving Cars (NVIDIA Drive PX2)
https://www.youtube.com/watch?v=84M3ghUKlLk

What is CUDA?

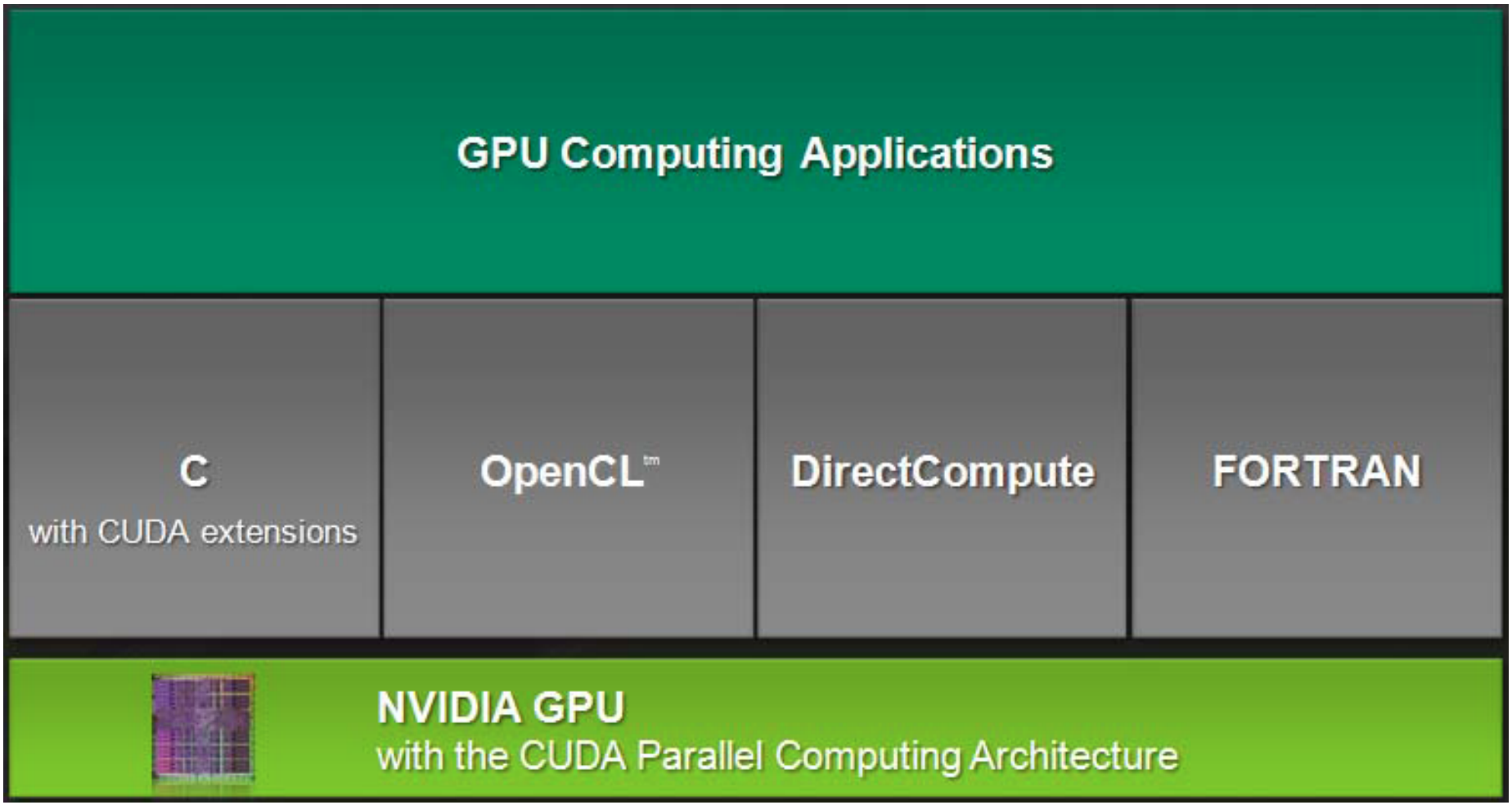
Great Barracuda
(Source: oceanwideimages.com)

What is CUDA?

Plymouth Hemi Cuda, 1971
(Source: carfunblog.com)

NVIDIA CUDA

- “Compute Unified Device Architecture”
  - Or simply, “COMPUTE”
- “A General-Purpose Parallel Computing Environment”
  - minimal set of C language extensions to harness GPU’s computational resources
- CUDA toolset includes compiler, SDK and profiler, etc
  - “nvcc hello_cuda.cu” v.s. “gcc hello_world.c”



Source: NVIDIA

Lecture Plans

We can only briefly cover the highlighted sections today.

- Lecture 1: CUDA Crash Course
- Lecture 2: OpenGL Shading Language / GPU Architecture
- Lecture 3: Parallel Computing Basics
- Lecture 4: CUDA Thread Models
- Lecture 5: CUDA Memory / Tools Walkthrough
- Lecture 6: CUDA Control Flows & Floating Points
- Lecture 7: Parallel Algorithms
- Lecture 8: CUDA Optimization
- Lecture 9+: Guest Lectures, Case Studies, Computer Vision, Other GPU Architectures, etc

GPU Programming #1: CUDA Crash Course

Wei-Chao Chen

Visiting Professor, National Taiwan University

weichao.chen@gmail.com

CUDA Workflow

- Get a CUDA-enabled GPU
- Write C/C++ like code (*.cu)
- Compile with CUDA compiler (*nvcc*)
  - Generates PTX code (“Parallel Thread Execution”)
- Applications auto-magically run on GPUs
  - Many many parallel threads
  - CUDA driver translate PTX code into HW

CUDA Overview

- CUDA C/C++ language extensions
  - Small sets of extensions for writing kernels - sub-routines that runs multi-threaded on GPUs
- CUDA Programming model abstraction
  - For fine-grained data / thread parallelism, including
    - Thread group hierarchy
    - Shared memories
    - Synchronization barriers

CUDA Overview

- CUDA C/C++ language extensions
  - Small sets of extensions for writing kernels - sub-routines that runs multi-threaded on GPUs
- CUDA Programming model abstraction
  - For fine-grained data / thread parallelism, including
    - Thread group hierarchy
    - Shared memories
    - Synchronization barriers

C/C++ Language Extensions

CPU (host)

```
int main()
{
...
    cudaMalloc(...)
    cudaMemcpy(...)
...
    my_kernel<<<nblock,
blocksize>>>(...)
...
    cudaMemcpy(...)
...
}
```

GPU (device)

```
__global__ void my_kernel(...)
{
   ...
   __shared__ float...
   ...blockIdx...
   ...threadIdx...
   int i = gpu_func(...)
...
}

__device__ int gpu_func(...) {
...
}
```

C/C++ Language Extensions

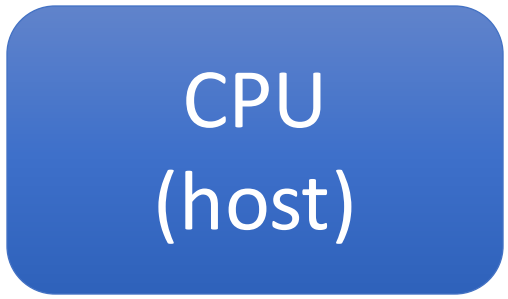
CPU (host)

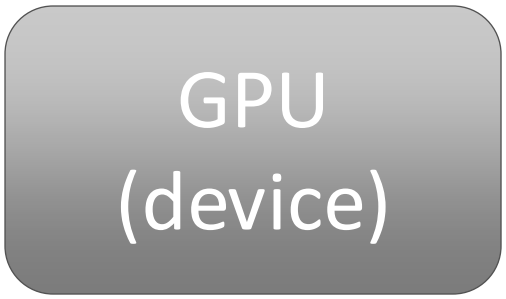
GPU (device)

```
int main()
{
...
    cudaMalloc(...)
    cudaMemcpy(...)
...
    my_kernel<<<nblock,
blocksize>>>(...)
...
    cudaMemcpy(...)
...
}
```

```
__global__ void my_kernel(...)
{
    ...
    __shared__ float...
    ...blockIdx...
    ...threadIdx...
    int i = gpu_func(...)
...
}

__device__ int gpu_func(...) {
...
}
```

C/C++ Language Extensions





```
int main()
{
...
    cudaMalloc(...)
    cudaMemcpy(...)
...
    my_kernel<<<nblock,
blocksize>>>(...)
...
    cudaMemcpy(...)
...
}
```

```
__global__ void my_kernel(...)
{
    ...
    __shared__ float...
    ...blockIdx...
    ...threadIdx...
    int i = gpu_func(...)
...
}

__device__ int gpu_func(...) {
...
}
```

C/C++ Language Extensions

CPU (host)

GPU (device)

```
int main()
{
...
    cudaMalloc(...)
    cudaMemcpy(...)
...
    my_kernel<<<nblock,
blocksize>>>(...)
...
    cudaMemcpy(...)
...
}
```

```
__global__ void my_kernel(...)
{
    ...
    __shared__ float...
    ...blockIdx...
    ...threadIdx...
    int i = gpu_func(...)
...
}

__device__ int gpu_func(...) {
...
}
```

C/C++ Language Extensions

CPU (host)

GPU (device)

```
int main()
{
...
    cudaMalloc(...)
    cudaMemcpy(...)
...
    my_kernel<<<nblock,
blocksize>>>(...)
...
    cudaMemcpy(...)
...
}
```

```
__global__ void my_kernel(...)
{
    ...
    __shared__ float...
    ...blockIdx...
    ...threadIdx...
    int i = gpu_func(...)
...
}

__device__ int gpu_func(...) {
...
}
```

C/C++ Language Extensions

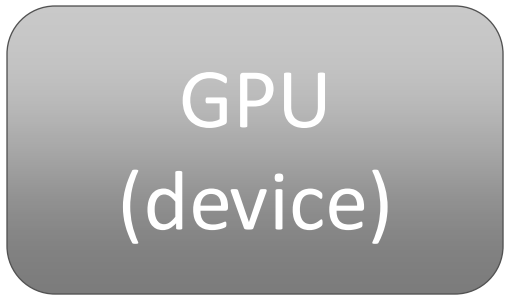
CPU (host)

GPU (device)

```
int main()
{
...
    cudaMalloc(...)
    cudaMemcpy(...)
...
    my_kernel<<<nblock,
blocksize>>>(...)
...
    cudaMemcpy(...)
...
}
```

```
__global__ void my_kernel(...)
{
    ...
    __shared__ float...
    ...blockIdx...
    ...threadIdx...
    int i = gpu_func(...)
...
}

__device__ int gpu_func(...) {
...
}
```

C/C++ Language Extensions




```
int main()
{
...
    cudaMalloc(...)
    cudaMemcpy(...)
...
    my_kernel<<<nblock,
blocksize>>>(...)
...
    cudaMemcpy(...)
...
}
```

```
__global__ void my_kernel(...)
{
   ...
   __shared__ float...
   ...blockIdx...
   ...threadIdx...
   int i = gpu_func(...)
...
}

__device__ int gpu_func(...) {
...
}
```

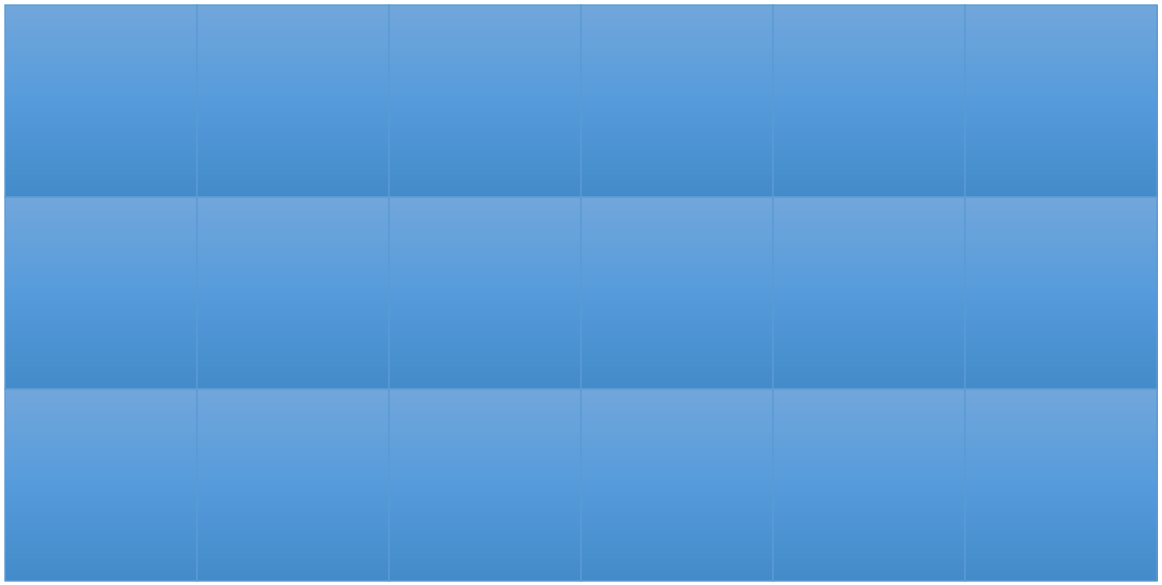
CUDA Overview

- CUDA C/C++ language extensions
  - Small sets of extensions for writing kernels - sub-routines that runs multi-threaded on GPUs
- CUDA Programming model abstraction
  - For fine-grained data / thread parallelism, including
    - Thread group hierarchy
    - Shared memories
    - Synchronization barriers

CUDA Programming Model Abstraction

- Thread group hierarchy

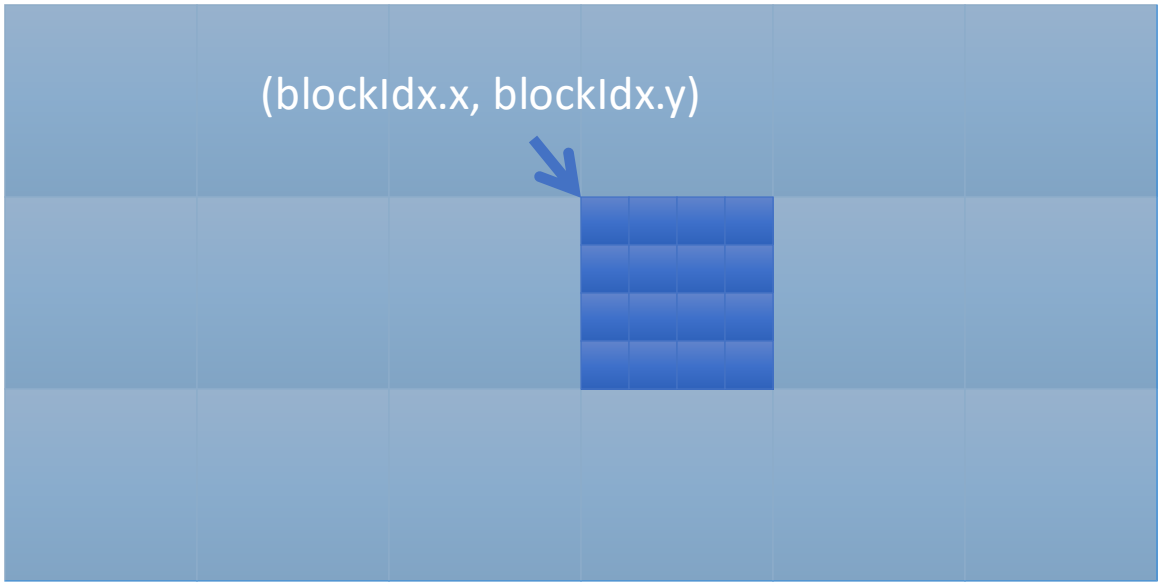
Grid



CUDA Programming Model Abstraction

- Thread group hierarchy

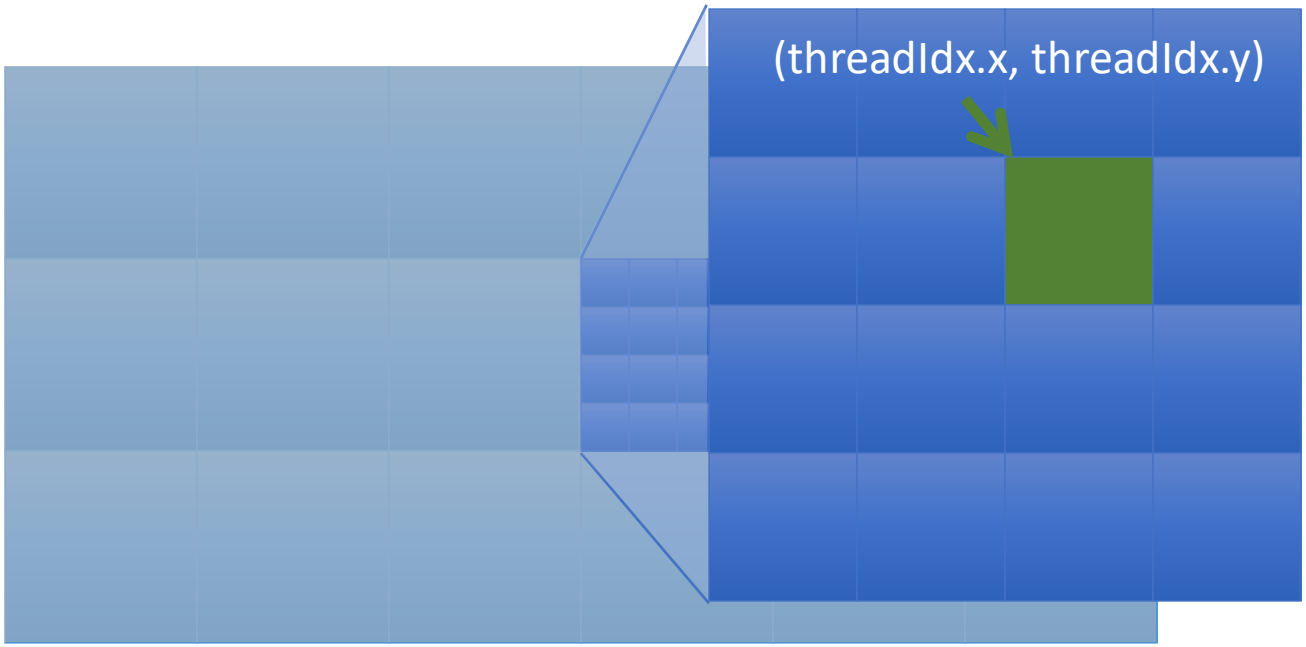
  Grid of **blocks**



CUDA Programming Model Abstraction

- Thread group hierarchy

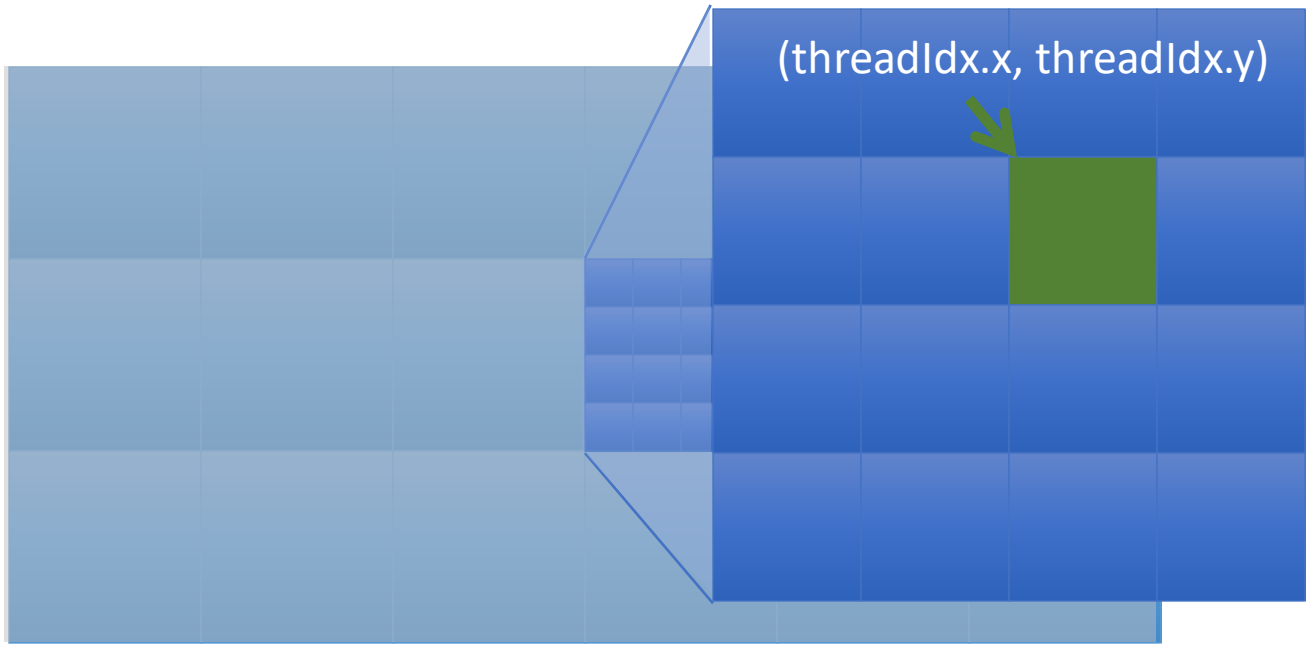
  Grid of blocks of **threads**



CUDA Programming Model Abstraction

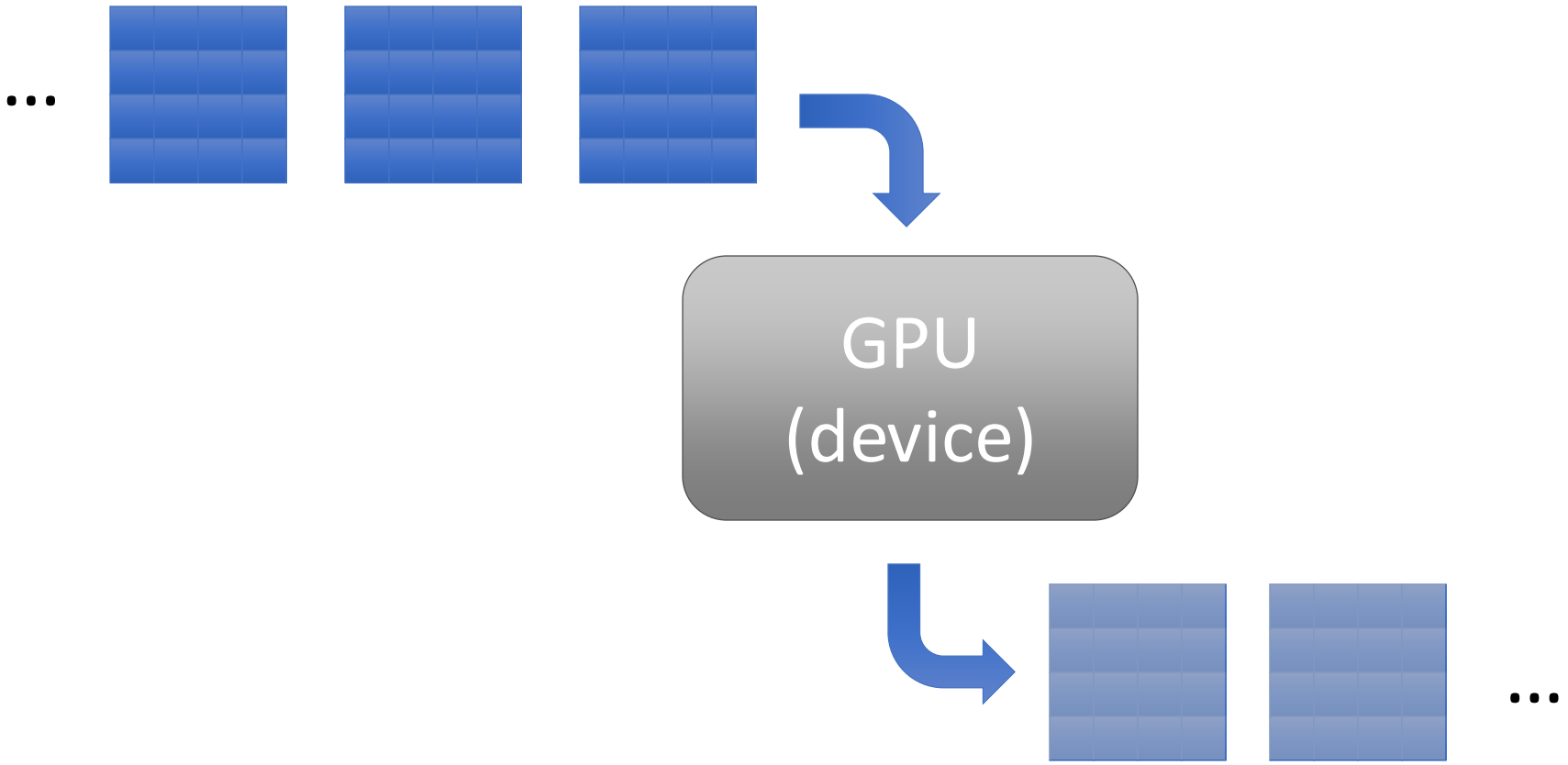
- Thread group hierarchy

```
__global__ myKernel<<<nBlocks, nThreads>>>(…)
```



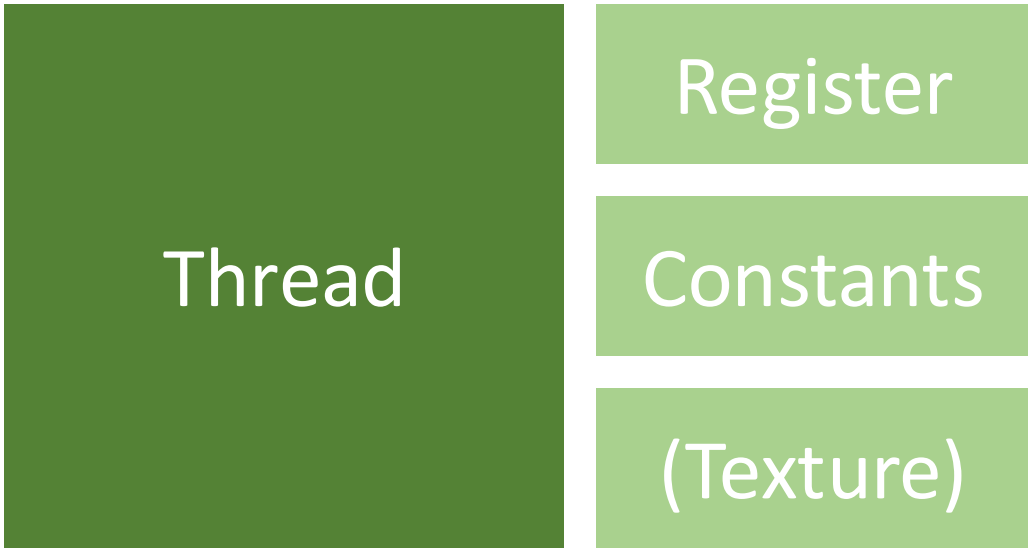
CUDA Programming Model Abstraction

- Thread group hierarchy



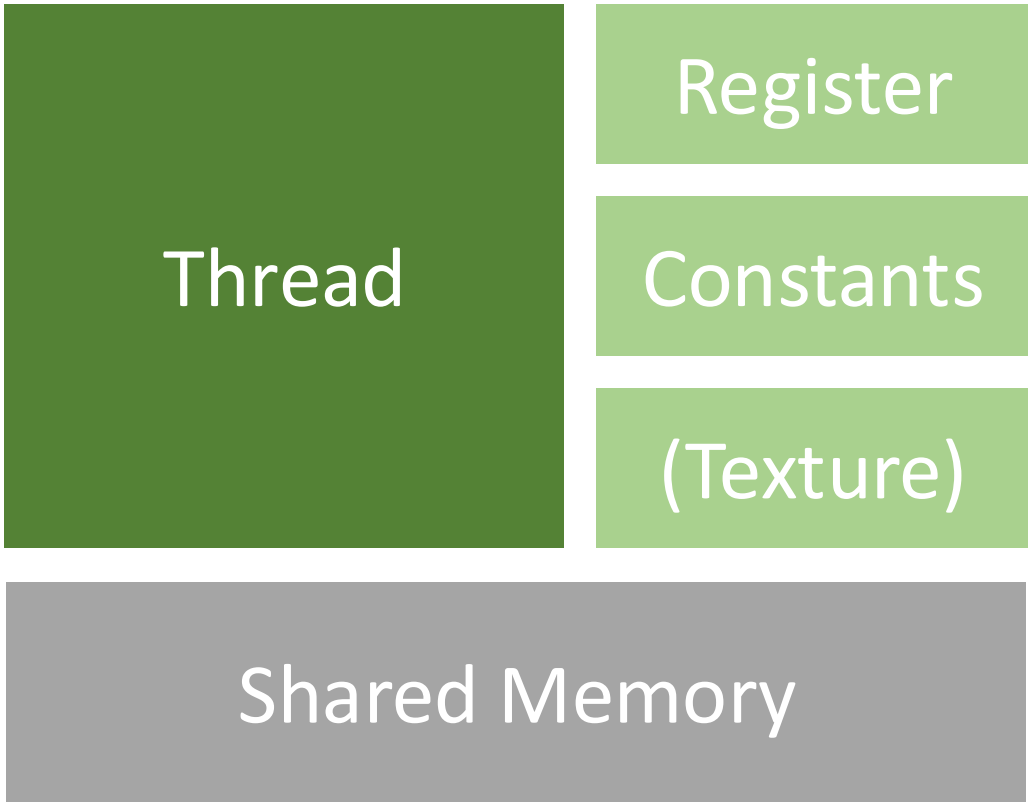
CUDA Programming Model Abstraction

- Shared memory



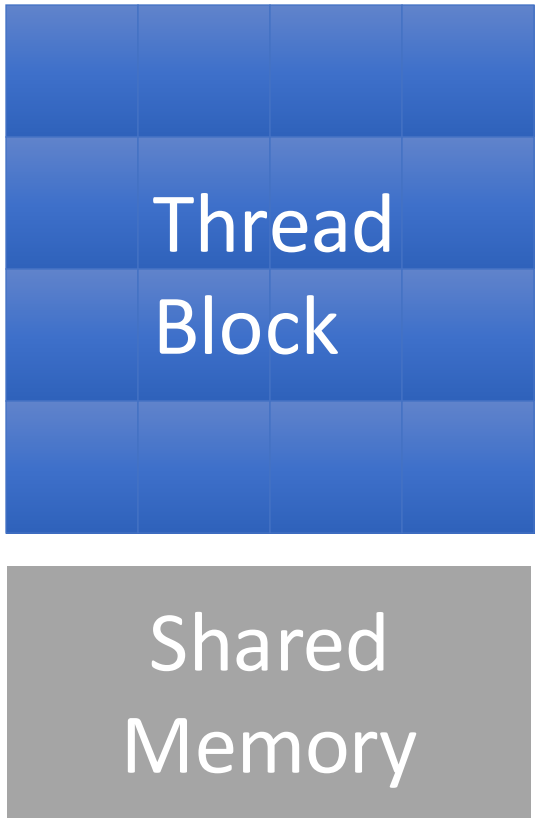
CUDA Programming Model Abstraction

- Shared memory



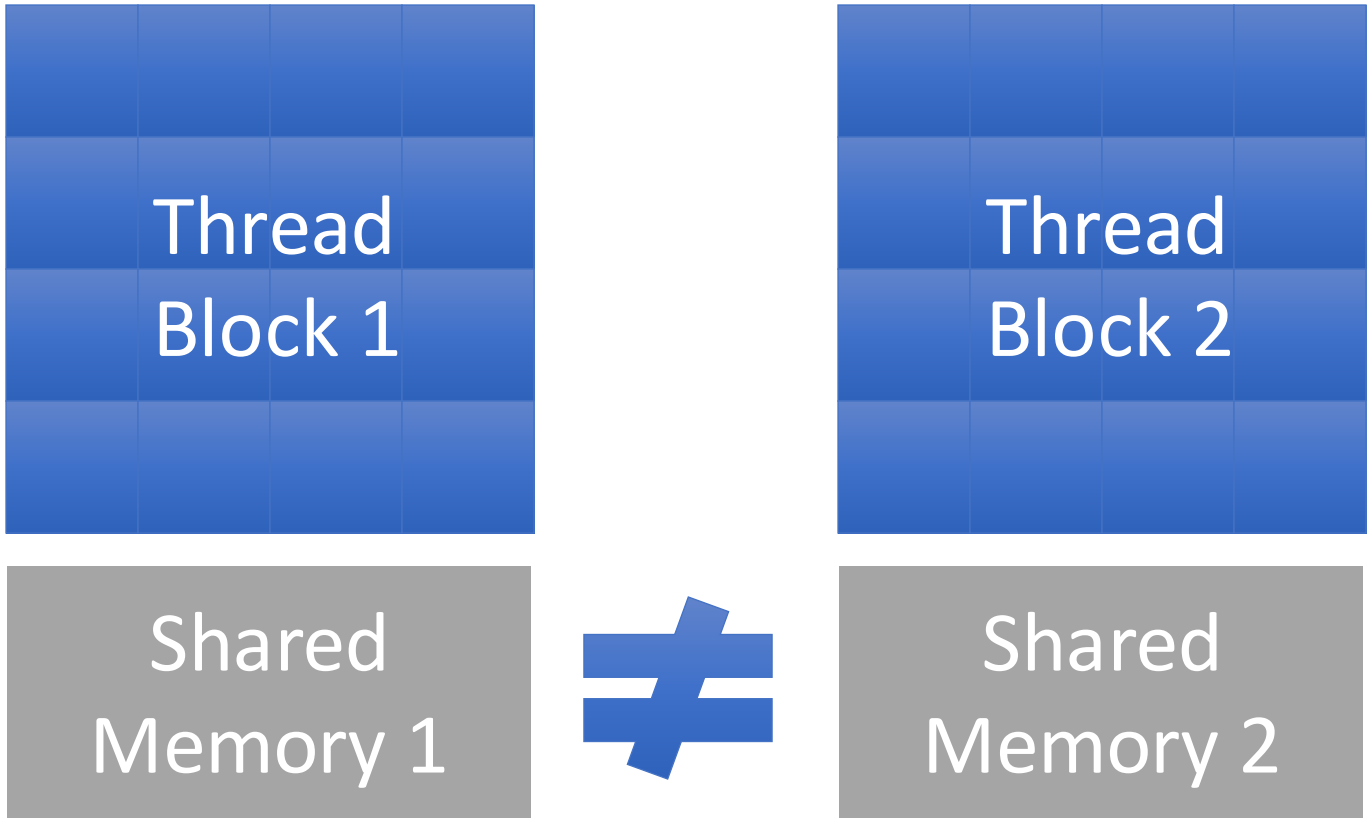
CUDA Programming Model Abstraction

- Shared memory



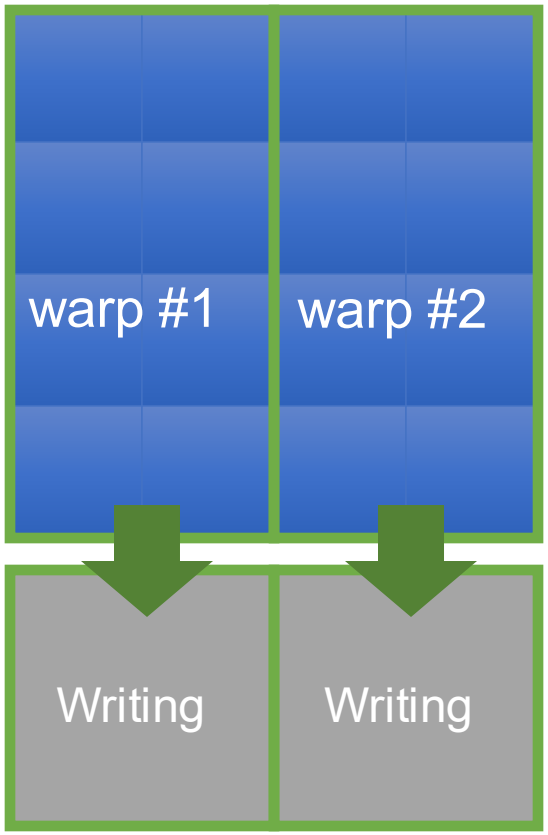
CUDA Programming Model Abstraction

- Shared memory
  - Logically, they are mutually invisible
  - Physically they may lie on the same memory unit

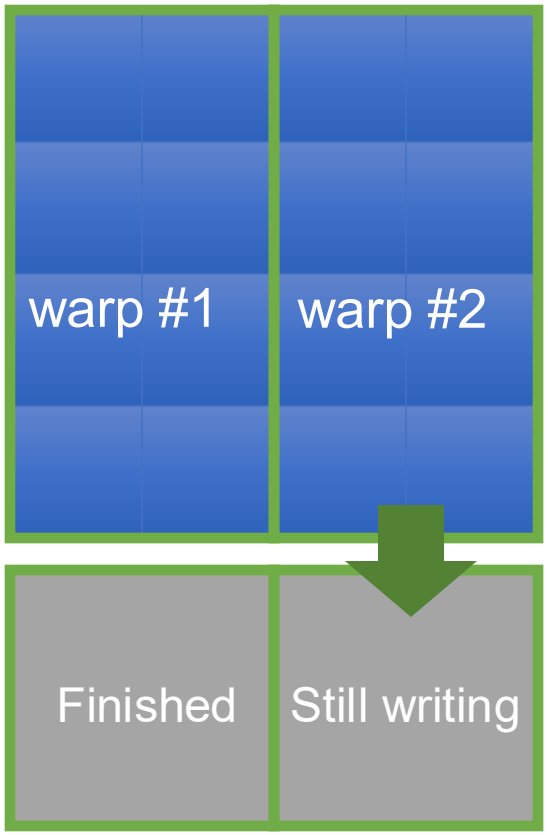


CUDA Programming Model Abstraction

- Synchronization Barrier
  - SIMT threads launched in a unit called a *warp*

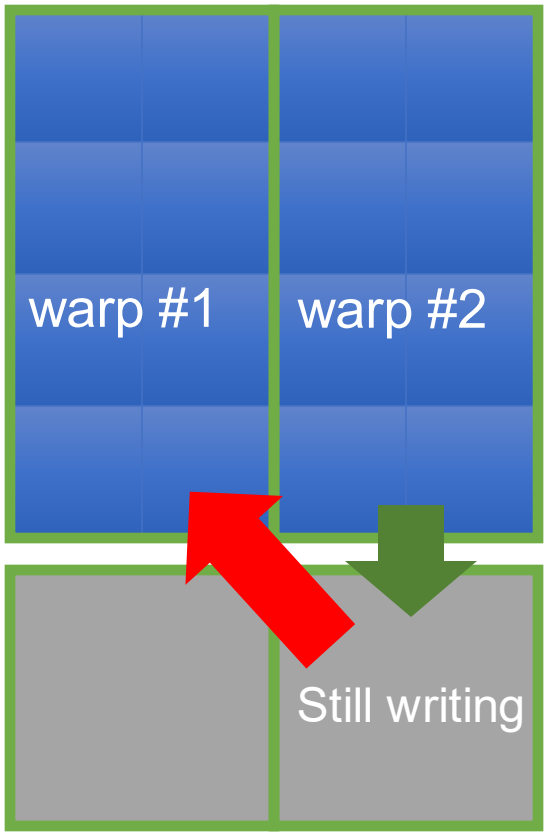


CUDA Programming Model Abstraction



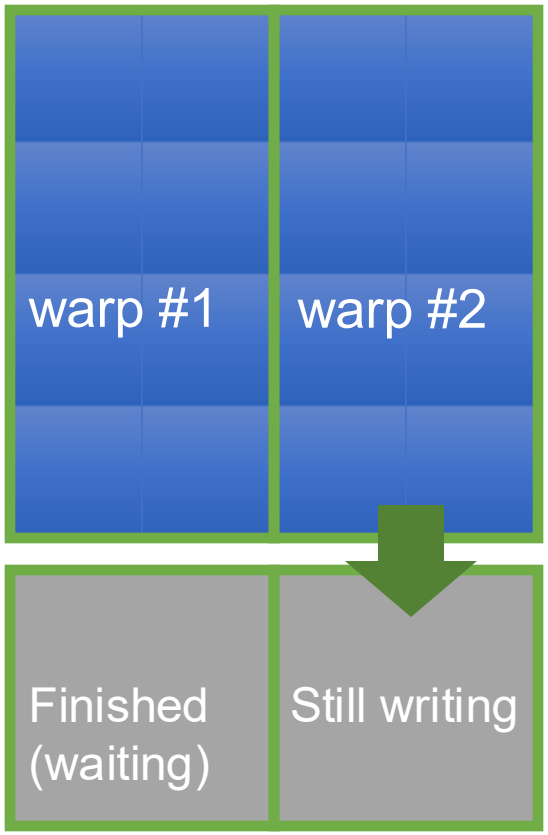
- Synchronization Barrier
  - SIMT threads launched in a unit called a *warp*
  - Problems occur when one warp reads from another before it's finished

CUDA Programming Model Abstraction



- Synchronization Barrier
  - SIMT threads launched in a unit called a *warp*
  - Problems occur when one warp reads from another before it's finished

CUDA Programming Model Abstraction



- Synchronization Barrier
  - SIMT threads launched in a unit called a *warp*
  - Problems occur when one warp reads from another before it's finished
  - **__syncthreads()** prevents the read-after-write hazard
  - BTW, a warp doesn't branch
    - data-dependent conditional branch only

A Little More on Synchronization

- All synchronization points must be encountered by all threads of a block.
- Undefined if threads met different synchronization points.
  - if (cond)
    __syncthreads();
    else
    __syncthreads();
- Undefined if some threads do not met synchronization points.
  - if (idx < total_num)
    return;
    __syncthreads();

__syncthreads() is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects.

Hello CUDA!

