

# GPU Programming Introduction

National Tsing Hua University  
Instructor: 周志遠 (Jerry Chou)



# Outline

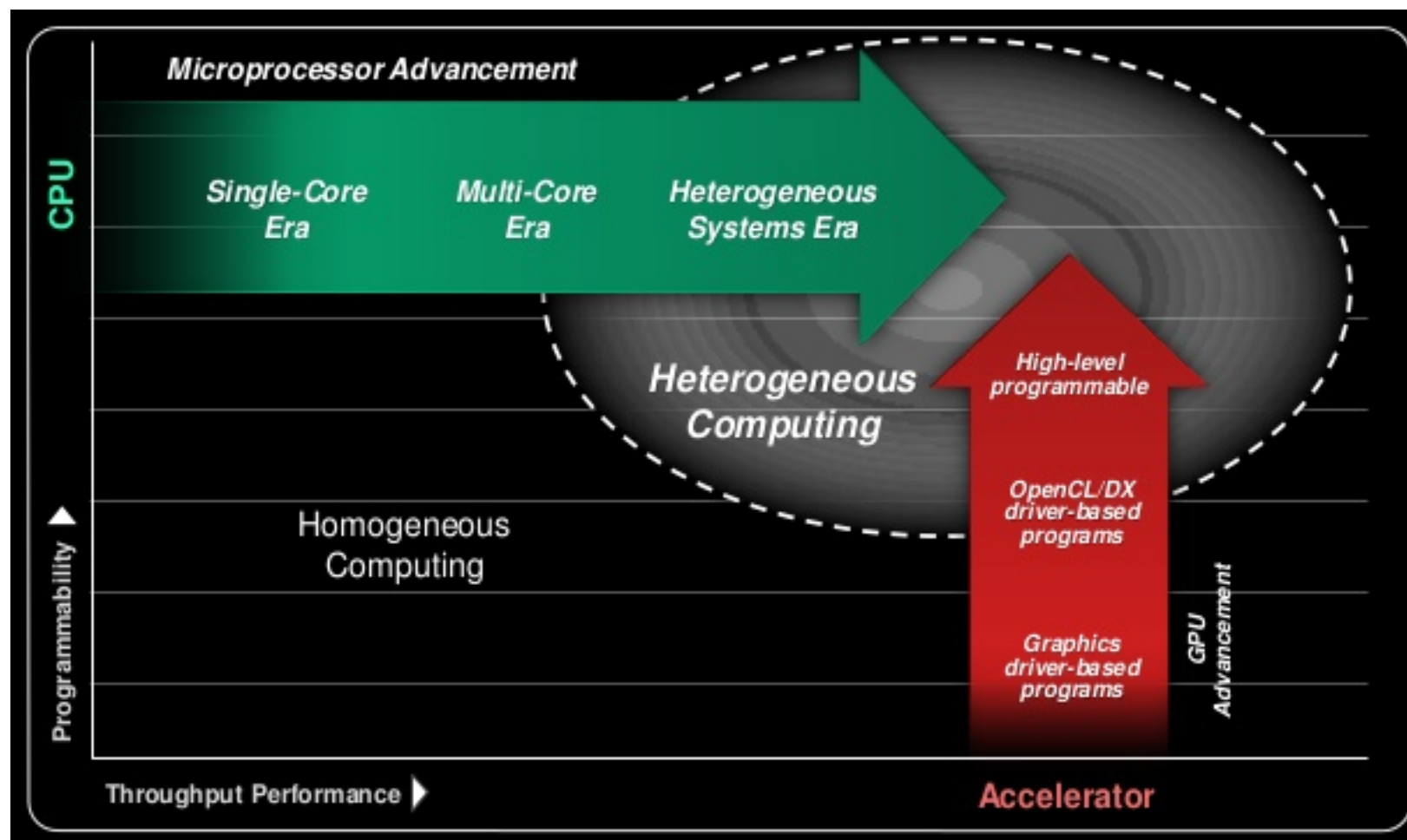
- Heterogeneous Computing & GPU Intro
- GPU Memory Hierarchy & Execution Model
- CUDA Programming Model & API
- Coding Example & Share Memory Optimization