- A simple example that
  - Initializes an empty buffer on CPU *(host)*
  - Get GPU (*device)* to fill the buffer with a string
  - Transfer result back to *host*

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
__global__ void hello_kernel(char
*odata, int num)
{
    char hello_str[12]="Hello CUDA!";
    int idx = blockIdx.x * blockDim.x
+ threadIdx.x;
    if(idx < num)
        odata[idx] = hello_str[idx];
}
int main(void)
{
    char *h_data, *d_data;
    const int strlen = 12;
    size_t strsize = strlen *
sizeof(char);
    h_data = (char *) malloc(strsize);
    memset(h_data, 0, strlen);
    cudaMalloc((void **) &d_data,
strsize);
    cudaMemcpy(d_data, h_data,
strsize, cudaMemcpyHostToDevice);
    int blocksize = 8;
    int nblock = strlen/blocksize +
(strlen % blocksize == 0 ? 0 : 1);

hello_kernel<<<nblock,blocksize>>>(d_data
, strlen);

    cudaMemcpy(h_data, d_data,
sizeof(char)*strlen,
cudaMemcpyDeviceToHost);
    printf("%s\n", h_data);

    free(h_data);
    cudaFree(d_data);
}
```

Hello CUDA! Program

- “nvcc hello_cuda.cu” to build

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
__global__ void hello_kernel(char
*odata, int num)
{
    char hello_str[12]="Hello CUDA!";
    int idx = blockIdx.x * blockDim.x
+ threadIdx.x;
    if(idx < num)
        odata[idx] = hello_str[idx];
}
int main(void)
{
    char *h_data, *d_data;
    const int strlen = 12;
    size_t strsize = strlen *
sizeof(char);
    h_data = (char *) malloc(strsize);
    memset(h_data, 0, strlen);
    cudaMalloc((void **) &d_data,
strsize);
    cudaMemcpy(d_data, h_data,
strsize, cudaMemcpyHostToDevice);
    int blocksize = 8;
    int nblock = strlen/blocksize +
(strlen % blocksize == 0 ? 0 : 1);

hello_kernel<<<nblock,blocksize>>>(d_data
, strlen);

    cudaMemcpy(h_data, d_data,
sizeof(char)*strlen,
cudaMemcpyDeviceToHost);
    printf("%s\n", h_data);

    free(h_data);
    cudaFree(d_data);
}
```

1. Initializes device buffer
- *cudaMalloc()*
- *cudaMemcpy()*

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
__global__ void hello_kernel(char
*odata, int num)
{
    char hello_str[12]="Hello CUDA!";
    int idx = blockIdx.x * blockDim.x
+ threadIdx.x;
    if(idx < num)
        odata[idx] = hello_str[idx];
}
int main(void)
{
    char *h_data, *d_data;
    const int strlen = 12;
    size_t strsize = strlen *
sizeof(char);
    h_data = (char *) malloc(strsize);
    memset(h_data, 0, strlen);
    cudaMalloc((void **) &d_data,
strsize);
    cudaMemcpy(d_data, h_data,
strsize, cudaMemcpyHostToDevice);
```

```
    int blocksize = 8;
    int nblock = strlen/blocksize +
(strlen % blocksize == 0 ? 0 : 1);

hello_kernel<<<nblock,blocksize>>>(d_data
, strlen);

    cudaMemcpy(h_data, d_data,
sizeof(char)*strlen,
cudaMemcpyDeviceToHost);
    printf("%s\n", h_data);

    free(h_data);
    cudaFree(d_data);
}
```

2. Launch the kernel
- compute block / grid size
- launch:
kernel<<<nBlocks, nThreads>>>()

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
__global__ void hello_kernel(char
*odata, int num)
{
    char hello_str[12]="Hello CUDA!";
    int idx = blockIdx.x * blockDim.x
+ threadIdx.x;
    if(idx < num)
        odata[idx] = hello_str[idx];
}
int main(void)
{
    char *h_data, *d_data;
    const int strlen = 12;
    size_t strsize = strlen *
sizeof(char);
    h_data = (char *) malloc(strsize);
    memset(h_data, 0, strlen);
    cudaMalloc((void **) &d_data,
strsize);
    cudaMemcpy(d_data, h_data,
strsize, cudaMemcpyHostToDevice);
```

```
    int blocksize = 8;
    int nblock = strlen/blocksize +
(strlen % blocksize == 0 ? 0 : 1);

hello_kernel<<<nblock,blocksize>>>(d_data
, strlen);

    cudaMemcpy(h_data, d_data,
sizeof(char)*strlen,
cudaMemcpyDeviceToHost);
    printf("%s\n", h_data);

    free(h_data);
    cudaFree(d_data);
}
```

3. Transfer result back to host

- *cudaMemcpy()*
- *cudaFree()*

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
__global__ void hello_kernel(char
*odata, int num)
{
    char hello_str[12]="Hello CUDA!";
    int idx = blockIdx.x * blockDim.x
+ threadIdx.x;
    if(idx < num)
        odata[idx] = hello_str[idx];
}
int main(void)
{
    char *h_data, *d_data;
    const int strlen = 12;
    size_t strsize = strlen *
sizeof(char);
    h_data = (char *) malloc(strsize);
    memset(h_data, 0, strlen);
    cudaMalloc((void **) &d_data,
strsize);
    cudaMemcpy(d_data, h_data,
strsize, cudaMemcpyHostToDevice);
    int blocksize = 8;
    int nblock = strlen/blocksize +
(strlen % blocksize == 0 ? 0 : 1);

hello_kernel<<<nblock,blocksize>>>(d_data
, strlen);

    cudaMemcpy(h_data, d_data,
sizeof(char)*strlen,
cudaMemcpyDeviceToHost);
    printf("%s\n", h_data);

    free(h_data);
    cudaFree(d_data);
}
```

The kernel:
- executes in parallel on device
- thread identifier:
  (blockIdx, threadIdx)
- output to device buffer

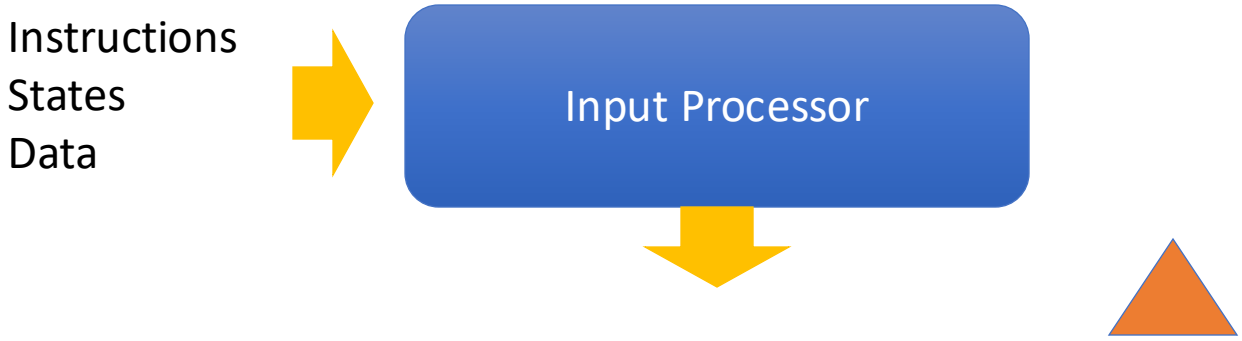
GPU Programming #2: GPU Architecture

Wei-Chao Chen

Visiting Professor, National Taiwan University

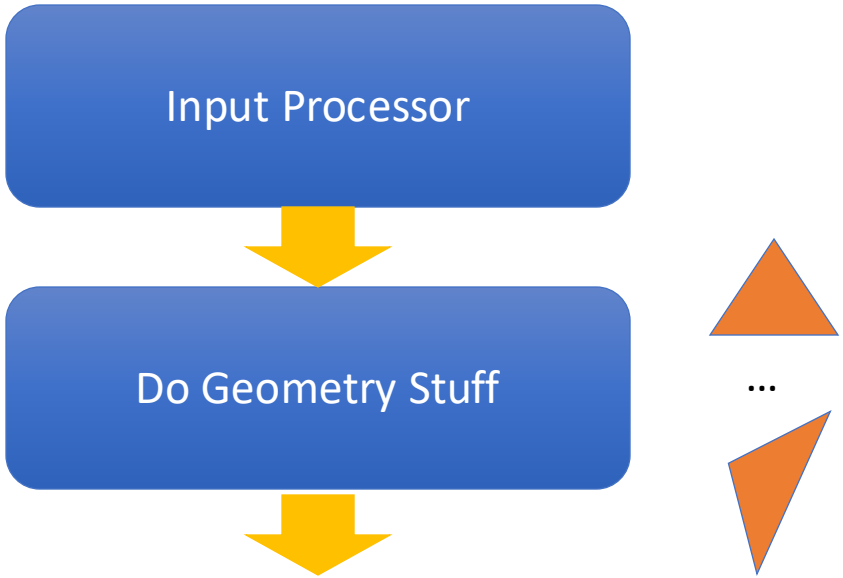
weichao.chen@gmail.com

Simplified Graphics Pipeline

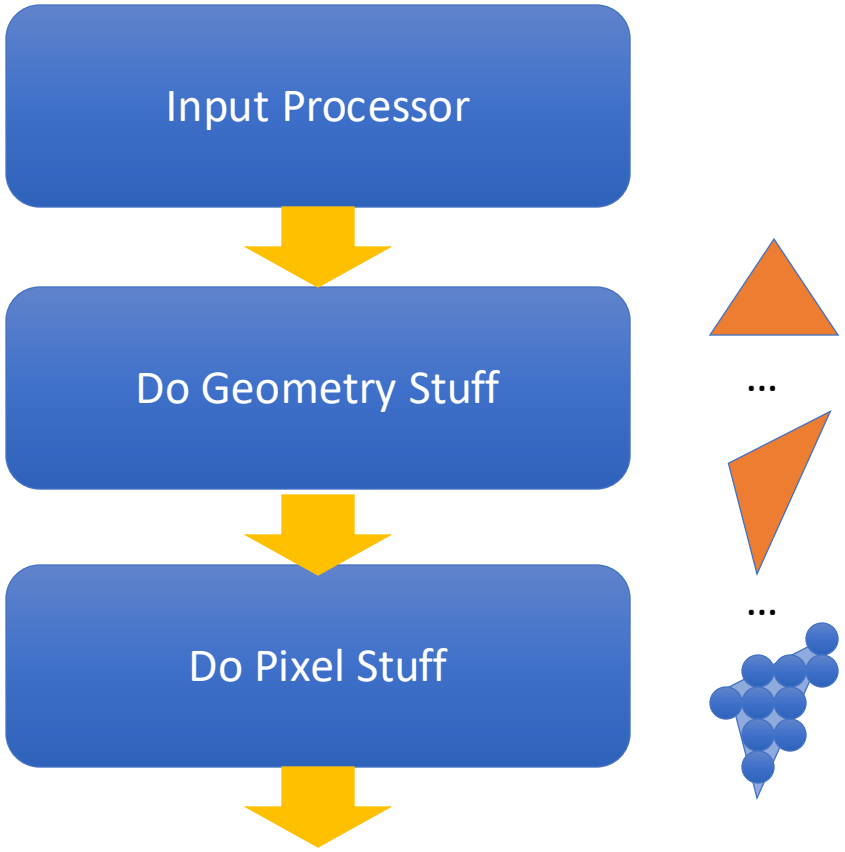


Simplified Graphics Pipeline

Transform
Lighting, etc

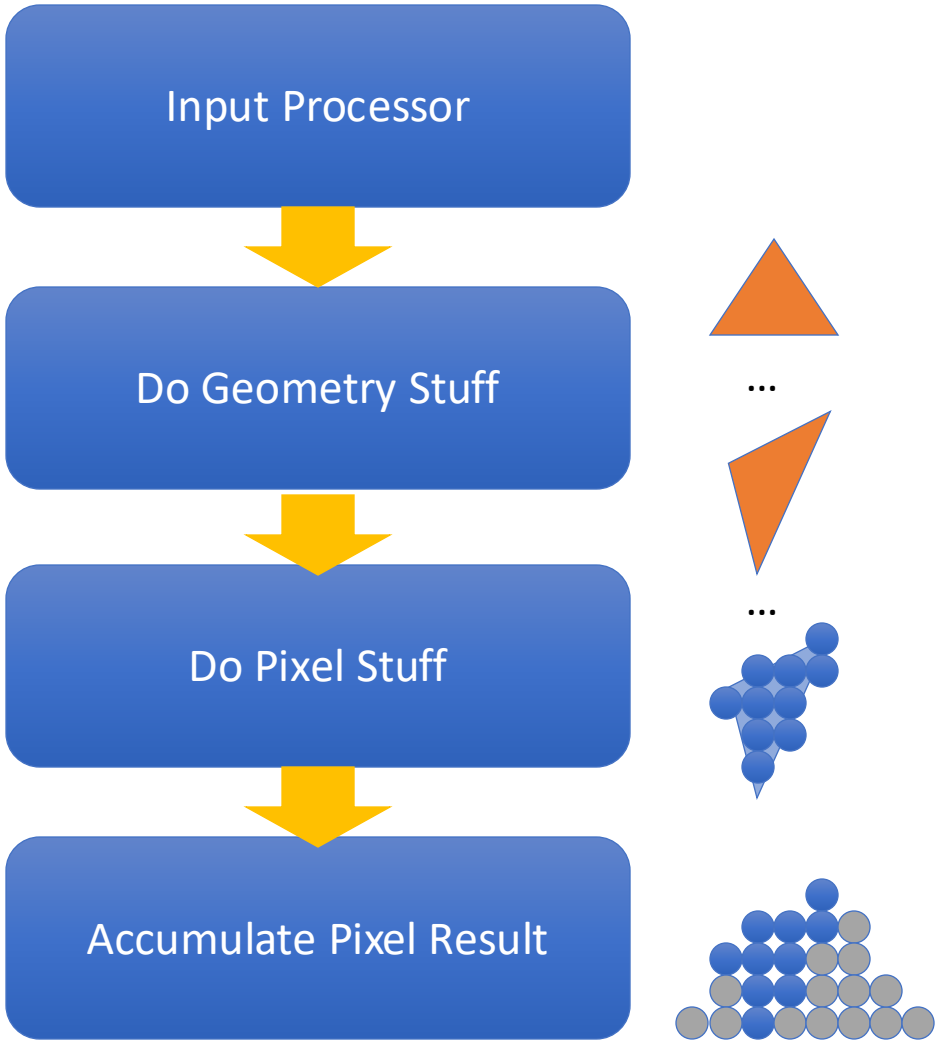


Simplified Graphics Pipeline



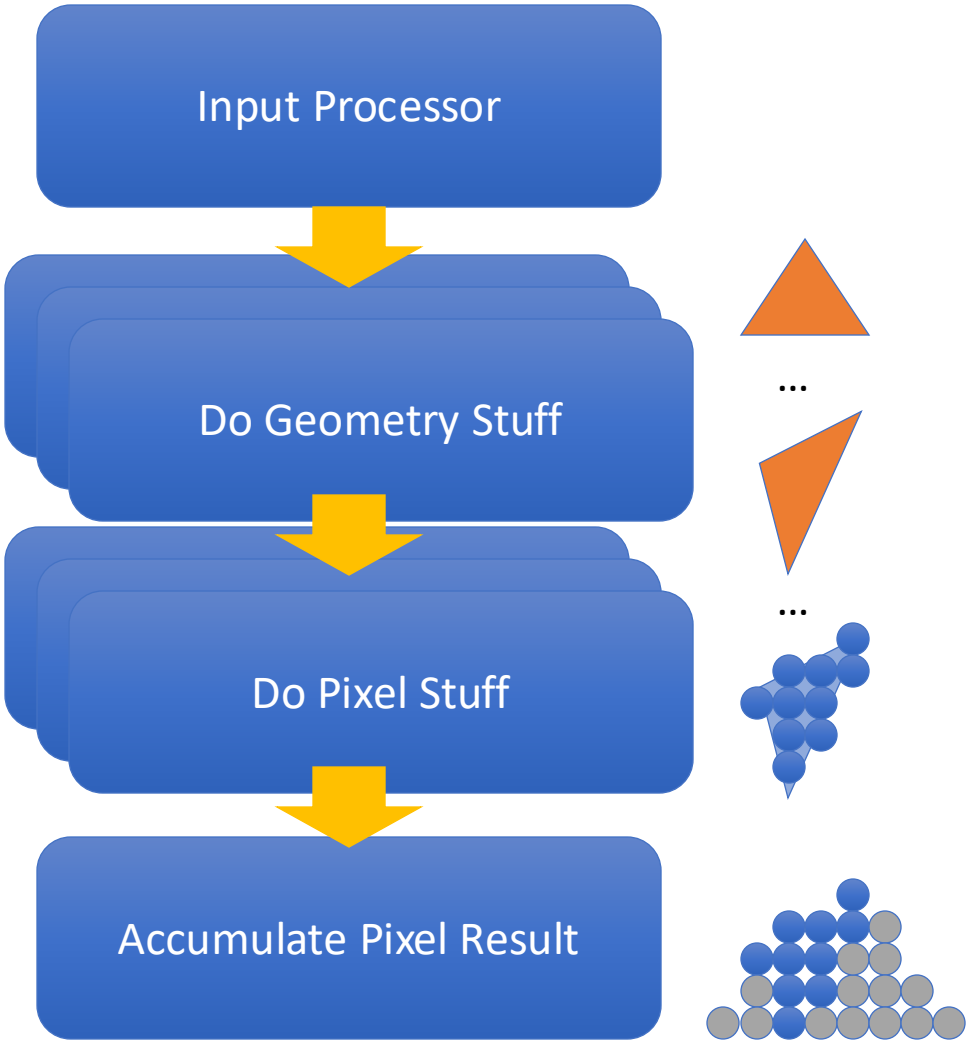
Rasterize
Pixel Shading

Simplified Graphics Pipeline

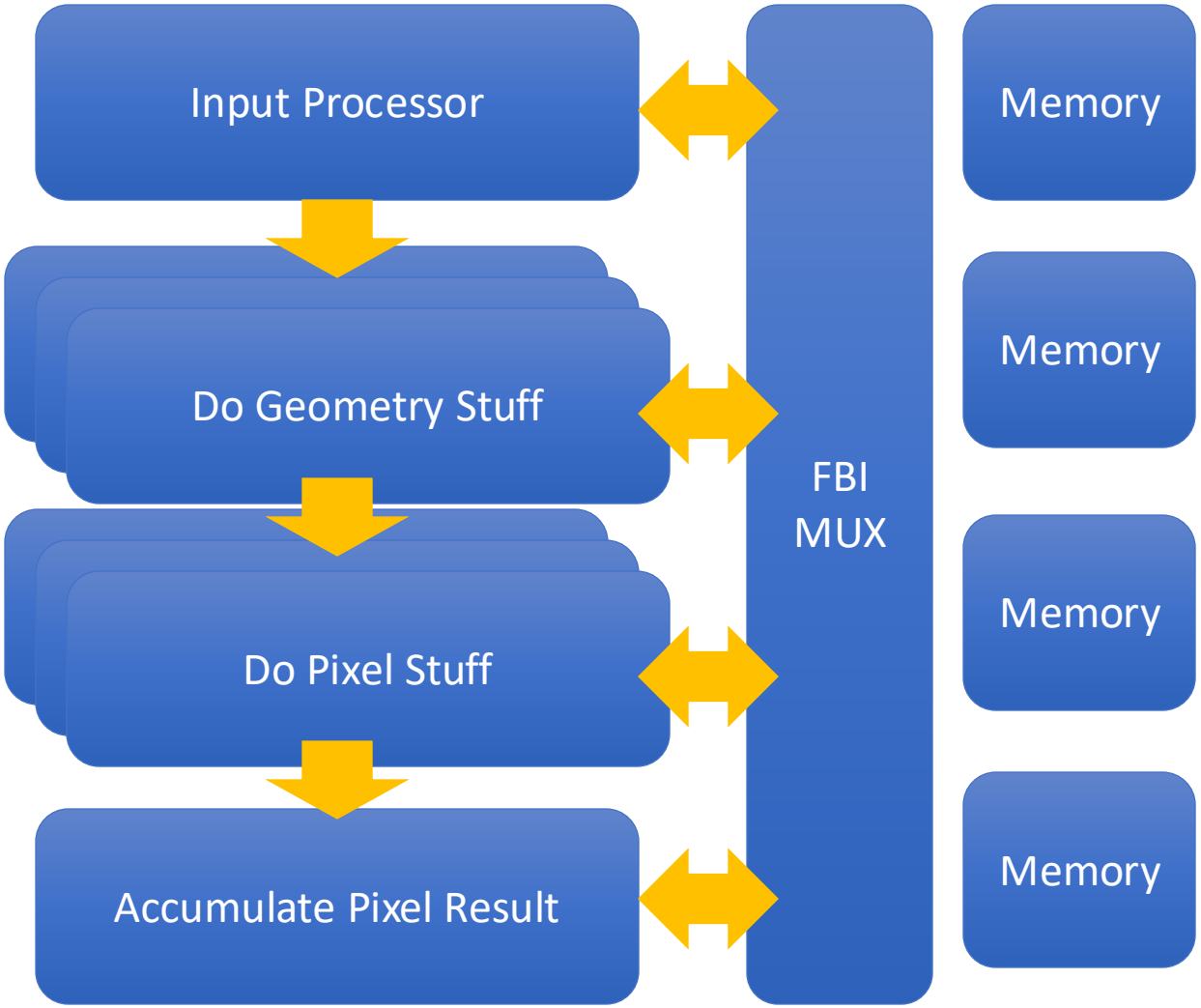


Sorting Stage
(Z-buffer)
Transparency

Making It Faster



Add Framebuffer Access



The Rendering Process

- Setup geometry
- Setup lighting
  - Directional & point light sources
  - Environment maps, etc
- Setup material attributes
  - Texture, reflectance properties
  - Usually specified through shaders
- Run your favorite render
  - Light transport simulation (shadows, inter-reflectivity, etc)

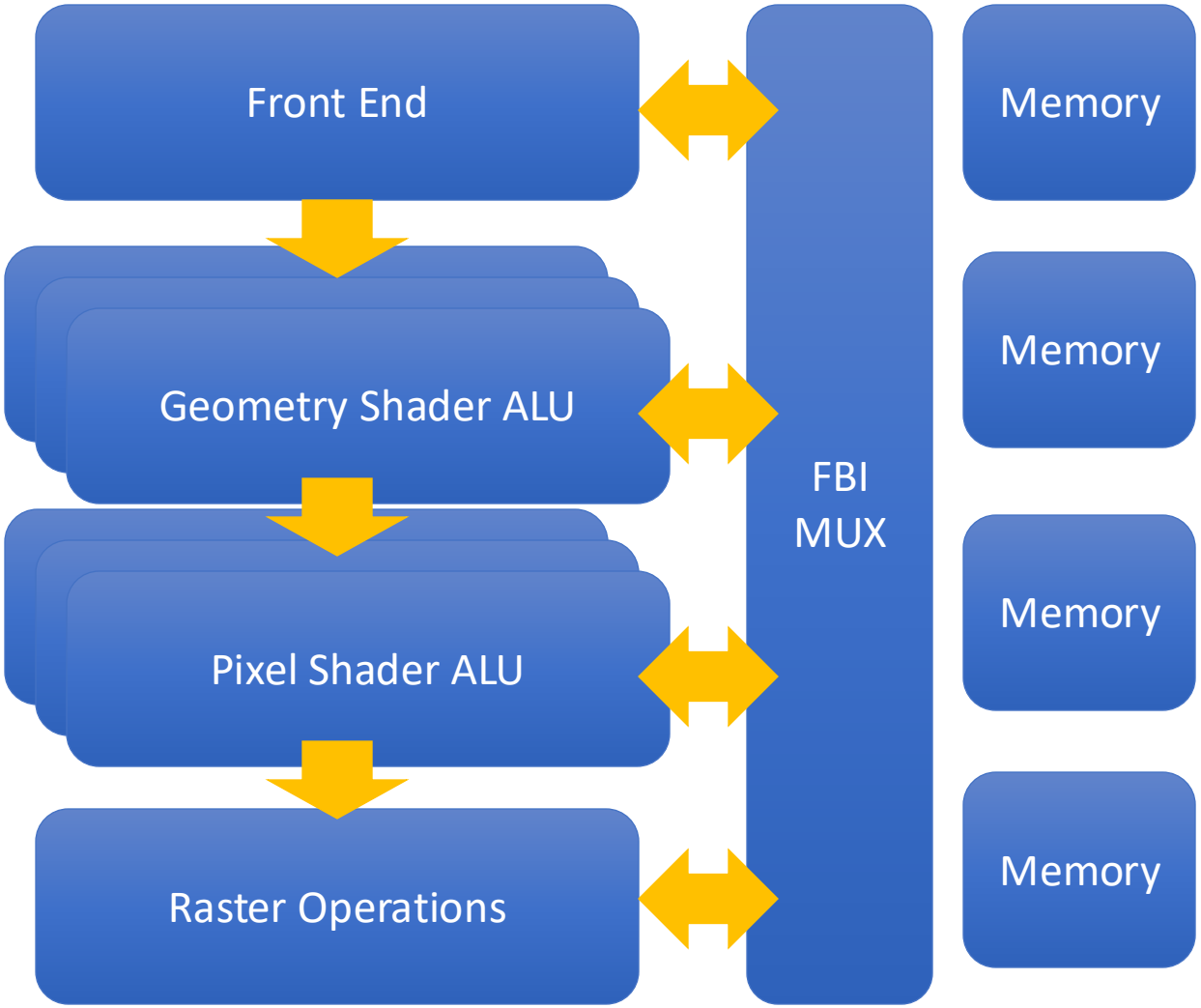
10 Years of Progress

- Hundreds of times improvements on
  - Power consumption, Size, Speed
  - Thank you, Moore's Law
- What Else?

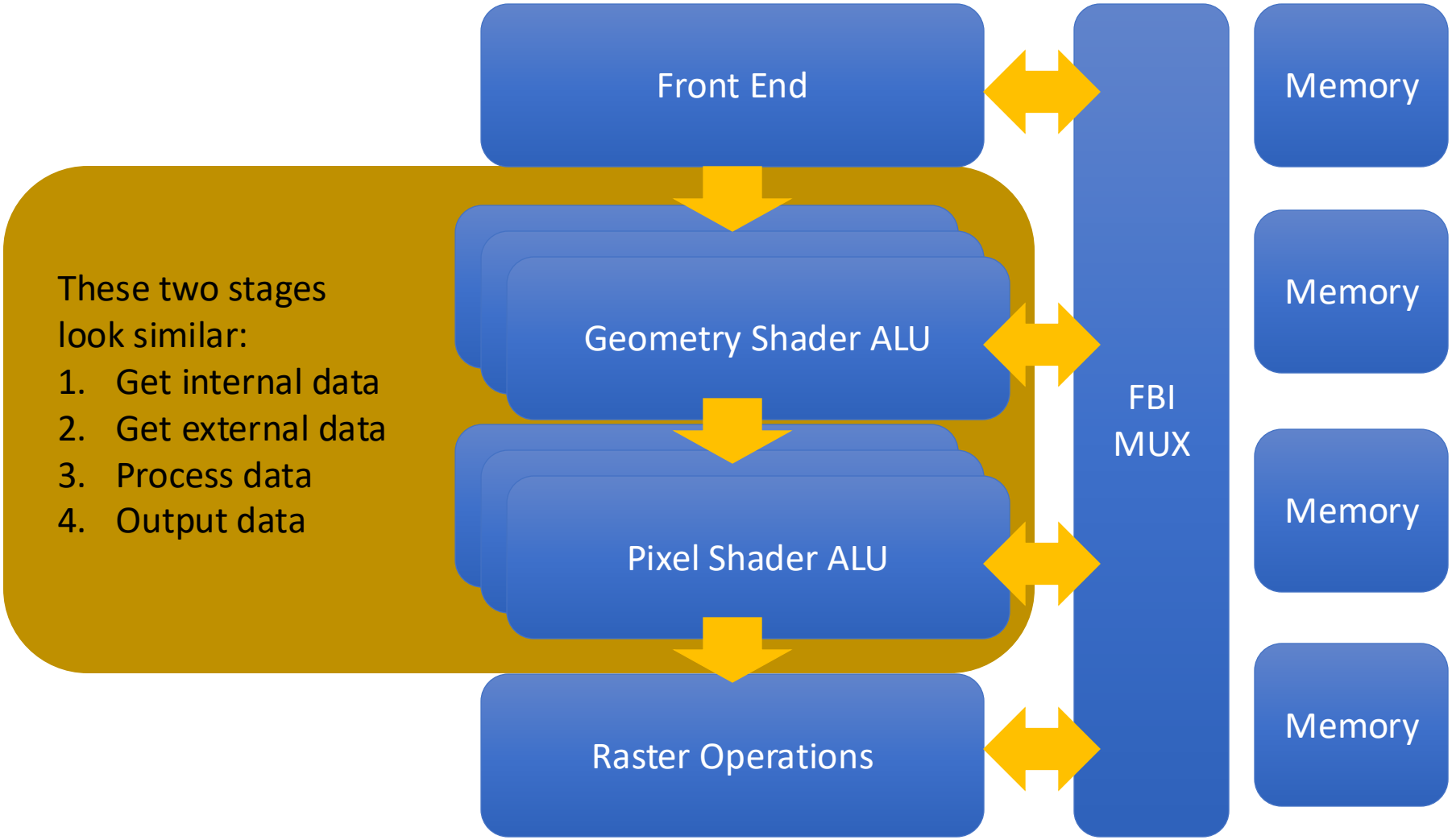
1996 (30M tris/sec, USD $1M)

2006 (>100M tris/sec, <USD $100)

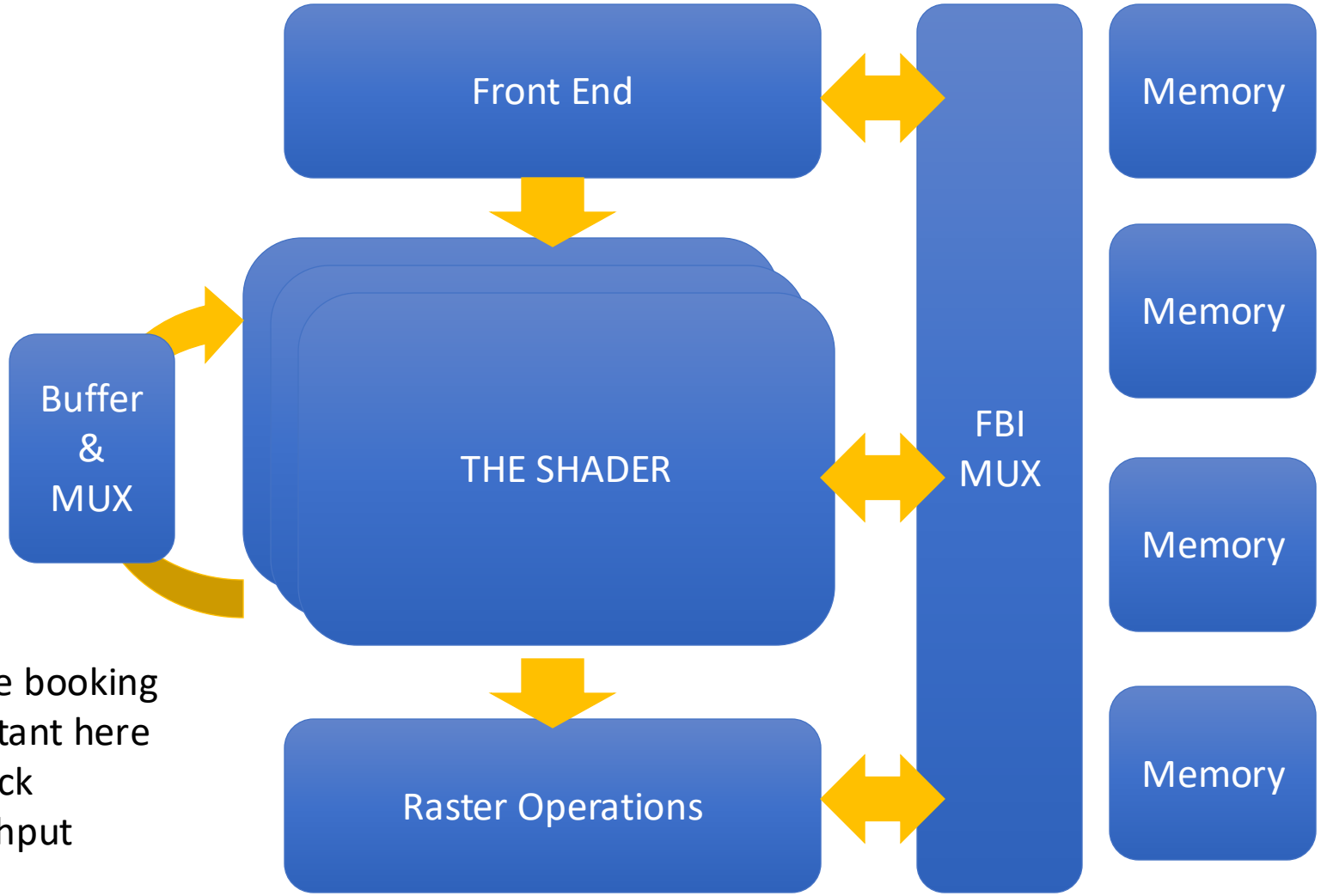
Add Programmability



Add Programmability

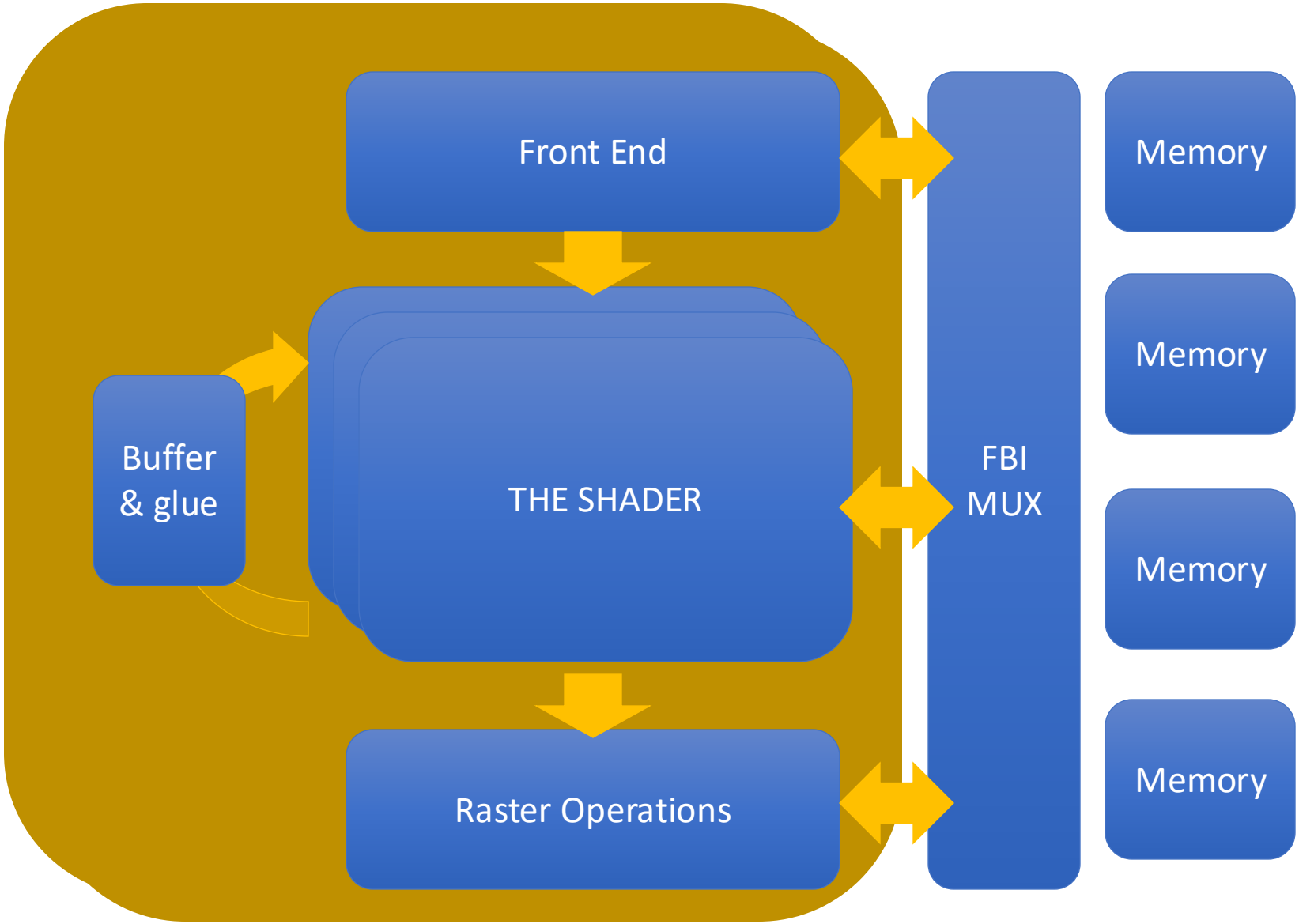


Unified Shader

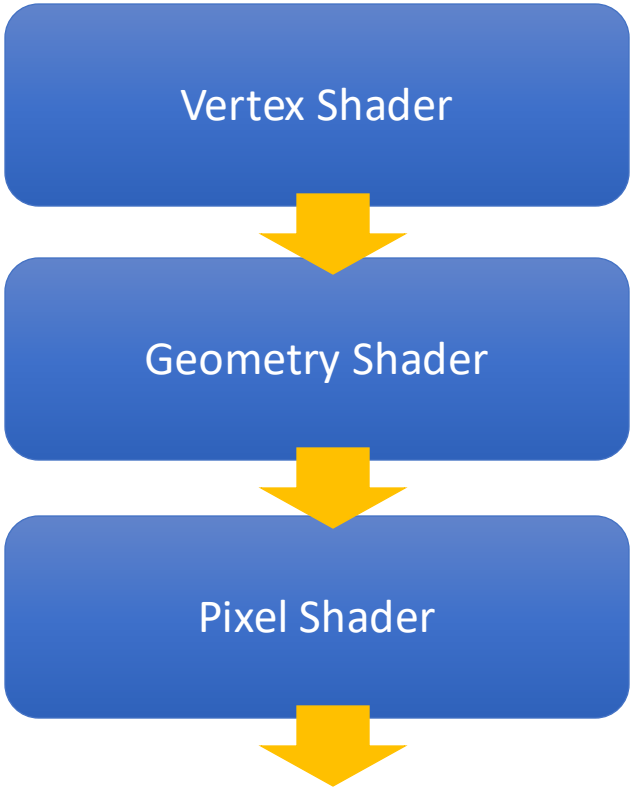


Resource booking
is important here
- deadlock
- throughput

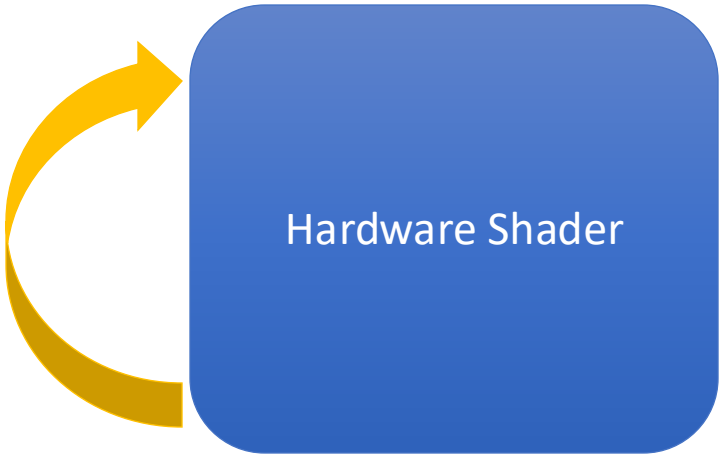
Scaling Up Again



Software Pipeline v.s. Hardware Pipeline



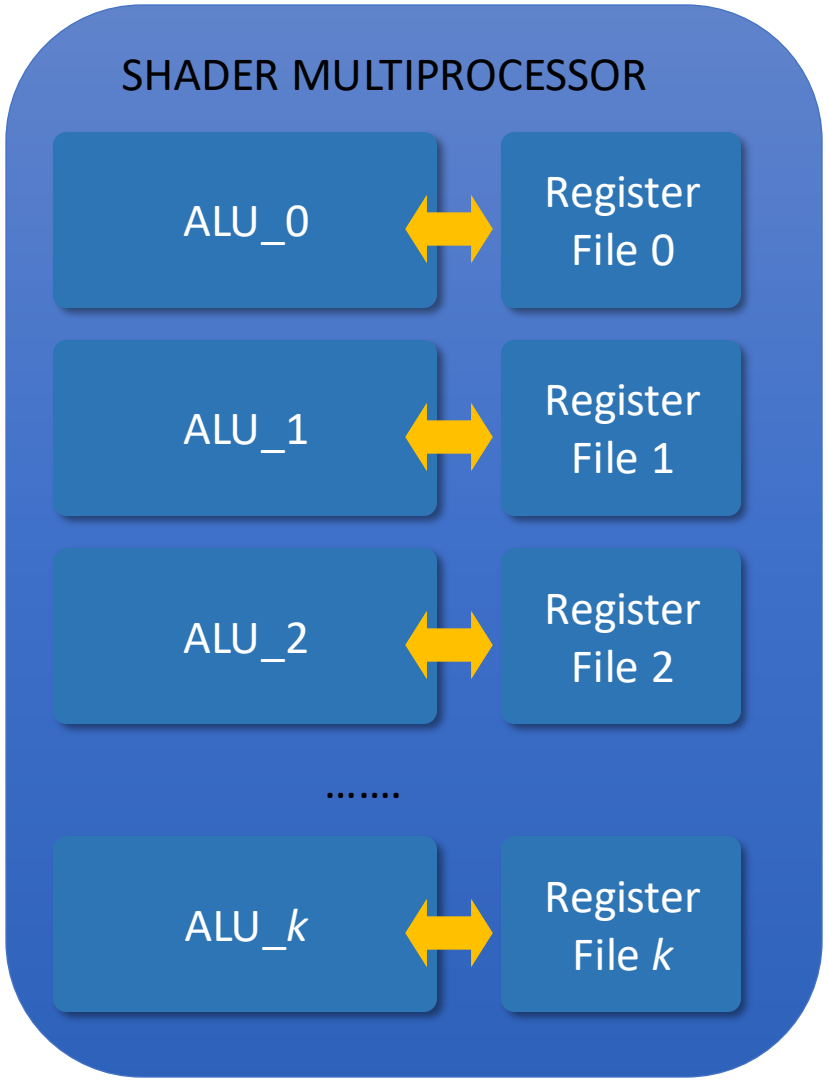
Software Pipeline



Hardware Pipeline

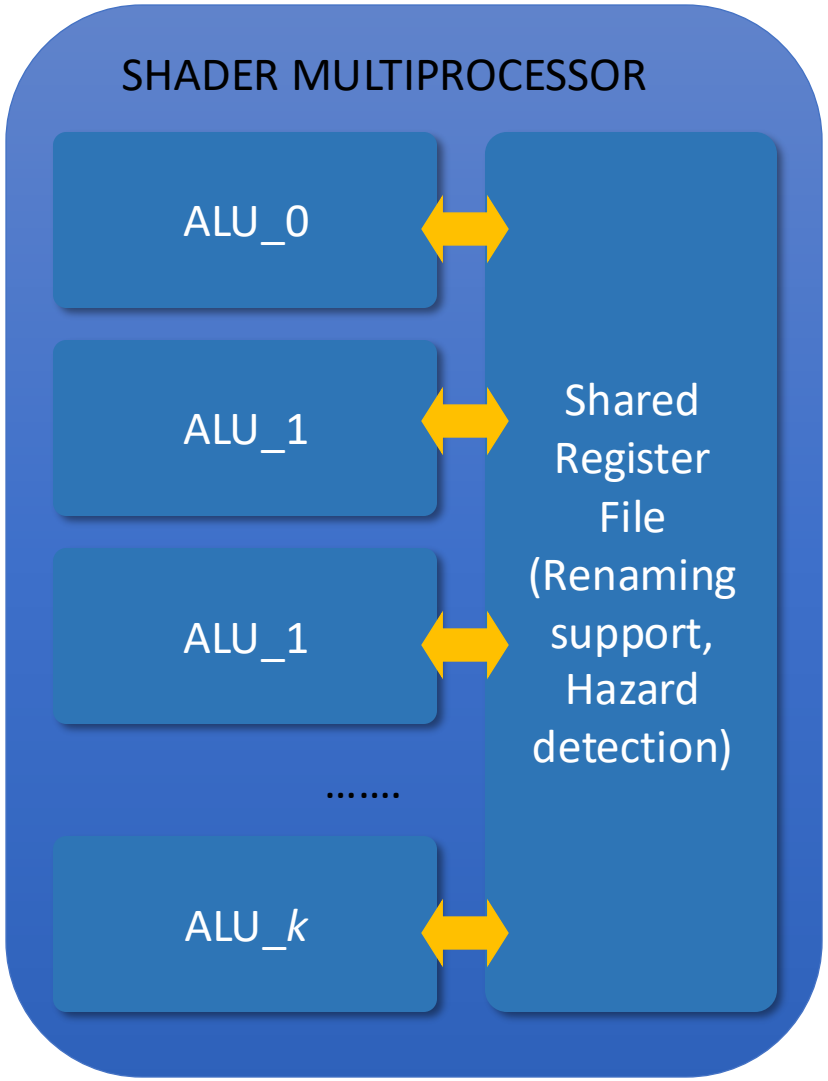
Shared Register File/Cache

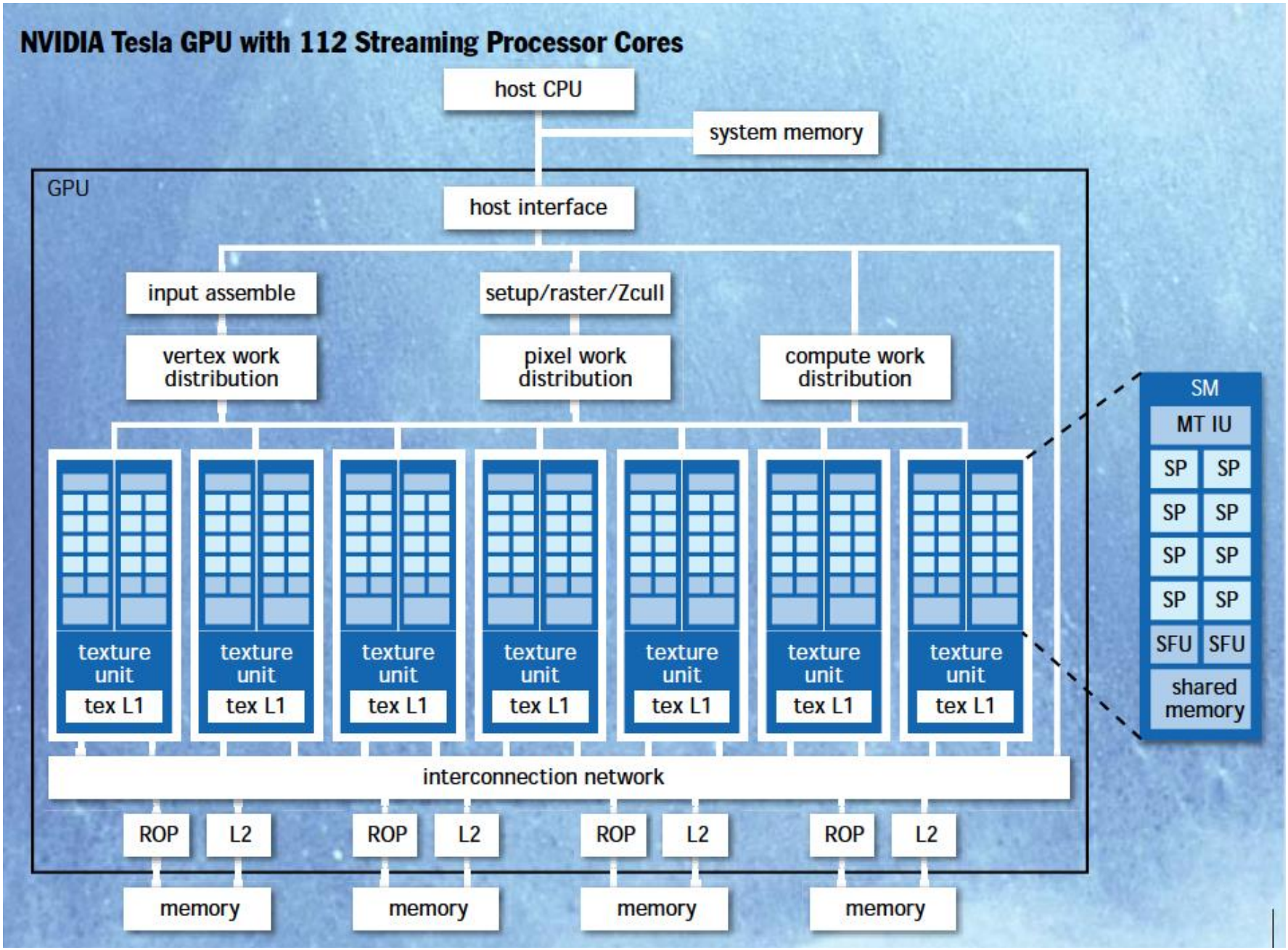
- Separate register files
  - Strict streaming processing mode
  - No data sharing at instruction-level granularity



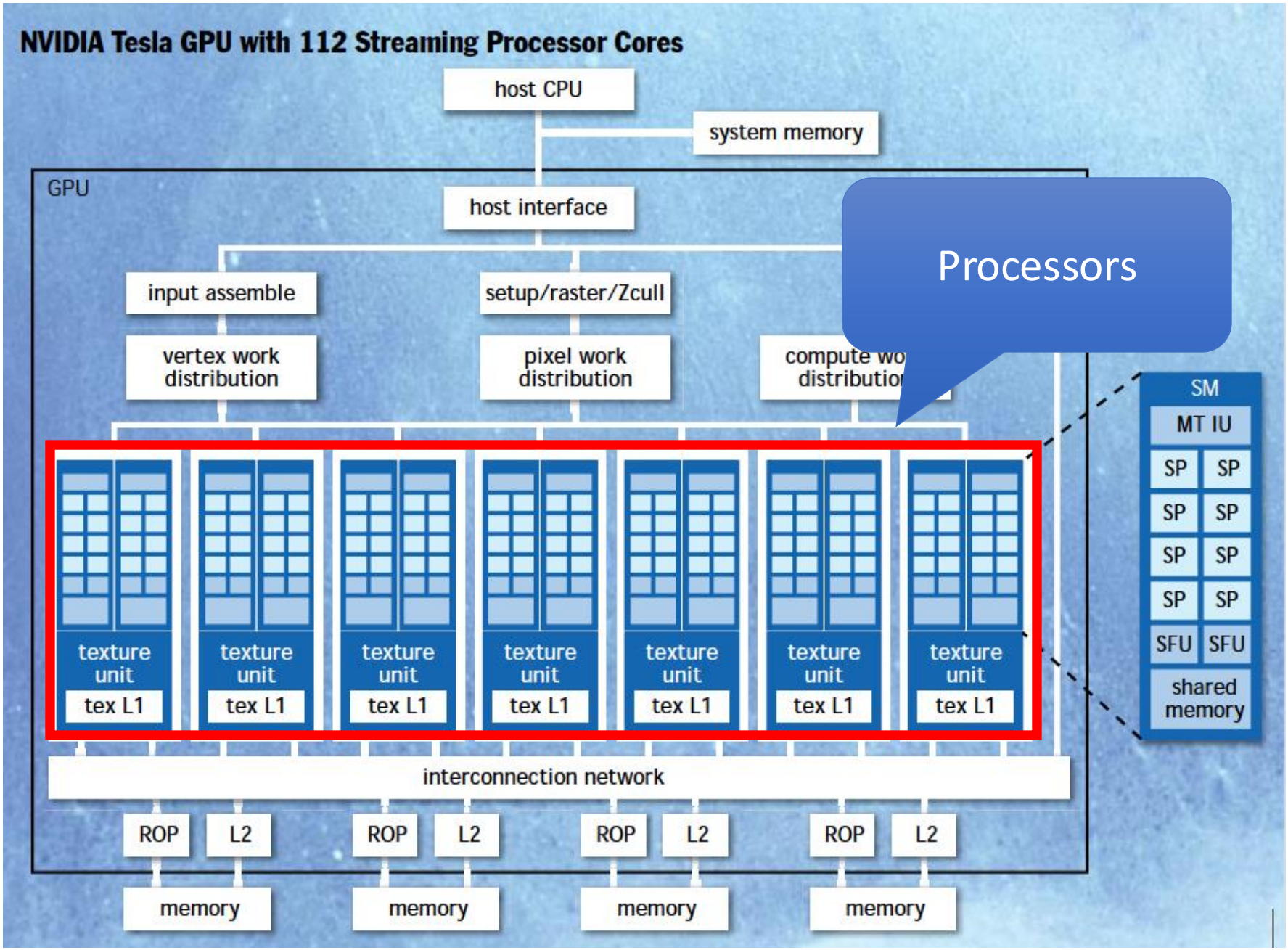
Shared Register File/Cache

- Shared register file
  - Extra memory hierarchy between registers and texture cache
  - Allows fine-grain data sharing between threads

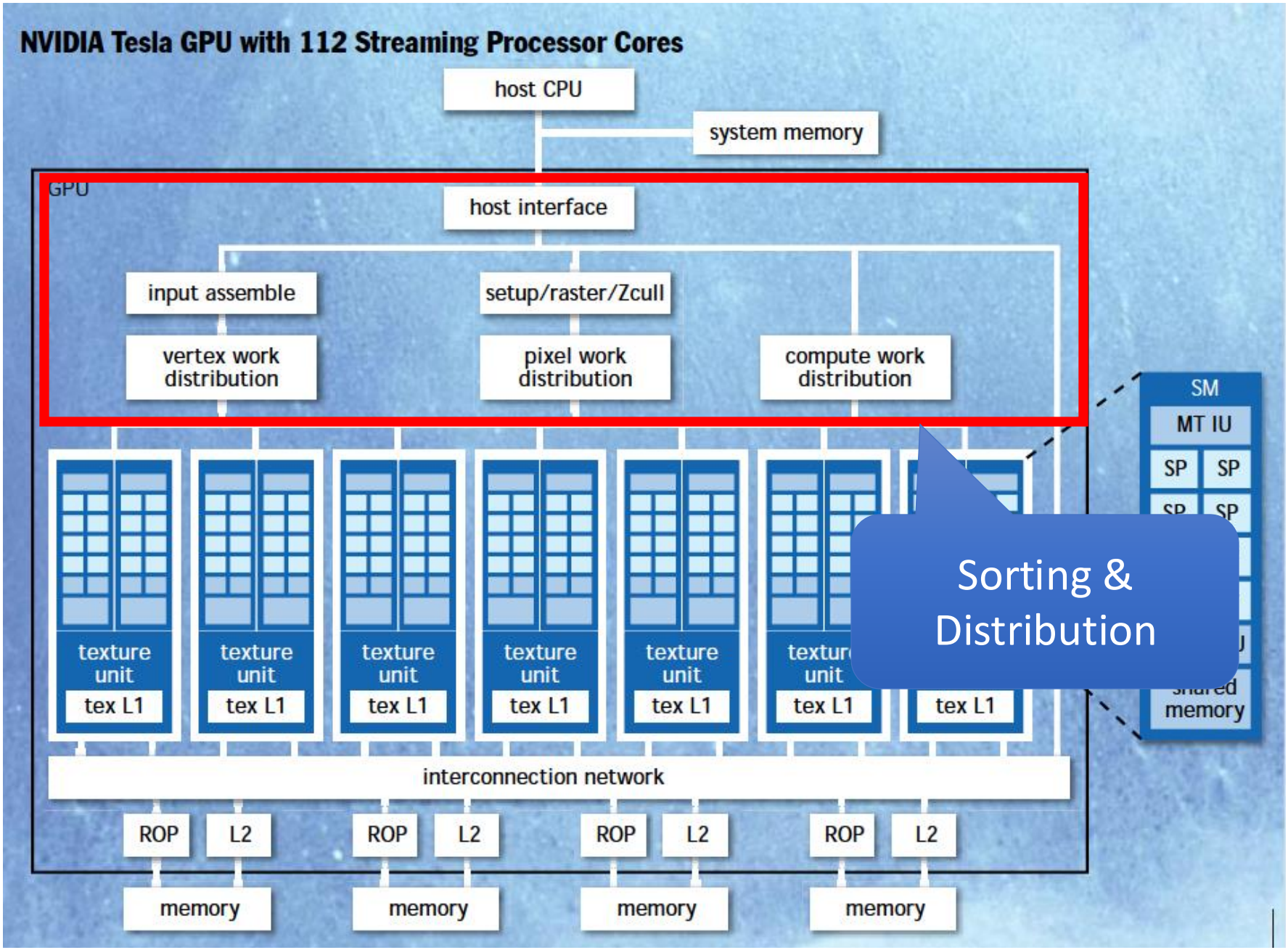




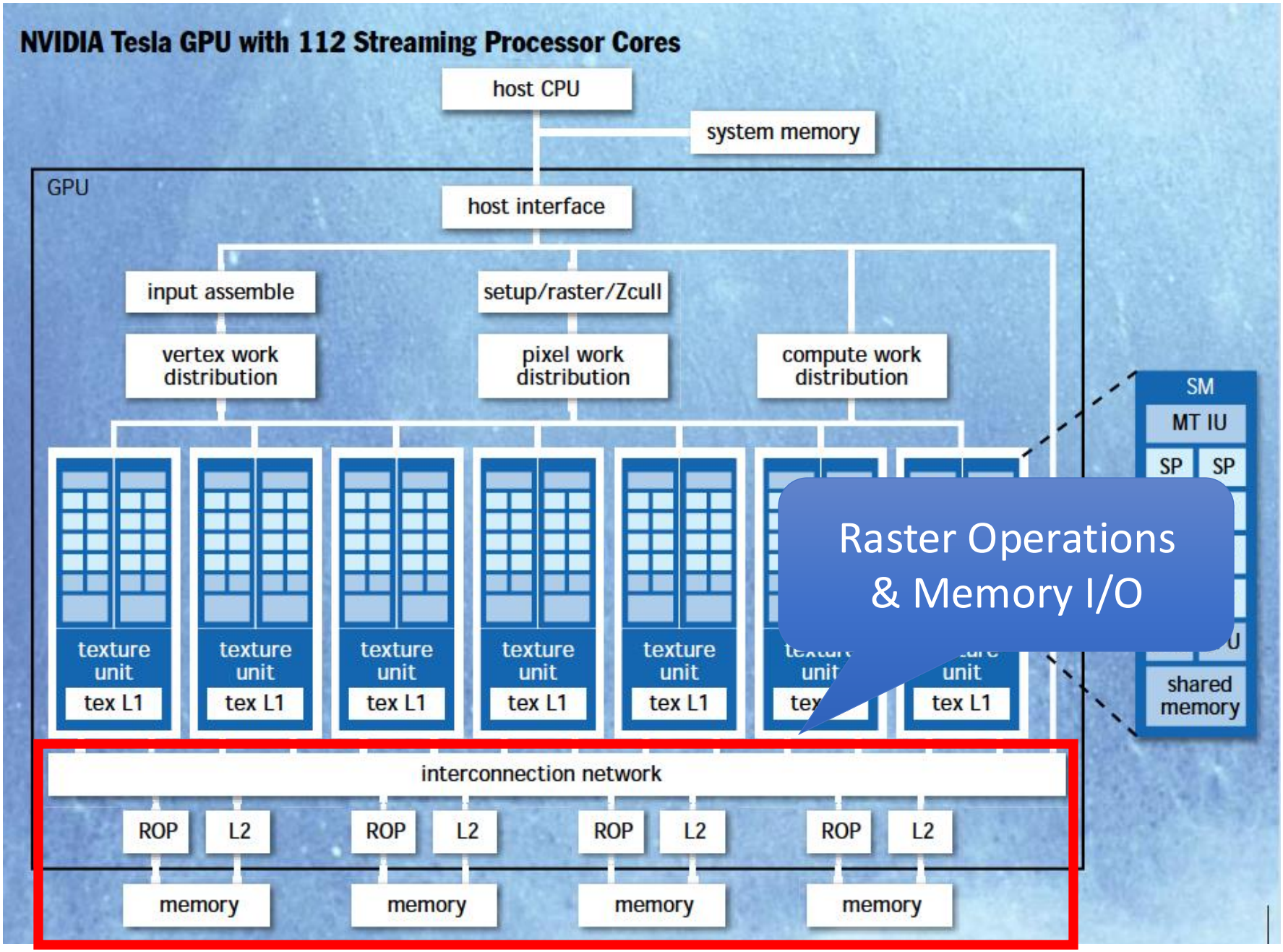
John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. Queue, 6(2):40{53, 2008.



John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. Queue, 6(2):40{53, 2008.



John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. Queue, 6(2):40{53, 2008.



John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. Queue, 6(2):40{53, 2008.

GM204 consists of four GPCs (Graphics Processing Clusters)
16 Maxwell SMs (SMM)

GPU Programming #3: Parallel Computing

Wei-Chao Chen

Visiting Professor, National Taiwan University

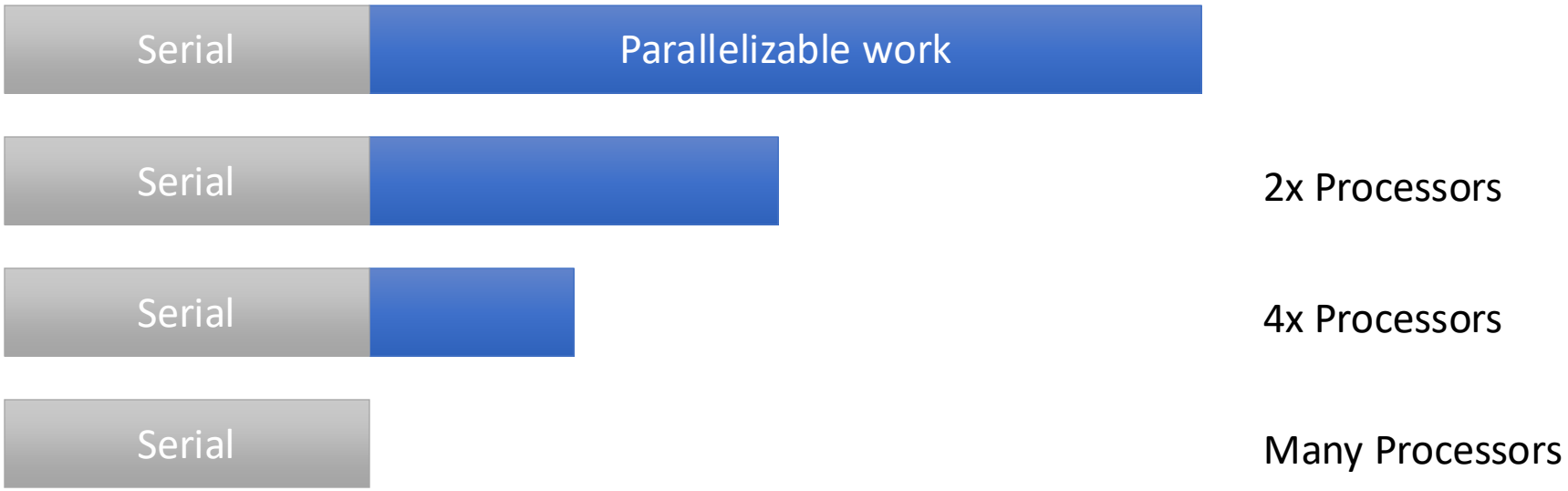
weichao.chen@gmail.com

Parallel Computing Goals

- To solve your problem in less time
  - Divide one big problem into smaller pieces
  - Solve smaller problems concurrently
  - Allows us to solve a bigger problem
- To parallelize a problem
  - Identify dependencies in the problem
  - Identify critical paths in the algorithm
  - Modify dependencies to shorten the critical paths

Amdahl's Law

- Named after computer architect Gene Amdahl
- Speedup of a parallel computer is limited by the amount of serial work



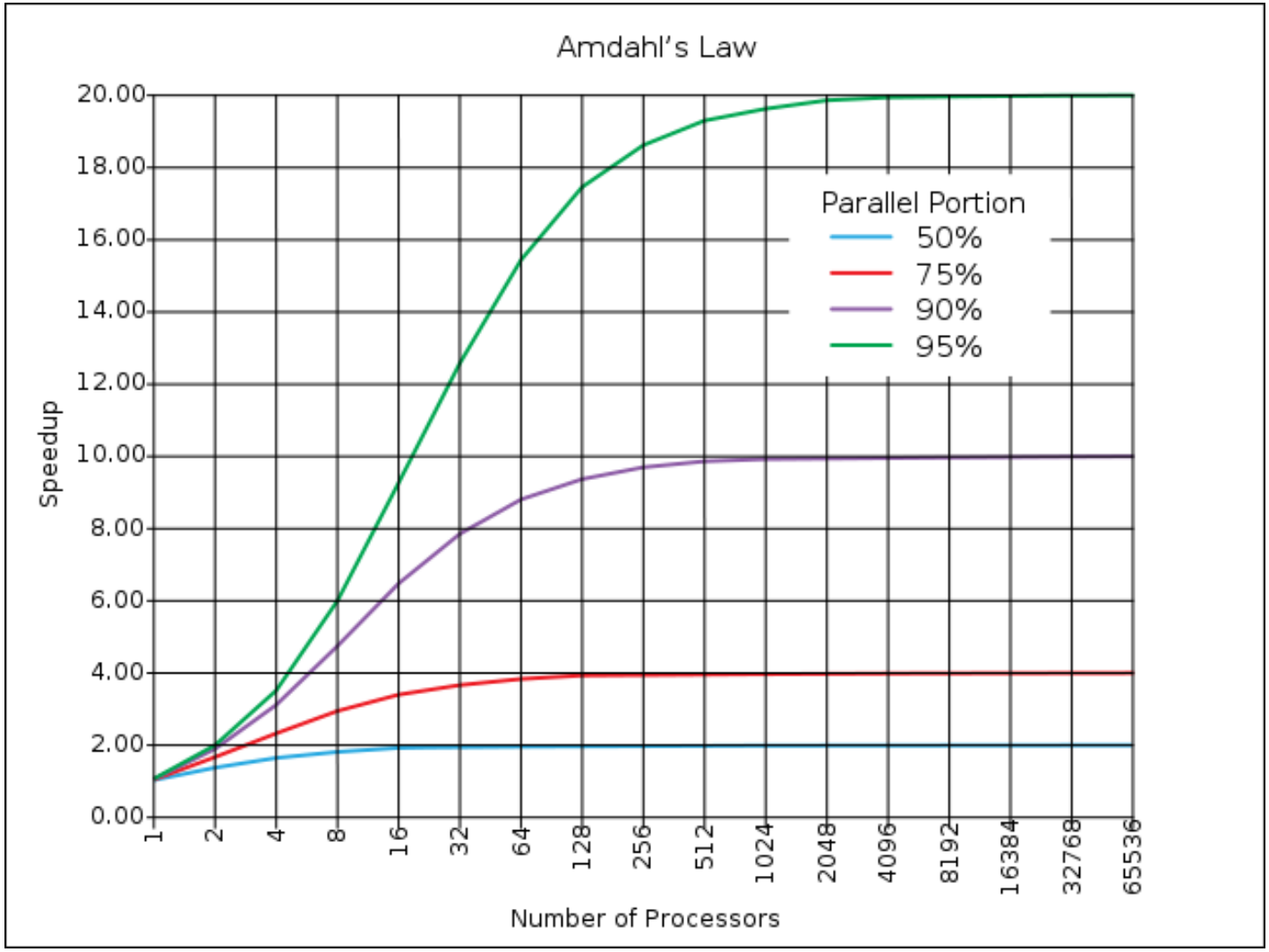
Amdahl's Law

- Total speedup:

$$\frac{1}{(1-P) + (P/S)}$$

  - *P:* parallelizable work
  - *S:* speedup for parallizable work

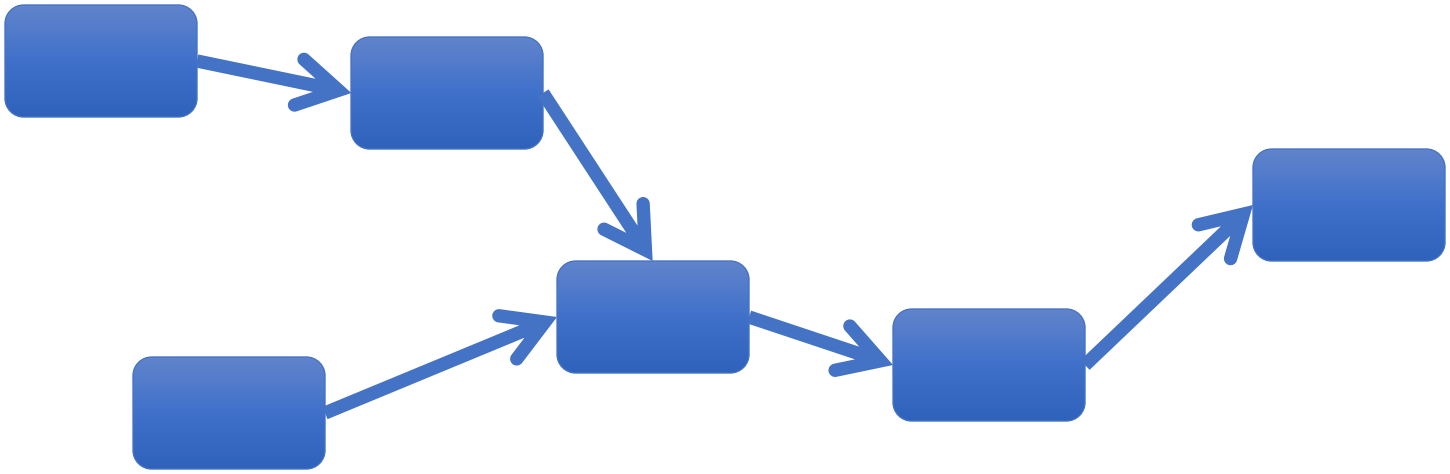
Amdahl's Law



Source: wikipedia.org

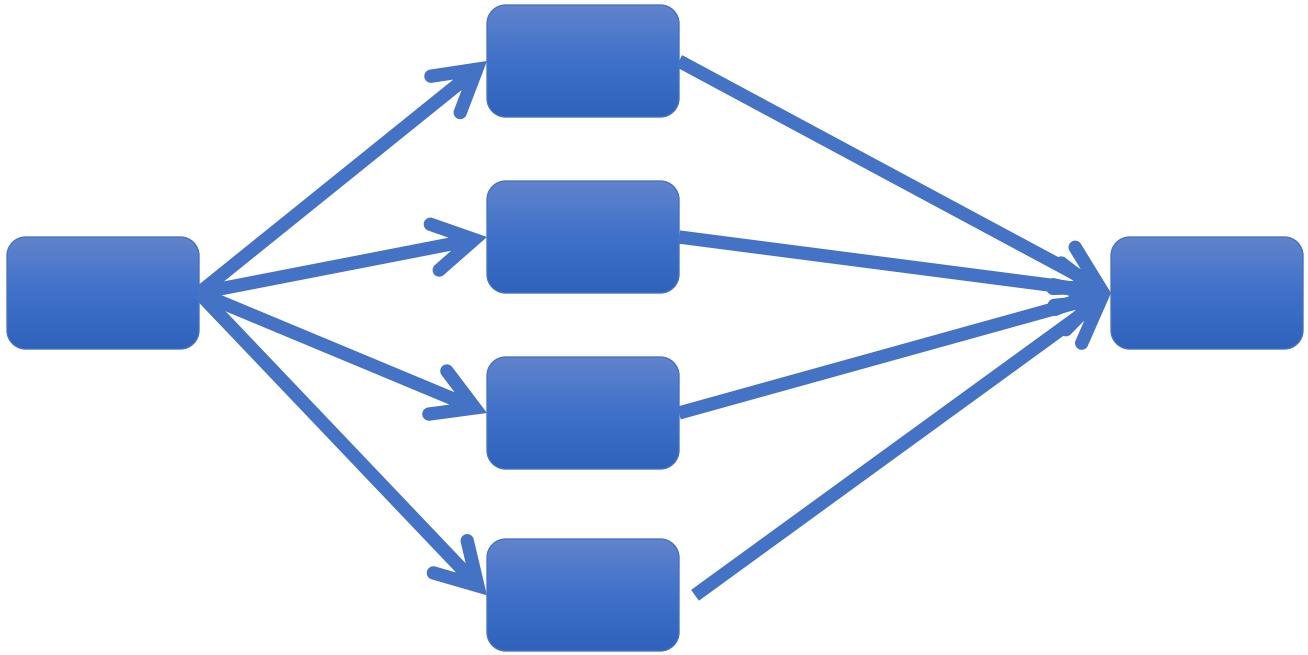
Dependency

- Data/Instruction dependency



Dependency

- Data/Instruction dependency
  - Shorter critical path, more parallelism opportunities



Race Conditions

- Consider the following program:

```
Thread 1          Thread 2

Load R, A         Load R, A
R:= R+1           R:= R+1
Save R, A         Save R, A
```

- What is the outcome?

Race Conditions

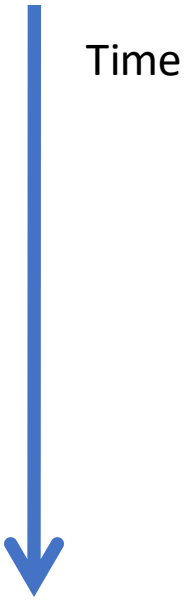
- Scenario 1:

Thread 1	**Thread 2**
`Load R, A`	
	`Load R, A`
`R:= R+1`	
	`R:= R+1`
`Save R, A`	
	`Save R, A`

Race Conditions

- Scenario 1.5:

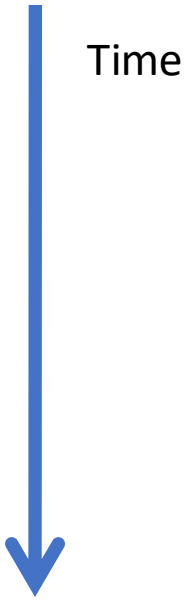
```
Thread 1          Thread 2

Load R, A

                  Load R, A
                  R:= R+1
                  Save R, A

R:= R+1
Save R, A
```



Race Conditions

- Scenario 2:

Thread 1	**Thread 2**
`Load R, A`	
`R:= R+1`	
`Save R, A`	
	`Load R, A`
	`R:= R+1`
	`Save R, A`



Race Conditions

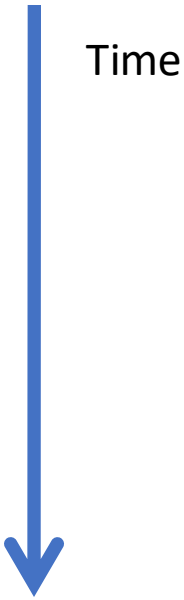
- Modification: lock resources

```
Thread 1          Thread 2

Lock A            Lock A
Load R, A         Load R, A
R:= R+1           R:= R+1
Save R, A         Save R, A
Release A         Release A
```

Race Conditions

- Modification: lock resources

```
Thread 1          Thread 2

Lock A
Load R, A
R:= R+1
Save R, A
Release A
                  Lock A
                  Load R, A
                  R:= R+1
                  Save R, A
                  Release A
```



Semaphores

- Multi-valued lock
  - Dutch words - P: passing (passering); V: release (vrijgave)

```
procedure P (S : Semaphore; I : Integer);
begin
  repeat Wait() until S >= I;
  S := S - I
end;

procedure V (S : Semaphore; I : Integer);
begin
  S := S + I
end;
```

Code within the boxes are *atomic* operations

Semaphores

- Multi-valued lock
  - Dutch words - P: passing (passering); V: release (vrijgave)

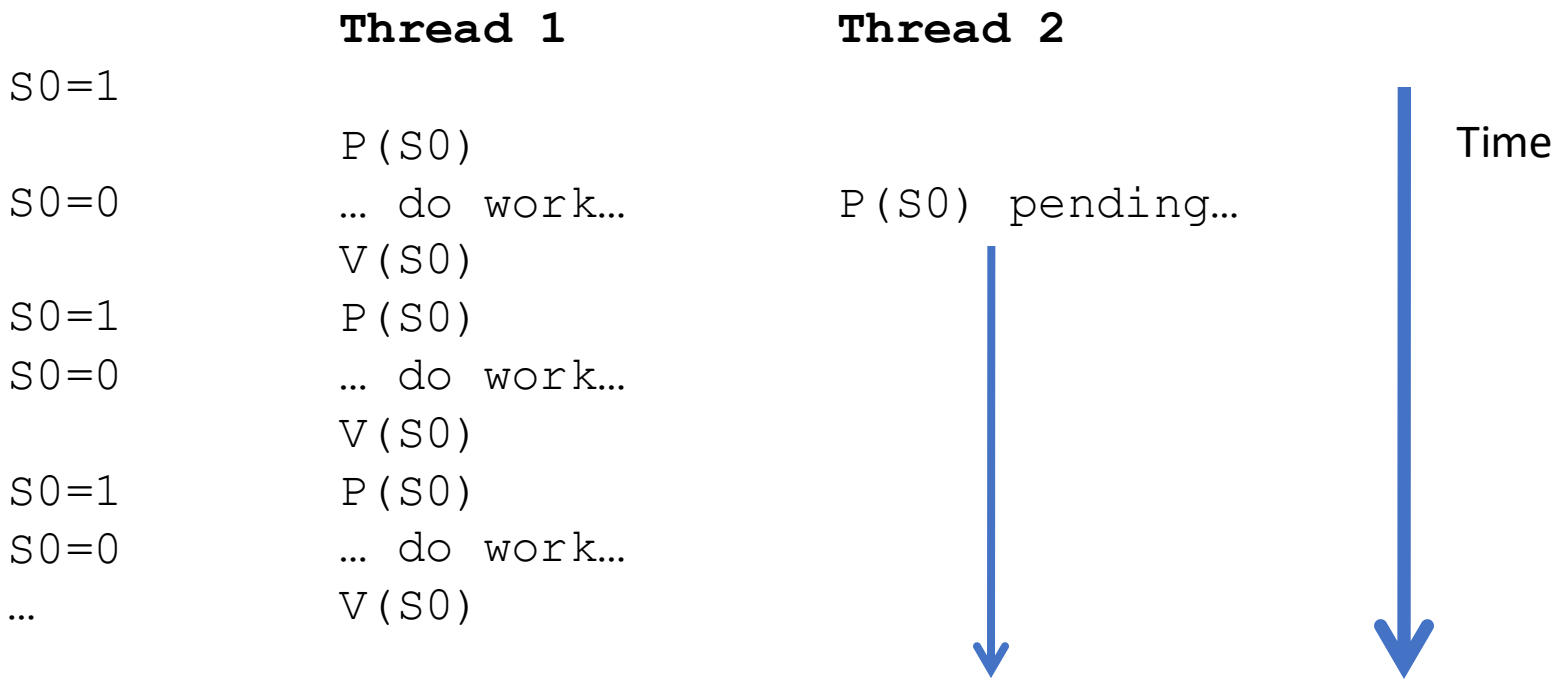
```
procedure P (S : Semaphore; I : Integer);
begin
  repeat Wait() until S >= I;
  S := S - I
end;
```

```
procedure V (S : Semaphore; I : Integer);
begin
  S := S + I
end;
```

Code within the boxes are *atomic* operations

Semaphore: Issue

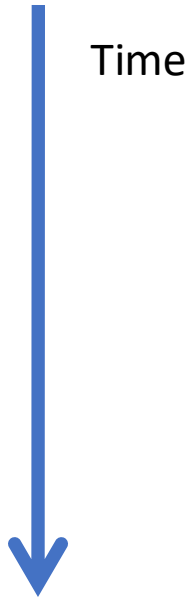
- Thread 2 may never get through



Semaphore Queue

- With a request queue, thread 2 won't starve (FIFO)

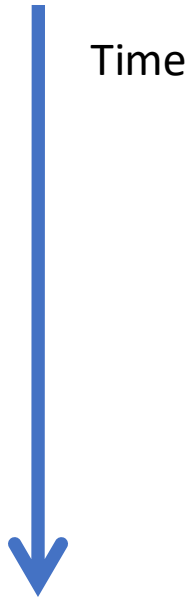
	Thread 1	**Thread 2**
S0=1, Q={}		
	P(S0)	
S0=0 Q={}	… do work…	P(S0) requested



Semaphore Queue

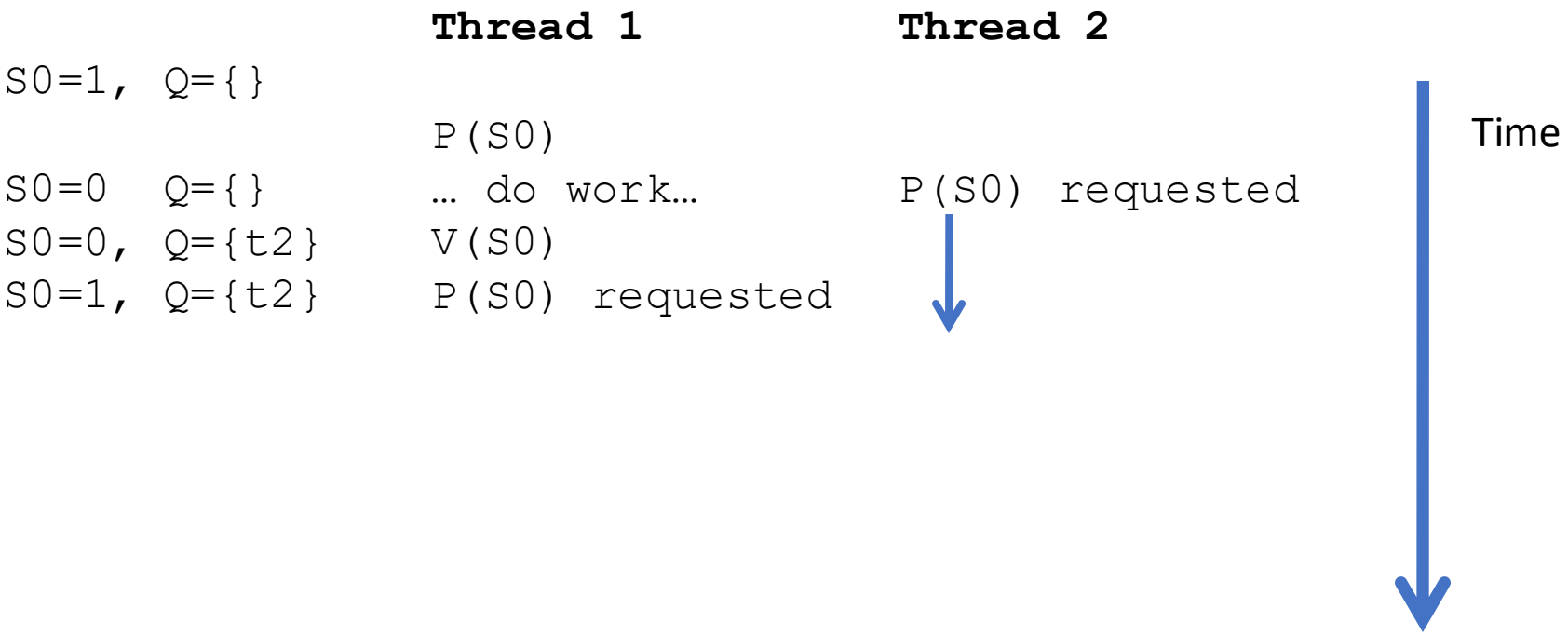
- With a request queue, thread 2 won't starve (FIFO)

	Thread 1	**Thread 2**
S0=1, Q={}		
	P(S0)	
S0=0 Q={}	… do work…	P(S0) requested
S0=0, Q={t2}	V(S0)	
S0=1, Q={t2}		



Semaphore Queue

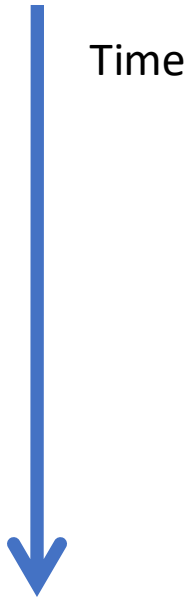
- With a request queue, thread 2 won't starve (FIFO)



Semaphore Queue

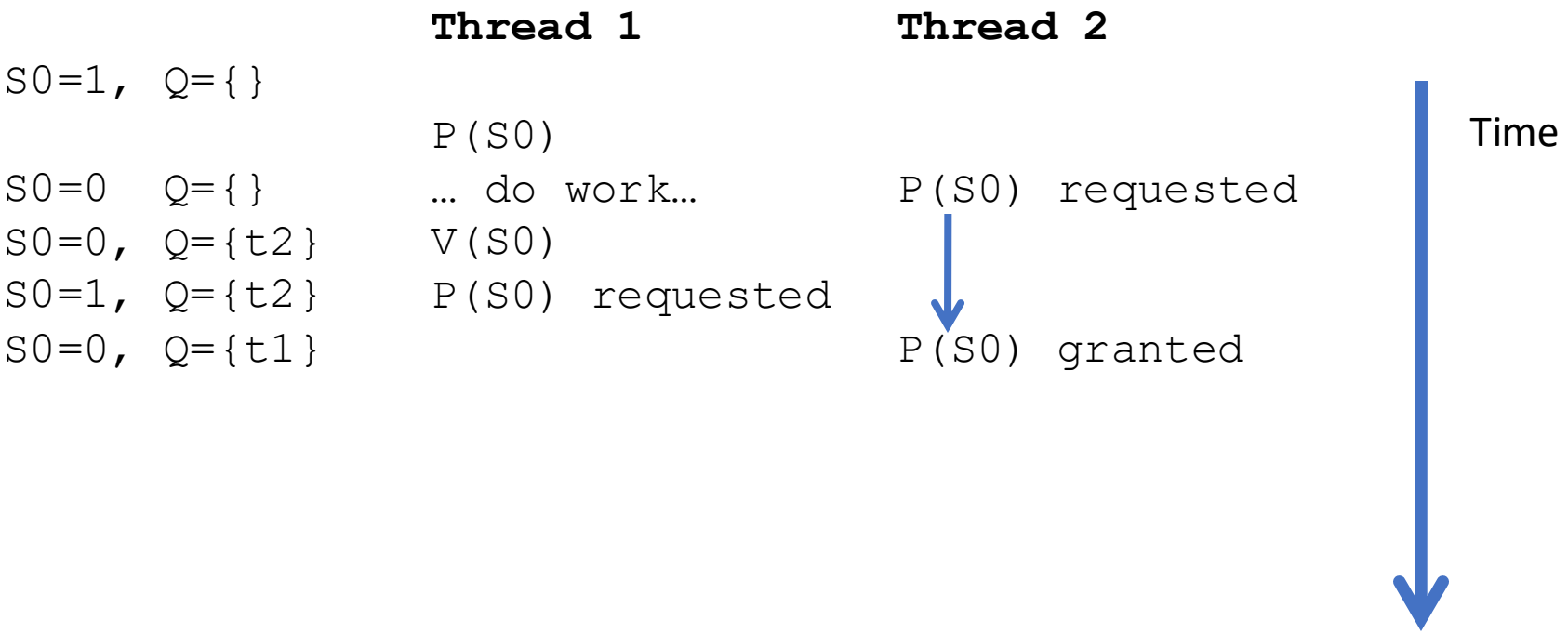
- With a request queue, thread 2 won't starve (FIFO)