# Heterogeneous Computing

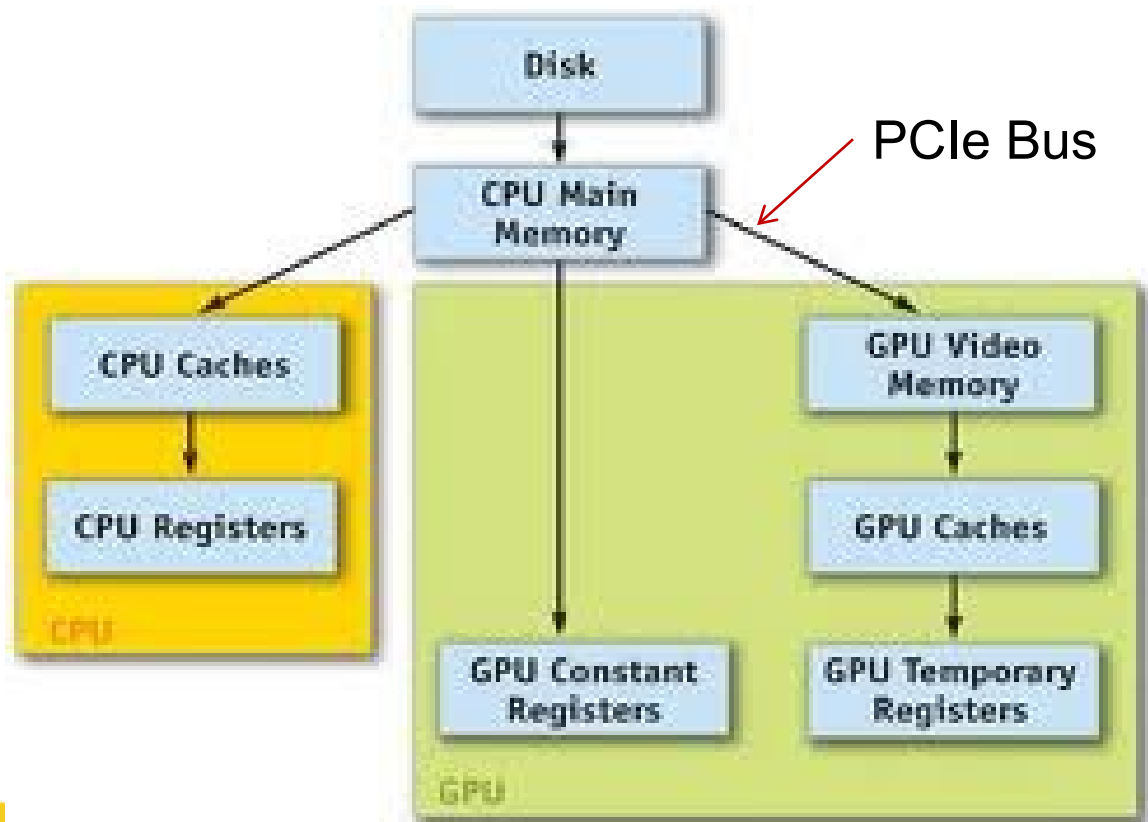
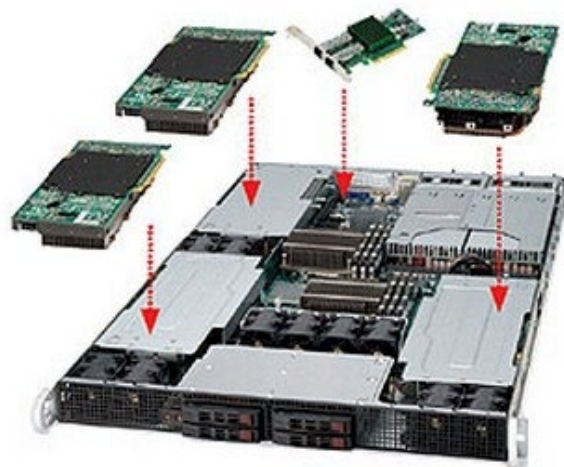
- Heterogeneous computing is an integrated system that consists of different types of (programmable) computing units.
  - DSP (digital signal processor)
  - FPGA (field-programmable gate array)
  - ASIC (application-specific integrated circuit)
  - GPU (graphics processing unit)
  - Co-processor (Intel Xeon Phi)
- A system can be a cell phone or a supercomputer

# Shift of Computing Paradigm

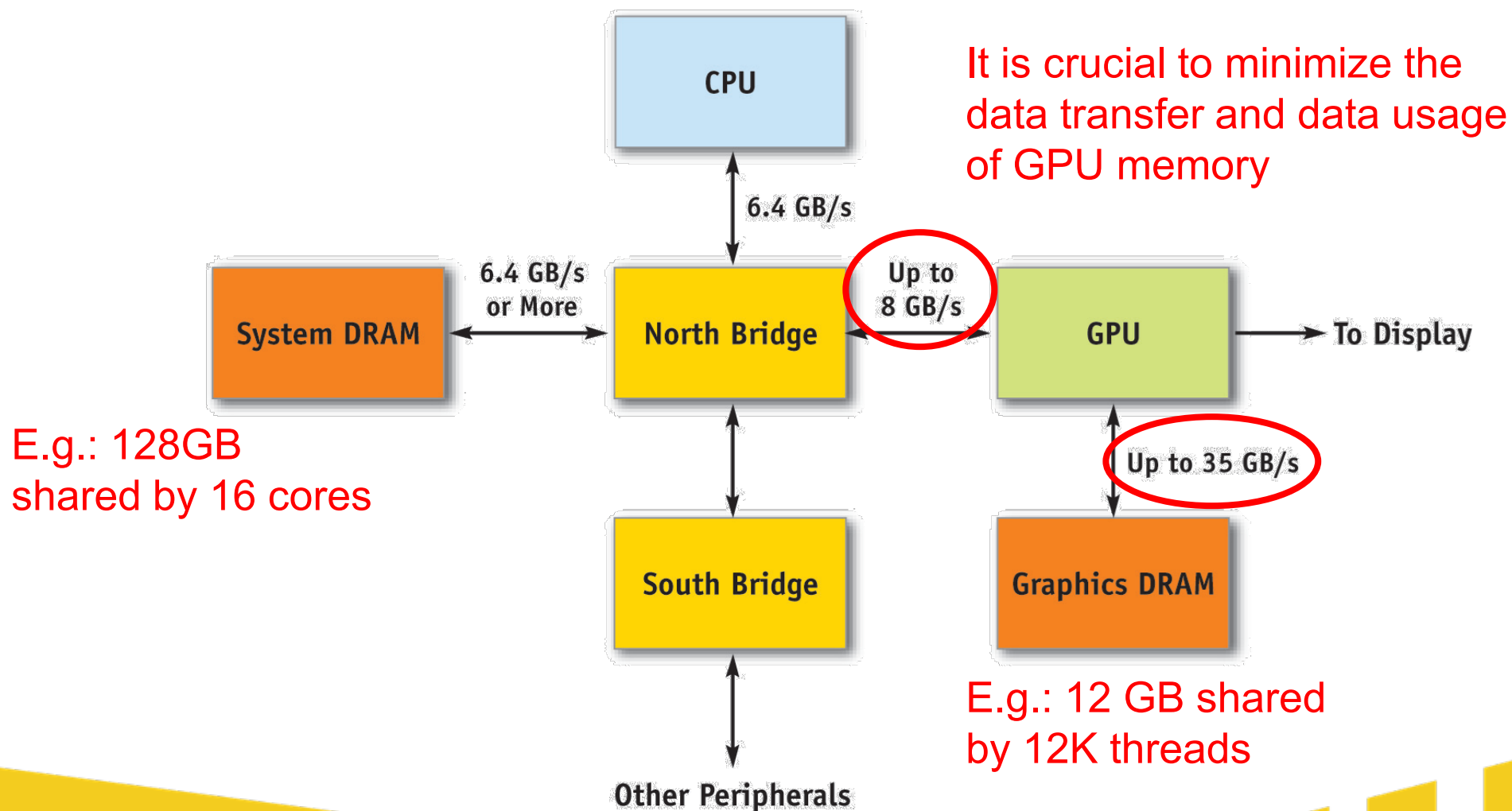


# GPU Servers

- Same HW architecture as commodity server, but memory copy between CPU and GPU becomes the main bottleneck