```
                  Thread 1          Thread 2
S0=1, Q={}
                  P(S0)
S0=0  Q={}        … do work…        P(S0) requested
S0=0, Q={t2}      V(S0)
S0=1, Q={t2}      P(S0) requested
S0=1, Q={t2,t1}
```



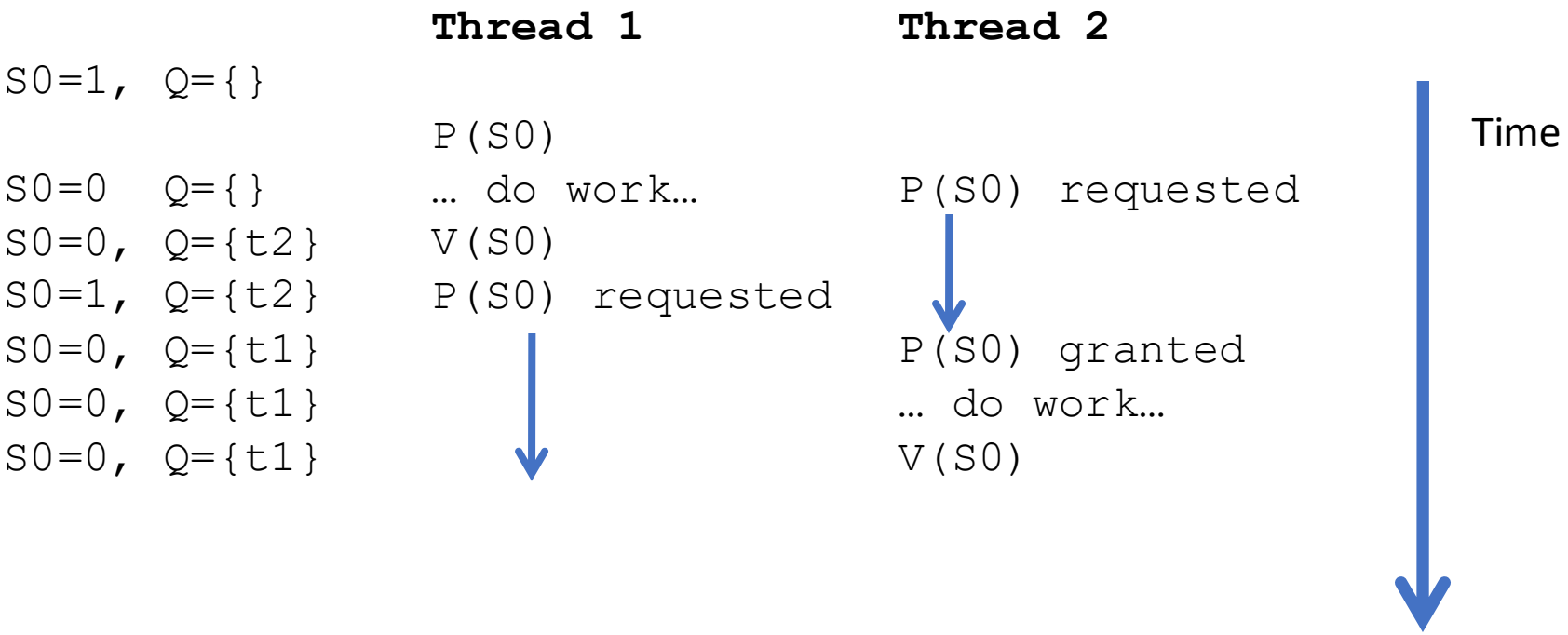
Semaphore Queue

- With a request queue, thread 2 won't starve (FIFO)



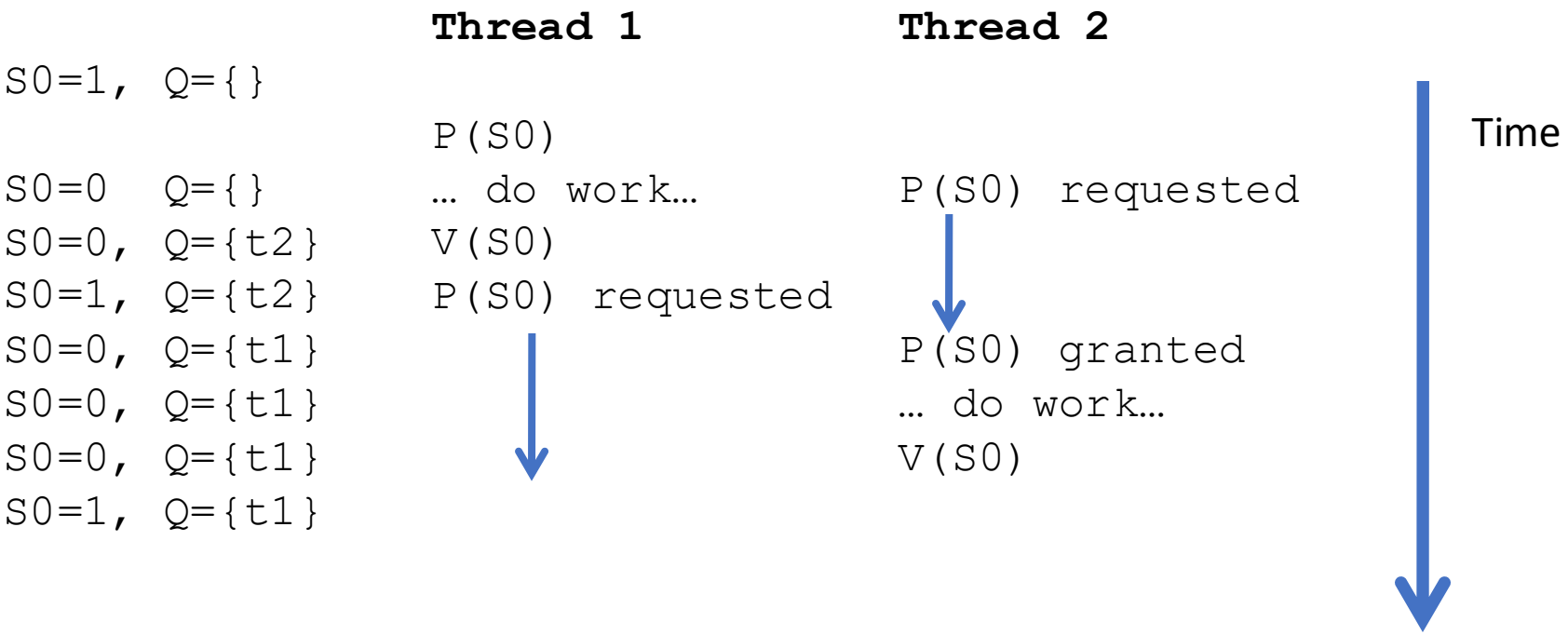
Semaphore Queue

- With a request queue, thread 2 won't starve (FIFO)



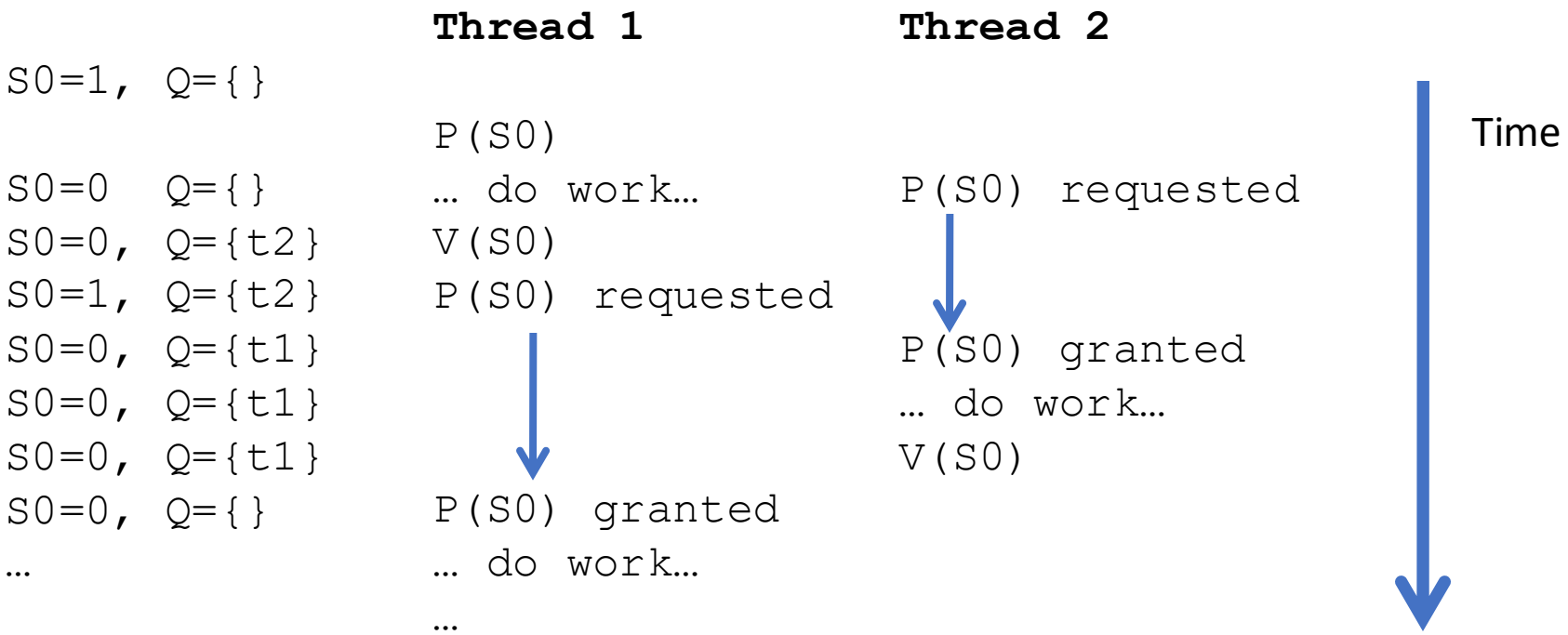
Semaphore Queue

- With a request queue, thread 2 won't starve (FIFO)



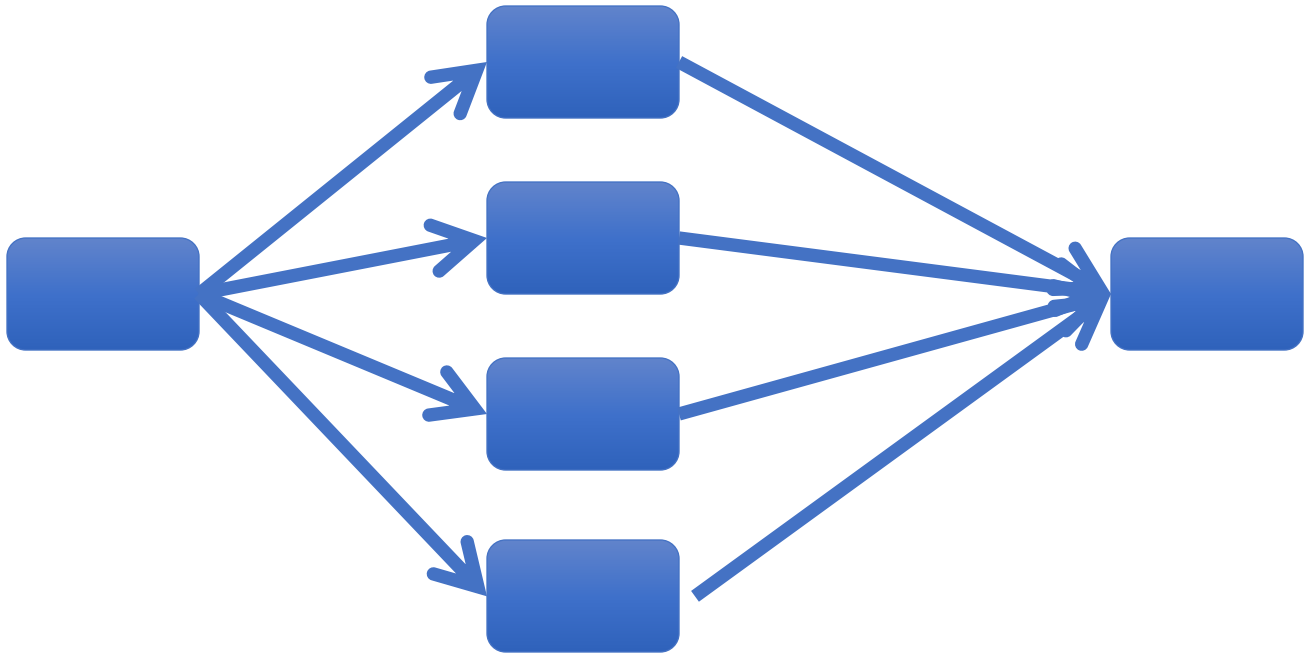
Semaphore Queue

- With a request queue, thread 2 won't starve (FIFO)



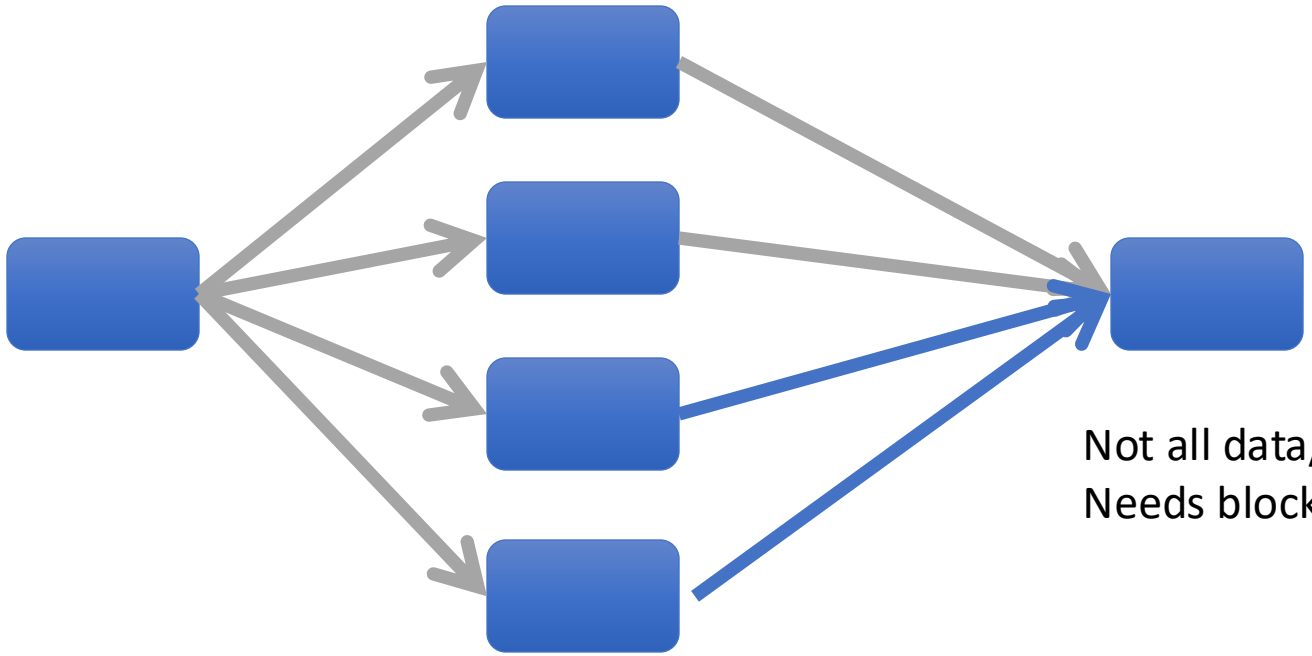
Synchronization

- What's wrong with this picture?



Synchronization

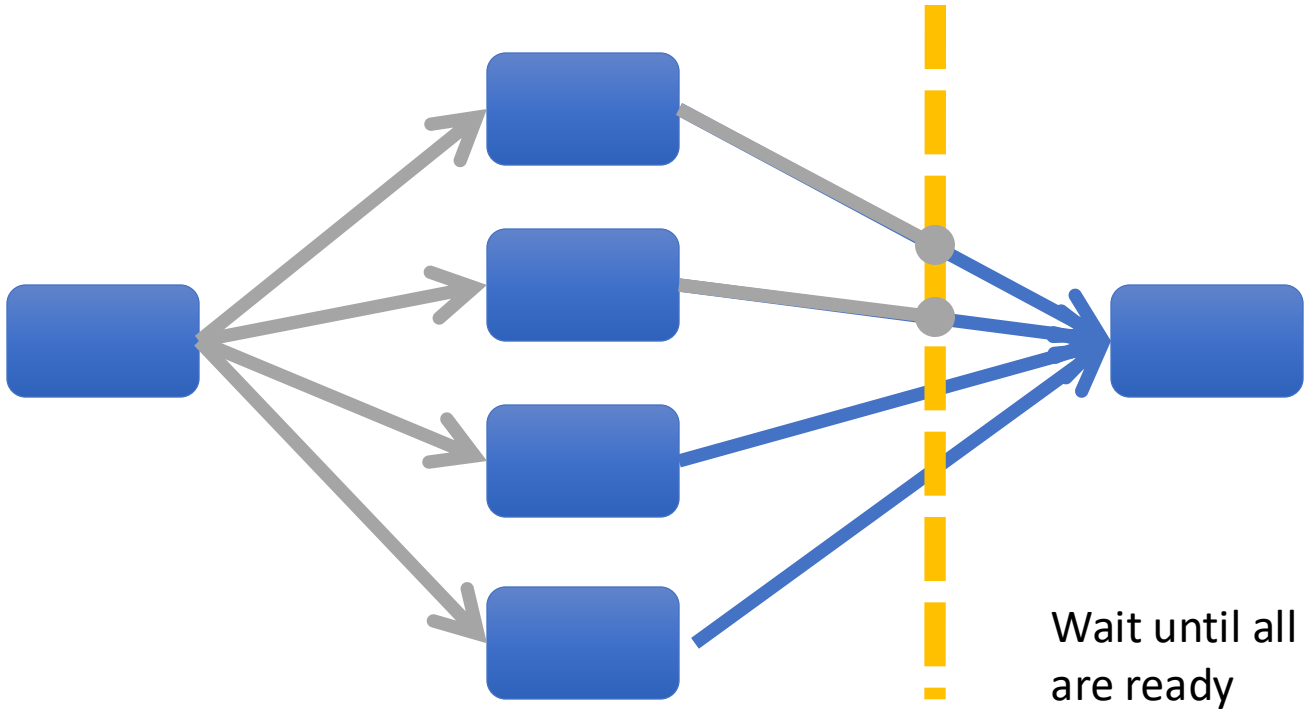
- What's wrong with this picture?



Not all data/threads ready
Needs blocking mechanism

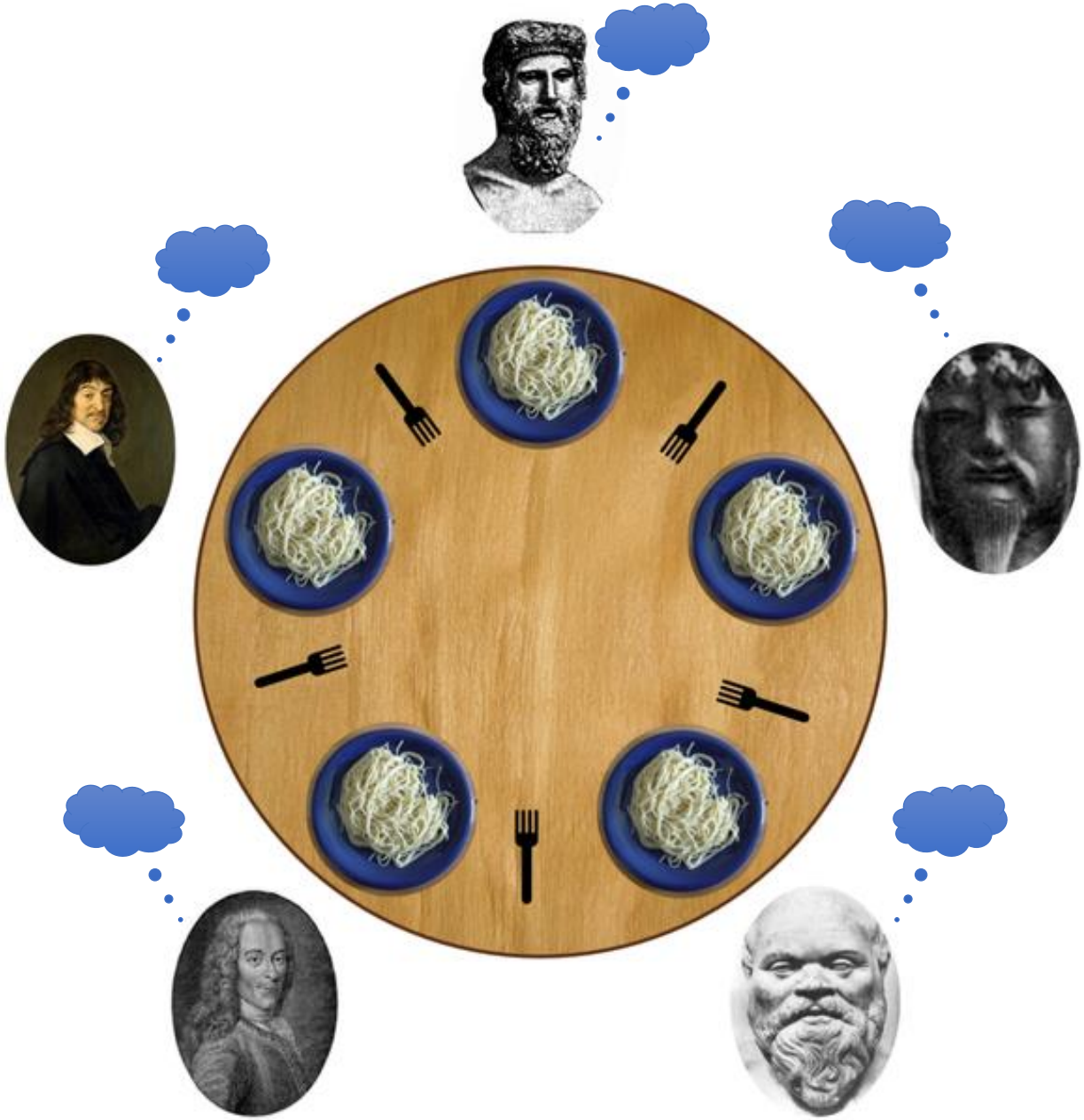
Synchronization

- Synchronize at *barriers*

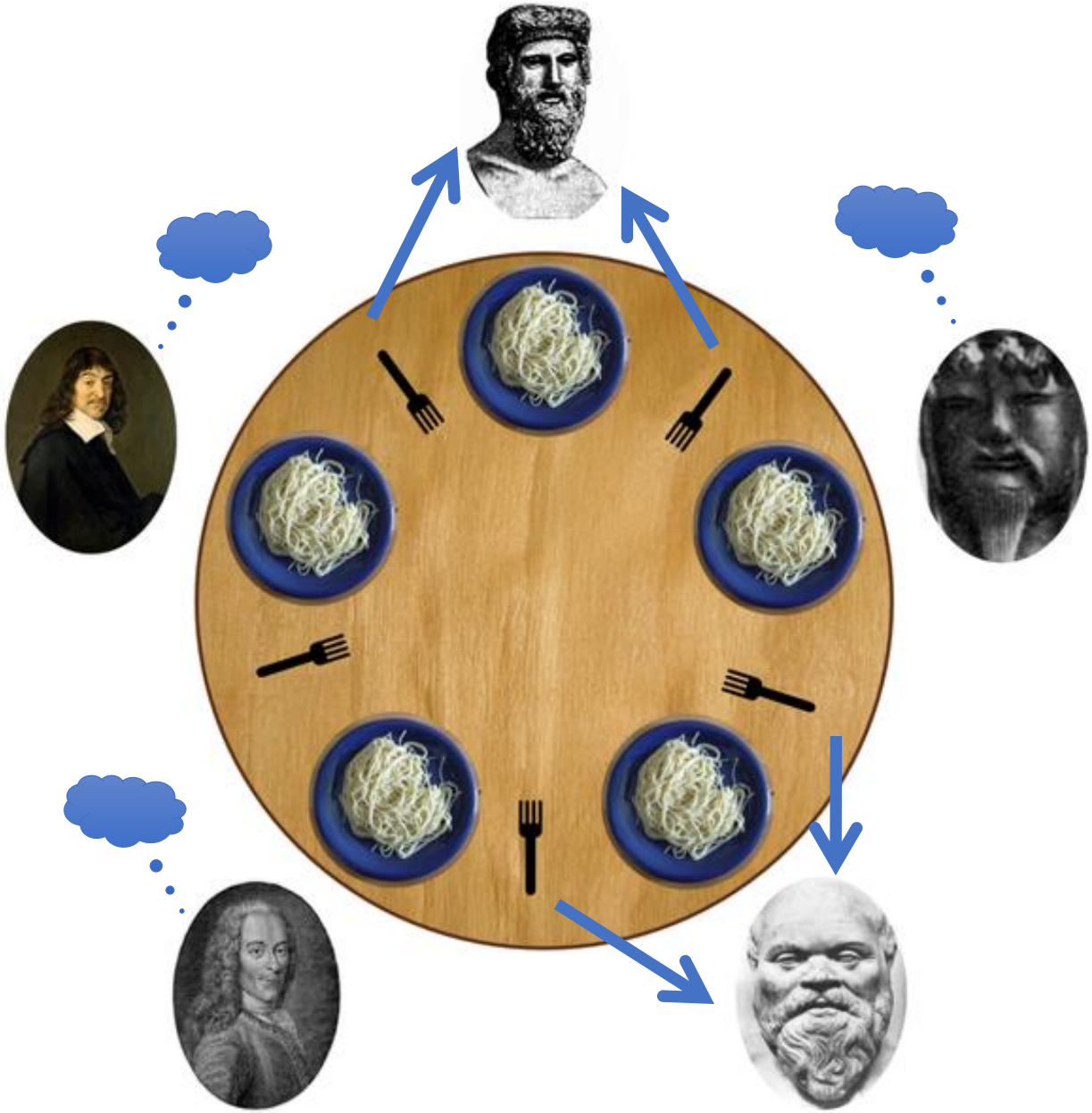


Resource Management

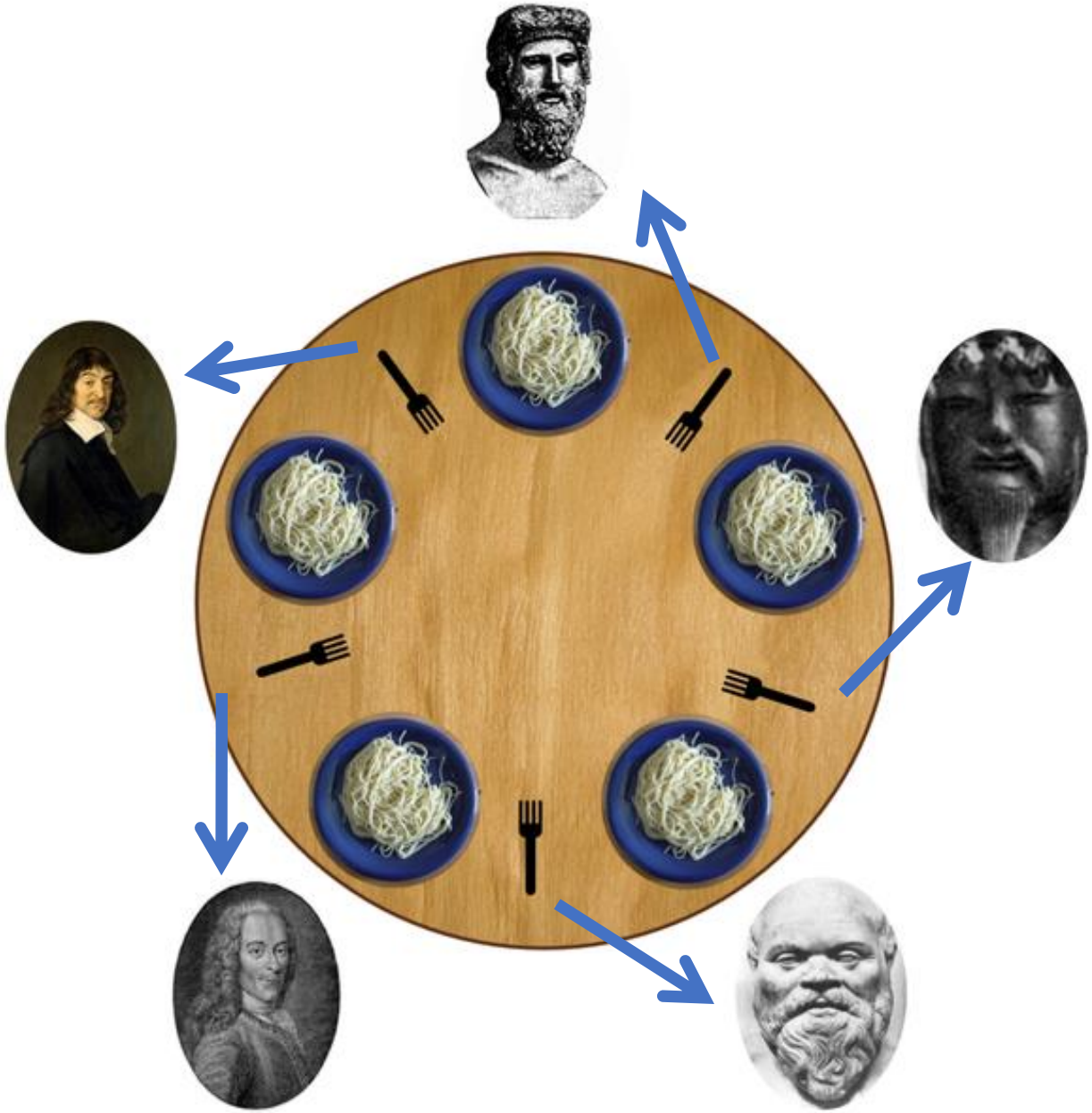
- Proper resource management is important!
  - Deadlock - dining philosophers' problem
    - “Starvation”
  - Can semaphores solve this problem?
    - “Ask the waiter”
    - Ref: A. Tannenbaum, “Modern Operating Systems”



Source: http://en.wikipedia.org/wiki/Dining_philosophers_problem

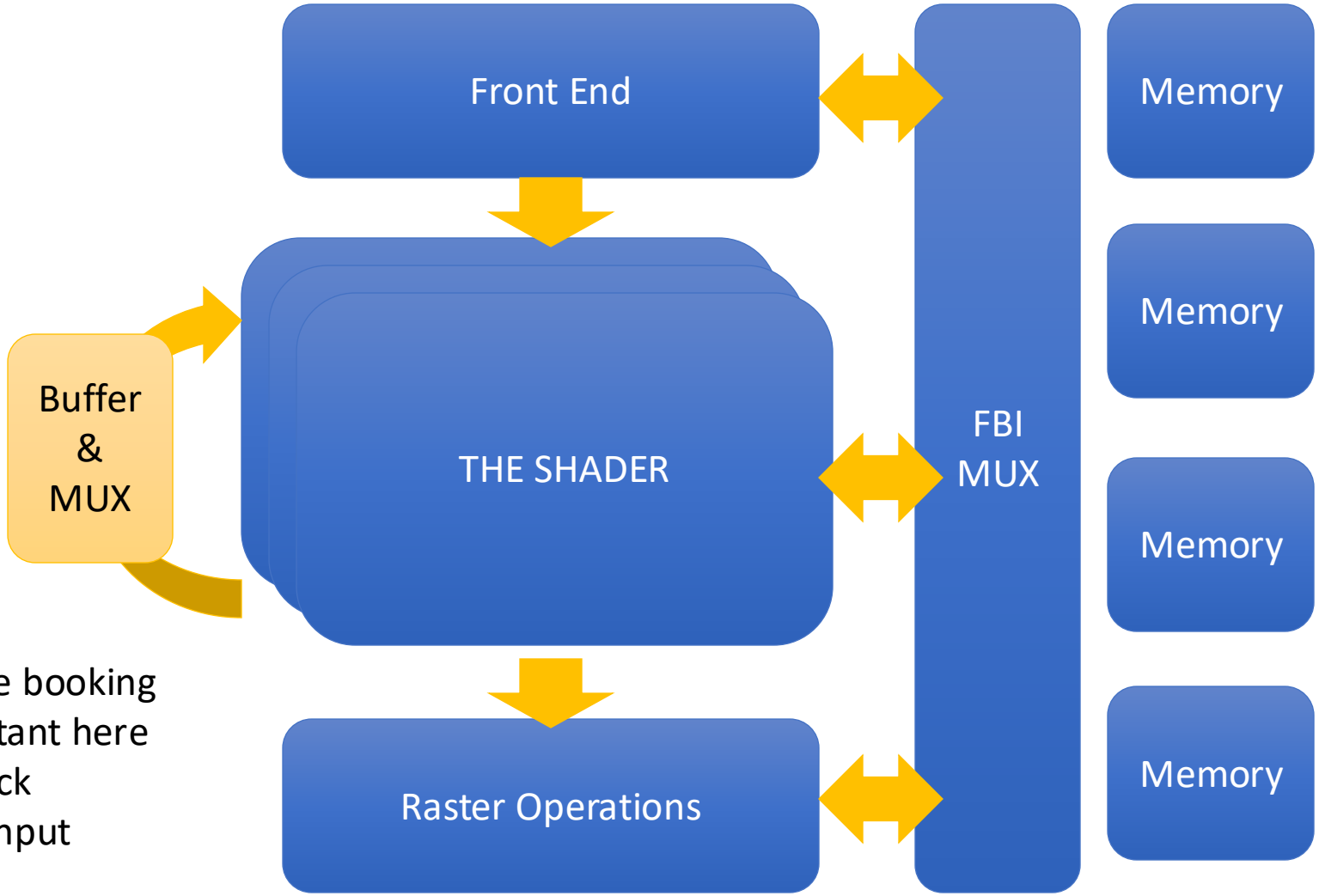


Source: http://en.wikipedia.org/wiki/Dining_philosophers_problem



Source: http://en.wikipedia.org/wiki/Dining_philosophers_problem

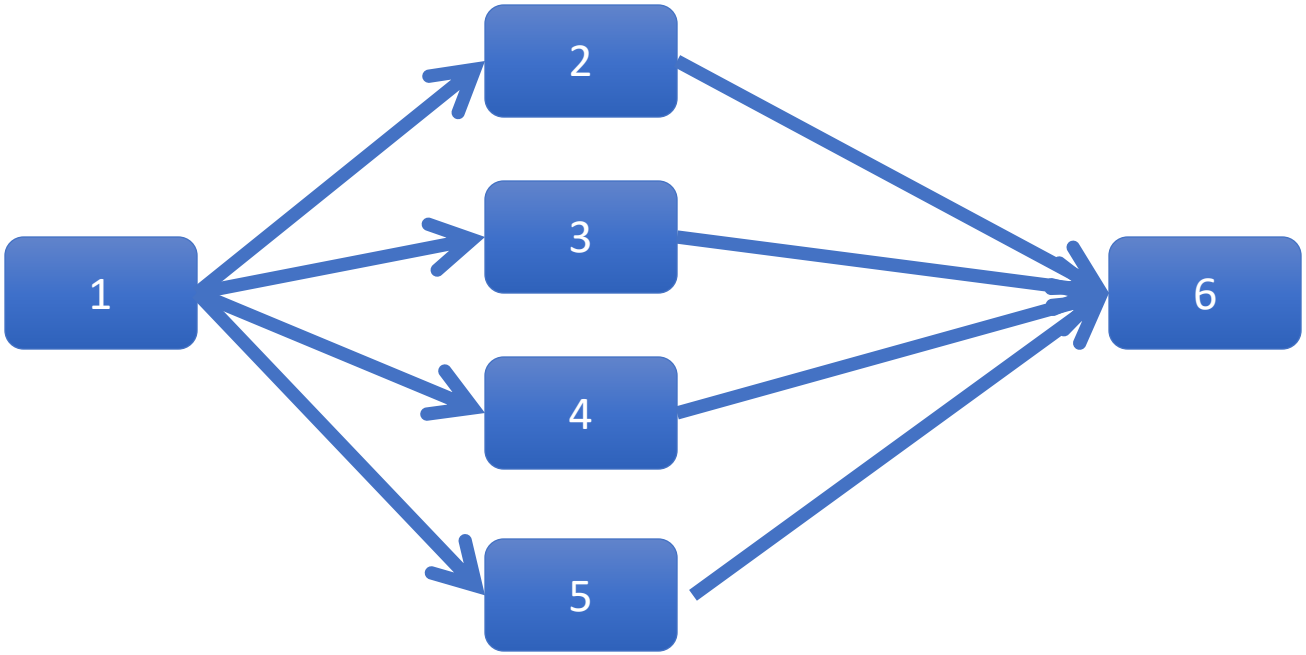
Remember This?



Resource booking
is important here
- deadlock
- throughput

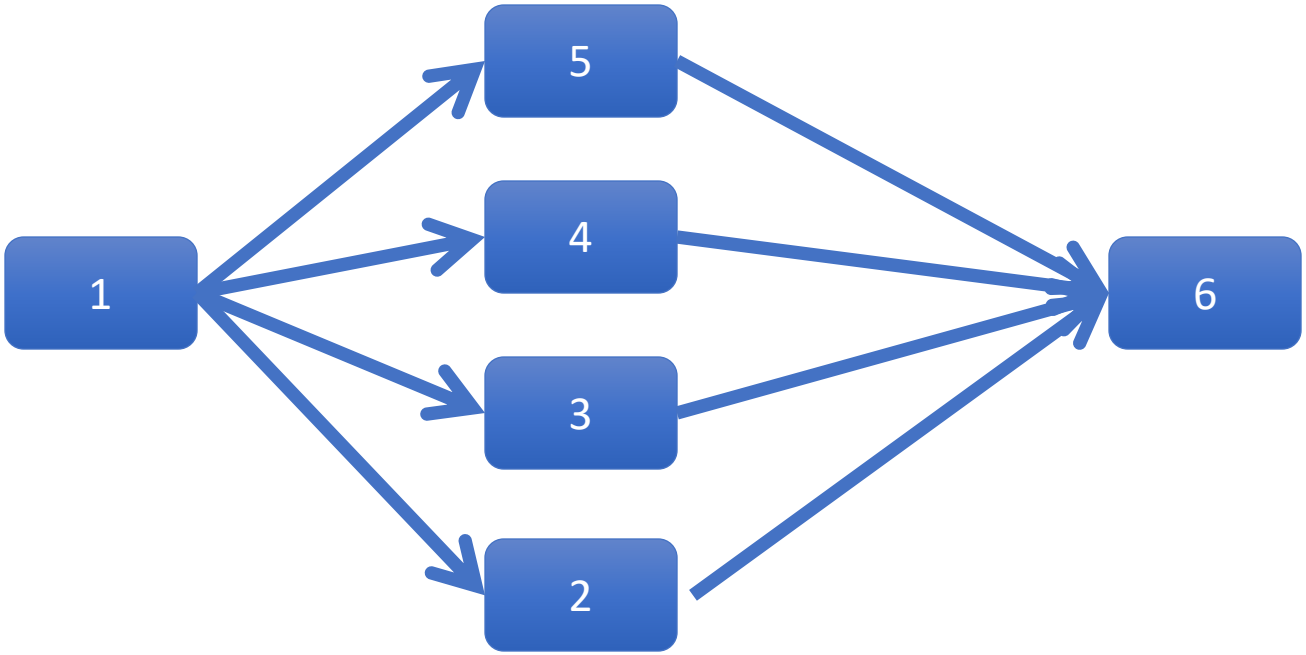
Consistency Models

- Serial version of the program



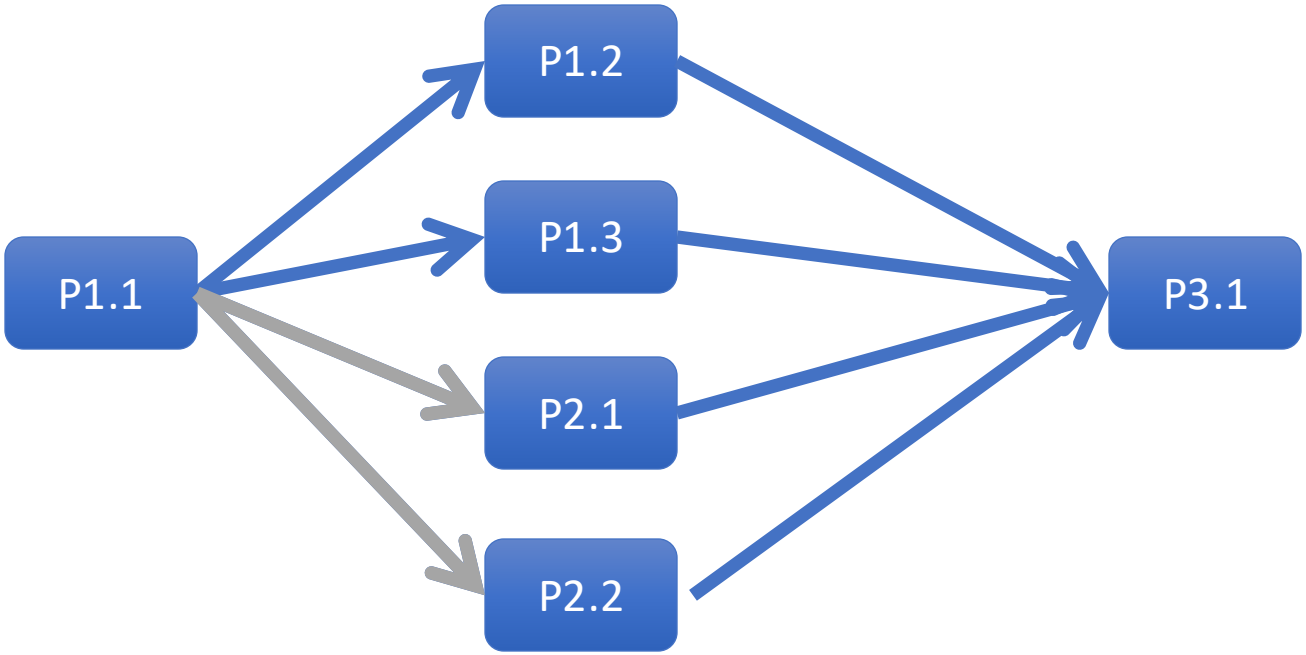
Consistency Models

- Does this give a different result?



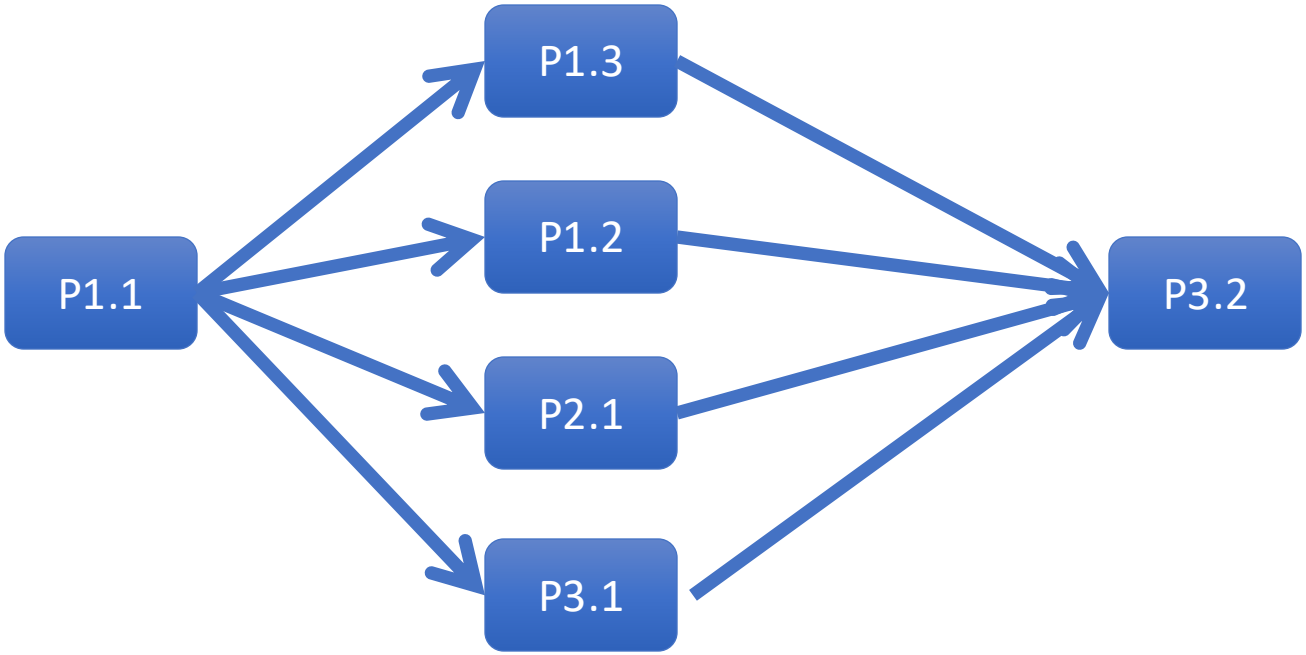
Consistency Models

- Notion of time vanishes with distributed systems



Consistency Models

- Notion of time vanishes with distributed systems
  - Dataflow architecture?



Types of Parallelism

- Multiple programs
  - Multi-tasking
  - Multi-threading
- Single program
  - Instruction-level parallelism
  - Data parallelism

Instruction-Level Parallelism

- Multiple instructions in a serial program get executed simultaneously
  - Superscalar, etc

```
A=A+1
B=B+1
C=C+A
C=C+B
…
```

T=0

Instruction-Level Parallelism

- Multiple instructions in a serial program get executed simultaneously
  - Superscalar, etc

```
A=A+1
B=B+1
C=C+A
C=C+B
...
```

T=1

(failed to issue because of dependency)

Instruction-Level Parallelism

- Multiple instructions in a serial program get executed simultaneously
  - Superscalar, etc

```
A=A+1
B=B+1
C=C+A
C=C+B
...
```

T=2

Data-Level Parallelism

- Data-centric rather than instruction-centric

```
for i:=1 to 100,
    c[i] = a[i] + b[i]
end
```

Data-Level Parallelism

- Data-centric rather than instruction-centric
  - Single operation repeated over different data

```
c[1] = a[1] + b[1]
c[2] = a[2] + b[2]
c[3] = a[3] + b[3]
c[4] = a[4] + b[4]
…
```

Flynn's Taxonomy

- Classification for parallel computers and programs

	Single Instruction	Multiple Instruction
Single Data	SISD (Single-core, single-threaded CPU)	MISD (Very rare)
Multiple Data	SIMD (GPU, Vector processors)	MIMD (Multi-core, multi-programming)

GPU == SIMD?

- Suppose we have this program:

```
c[i] = a[i] + b[i]
if (c[i]>pi)
    path-A
else
    path-B
…
```

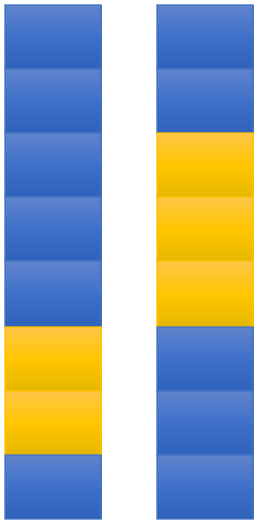
- What happens at the if... statement?

Divergence

- Some say yay, some say nay

yay

```
c[i] = a[i] + b[i]
if (c[i]>pi)
    path-A
    path-A
    path-A
else
    path-B
    path-B
 …
```

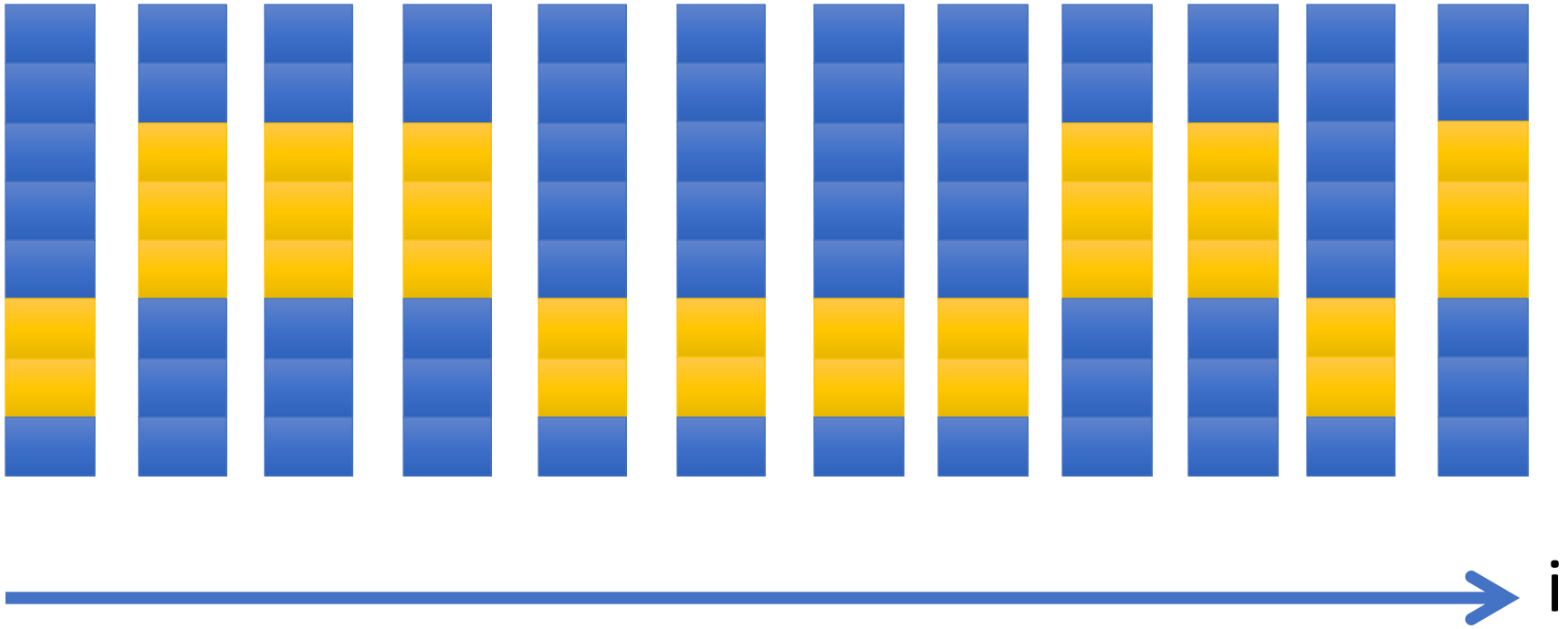


nay