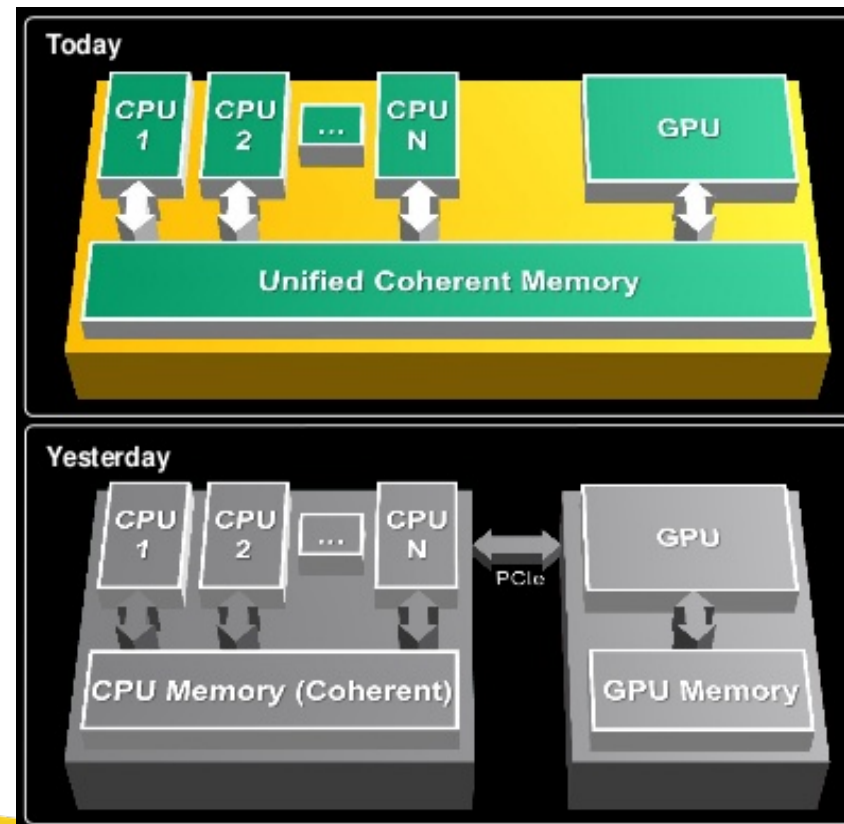


# Memory Bottleneck



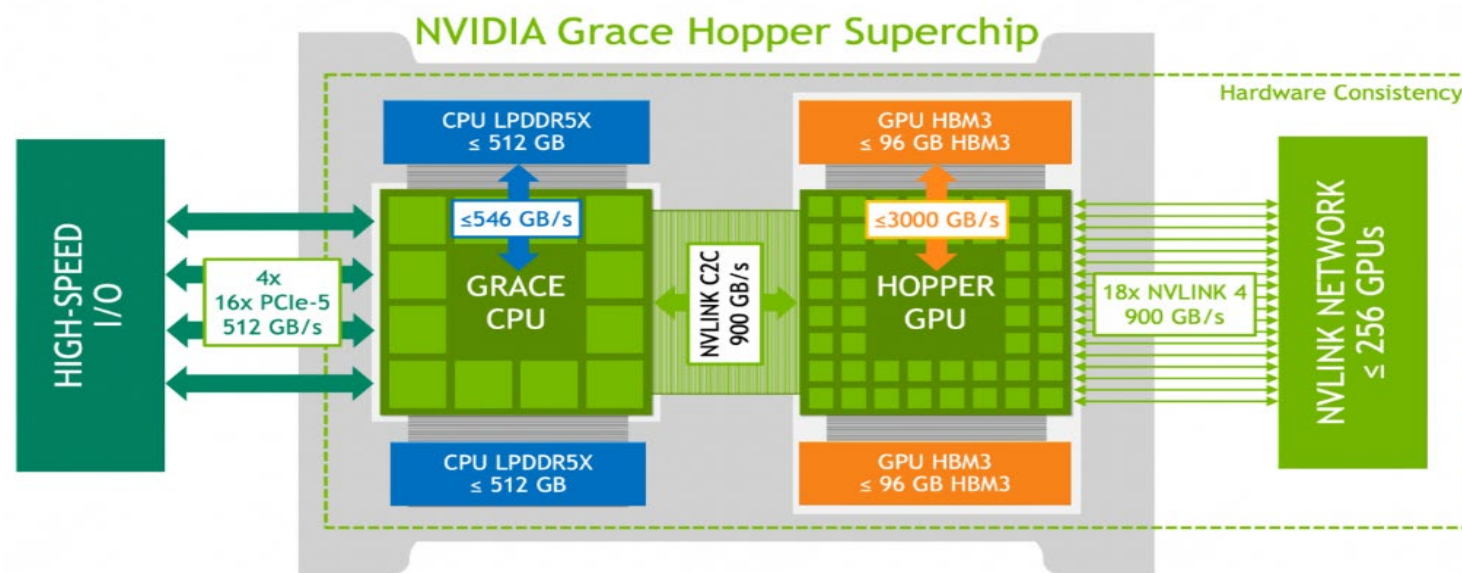
# Heterogeneous System Architecture (HSA)