```
c[i] = a[i] + b[i]
if (c[i]>pi)
    path-A
    path-A
    path-A
else
    path-B
    path-B
 …
```

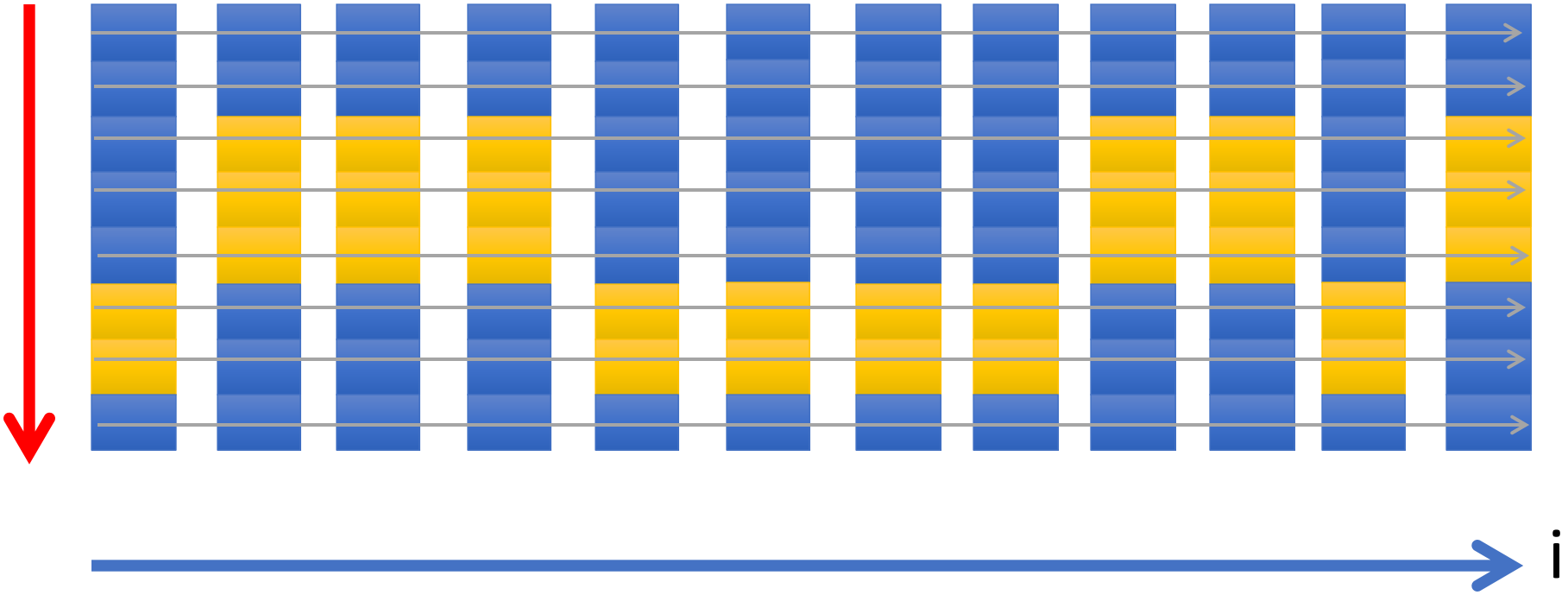
Divergence

- Some say yay, some say nay
  - How do you run this efficiently?



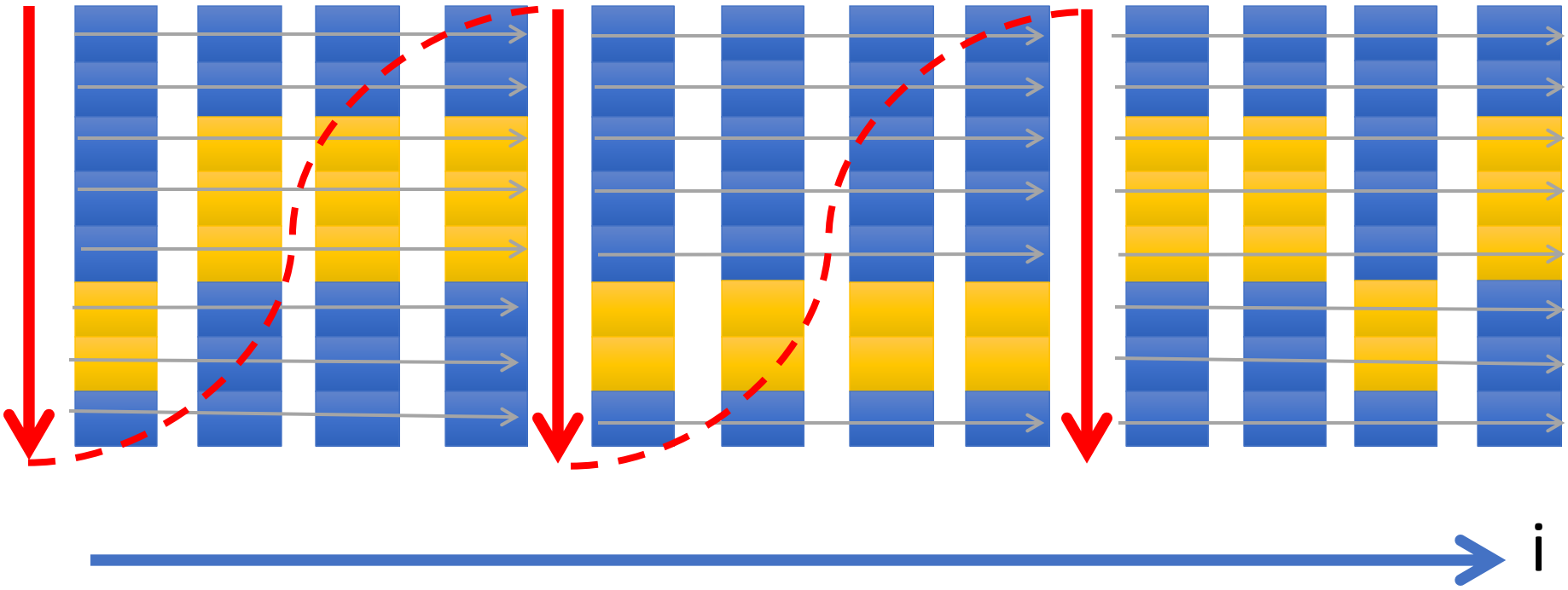
Divergence

- SIMD-like processing; lots of NOPs
  - 31% bubble (30/96)



Divergence

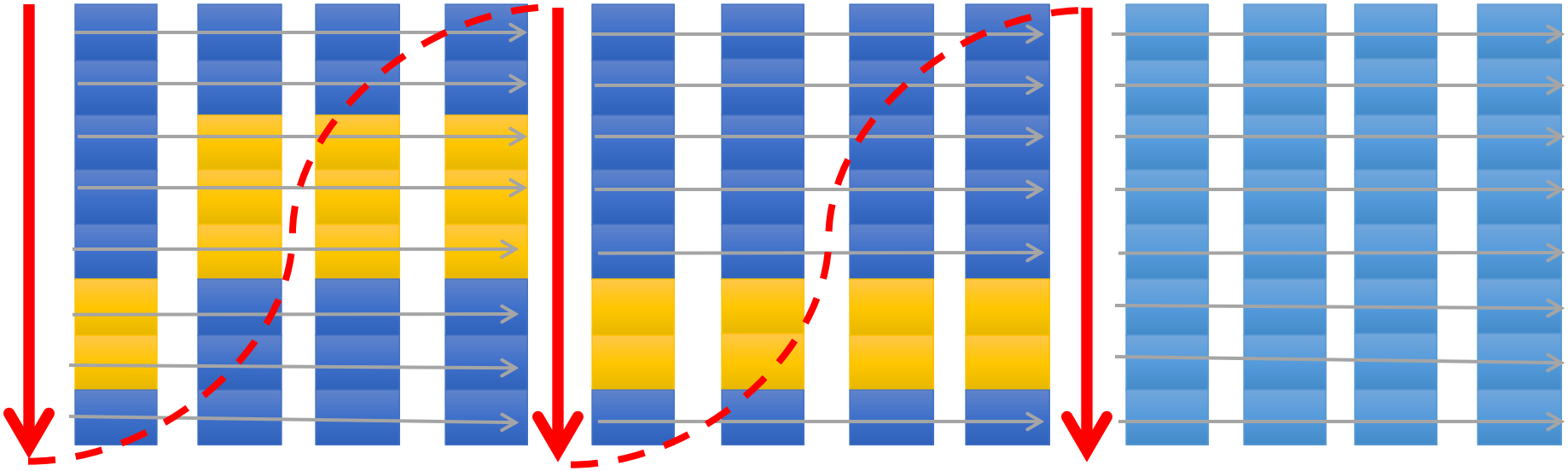
- SIMD-like processing with insufficient resources
  - 25% bubble (22/88) - if we can predict branch outcome



Multiple Programs – GPUs?

- SIMD-like processing with insufficient resources + multiple programs

time

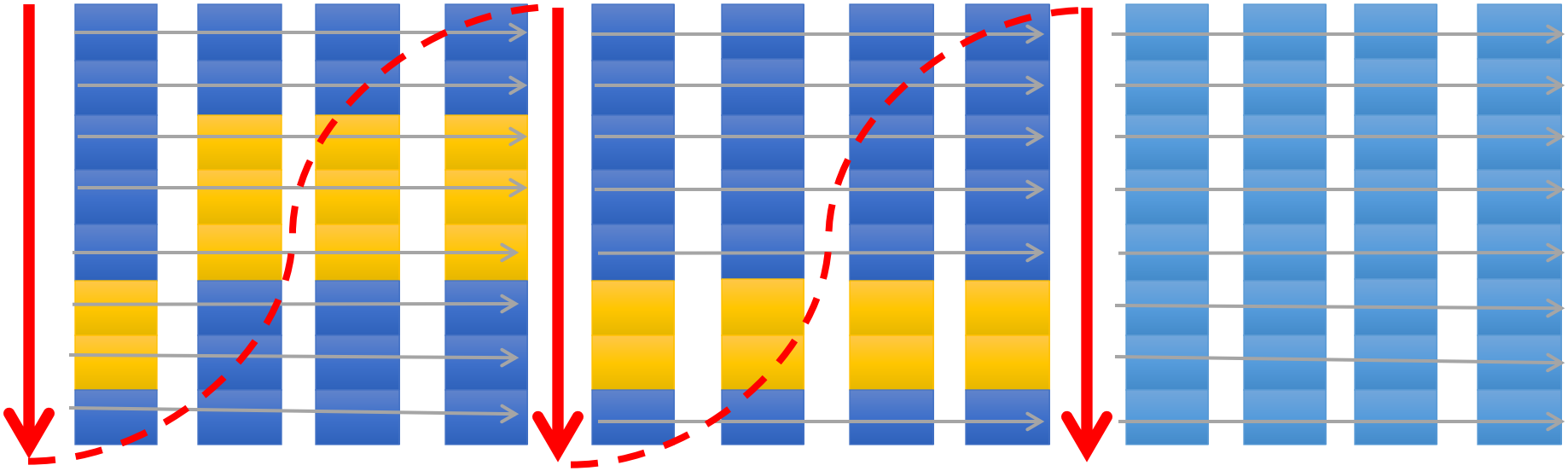


- Most people told you this is how GPUs work, but they'd be wrong

What Can Go Wrong?

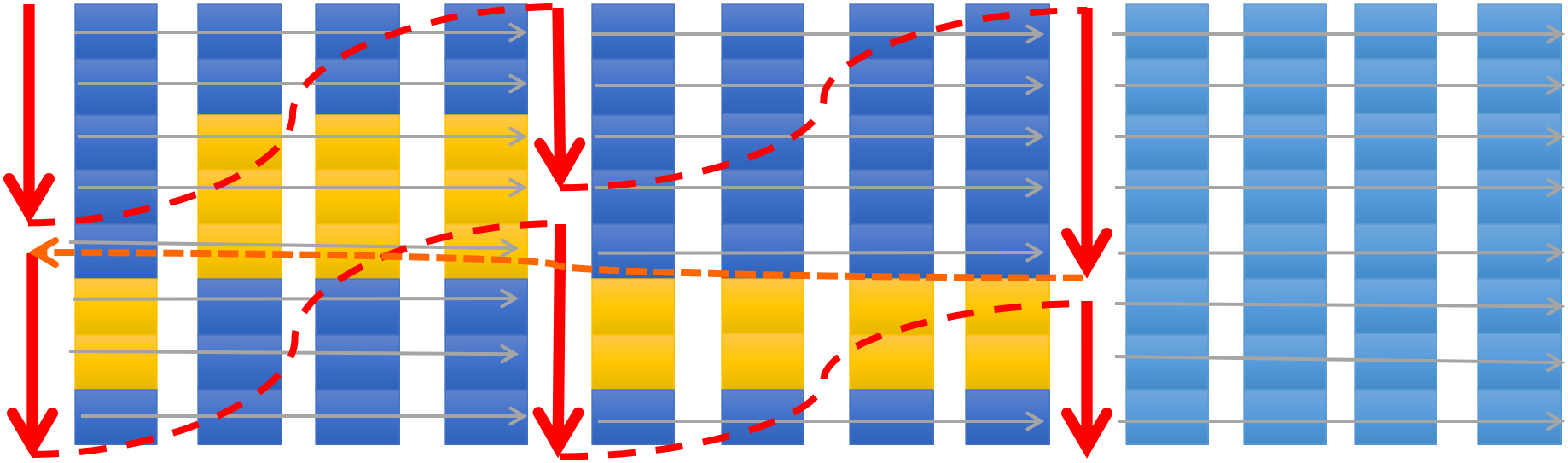
- Cache miss stall
- Can’t predict if branch should be taken

time



SIMT (T for Threads)

- NVIDIA Tesla GPU Architecture, 2006
- Jump to different program upon stall, bubble, etc



- Hint: in reality it is more like a priority queue

GPU Programming
#4 CUDA Thread Model

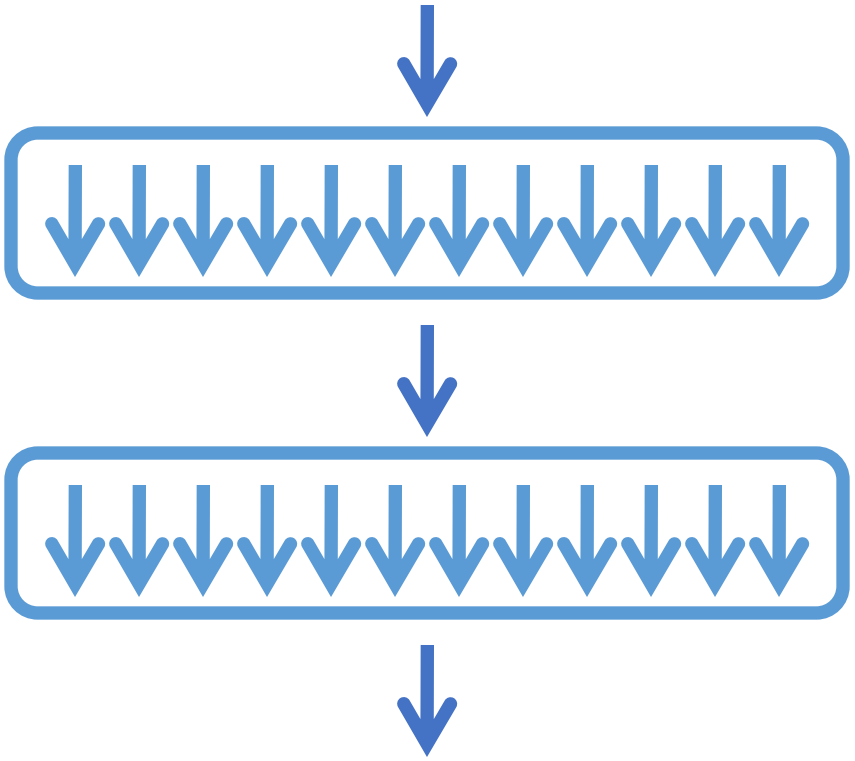
Wei-Chao Chen

Visiting Professor, National Taiwan University

weichao.chen@gmail.com

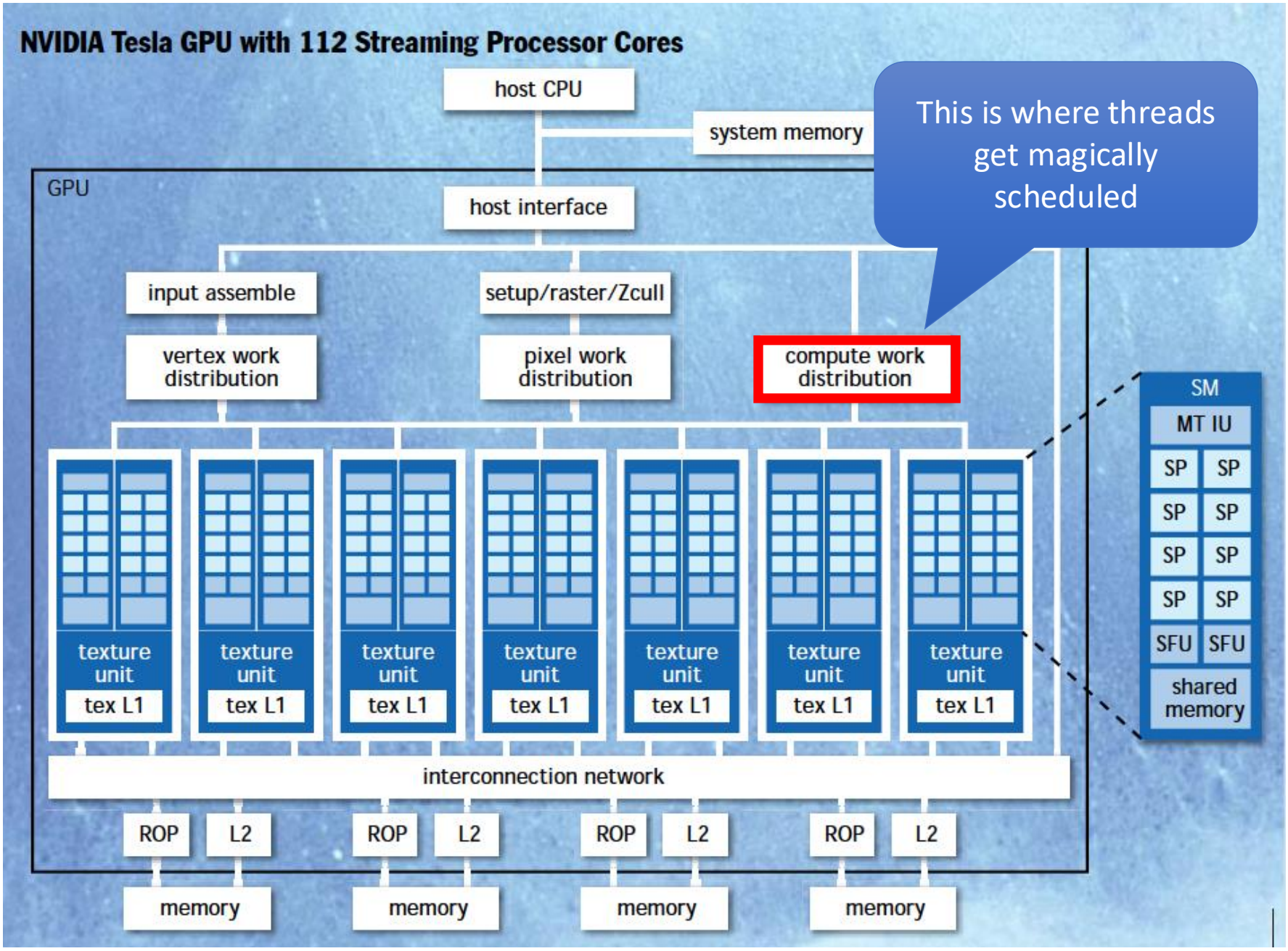
Thread Model Basics

- Serial code runs on the *host* (CPU)
- Parallel code runs on the *device* (GPU)

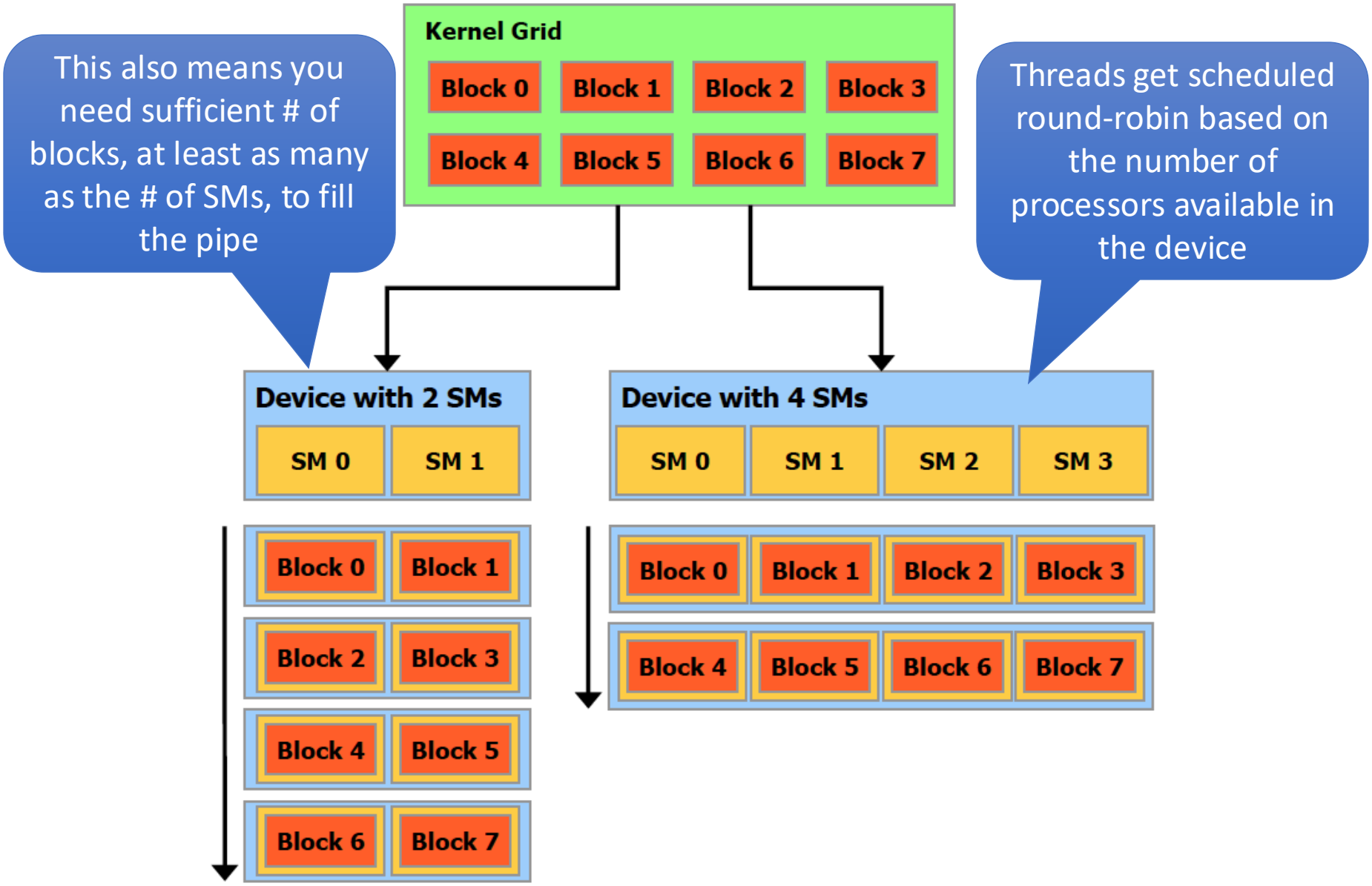


Thread Model Basics

- Two-level thread organization
  - *Grid* of *blocks(concurrent thread array, CTA)* of threads
- Serial *host* code responsible for launching parallel code to *device*
  - Kernel defined with *__global__ myKernel(args)*
  - Kernel launched with *myKernel<<nBlocks, nThreads>>(args)*
- Kernel scheduling done transparently for the user
  - Scheduling done by the hardware
  - Round-robin



John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. Queue, 6(2):40{53, 2008.



Source: NVIDIA

The Size of a Block?

- Rule 1: Fill the pipe
  - More threads makes GPU happy – they hide memory latency
  - E.g. SM of Tesla, Fermi, Kepler+ can hold 768, 1536, 2048 threads
  - BTW, the block size is 512, 1024, 1024

The Size of a Block?

- Rule 2: Don’t overfill
  - Threads compete for resources – registers, shared memory
  - You can’t anyways – the maximum size is limited; use cudaGetDeviceProperties() to obtain the maximum size of a thread block and grid supported by a particular device.
  - To control the fill, use:
    - *–maxrregcount* nvcc option,
    - CUDA API: cudaOccupancyMaxActiveBlocksPerMultiprocessor

Block IDs & Thread IDs

- Block IDs: 1D / 2D / 3D (Fermi+)
- Thread IDs: 1D / 2D / 3D (Fermi+)
- All __*global*__ and __*device*__ functions can access these variables:
  - `dim3 gridDim;` (grid dimensions)
  - `dim3 blockDim;` (block dimensions)
  - `dim3 blockIdx;` (current block index within the grid)
  - `dim3 threadIdx;` (current thread index within the block)

Block IDs & Thread IDs

- To specify the grid/block size, do this:

```
dim3 gridSize(nBlockX, nBlockY);
dim3 blockSize(nThreadX, nThreadY,
  nThreadZ);
myKernel<<<gridSize, blockSize>>>(args)
```

- In kernel function, a common pattern for computing thread index in 1D:

```
int idx = blockDim.x*blockIdx.x +
threadIdx.x;
```

Block IDs & Thread IDs

- In kernel function, a common pattern for computing thread index in 2D:

```
int iy = blockDim.y * blockIdx.y + threadIdx.y;
int ix = blockDim.x * blockIdx.x + threadIdx.x;
int idx = iy * w + ix;
```

(follow similar patterns for 3D blocks)

Functional Declarations

Qualifier	Executed on…	Callable from…
__global__	device	host
__device__	device	device
(__host__)	host	host

- __host__ is implicit
  - Default is running on host if no qualifier is given
- __host__ can be used in conjunction with __device__
  - Code will be compiled for both CPU and GPU

Limitations on Functional Qualifiers

Qualifier	__global__	__device__	__host__
Recursion	Dynamic parallelism	Yes *1	Yes
Static Variables	No	No	Yes
Variable Number of Arguments	No	No	Yes
Return value	Void	Any	Any
Asynchronous	Yes	No	By OS API

*1 Require large stack and __device__ functions are usually inlined if possible.

Variable Qualifiers in Kernel

Qualifier	Physical Location	Allocation	Accessibility	Lifetime
__device__	device (DRAM)	cudaMalloc()	All threads	Application
__shared__	shader (SRAM)	execution / compile time	Threads in the same block	Kernel Execution
__costant__	device (DRAM)	Compile time (per compile unit)	All threads	Application
(Unqualified)	shader registers or device stack *1	Execution / PTX code	Single thread	Thread

* Arrays in shader are stored in stack only if compiler cannot allocate it statically.

Shared Memory Allocation

- Compile time allocation

```
__global__ void myKernel(…) {
  …
  __shared__ float sData[16];
  …
}
void main(…) {
  …
  myKernel<<<gridSize,blockSize>>>(…)
  …
}
```

Shared Memory Allocation

- Execution-time allocation

```
extern __shared__ char array[];
__global__ void myKernel(…) {
  …
  float *sData[] = (float*) array;
  …
}
void main(…) {
  …
  numBytes = 16*sizeof(float)
  myKernel<<<gridSize,blockSize,numBytes>>>
  (…)
  …
}
```

Shared Memory Allocation

- Execution-time allocation (more than one shared memory)

```
extern __shared__ char array[];
__global__ void myKernel(…) {
  …
  float *f1Data[] = (float*) array;
  float4 *f4Data[] = (float4*) (array+ofs1);
  …
}
void main(…) {
  …
  numBytes = 7*sizeof(float)
  // We usually use __alignof(float4) to obtain 16
  ofs1 = ((numBytes-1)/16+1)*16
  numBytes += 12*sizeof(float4)
  myKernel<<<gridSize,blockSize,numBytes>>>(…)
  …
}
```

Built-in Types & Registers

- These can be used for both CPU and GPU
- Up to 4 dimensions, no more than 16B
  - R,G,B,A, historical reasons
  - Types of size 3 usually have poor performance

`[u]char[1…4]`

`[u]short[1…4]`

`[u]int[1…4]`

`[u]long[1…4]`

`float[1…4]`

`double[1...2], longlong[1…2]`

On Vector Processing

- Classic SIMD processing model
- Perfect for parallelizing a for-loop

```
for i=0:15,
  a[i] = b[i] * c[i] + d[i]
end
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	…	a[15]
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	…	b[15]
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	…	c[15]
d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	…	d[15]

On Vector Processing

```
va = vb × vc + vd
```

➔ `vt = vb × vc`

T=0

								…	
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	…	b[15]
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	…	c[15]
d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	…	d[15]
P0	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**…**	**P15**

Processors

On Vector Processing

```
va = vb × vc + vd
```

➔ `vt = vb × vc`

T=1

➔

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	…	t[15]
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	…	b[15]
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	…	c[15]
d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	…	d[15]
P0	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**…**	**P15**

Processors

On Vector Processing

- Pros:
  - Easy to understand and implement (hardware)
  - Simple programming model (software)
- Cons:
  - Expensive
    - Lots of memory reads/writes
    - Lots of registers
  - Not branching friendly
    - Synchronous execution within a vector
  - Pipeline bubbles
    - e.g., `for i=0:19`

SIMT: Single Instruction, Multiple Threads

- SIMD in spirit
  - A mix between SIMD and SMT (Simultaneous Multithreading)
- Pipelining differences – Vertical v.s. Horizontal

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	…	a[15]
	b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	…	b[15]
	c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	…	c[15]
	d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	…	d[15]
Processors	**P0**	**P0**	**P0**	**P0**	**P1**	**P1**	**P1**	**P1**	**…**	**P3**
Threads	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**…**	**15**

SIMT: Single Instruction, Multiple Threads

- Possibility I: Purely Vertical

T=0

								…	
b[0]	b[1]	b[2]	b[3]	**b[4]**	b[5]	b[6]	b[7]	…	b[15]
c[0]	c[1]	c[2]	c[3]	**c[4]**	c[5]	c[6]	c[7]	…	c[15]
d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	…	d[15]
P0				**P1**				**…**	

SIMT: Single Instruction, Multiple Threads

- Possibility I: Purely Vertical

T=1

t[0]				**t[4]**				…	
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	…	b[15]
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	…	c[15]
d[0]	d[1]	d[2]	d[3]	**d[4]**	d[5]	d[6]	d[7]	…	d[15]
P0				**P1**				**…**	

SIMT: Single Instruction, Multiple Threads

- Possibility I: Purely Vertical

T=2

a[0]				**a[4]**				**…**	
b[0]	**b[1]**	b[2]	b[3]	b[4]	**b[5]**	b[6]	b[7]	…	b[15]
c[0]	**c[1]**	c[2]	c[3]	c[4]	**c[5]**	c[6]	c[7]	…	c[15]
d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	…	d[15]
	P0				**P1**			**…**	

SIMT: Single Instruction, Multiple Threads

- Possibility I: Purely Vertical

T=3

a[0]	**t[1]**			**a[4]**	**t[5]**			**…**	
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	…	b[15]
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	…	c[15]
d[0]	**d[1]**	d[2]	d[3]	d[4]	**d[5]**	d[6]	d[7]	…	d[15]
	P0				**P1**			**…**	

SIMT: Single Instruction, Multiple Threads

- Possibility I: Purely Vertical

T=4

a[0]	a[1]			a[4]	a[5]			…	
b[0]	b[1]	**b[2]**	b[3]	b[4]	b[5]	**b[6]**	b[7]	…	b[15]
c[0]	c[1]	**c[2]**	c[3]	c[4]	c[5]	**c[6]**	c[7]	…	c[15]
d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	…	d[15]
		P0				**P1**		**…**	

SIMT: Single Instruction, Multiple Threads

- Purely Vertical Threads
  - Complete one thread before moving to the next
  - Multi-Core like
  - Small register footprint
    - Only 1 copy of t is required.
  - Not cache miss / latency friendly

SIMT: Single Instruction, Multiple Threads

- Possibility II: Group Horizontal

T=0

								…	
b[0]	b[1]	b[2]	b[3]	**b[4]**	b[5]	b[6]	b[7]	…	b[15]
c[0]	c[1]	c[2]	c[3]	**c[4]**	c[5]	c[6]	c[7]	…	c[15]
d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	…	d[15]
P0				**P1**				**…**	

SIMT: Single Instruction, Multiple Threads

- Possibility II: Group Horizontal

T=1

t[0]				**t[4]**				**…**	
b[0]	**b[1]**	b[2]	b[3]	b[4]	**b[5]**	b[6]	b[7]	…	b[15]
c[0]	**c[1]**	c[2]	c[3]	c[4]	**c[5]**	c[6]	c[7]	…	c[15]
d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	…	d[15]
	P0				**P1**			**…**	

SIMT: Single Instruction, Multiple Threads

- Possibility II: Group Horizontal

T=2

t[0]	t[1]			t[4]	t[5]			…	
b[0]	b[1]	**b[2]**	b[3]	b[4]	b[5]	**b[6]**	b[7]	…	b[15]
c[0]	c[1]	**c[2]**	c[3]	c[4]	c[5]	**b[6]**	c[7]	…	c[15]
d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	…	d[15]
		P0				**P1**		**…**	

SIMT: Single Instruction, Multiple Threads

- Possibility II: Group Horizontal

T=3

t[0]	t[1]	t[2]		t[4]	t[5]	t[6]		…	
b[0]	b[1]	b[2]	**b[3]**	b[4]	b[5]	b[6]	**b[7]**	…	**b[0]**
b[0]	c[1]	c[2]	**c[3]**	c[4]	c[5]	c[6]	**c[7]**	…	**c[0]**
d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	…	d[15]
			P0				**P1**	**…**	**P3**

SIMT: Single Instruction, Multiple Threads

- Possibility II: Group Horizontal

T=4

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	…	t[15]
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	…	**b[15]**
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	…	**c[15]**
d[0]	d[1]	d[2]	d[3]	**d[4]**	d[5]	d[6]	d[7]	…	d[15]
P0				**P1**				**…**	

SIMT: Single Instruction, Multiple Threads

- Possibility II: Group Horizontal

T=5

a[0]	**t[1]**	**t[2]**	**t[3]**	**a[4]**	**t[5]**	**t[6]**	**t[7]**	**…**	**t[15]**
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	…	**b[15]**
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	…	**c[15]**
d[0]	**d[1]**	d[2]	d[3]	d[4]	**d[5]**	d[6]	d[7]	…	d[15]
	P0				**P1**			**…**	

SIMT: Single Instruction, Multiple Threads

- Possibility II: Group Horizontal

T=6

a[0]	a[1]	t[2]	t[3]	a[4]	a[5]	t[6]	t[7]	…	t[15]
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	…	**b[15]**
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	…	**c[15]**
d[0]	d[1]	**d[2]**	d[3]	d[4]	d[5]	**d[6]**	d[7]	…	d[15]
		P0				**P1**		**…**	

SIMT: Single Instruction, Multiple Threads

- Possibility II: Group Horizontal

T=7

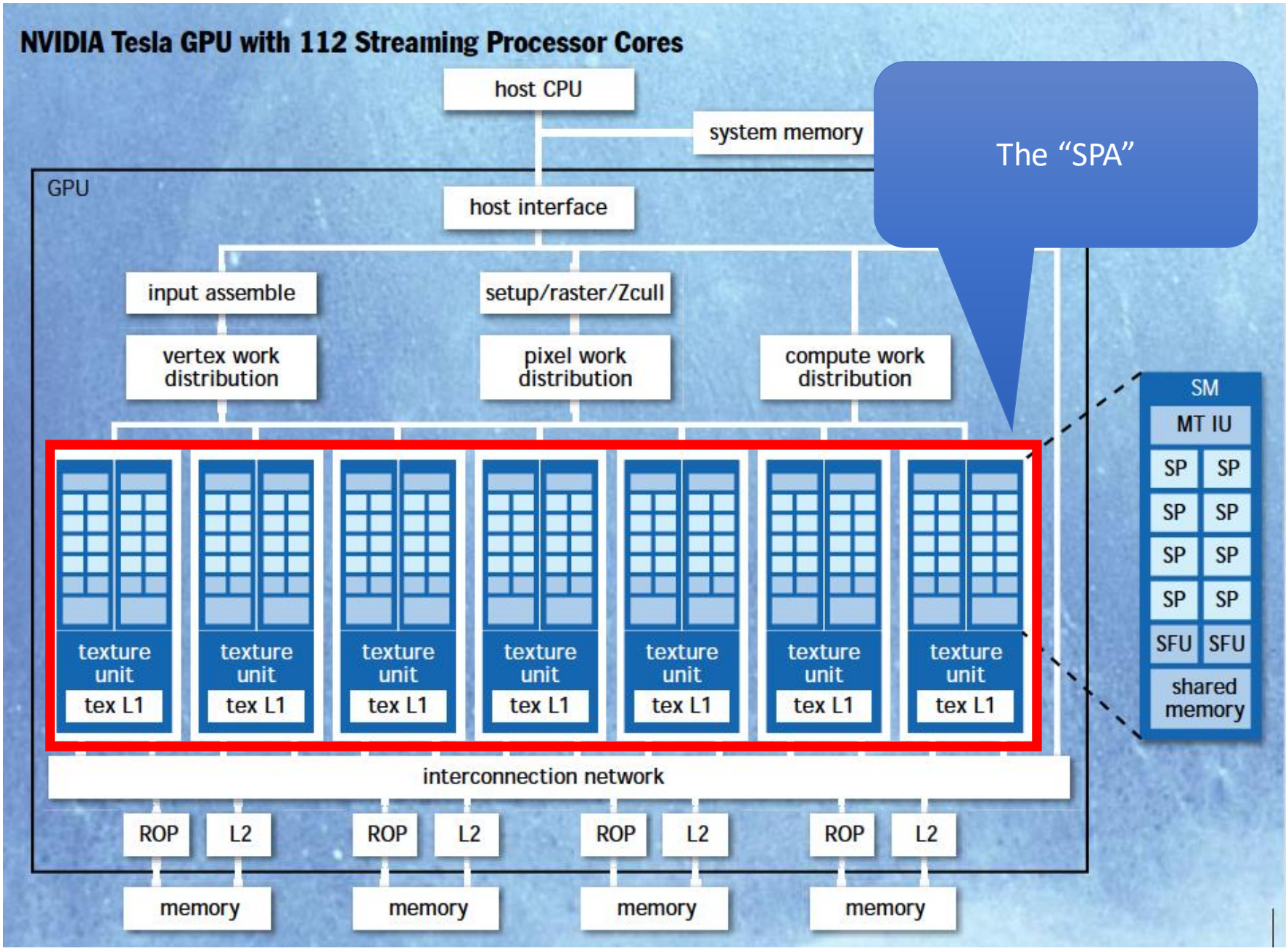
a[0]	**a[1]**	**a[2]**	**t[3]**	**a[4]**	**a[5]**	**a[6]**	**t[7]**	**…**	**t[15]**
b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	…	**b[15]**
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	…	**c[15]**
d[0]	d[1]	d[2]	**d[3]**	d[4]	d[5]	d[6]	**d[7]**	…	d[15]
			P0				**P1**	**…**	

SIMT: Single Instruction, Multiple Threads

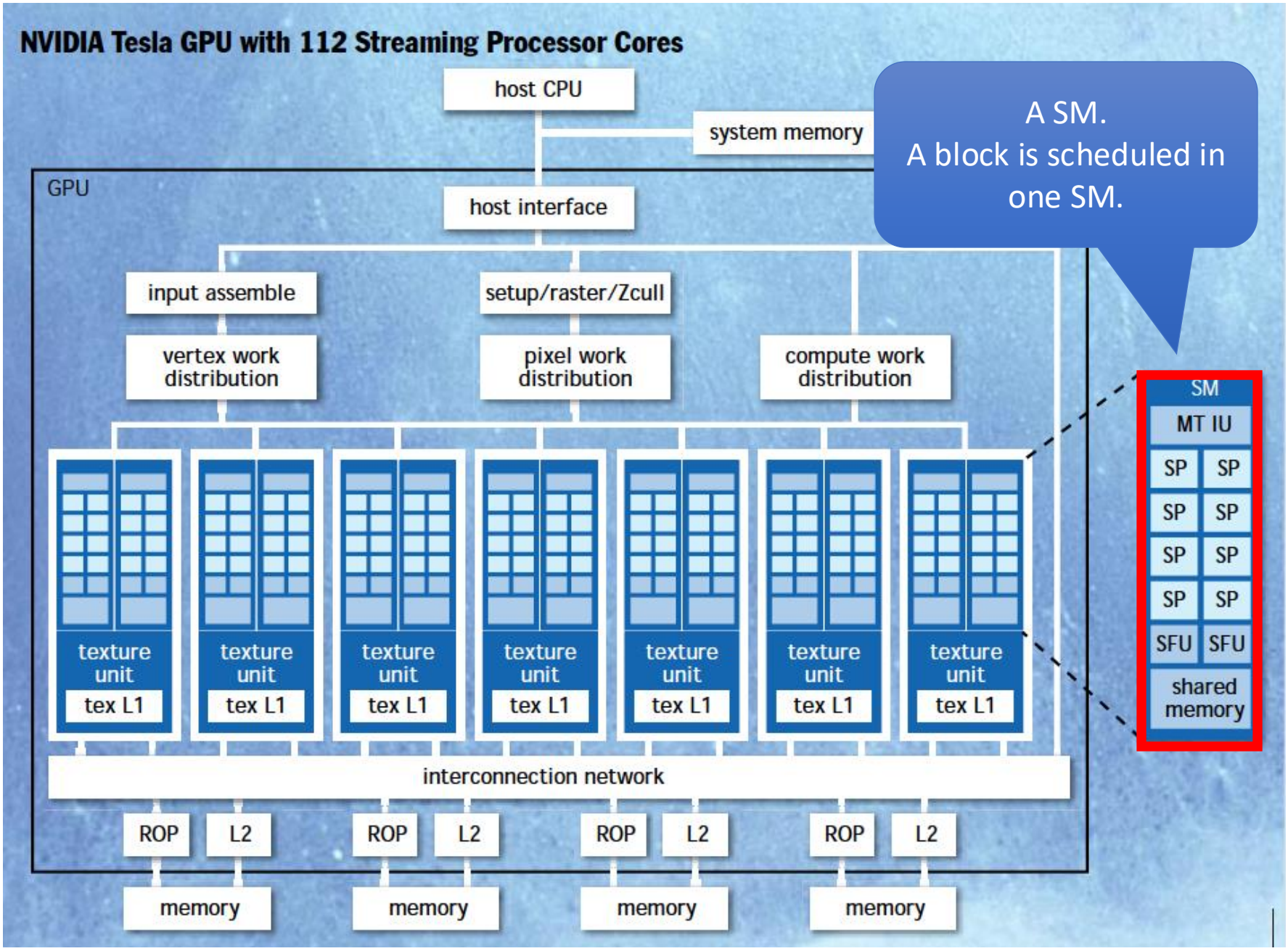
- Group Horizontal Threads
  - Complete the same instruction across different threads, before moving to the next instruction
  - Vector-processing-like
  - More cache miss / latency tolerance
  - Larger temporary register footprint

SIMT: Single Instruction, Multiple Threads

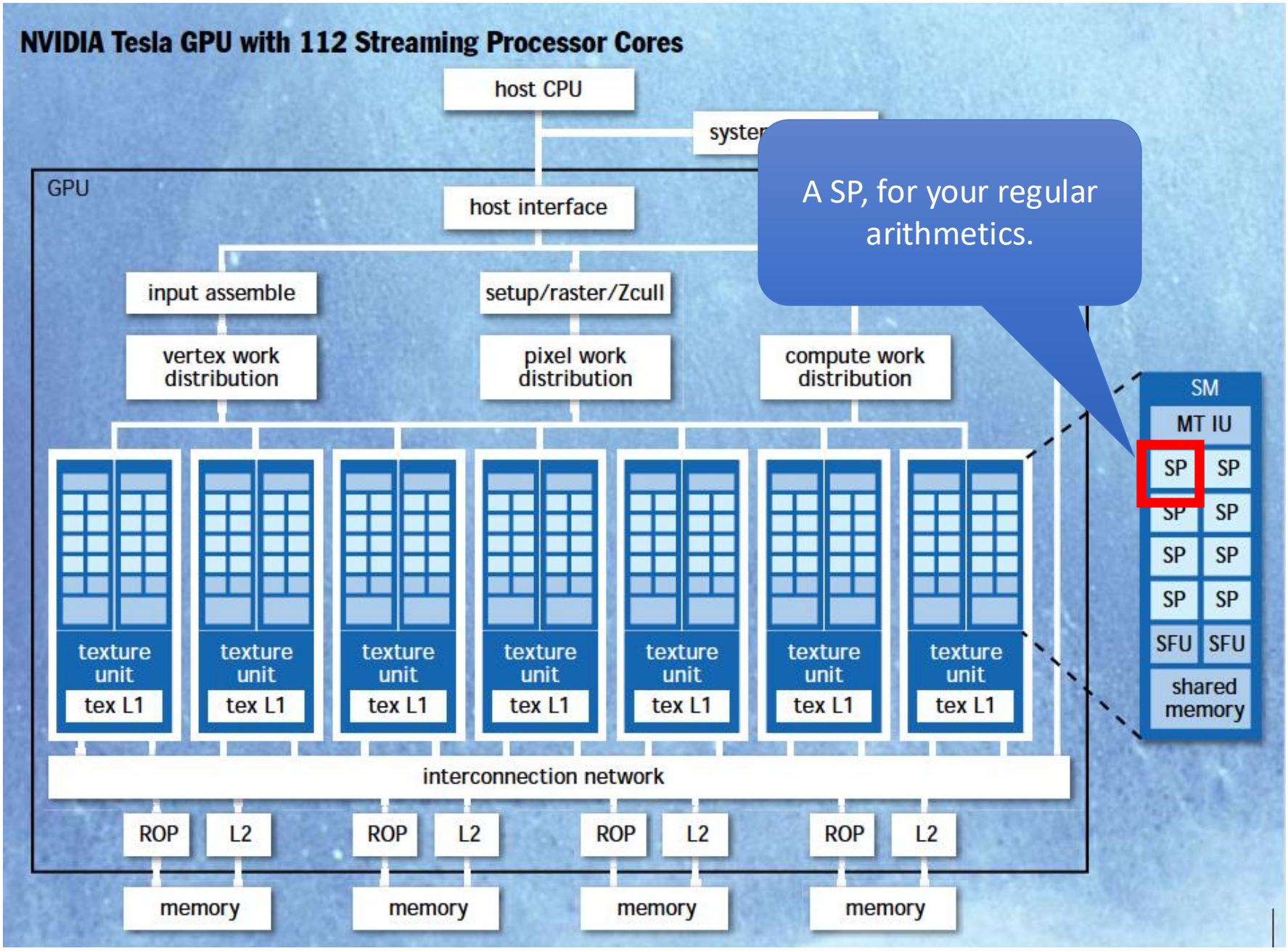
- Vertical v.s. Horizontal
  - No definite answers
  - Use more Horizontal threads when
    - Memory latency / cache miss is high
  - Use more Vertical threads when
    - Register memory circuits are expensive
    - Branching divergence occurs often
- In reality, we mix between Vertical / Horizontal
  - Remember warps?



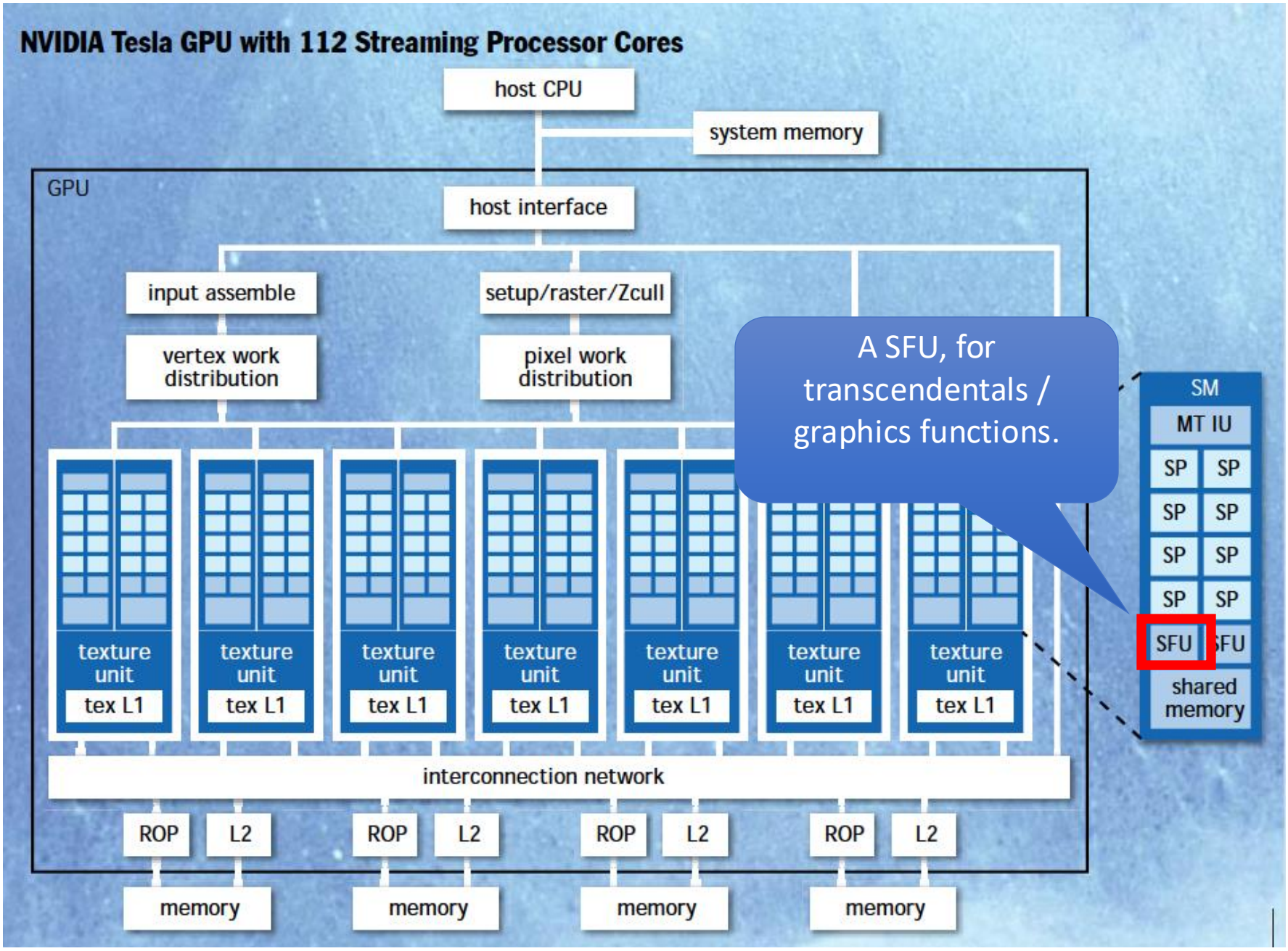
John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. Queue, 6(2):40{53, 2008.



John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. Queue, 6(2):40{53, 2008.



John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. Queue, 6(2):40{53, 2008.

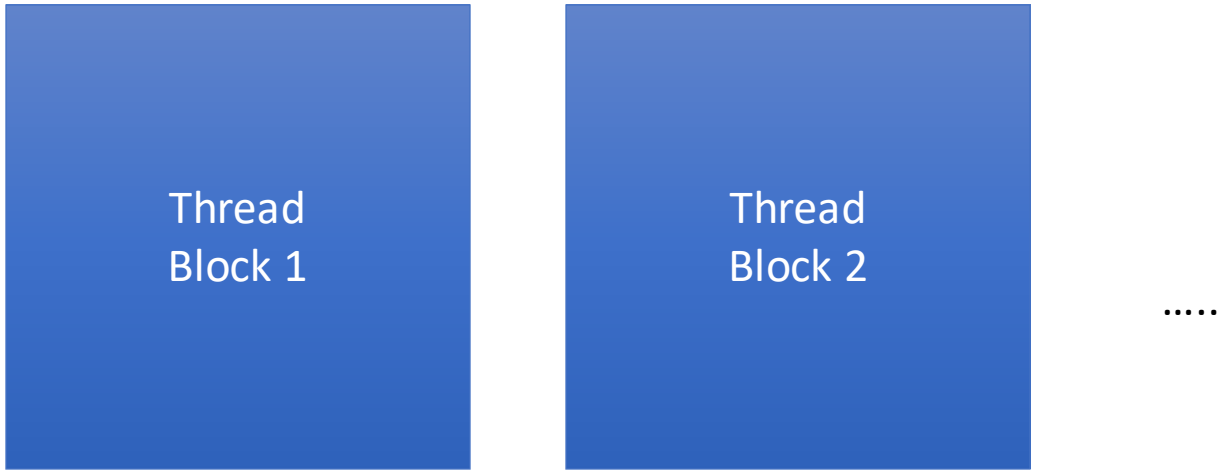


John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. Queue, 6(2):40{53, 2008.