- Aim to provide a common system architecture for designing higher-level programming models for all devices
- Unified coherent memory
  - Single virtual memory address space
  - Prevent memory copy



# NVIDIA Grace Hopper Superchip

- Massive Bandwidth for Compute Efficiency
- Using NVIDIA® **NVLink®**-C2C (900GB/s) for CPU-GPU connections
- High-speed I/O
- **HBM3 memory**
- **On-chip fabrics**
- System-on-chip (SoC) design
- **Arm-based processors**





# GPU (Graphic Processing Unit)

- A **specialized chip** designed for rapidly display and visualization
  - **SIMD architecture**
- Massively multithreaded manycore chips
  - NVIDIA Tesla products have up to **5120 scalar processors**
  - Over **12,000 concurrent threads**
  - Over **470 GFLOPS** sustained performance









# GPGPU (General-Purpose Graphic Processing Unit)

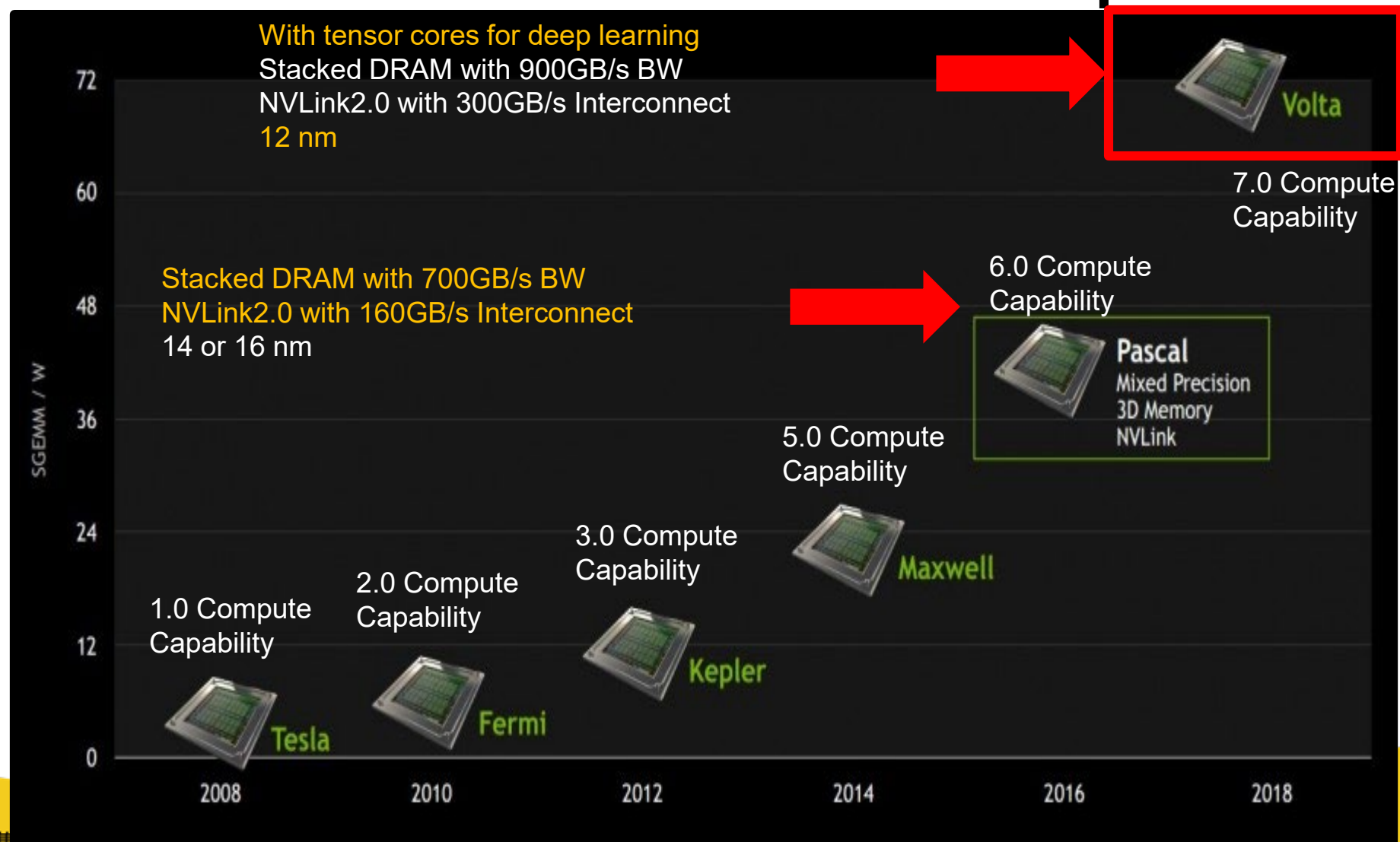


- Expose the horse power of GPUs for general purpose computations
  - Exploit **data parallelism** for solving embarrassingly parallel tasks and numeric computations
  - Users across science & engineering disciplines are achieving **100x or better speedups** on GPUs
- Programmable
  - Early GPGPU: using the libraries in computer graphics, such as OpenGL or DirectX, to perform the tasks other than the original hardware designed for.
  - Now **CUDA and openCL** provides an extension to C and C++ that enables parallel programming on GPUs