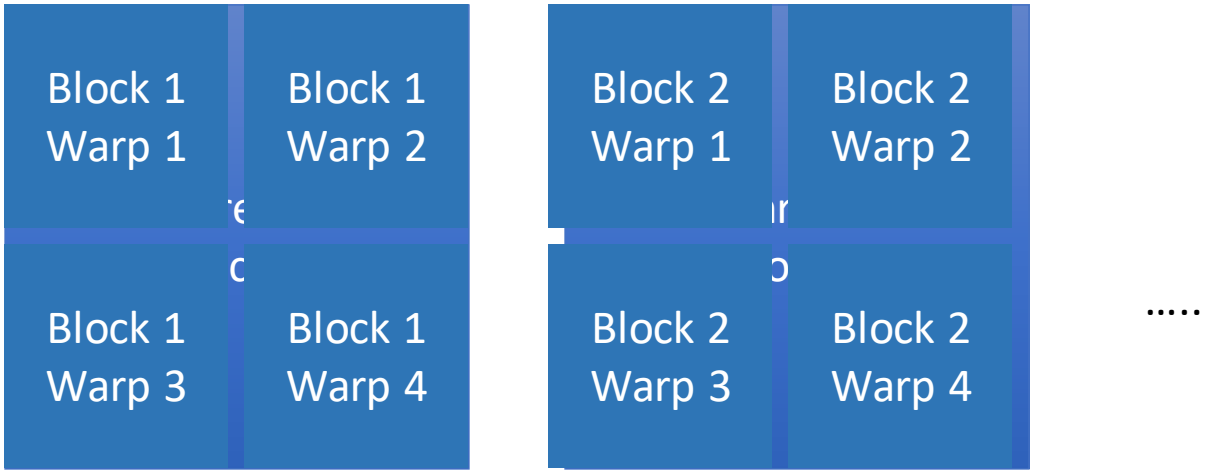
Warps

- Unrelated to CUDA design, this is purely hardware
  - Designed to support effective divergence and latency hiding
  - “SIMT” scheduling
- A block is divided into several warps
  - In G80, a warp == 32 threads
- Each warp get executed without divergence
  - If…then…else statement, if a branch is taken for one thread in the warp, all other threads will execute that path
- A SM can execute only ONE warp at a time
  - __syncthreads() not necessary for threads within the same warp

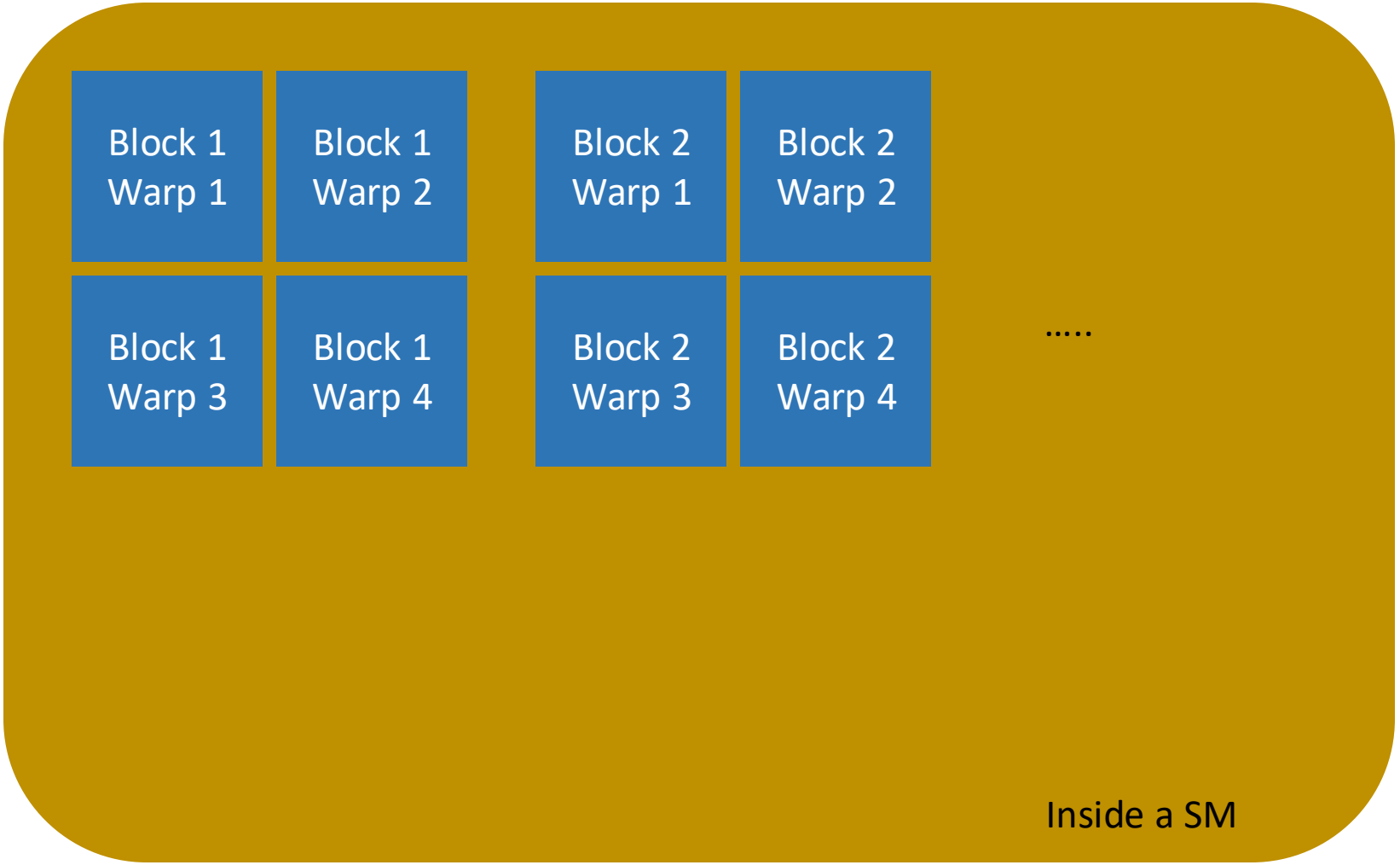
Warps Scheduling



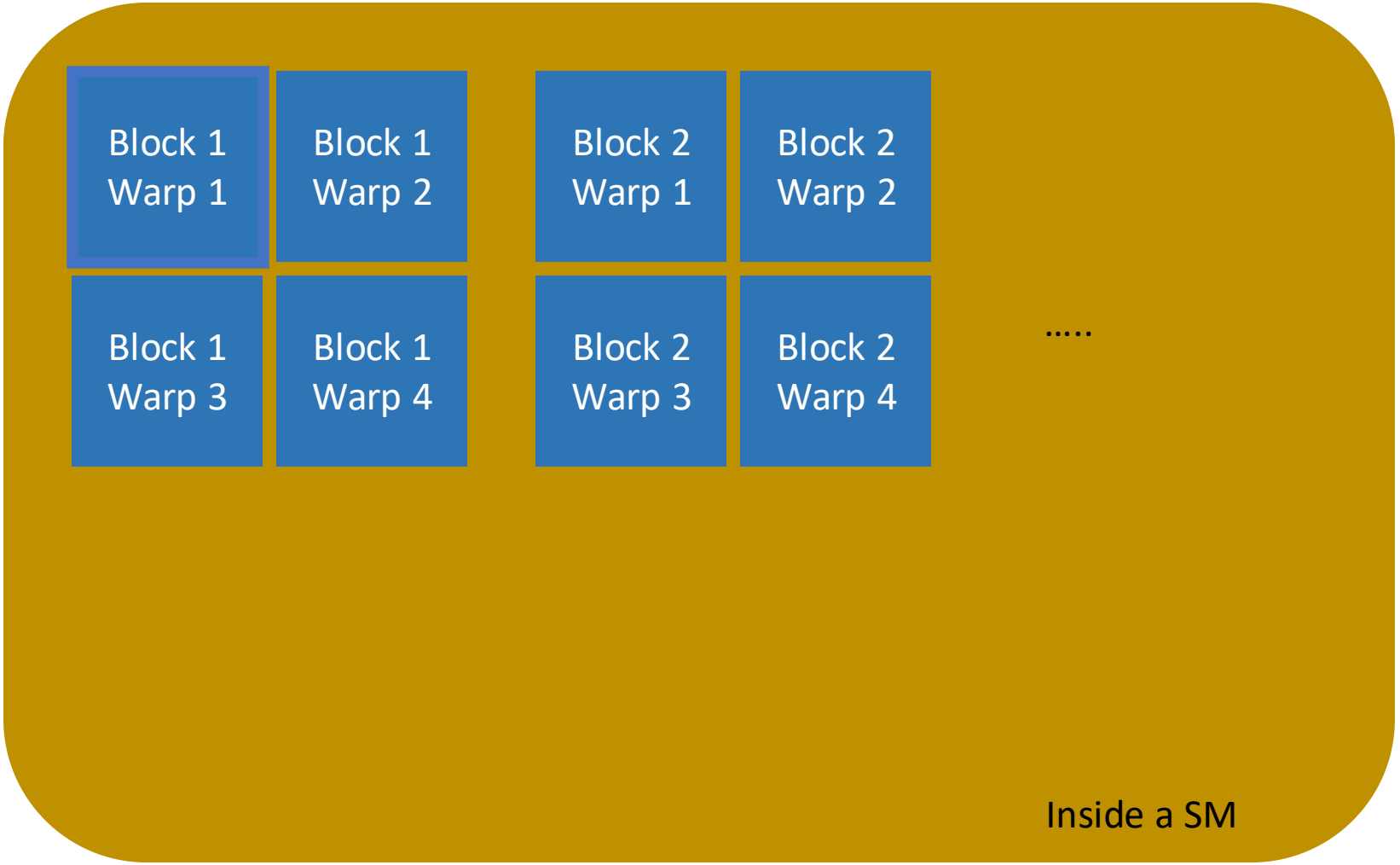
Warps Scheduling



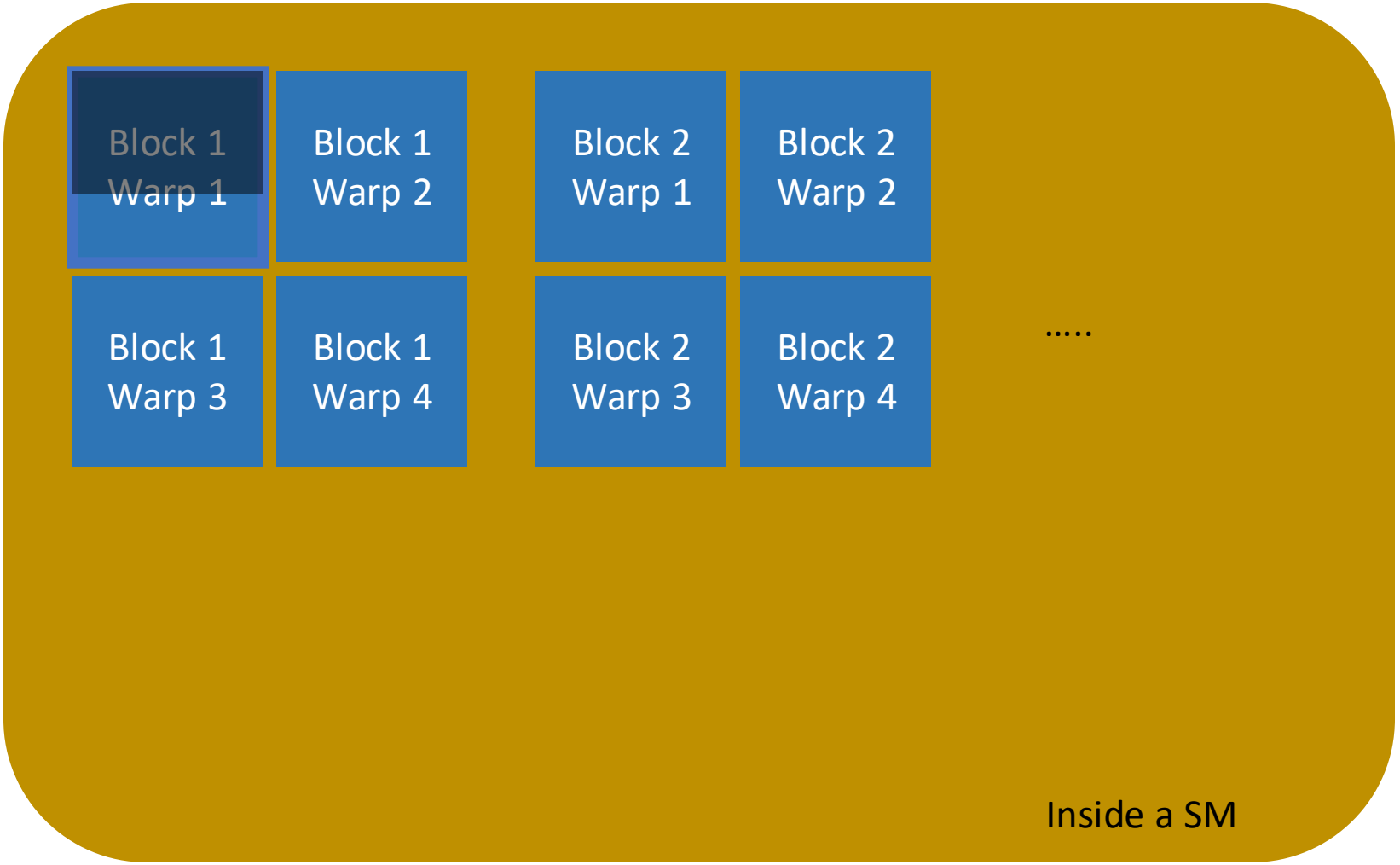
Warps Scheduling



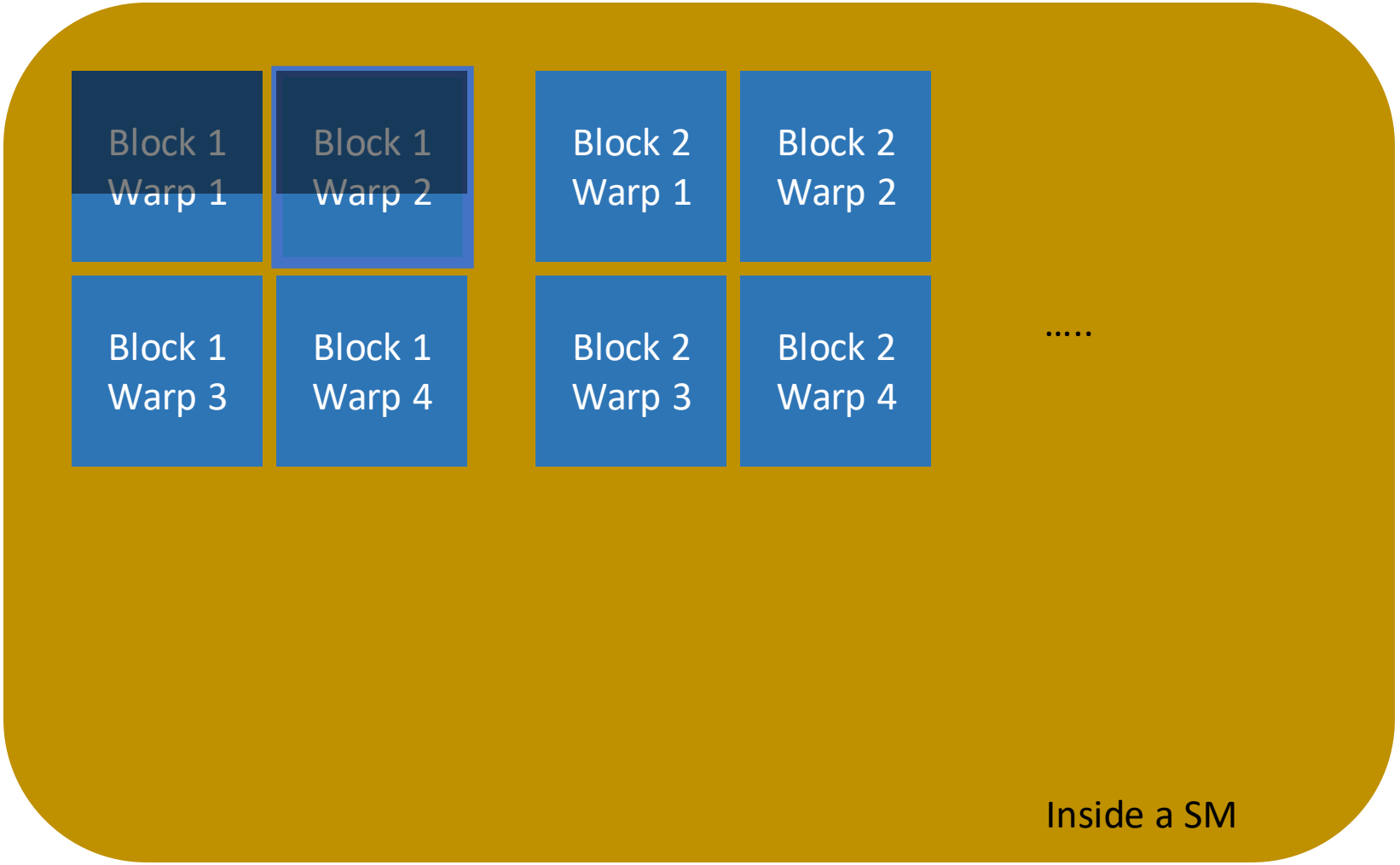
Warps Scheduling



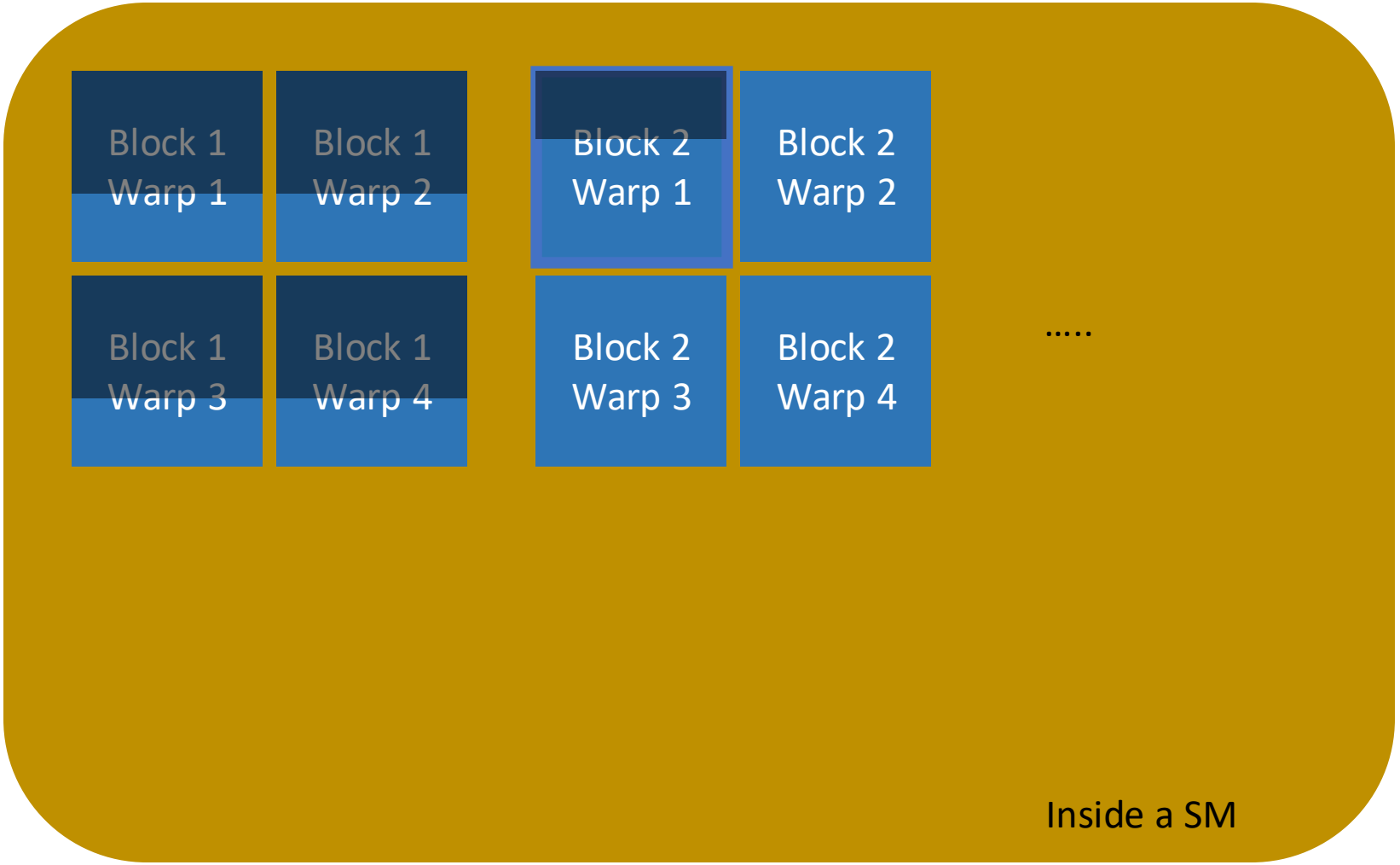
Warps Scheduling



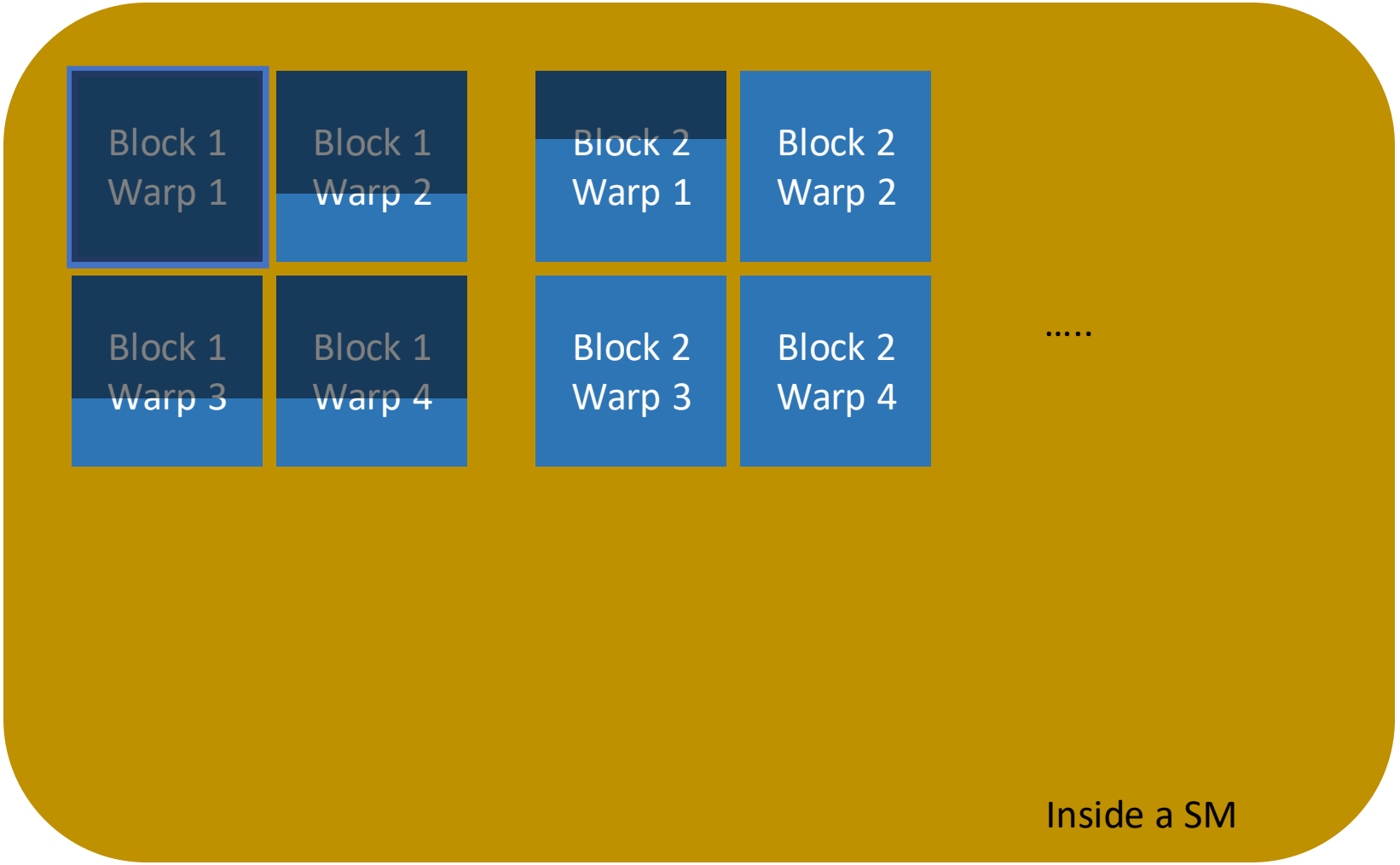
Warps Scheduling



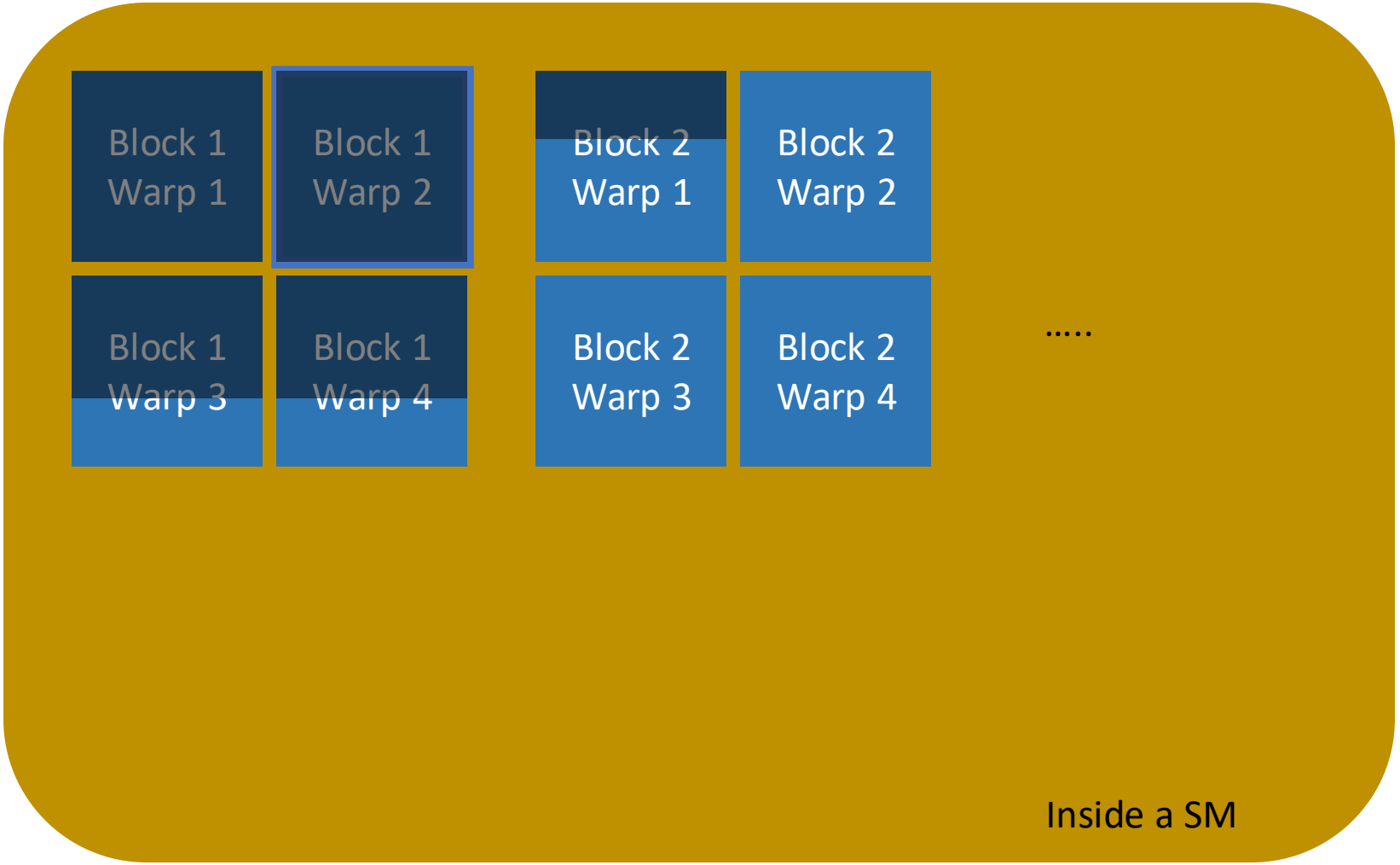
Warps Scheduling



Warps Scheduling



Warps Scheduling

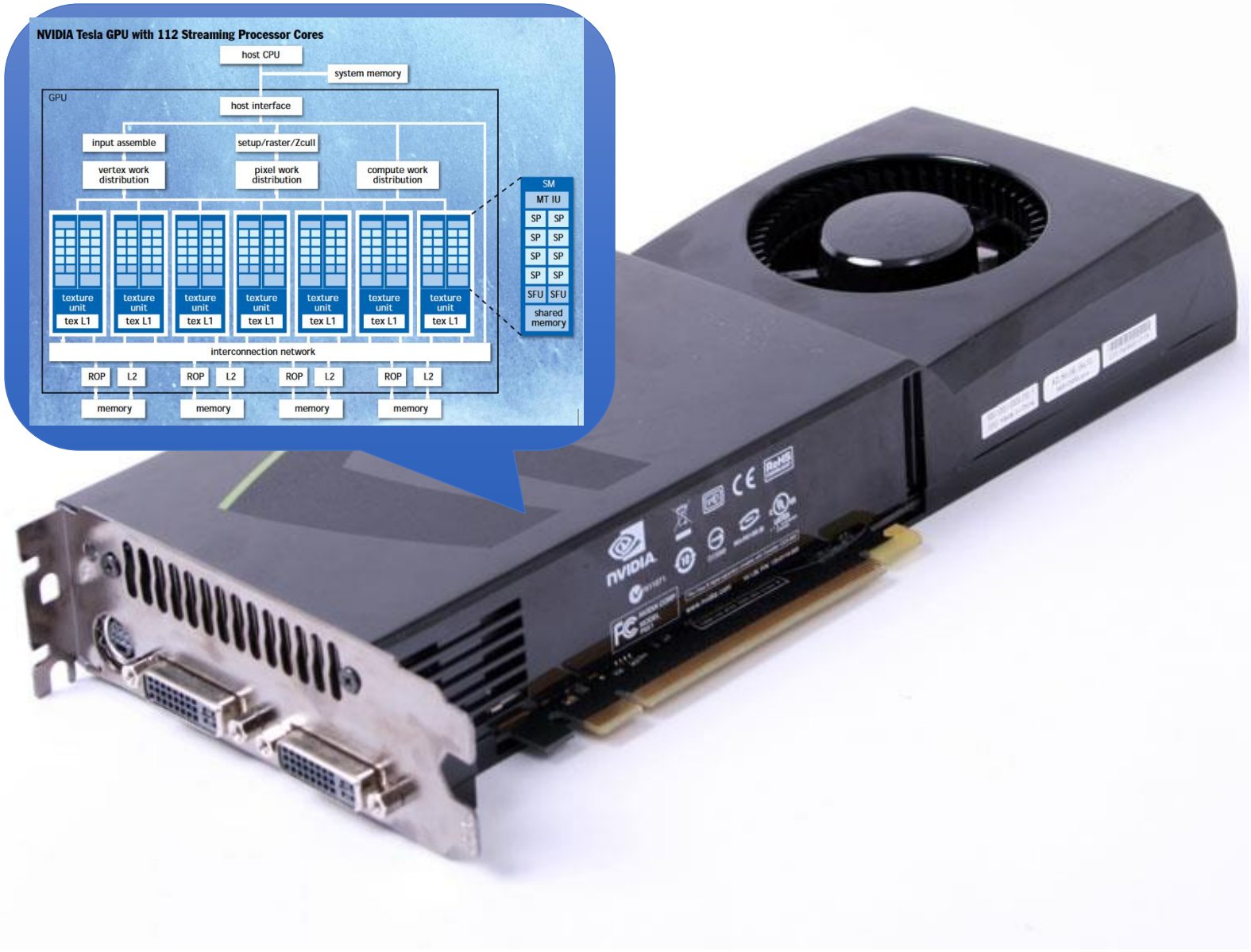


GPU Programming #5 CUDA Memory

Wei-Chao Chen

Visiting Professor, National Taiwan University

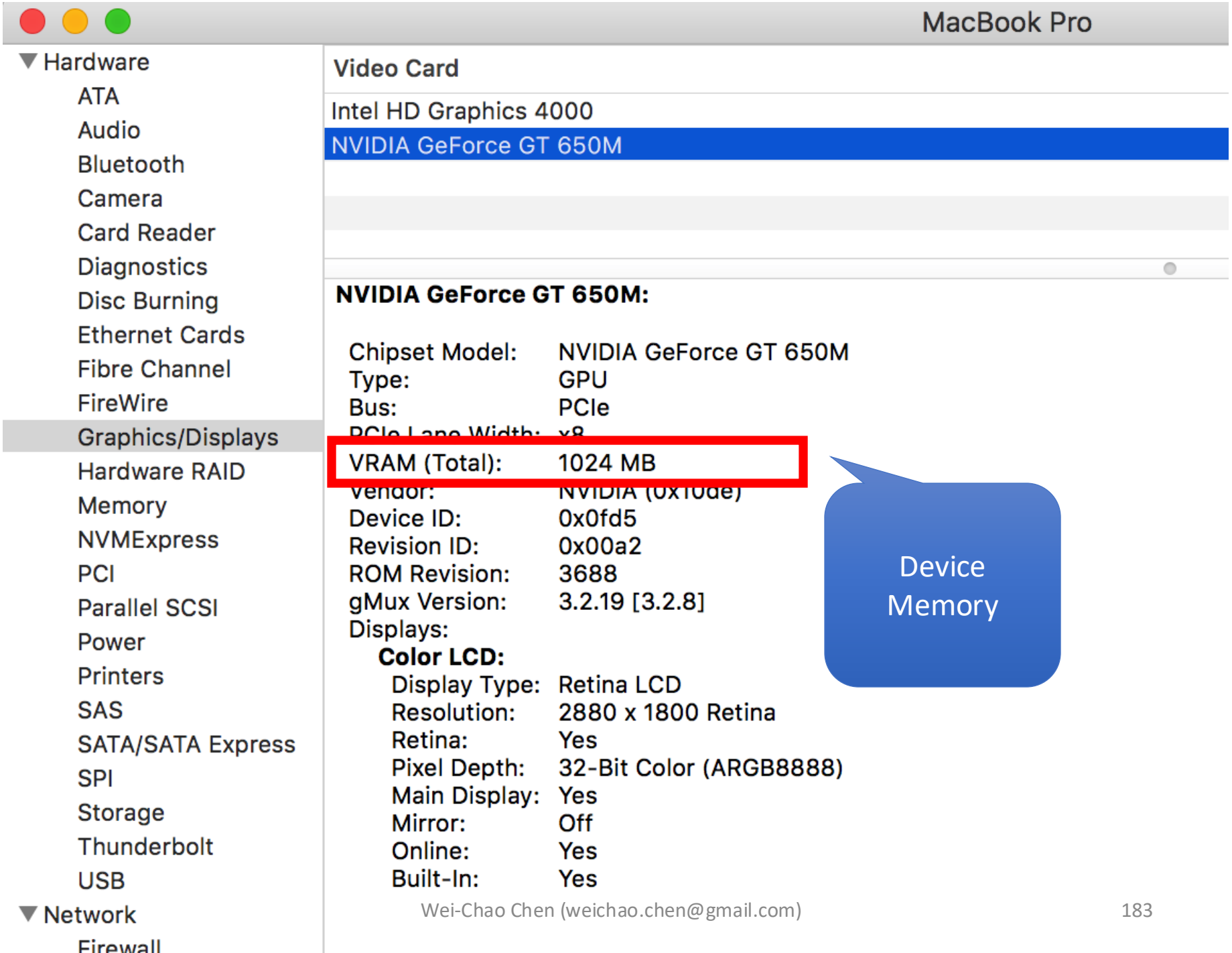
weichao.chen@gmail.com

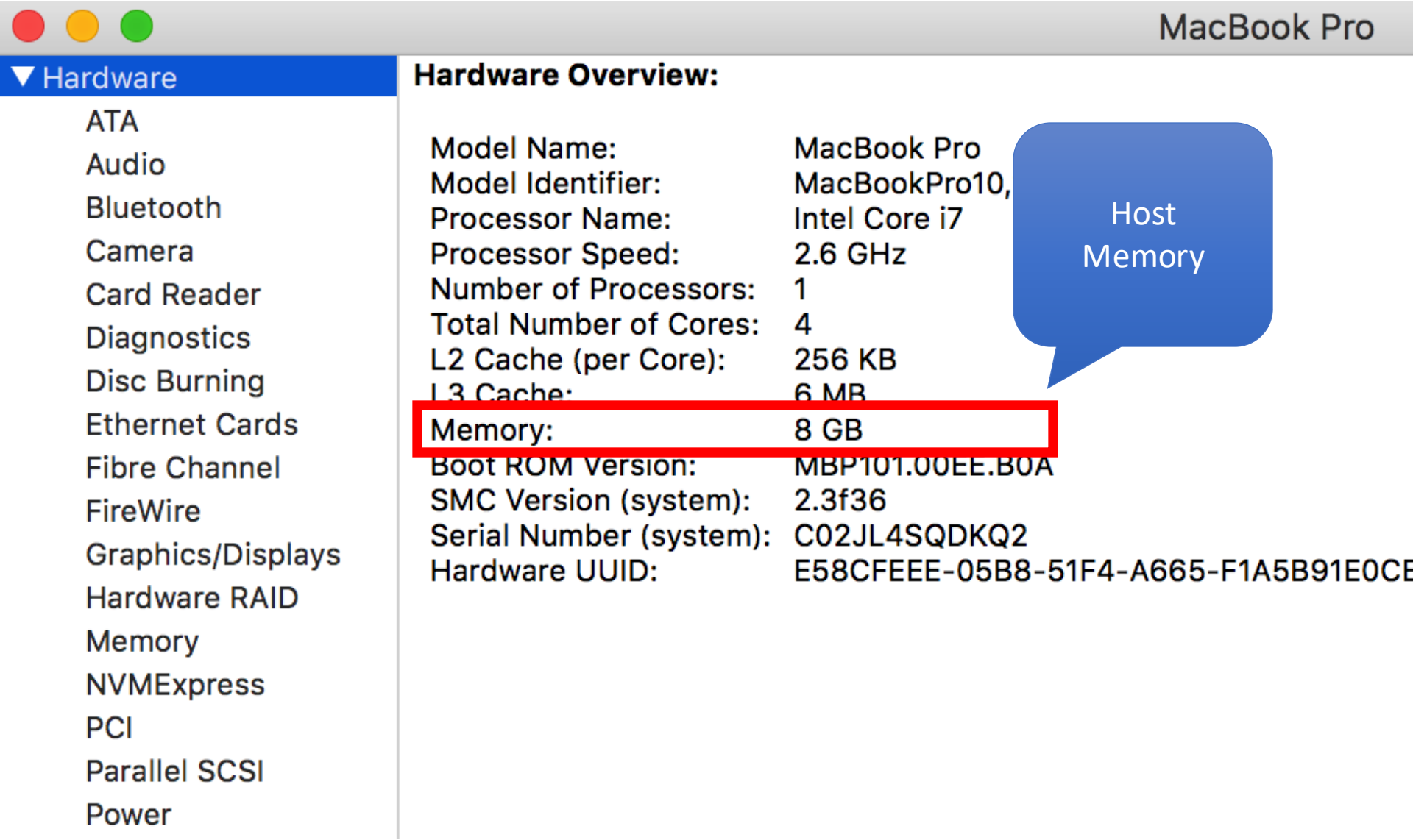


NVIDIA GTX 280

Memory Hierarchy

- Host Memory
  - = DRAM, addressable by your CPU
- Device Memory
  - = Graphics Card Memory
  - = Video Memory
  - = Global Memory in CUDA terminology
  - = Where your pixels reside
- Inside GPU
  - Shared Memory
  - Registers
  - Tons of Cache


MacBook Pro
Hardware
ATA
Audio
Bluetooth
Camera
Card Reader
Diagnostics
Disc Burning
Ethernet Cards
Fibre Channel
FireWire
Graphics/Displays
Hardware RAID
Memory
NVMExpress
PCI
Parallel SCSI
Power
Printers
SAS
SATA/SATA Express
SPI
Storage
Thunderbolt
USB
Network
Firewall
Video Card
Intel HD Graphics 4000
NVIDIA GeForce GT 650M
NVIDIA GeForce GT 650M:
Chipset Model: NVIDIA GeForce GT 650M
Type: GPU
Bus: PCIe
VRAM (Total): 1024 MB
Device ID: 0x0fd5
Revision ID: 0x00a2
ROM Revision: 3688
gMux Version: 3.2.19 [3.2.8]
Displays:
Color LCD:
Display Type: Retina LCD
Resolution: 2880 x 1800 Retina
Retina: Yes
Pixel Depth: 32-Bit Color (ARGB8888)
Main Display: Yes
Mirror: Off
Online: Yes
Built-In: Yes
Device Memory

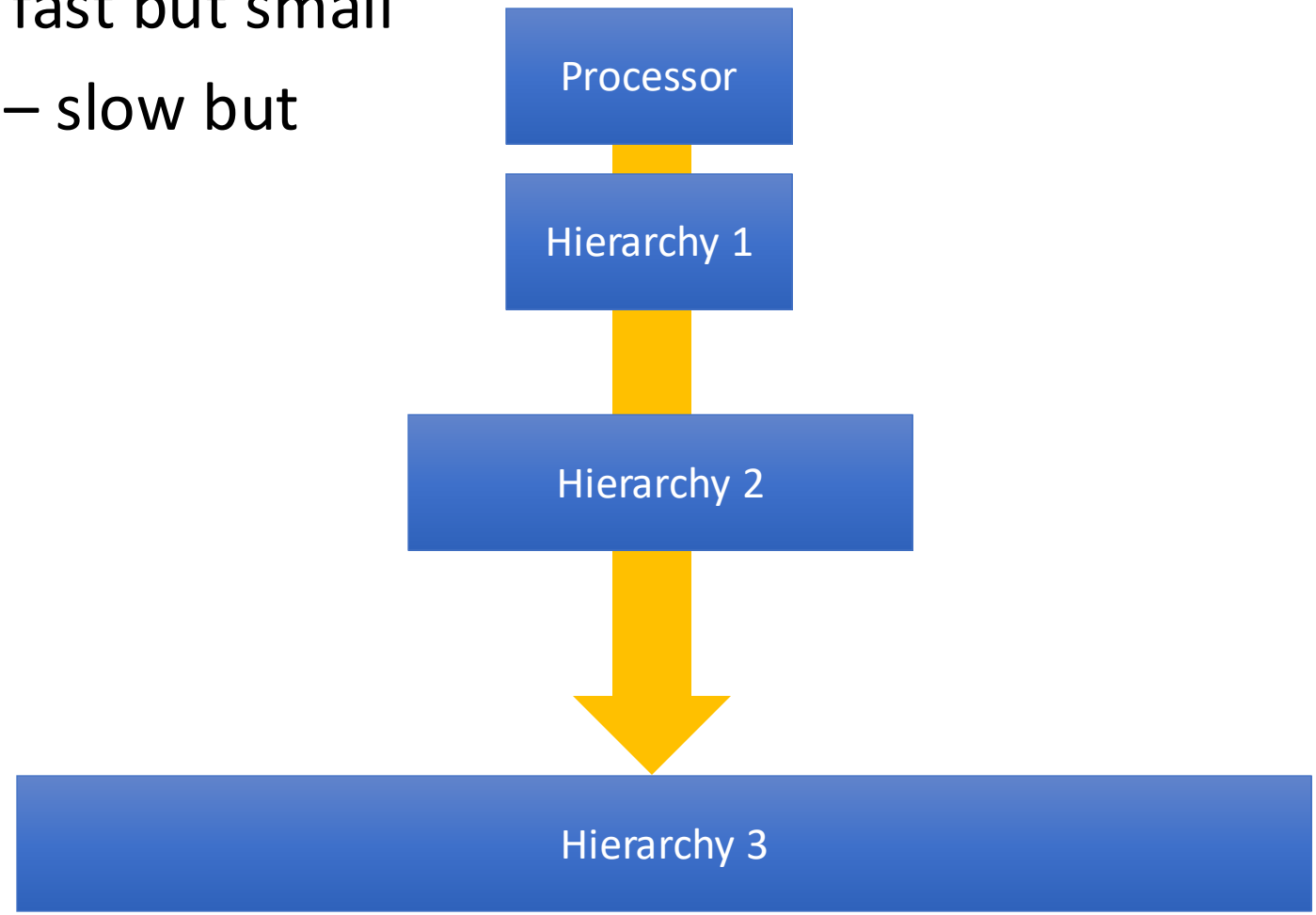

MacBook Pro
Hardware
ATA
Audio
Bluetooth
Camera
Card Reader
Diagnostics
Disc Burning
Ethernet Cards
Fibre Channel
FireWire
Graphics/Displays
Hardware RAID
Memory
NVMExpress
PCI
Parallel SCSI
Power
Hardware Overview:
Model Name: MacBook Pro
Model Identifier: MacBookPro10,
Processor Name: Intel Core i7
Processor Speed: 2.6 GHz
Number of Processors: 1
Total Number of Cores: 4
L2 Cache (per Core): 256 KB
L3 Cache: 6 MB
Memory: 8 GB
Boot ROM Version: MBP101.00EE.B0A
SMC Version (system): 2.3f36
Serial Number (system): C02JL4SQDKQ2
Hardware UUID: E58CFEEE-05B8-51F4-A665-F1A5B91E0CE
Host Memory

Memory Hierarchy

- Host & Device Memory don't mix
  - Copy back & forth through *cudaMemcpy()*
    - *cudaMemcpy2D(), cudaMemcpy3D()*
- Exception: Integrated Graphics Chips
  - Lower-end GPUs
  - Often sold as part of a chipset
  - Part of CPU memory occupied as video memory
  - DMA'ed between host/device automatically behind your back
    - Depends, sometimes there is no need to copy
  - Same programming model as dedicated video memory

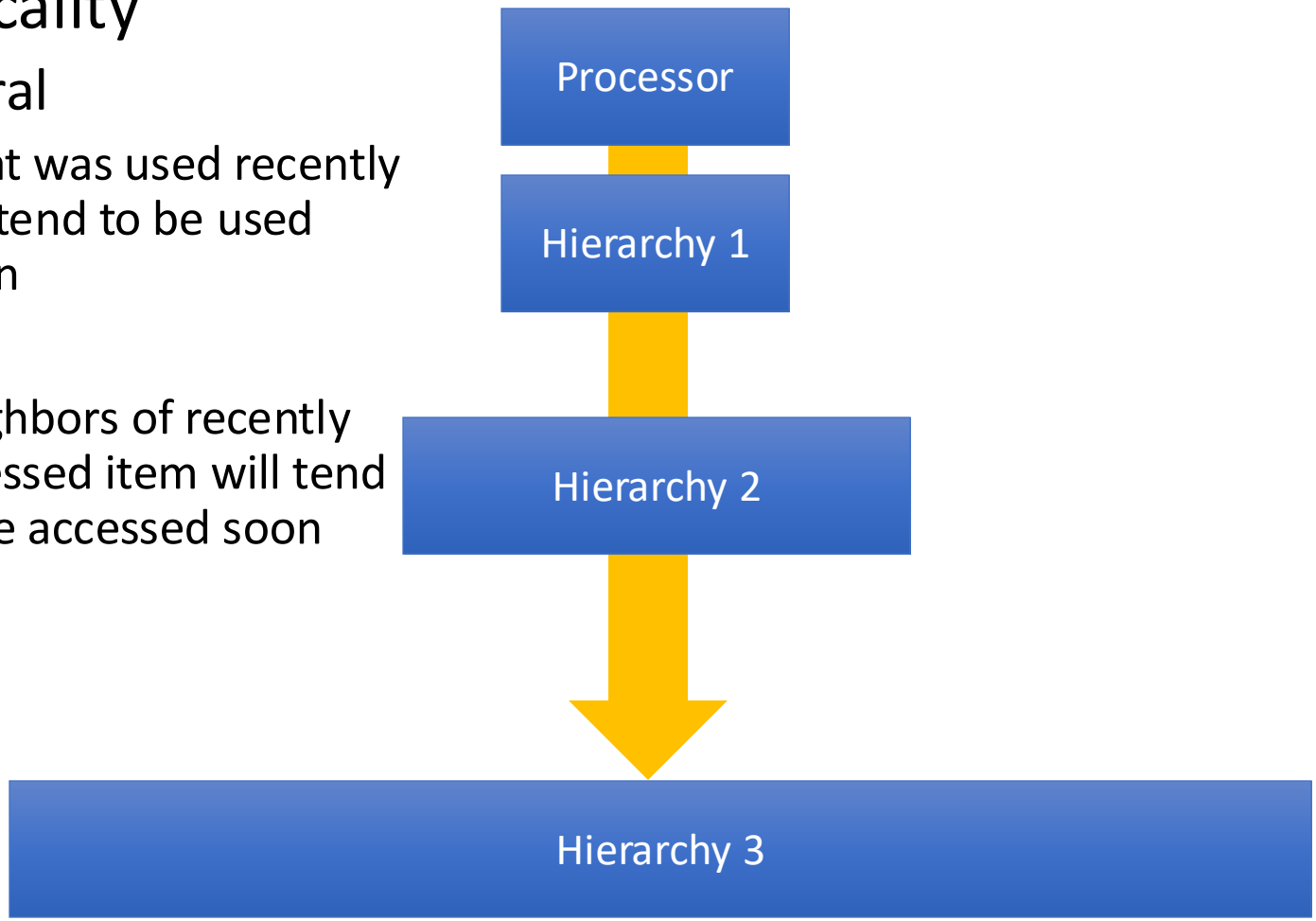
Memory Hierarchy Basics

- Nearby – fast but small
- Far away – slow but large



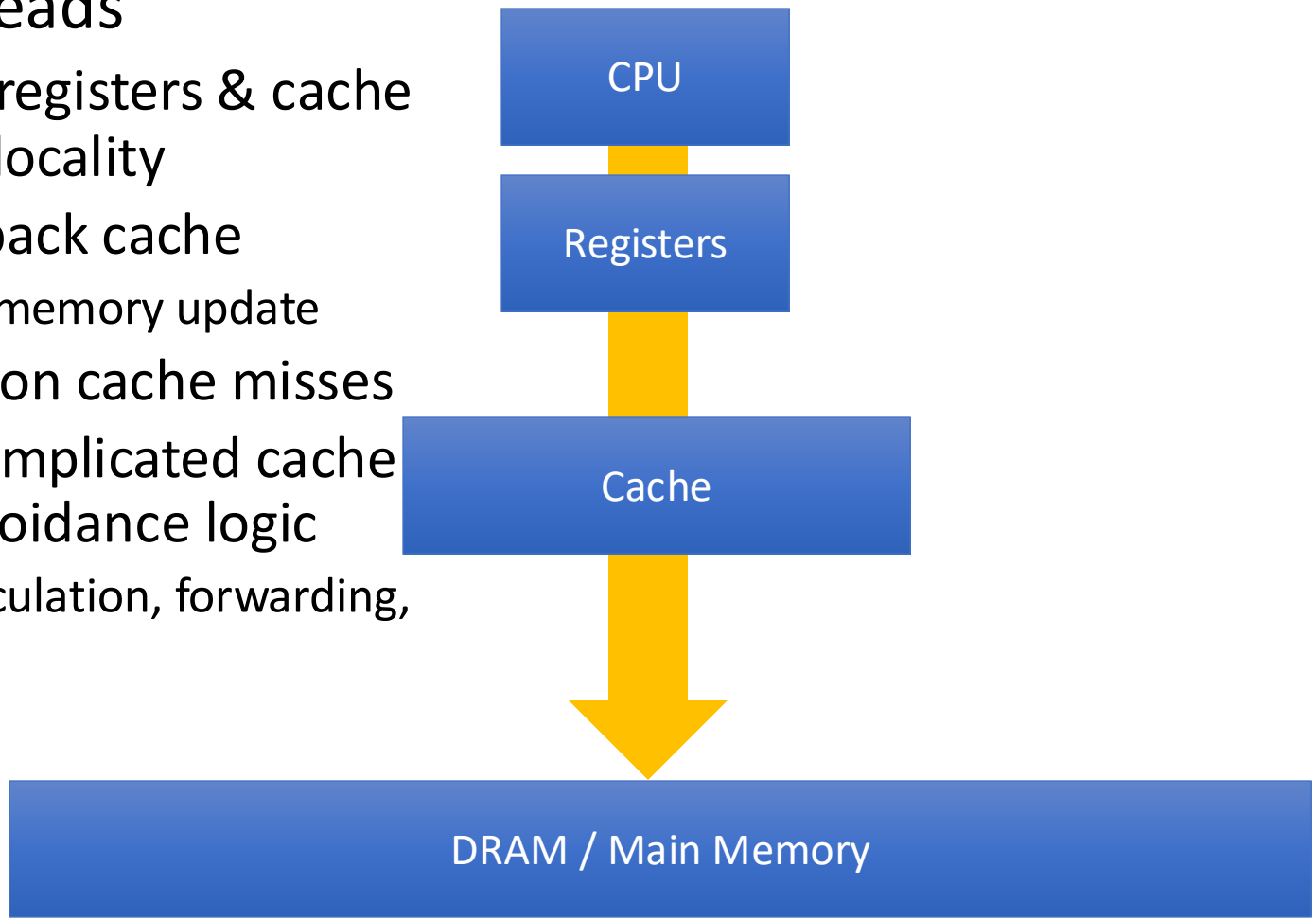
Memory Hierarchy Basics

- Access locality
  - Temporal
    - What was used recently will tend to be used again
  - Spatial
    - Neighbors of recently accessed item will tend to be accessed soon

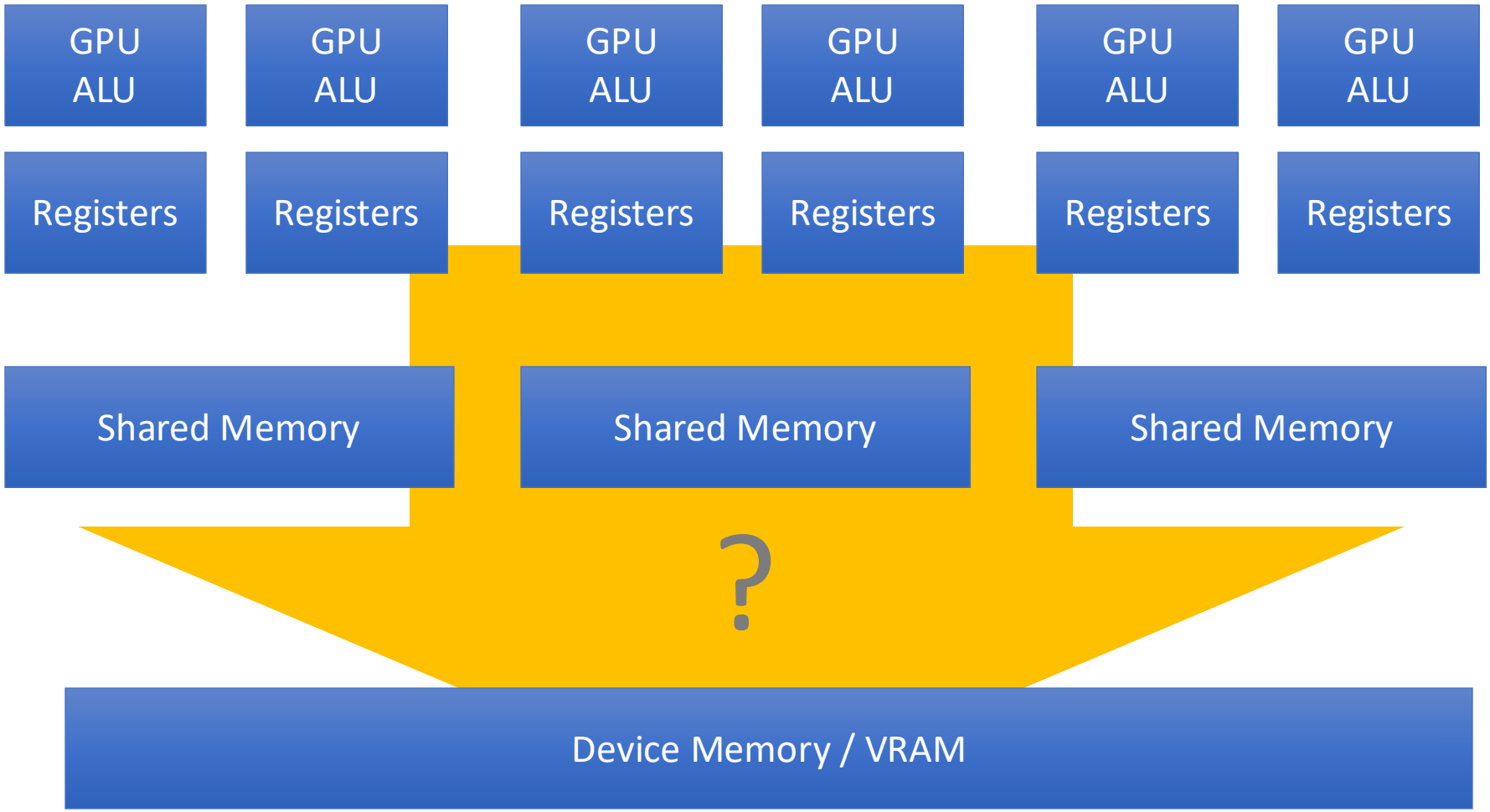


CPU Memory Hierarchy

- A few threads
  - Classic registers & cache access locality
  - Write-back cache
    = lazy memory update
  - Stalled on cache misses
  - Very complicated cache miss avoidance logic
    - Speculation, forwarding, etc



GPU Memory Hierarchy



GPU Memory Hierarchy

- Take home message
  - A lot of bandwidth
  - A lot more computation (>> bandwidth)
  - Access pattern can be chaotic if threads are not synchronized
    - (Remember “warps”?)
- Little or no cache miss avoidance mechanism
  - Crash & burn when cache miss happens
  - Employ lots of threads to cover the “cache miss” cost

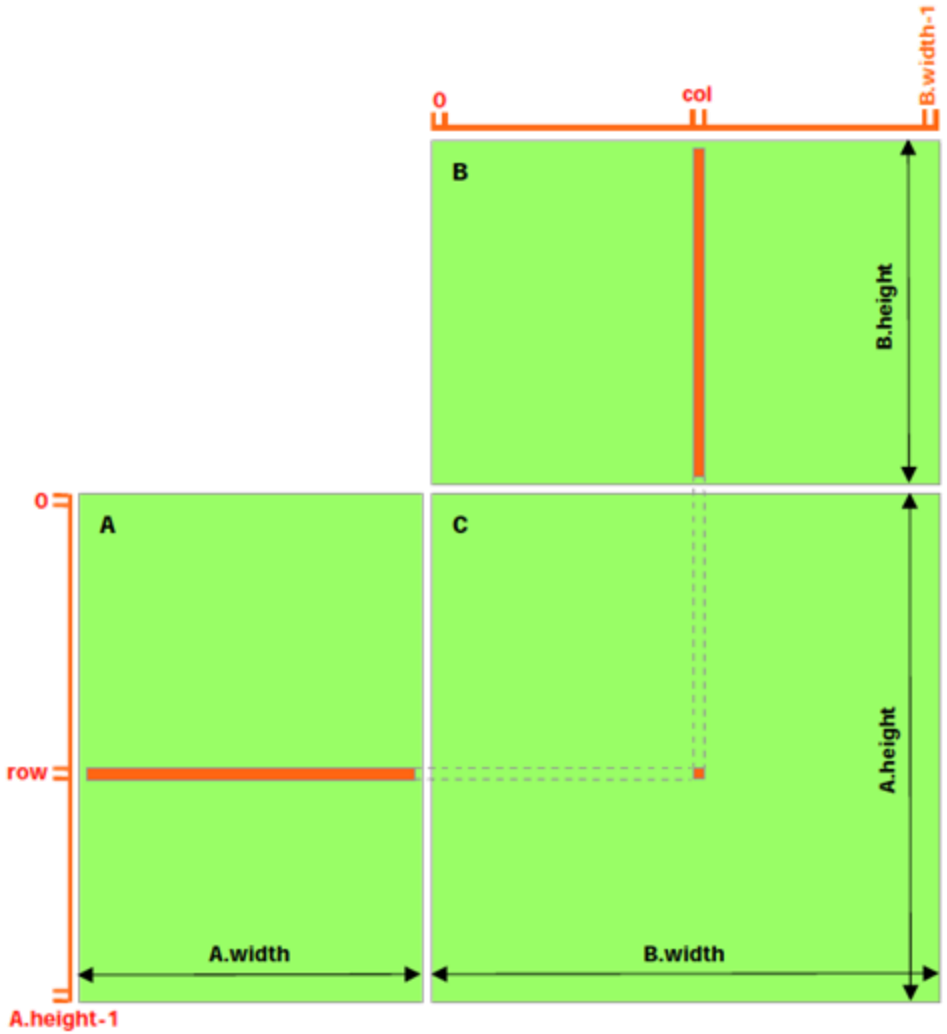
GPU Memory Hierarchy

- "Compute/Global Memory Access Ratio" is important
  (1) Count the # of FLOPS
  (2) Count the # of Global memory access
  CGMA raio = (1)/(2)

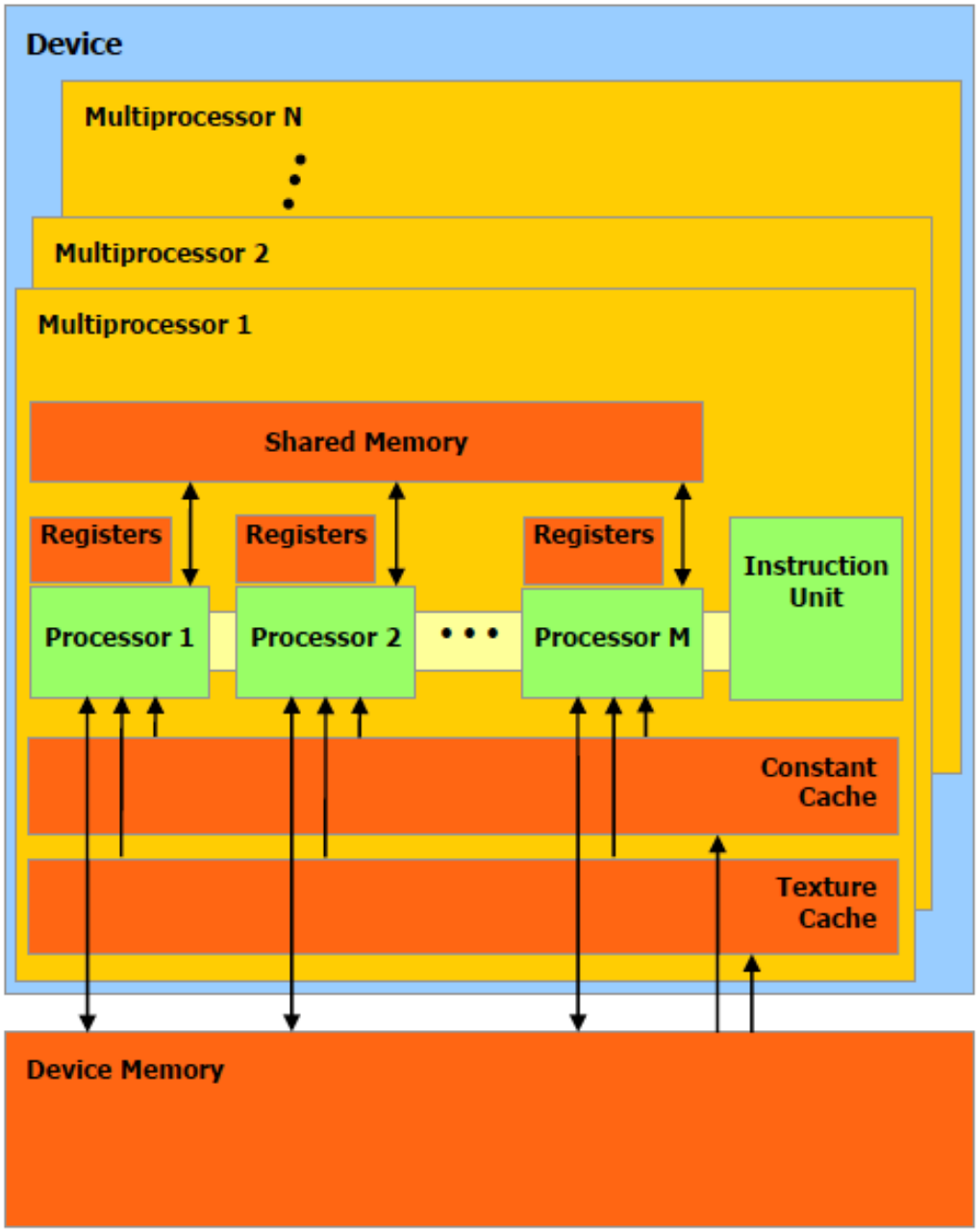
  Sometimes it is faster to recompute if CGMA ratio is low!

Matrix Multiplication: Revisited

- C = A x B
- Naïve implementation:
  - Read a row of A
  - Read a column of B
  - Dot product
- 2 memory reads
- 2 FLOPs
- CGMA = 2/2 = 1.0

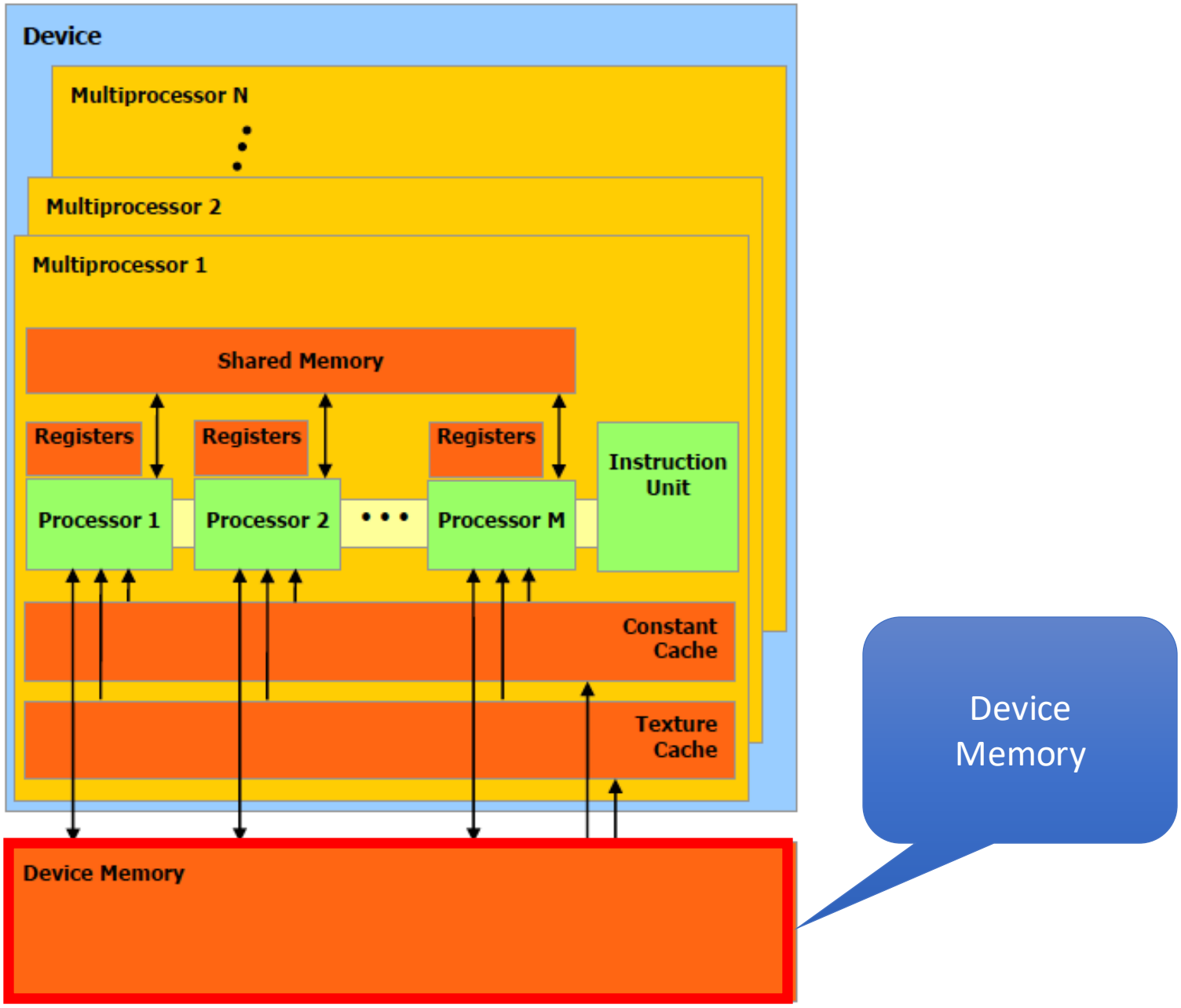


Source: NVIDIA CUDA Programming Guide



CUDA Memory Hierarchy Crib Sheet

Source: NVIDIA CUDA Programming Guide



Source: NVIDIA CUDA Programming Guide

Device Memory

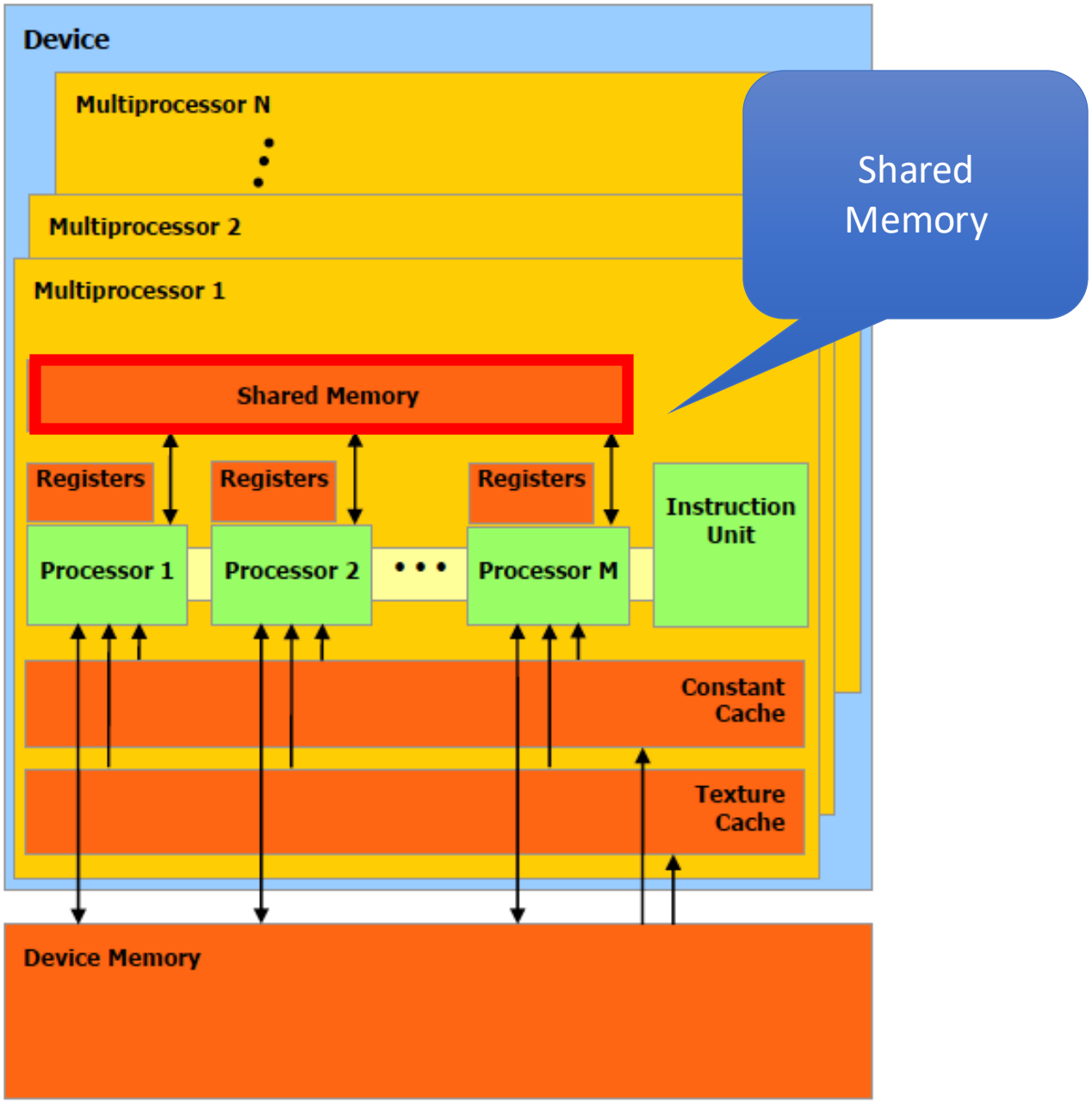
- Allocated as *linear memory* or *CUDA Array*
  - Same with constants
- *cudaMalloc(), cudaMemcpy(), cudaFree()*
  - Linear memory allocation
  - Usually, ptr returned from cudaMalloc are aligned to no less than 256B
- *cudaMemcpyToSymbol() / cudaMemcpyFromSymbol()*
  - Constant memory (A handy wrapper for cudaGetSymbolAddress+cudaMemcpy)
- *cudaMallocPitch()*
  - 2D Arrays allocation
- *cudaMalloc3D()*
  - 3D Arrays allocation

Device Memory – Warp Access

- Width of thread blocks and width of array should be multiples of warps
  - Pad / align your arrays to multiples of 32

```
// Host code
int width = 64, height = 64;
float* devPtr;
size_t pitch;
cudaMallocPitch(&devPtr, &pitch,
                width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// Device code
__global__ void MyKernel(float* devPtr,
                         size_t pitch, int width, int height)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

Source: NVIDIA CUDA Programming Guide



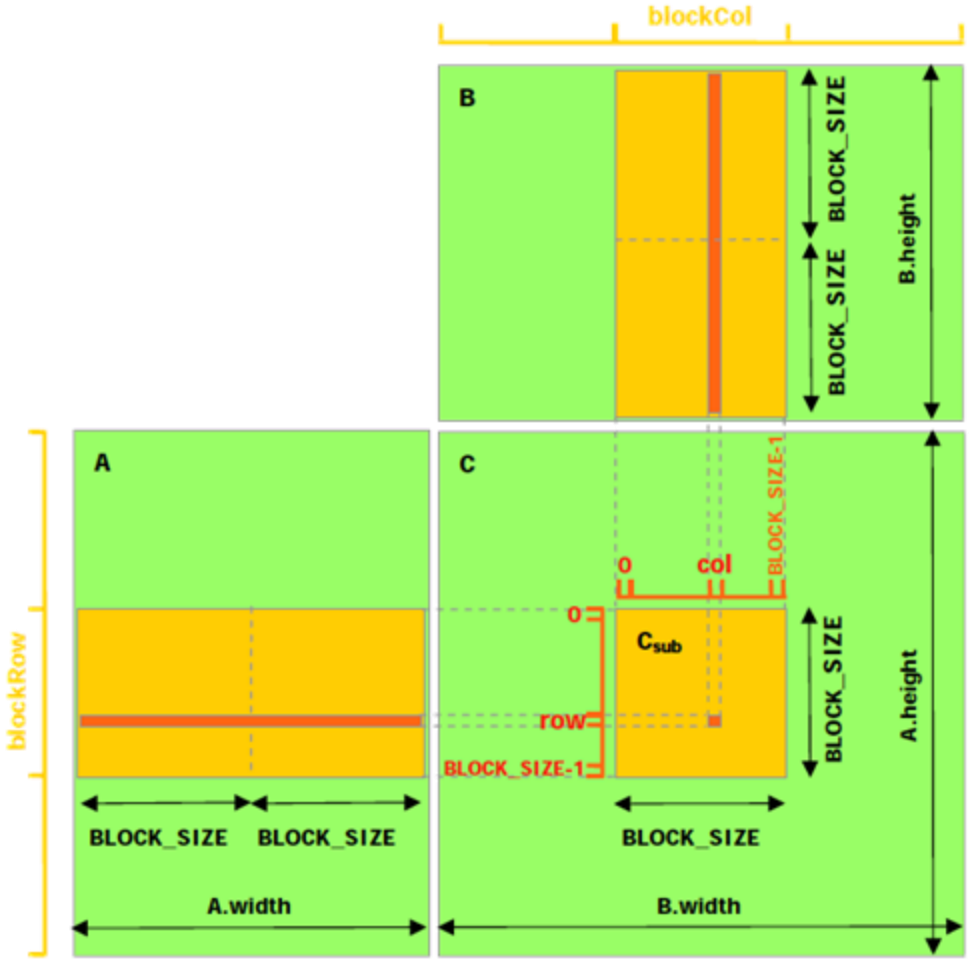
Source: NVIDIA CUDA Programming Guide

Shared Memory

- Per-block shared memory
  - Faster than global memory
  - Almost as good as registers (not counting the latency)
- Remember *__syncthreads()* to synchronize between different warps
  - No need to synchronize within a single warp
  - Fast if no warps are falling behind (scoreboard)
  - Slow if warps have skewed a lot from each other

Matrix Multiplication Revisited

- C = A x B
- Shared memory
- Source code in SDK



Source: NVIDIA CUDA Programming Guide

Common Program Pattern

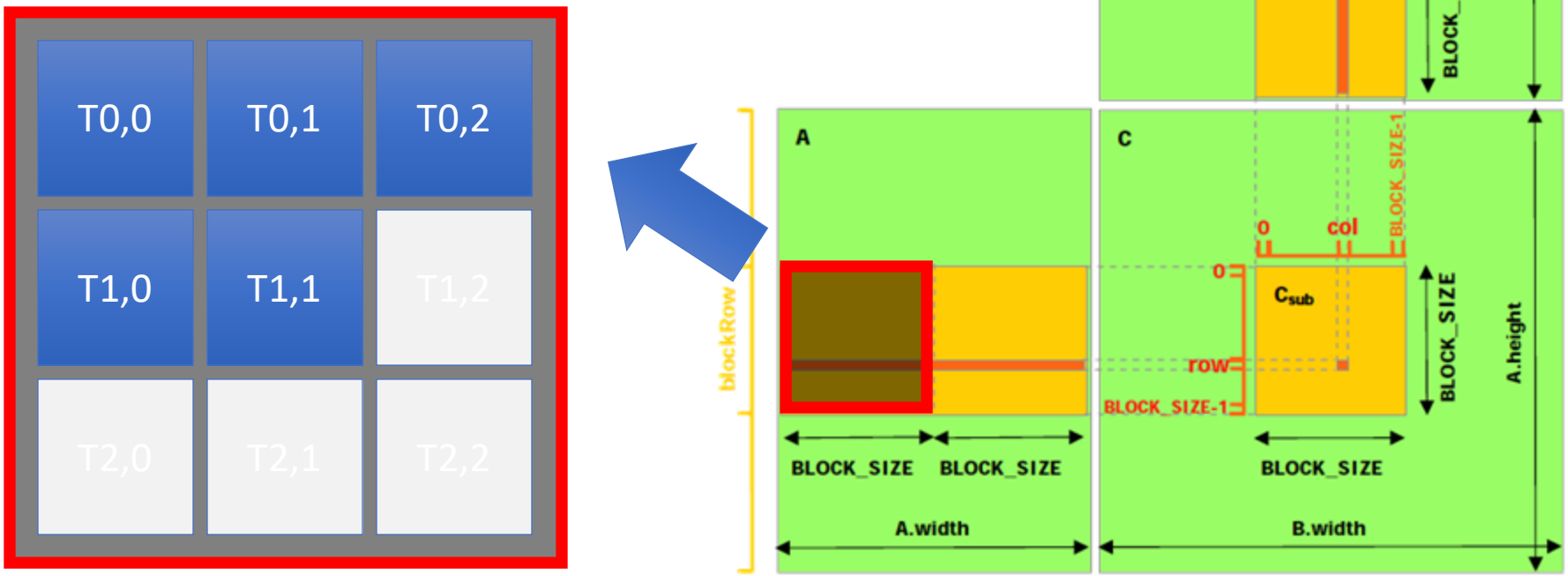
1. Load data from device to shared memory
2. Synchronize with all other threads in the same block
3. Process data in the shared memory
4. Synchronize again if necessary
5. Write results back to the device memory

Common Program Pattern

1. Load data from device to shared memory
2. Synchronize with all other threads in the same block
3. Process data in the shared memory
4. Synchronize again if necessary
5. Write results back to the device memory

Matrix Multiplication Revisited

- C = A x B
- Shared memory



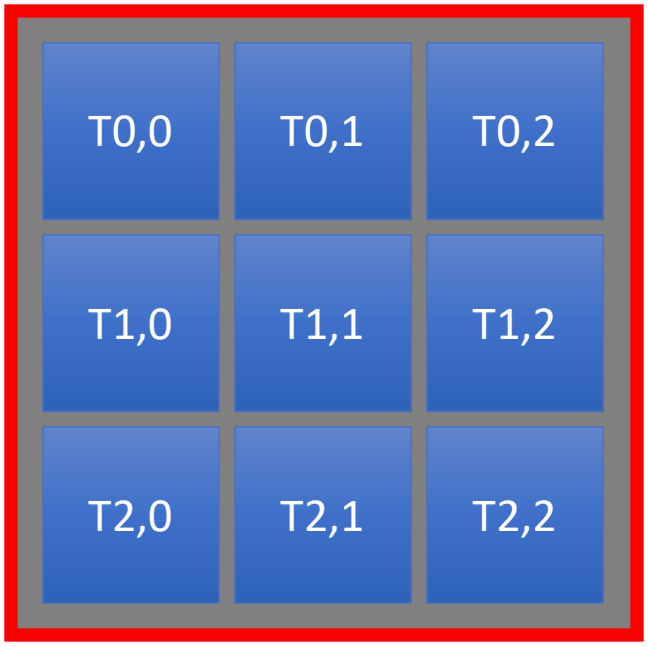
Source: NVIDIA CUDA Programming Guide

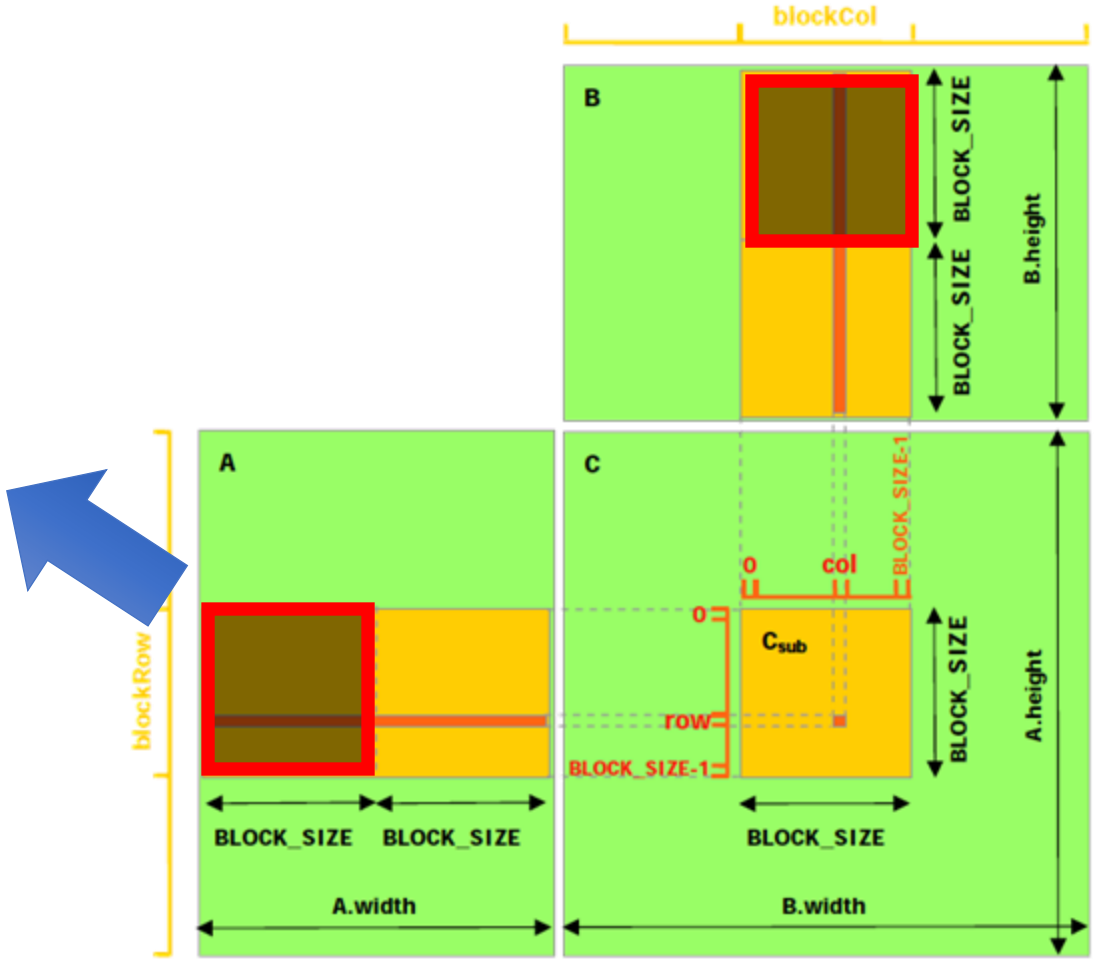
Common Program Pattern

1. Load data from device to shared memory
2. Synchronize with all other threads in the same block
3. Process data in the shared memory
4. Synchronize again if necessary
5. Write results back to the device memory

Matrix Multiplication Revisited

- C = A x B
- Shared memory





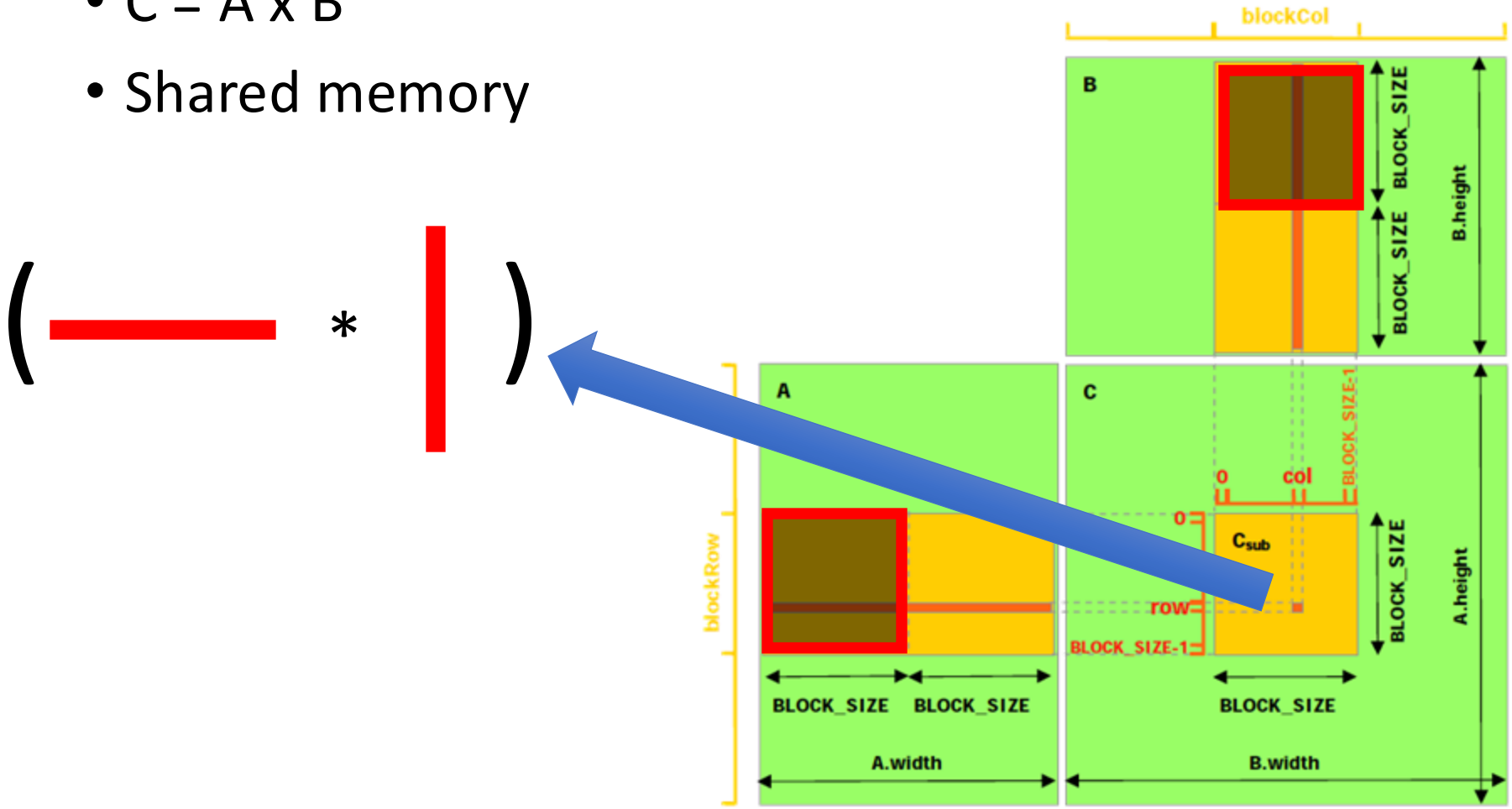
Source: NVIDIA CUDA Programming Guide

Common Program Pattern

1. Load data from device to shared memory
2. Synchronize with all other threads in the same block
3. Process data in the shared memory
4. Synchronize again if necessary
5. Write results back to the device memory

Matrix Multiplication Revisited

- C = A x B
- Shared memory



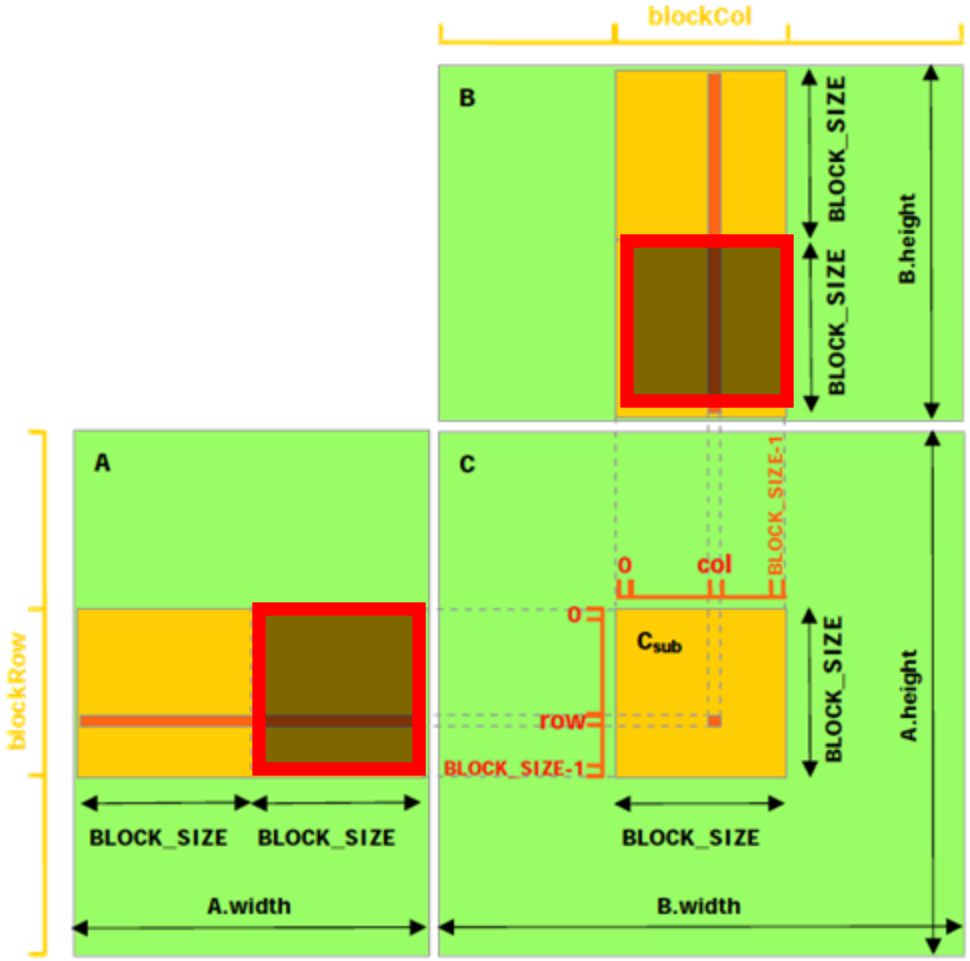
Source: NVIDIA CUDA Programming Guide

Common Program Pattern

1. Load data from device to shared memory
2. Synchronize with all other threads in the same block
3. Process data in the shared memory
4. Synchronize again if necessary
5. Write results back to the device memory

Matrix Multiplication Revisited

- C = A x B
- Shared memory



Source: NVIDIA CUDA Programming Guide

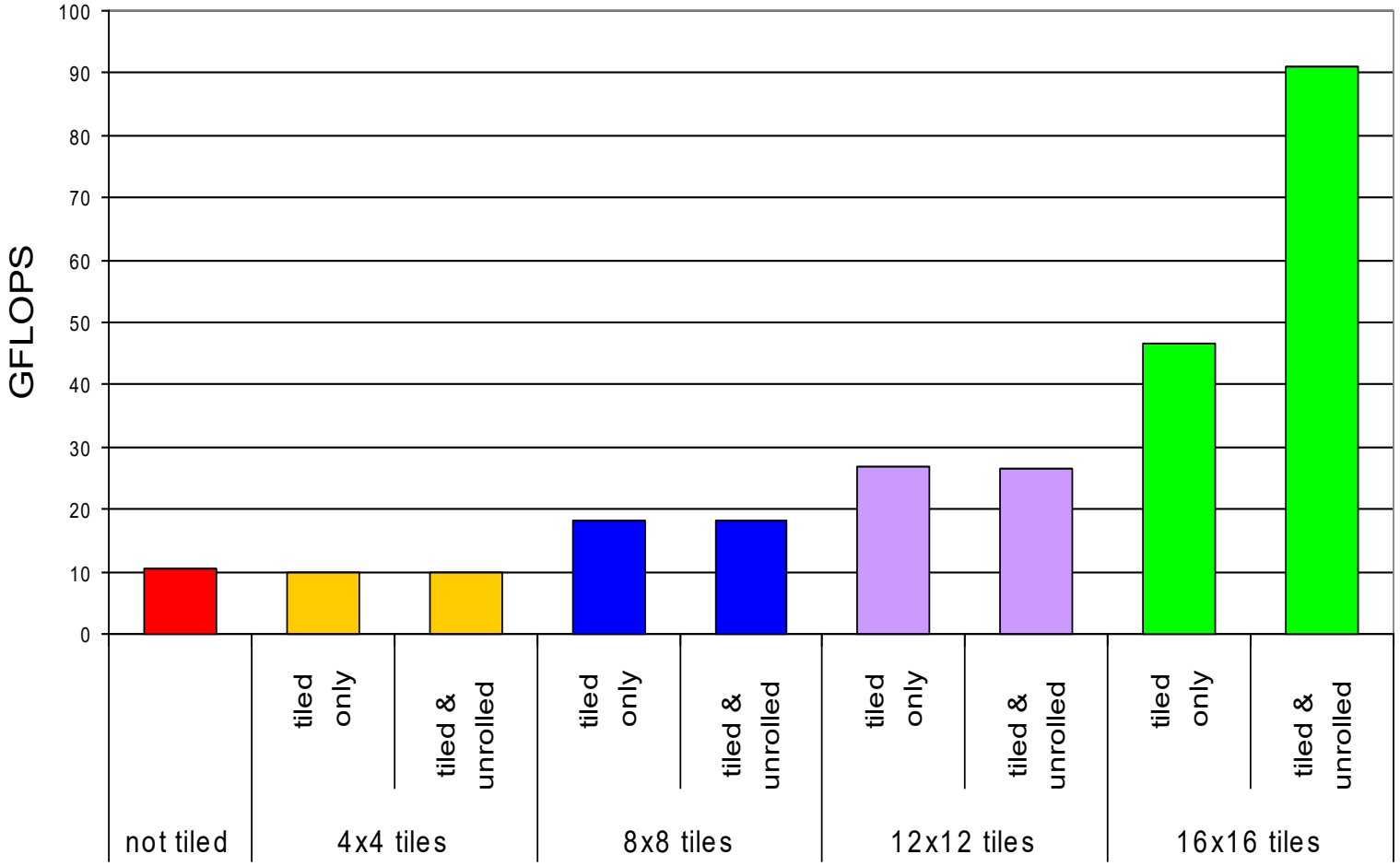
Common Program Pattern

1. Load data from device to shared memory
2. Synchronize with all other threads in the same block
3. Process data in the shared memory
4. Synchronize again if necessary
5. Write results back to the device memory

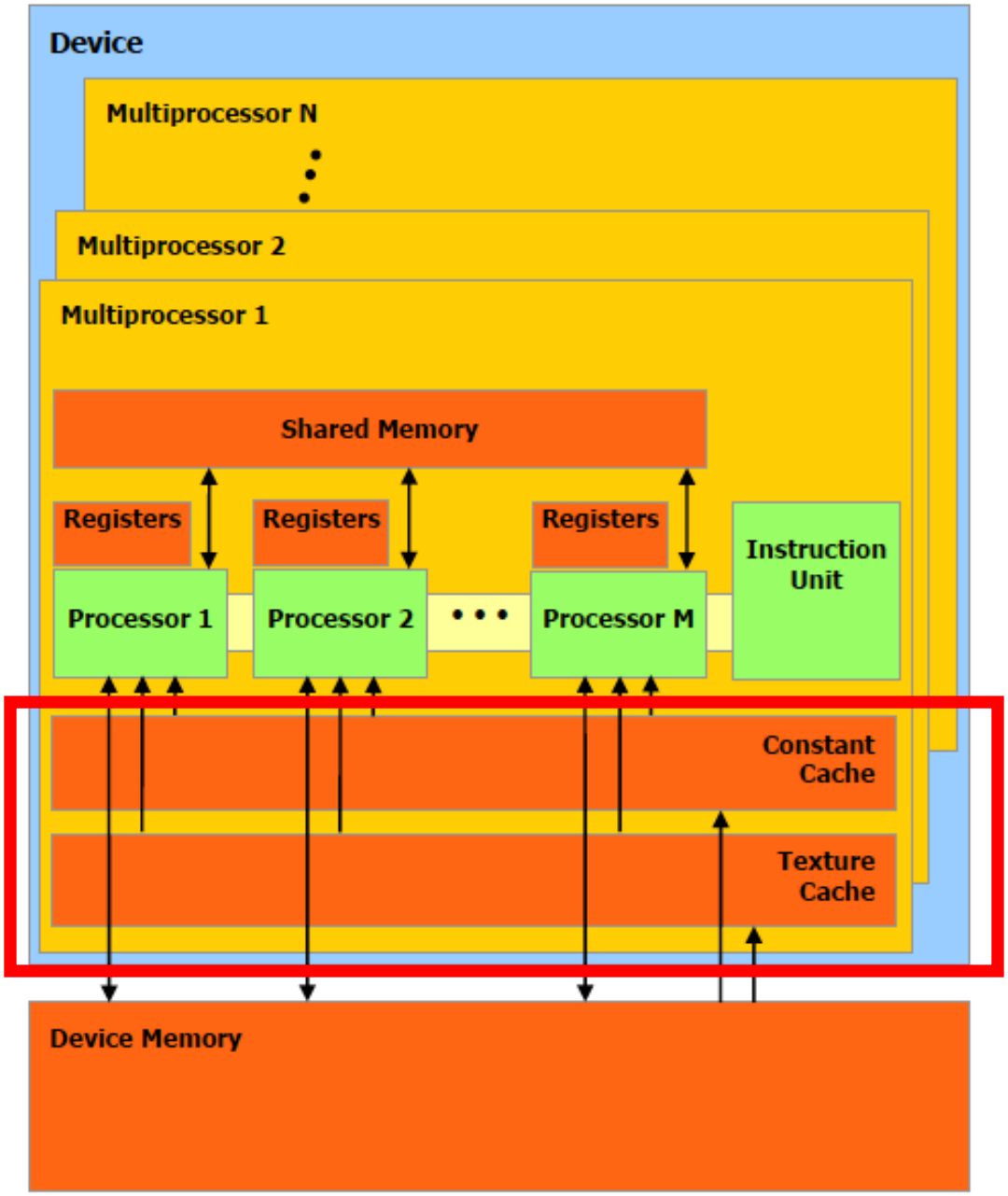
Matrix Multiplication: Complexity

- For a output block sized *b x b*,
  - Memory I/O count to global memory:
    - Read *N x b* elements from A
    - Read *N x b* elements from B
    - Write *b x b* elements
  - Operation count:
    - *N* adds/muls for each output element
    - Same amount of math as before
- New CGMA ratio: *b* (for *N >> b*)

Tiling Size Effects



Source: David Kirk/NVIDIA and Wen-mei W. Hwu,2007-2009
ECE498AL, University of Illinois, Urbana Champaign

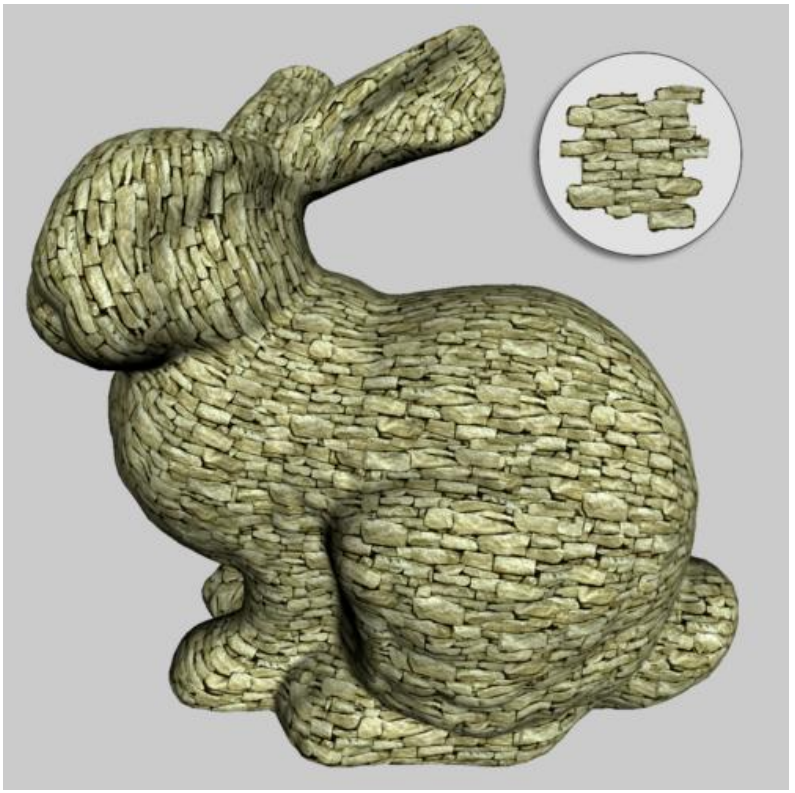


Texture Memory & Cache

Source: NVIDIA CUDA Programming Guide

Texture Mapping

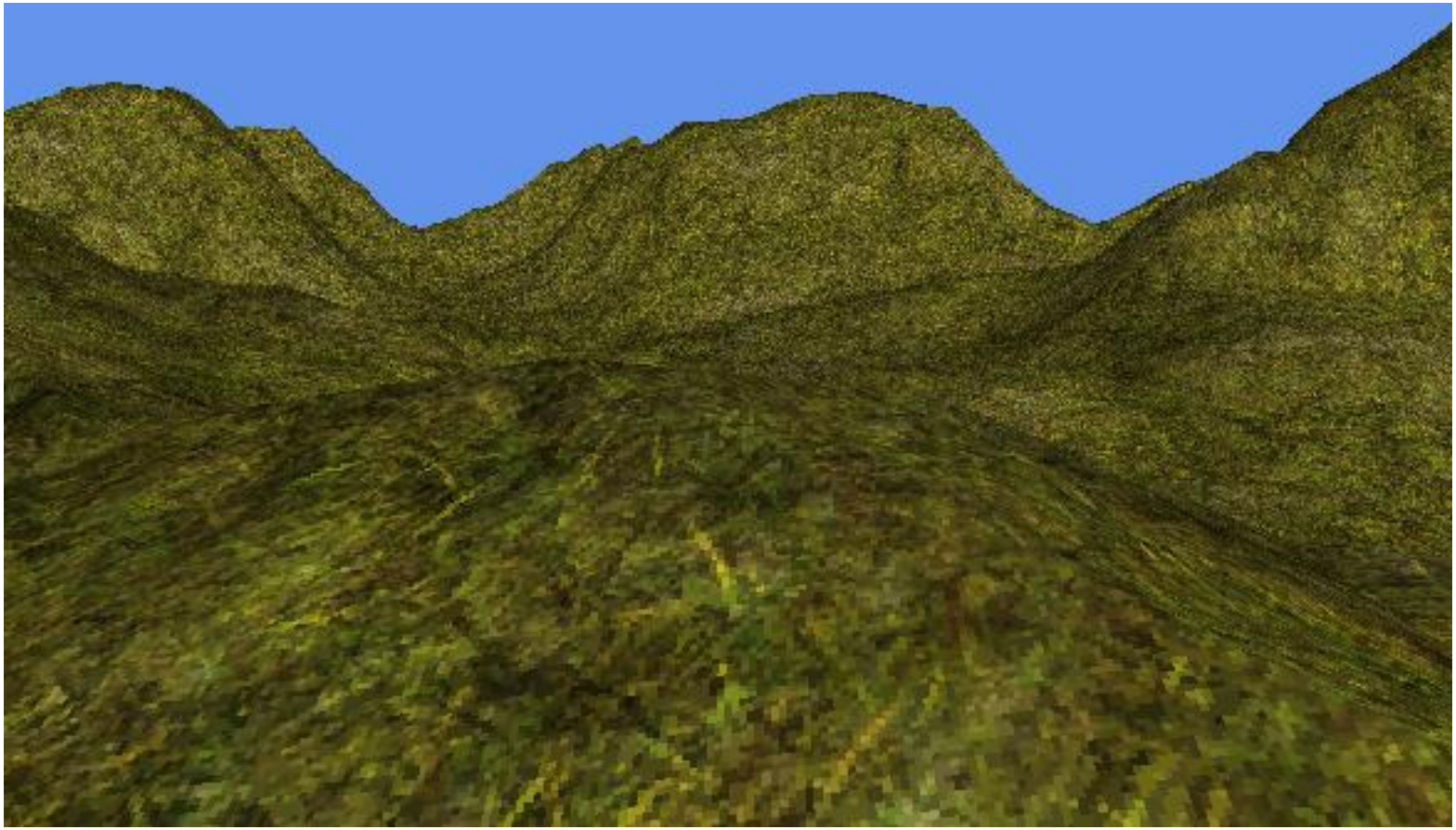
Source: graphics.stanford.edu



"Lapped Textures", Emil Praun, Adam Finkelstein, and Hugues Hoppe, Siggraph 2000

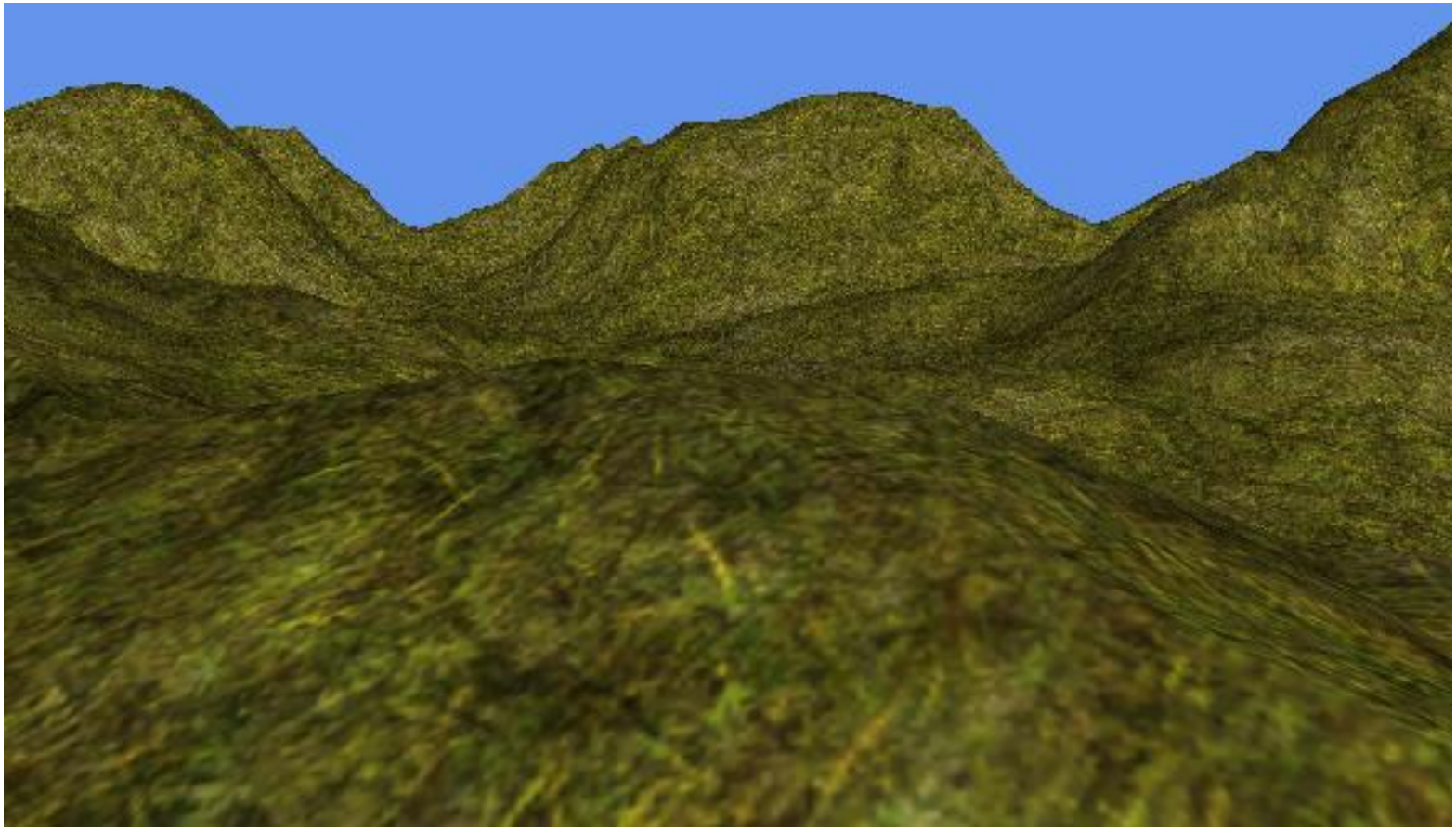
More at http://www.cc.gatech.edu/~turk/bunny/bunny.html

Texture Filtering



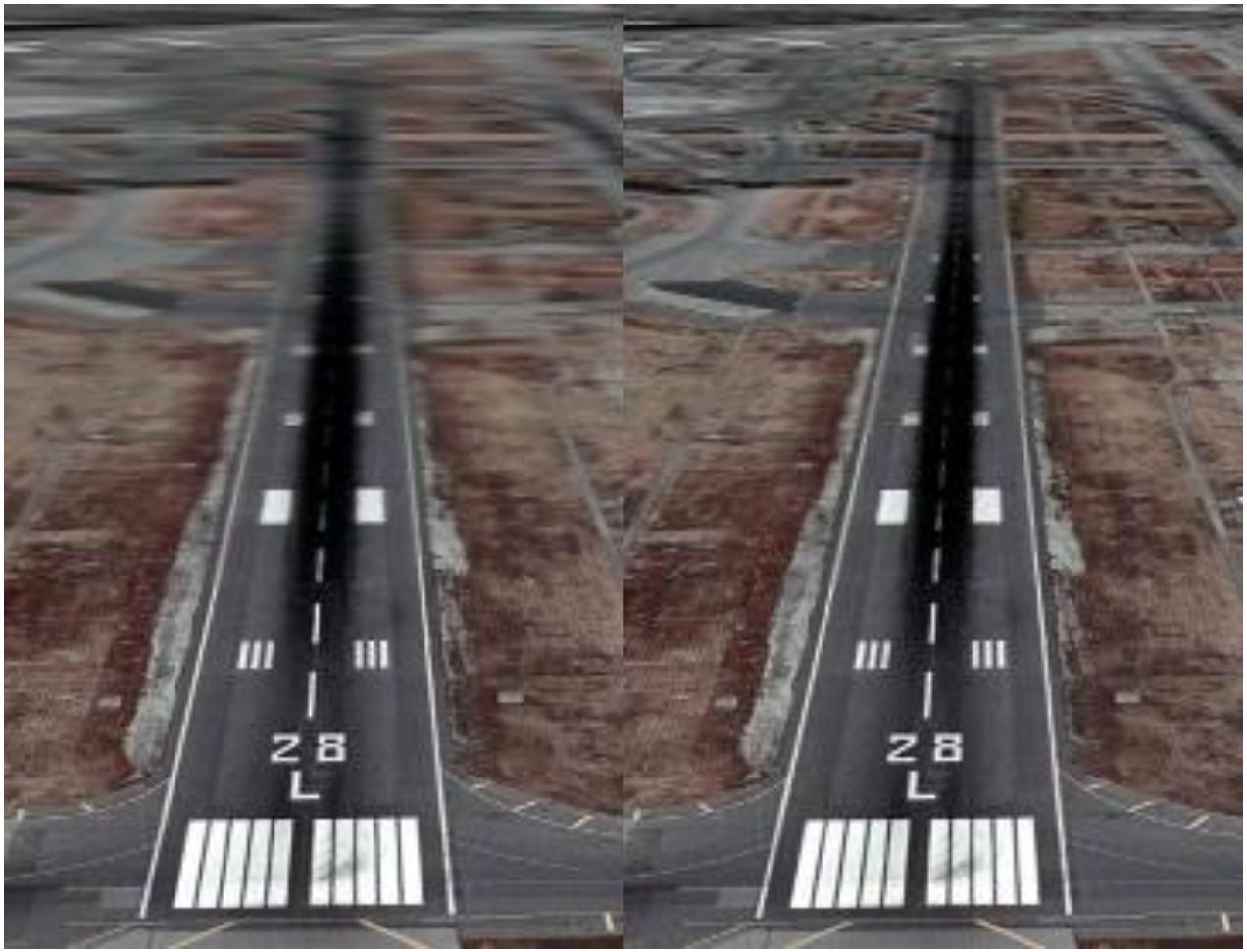
Source: http://blogs.msdn.com/shawnhar/archive/2009/09/08/texture-filtering.aspx

Texture Filtering

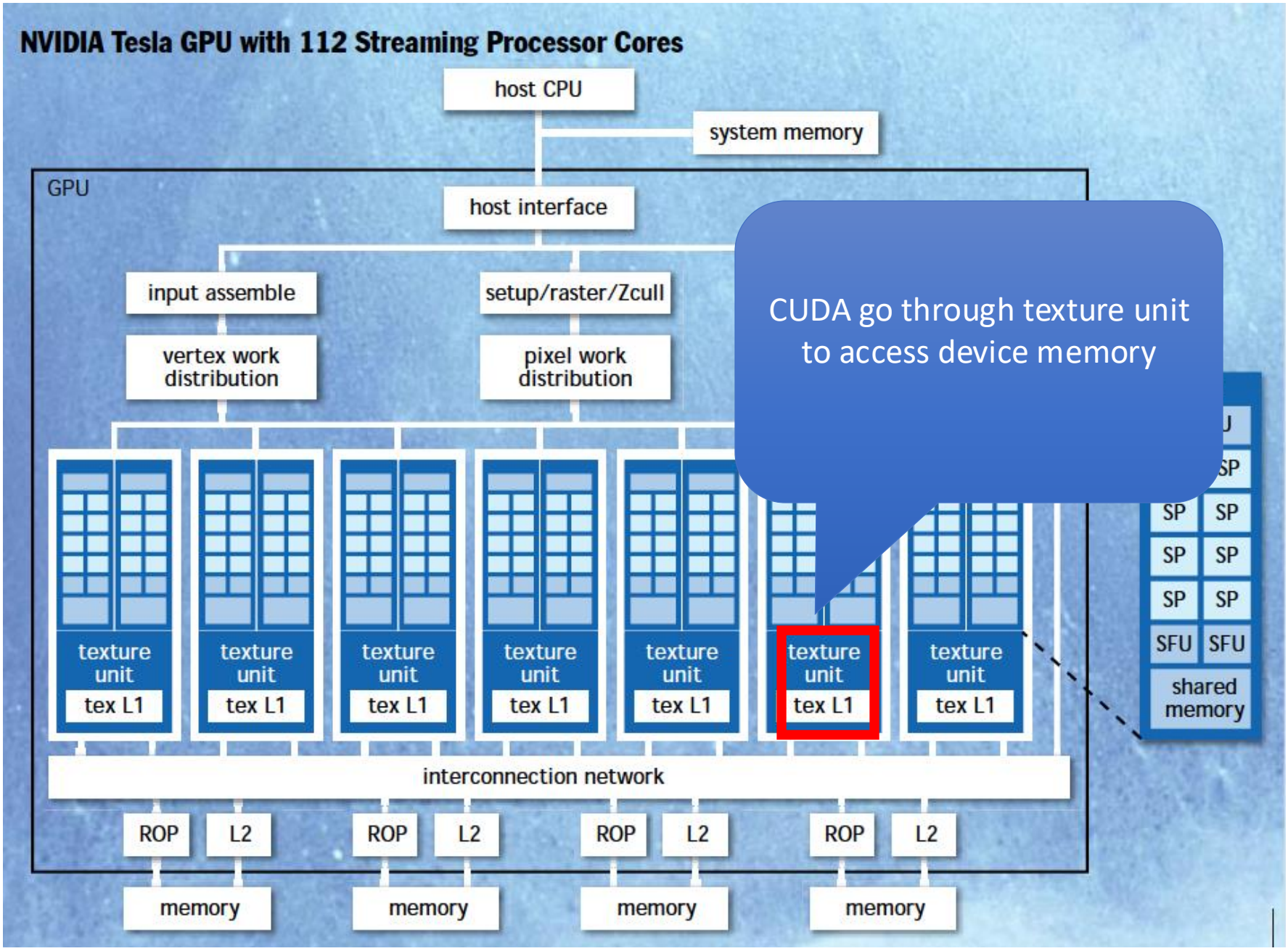


Source: http://blogs.msdn.com/shawnhar/archive/2009/09/08/texture-filtering.aspx

Texture Filtering



Source: wikipedia.org



John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. Queue, 6(2):40{53, 2008.

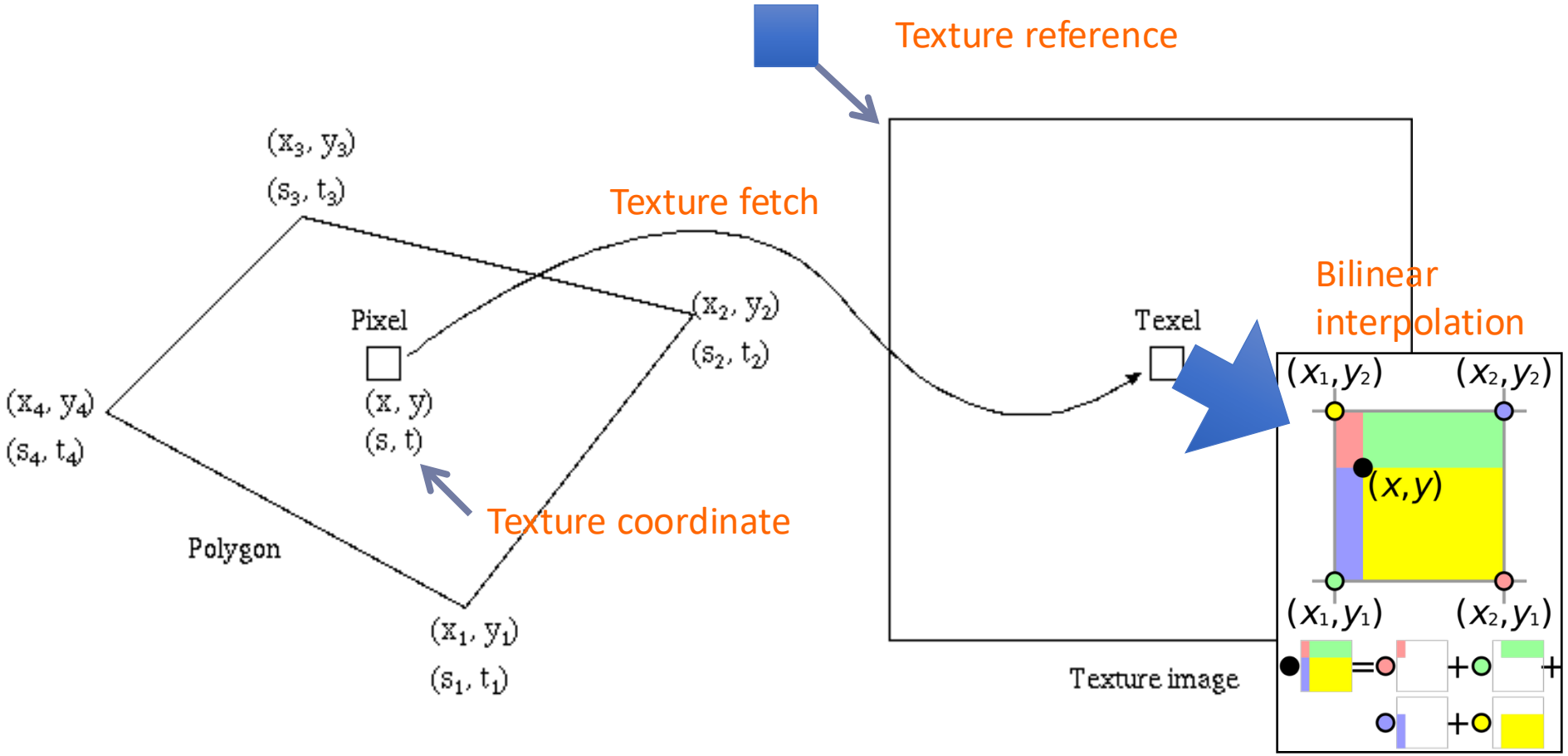
Texture Memory

- Texture reference, texture coordinate, texture fetch, texels
  - A texture *fetch* is an operation that reads several *texels* (texture pixels) near the texture *coordinates* in a texture pointed to by the texture *reference*.
  - The target texels are then *interpolated* into the final texture fetch result.

- CUDA Arrays
  - Opaque memory layout optimized for texture fetching
  - Tiled layout, *Euclidean* distance locality
  - Only readable through texture fetching
  - Read/write through surface memory

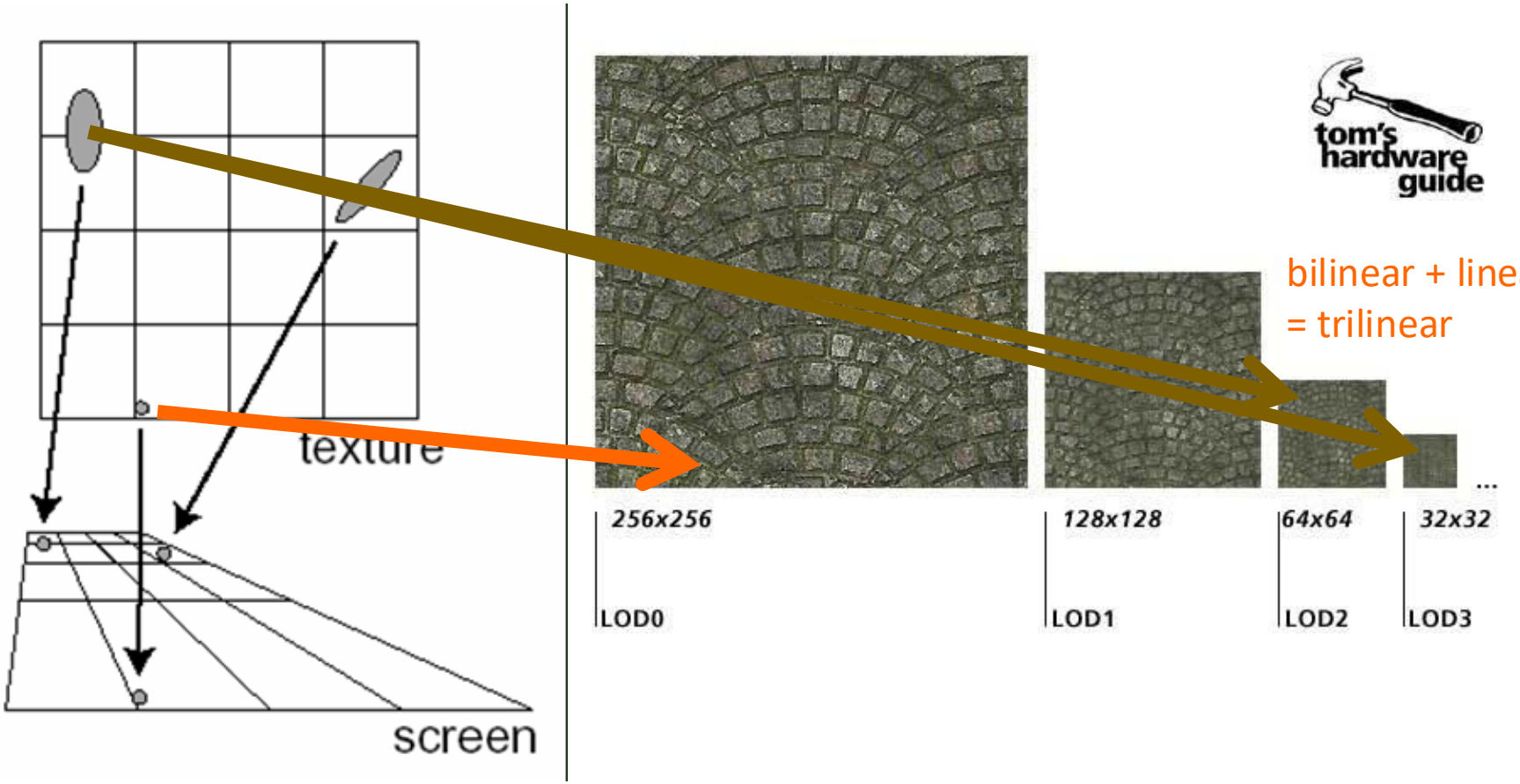
- Watch out for Read/Write Coherency
  - Not synchronized because we have texture cache!

Texture Fetching Explained



Source: http://idav.ucdavis.edu/~okreylos/PhDStudies/Winter2000/TextureMapping.html
& https://en.wikipedia.org/wiki/Bilinear_interpolation

Mipmapping / Anisotropic Texture Filtering



Source: http://www.tomshardware.com/

Texture Memory

- Get a texture (reference) with:

```
texture<Type, Dim, ReadMode> texRef;
```

Type: data type
Dim: dimensionality
ReadMode:

`cudaReadModeNormalizedFloat()`	Normalized to (0,1) or (-1,1)
`cudaReadModeElementType()`	Raw value

Texture Memory

- Binding a texture to linear memory:

```
texture<float, 2, cudaReadModeElementType> texRef;
cudaChannelFormatDesc channelDesc =
          cudaCreateChannelDesc<float>();
cudaBindTexture2D(0, texRef, devPtr, &channelDesc,
          width, height, pitch);
```

Source: NVIDIA CUDA Programming Guide
Texture Reference API

Texture Memory

- Binding a texture to CUDA Array:

```
texture<float, 2, cudaReadModeElementType>
texRef;
cudaBindTextureToArray(texRef, cuArray);
```

Source: NVIDIA CUDA Programming Guide
Texture Reference API

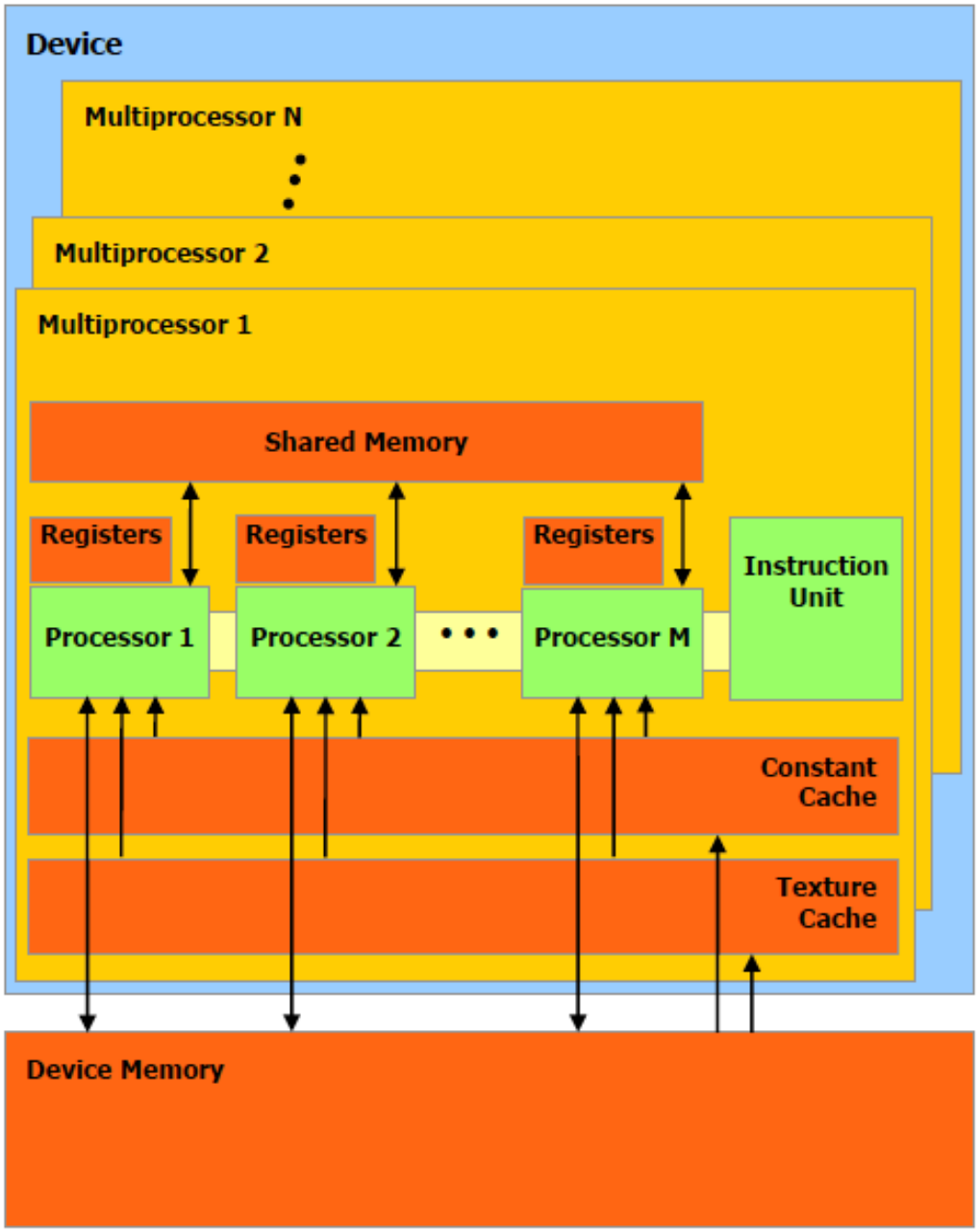
Surface Memory

- Read/write texture memory, basically
  - 2D only, no filtering
  - CUDA Compute 2.0+
  - Use flag: `cudaArraySurfaceLoadStore`

```
cudaArray* cuInputArray; cudaMallocArray(&cuInputArray,
&channelDesc, width, height, cudaArraySurfaceLoadStore);


cudaArray* cuOutputArray; cudaMallocArray(&cuOutputArray,
&channelDesc, width, height, cudaArraySurfaceLoadStore);


cudaMemcpyToArray(...)
```

Texture Memory v.s. Device Memory

- The Good:
  - Optimized for 2D locality
  - Good latency hiding for address calculation
  - Broadcasting of multiple values in single operation
  - Free integer -> float conversion
  - Free texture filtering (be sure to specify properly)
  - Much better caching through texture L1/L2
- The Bad:
  - Memory update problematic within the same kernel – writes garble up reads.
  - More code (!)



Host Memory

Source: NVIDIA CUDA Programming Guide

Host Memory

- Used to transfer between host to device
  - malloc(), free()
  - Old news to you

Page-Locked Host Memory

- *cudaHostAlloc(), cudaFreeHost()*
  - Won't be paged out
- The Good:
  - Higher bandwidth between GPU and CPU transfer
  - DMA transfer (parallel kernel/memory copy)
  - Can be mapped directly onto device without copying
    - Use *cudaHostAllocMapped* flag in *cudaHostAlloc()*
- The Bad:
  - Mapped memory writeable by both host/device, need stream synchronization / events.

Complete Variable Qualifiers

Qualifier	Physical Location	Allocation	Accessibility	Lifetime
__device__	device	cudaMalloc()	All threads & host	Application
__constant__	device*	compile time	All threads & host	Application
__shared__	shader	execution / compile time	Threads in the same block	Kernel Execution
(Unqualified)	shader (registers)	Execution / PTX code	Single thread	Thread
volatile	*Modifier to turn off memory read compiler optimization*			

* Design dependent

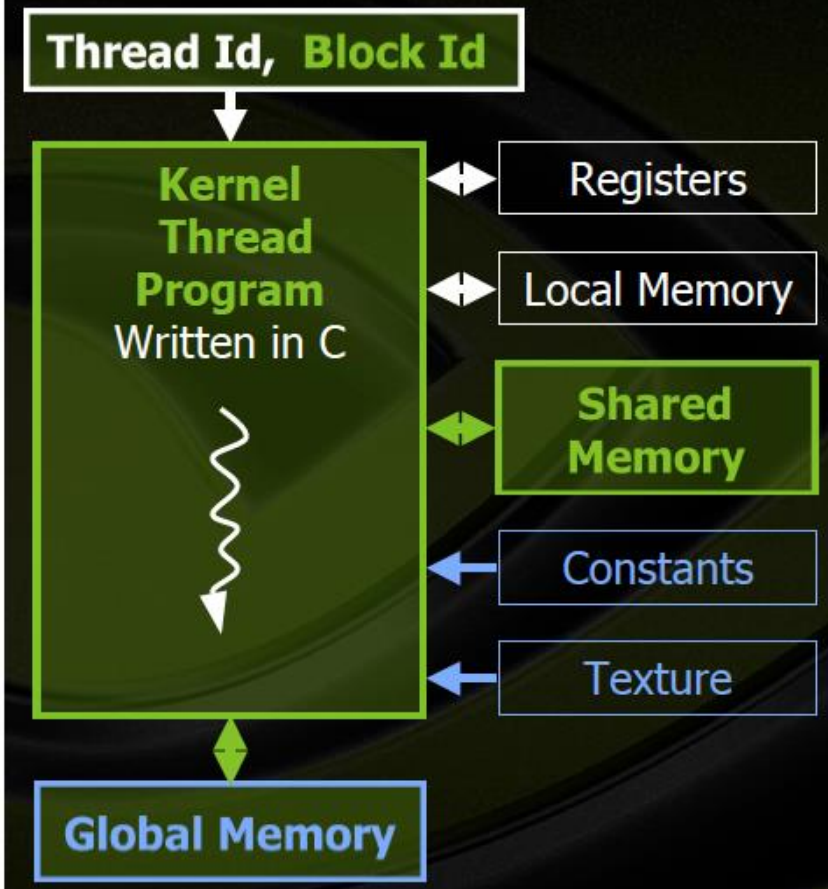
Volatile Modifier

- A global memory access may be converted into register read by the compiler
- Use *volatile* modifier to ensure reads will happen

```
__global__ void myKernel(int* result) {
    int tid = threadIdx.x;
    ...
    if (tid < warpSize) {
        int ref1 = myArray[tid] * 1;
        myArray[tid + 1] = 2;
        int ref2 = myArray[tid] * 1;
        result[tid] = ref1 * ref2;
    }...
}
```

Identical reads, compiler optimized this read away

CUDA Programming Model: Thread Memory Spaces



- Each kernel thread can read:
  - Thread Id — per thread
  - Block Id — per block
  - Constants — per grid
  - Texture — per grid
- Each thread can read and write:
  - Registers — per thread
  - Local memory — per thread
  - Shared memory — per block
  - Global memory — per grid
- Host CPU can read and write:
  - Constants — per grid
  - Texture — per grid
  - Global memory — per grid

Source: John Nickolls, Hot Chips 2007

Local Memory

- Essentially device memory stored using private address space
  - Equally as slow as device/global memory
- Code with caution
  - Compiler will try to put variables inside registers
  - Spill over to local memory for large structures & arrays
- PTX code tells it all
  - Compile with –ptx option, study the code

Memory Rule of Thumb

- Read-only: use constants / texture memory
- Read/Write within the block: use shared memory
- Read/Write within the thread: registers
- Read/Write I/O: global memory / surface memory

Epilogue

Why Bother?

- Bare metal engineering brings peace
  - You know how things work
  - You know how to make things better
  - You can design the next best thing

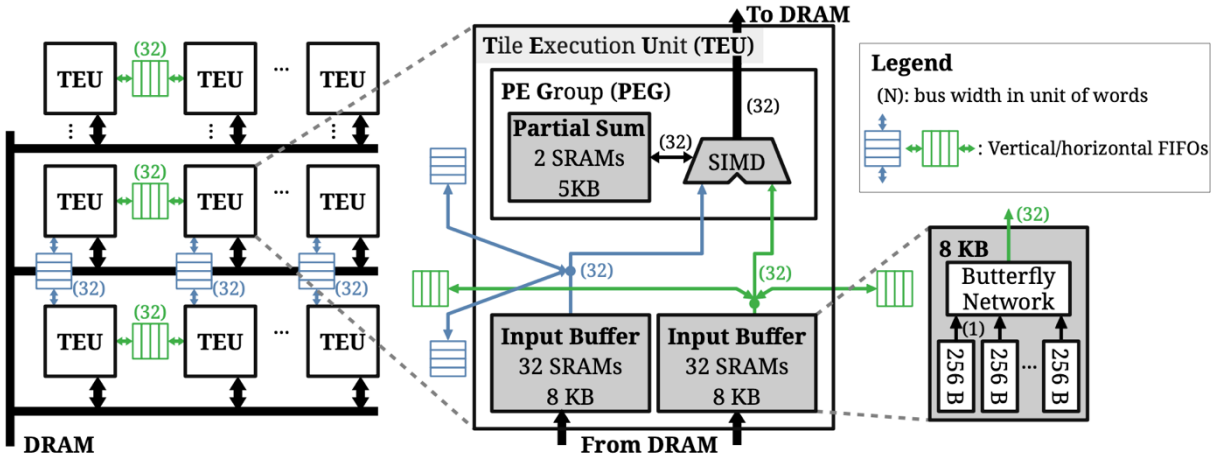
- Ref: The Future of AI Compute: DeepSeek's PTX Innovation and What It Means for Nvidia
  - https://medium.com/@noahbean3396/the-future-of-ai-compute-deepseeks-ptx-innovation-and-what-it-means-for-nvidia-f501b7a0f58e

How about Research?

- We published research papers (next page).
- We built a commercially viable AI processor IP.
  - https://money.udn.com/money/story/5612/8414489

Globally systolic, locally vector design

Memory-efficient with tensor-centric scheduling



References (VectorMesh)

- ¨ Yu-Sheng Lin, Wei-Chao Chen, Chia-Lin Yang, Shao-Yi Chien, “A Dense Tensor Accelerator with Data Exchange Mesh for DNN and Vision Workloads”, IEEE International Symposium on Circuits and Systems (ISCAS), May 2021.
- ¨ Yu-Sheng Lin, Hung-Chang Lu, Yang-Bin Tsao, Yi-Ming Chi, Wei-Chao Chen, Shao-Yi Chien, “*GrateTile: Kernel-Aware Memory Tiling for Efficient Sparse Tensor Transport for CNN*”, IEEE International Workshop on Signal Processing Systems (SiPS), October 2020.
- ¨ Yu-Sheng Lin, Wei-Chao Chen, Shao-Yi Chien, *"MERIT: Tensor Transform for Memory- Efficient Vision Processing on Parallel Architectures"*, IEEE Transactions on Very Large-Scale Integration (VLSI) Systems, Volume 28, Issue 3, March 2020.
- ¨ Yu-Sheng Lin, Wei-Chao Chen, Shao-Yi Chien, *"Unrolled Memory Inner-Products: An Abstract GPU Operator for Efficient Vision-Related Computations"*, International Conference on Computer Vision, Venice, Italy, Oct. 2017.

Discussions

Packaging & Form Factor

- What is the reason behind advanced packaging (e.g. CoWoS)?
- Why are companies selling whole servers (instead of individual chips)?

Power Usage

- What are the primary sources of power usage in AI servers?

- Can you think of ways of reducing power consumption by 10x on existing servers?

Applications

- How adaptable are GPUs for new applications or frameworks?

- How can one ensure that old GPUs work smoothly for new applications?
  - Or we don’t care, and transformer is all we need (how about context window length?)