# NVIDIA CUDA-Enabled GPUs Products

Architecture & Compute Capability		CUDA-Enabled NVIDIA GPUs			HPC (double precision)
 Volta Architecture (compute capabilities 7.x)	<b>Deep learning Inference</b>	<b>Visualization (single precision)</b>			Tesla V Series <b>V100</b>
	Pascal Architecture (compute capabilities 6.x)	Tegra X2, Jetson TX2	GeForce 1000 Series <b>GTX 1080</b>	Quadro P Series <b>P6000</b>	Tesla P Series <b>P100</b>
	Maxwell Architecture (compute capabilities 5.x)	Tegra X1 Jetson TX2	GeForce 900 Series	Quadro M Series	Tesla M Series
	Kepler Architecture (compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series
Applications		 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center

# NVIDIA GPU Architecture Roadmap



# CUDA SDK Device Query

- `deviceQuery.cpp`

```
Device 0: "Tesla M2090"
  CUDA Driver Version / Runtime Version      5.0 / 5.0
  CUDA Capability Major/Minor version number: 2.0
  Total amount of global memory:              5375 MBytes (5636554752 bytes)
  (16) Multiprocessors x ( 32) CUDA Cores/MP: 512 CUDA Cores
  GPU Clock rate:                            1301 MHz (1.30 GHz)
  Memory Clock rate:                         1848 Mhz
  Memory Bus Width:                          384-bit
  L2 Cache Size:                             786432 bytes
  Max Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536,65535), 3D=(16384,16384,16384)
  Max Layered Texture Size (dim) x layers    1D=(16384) x 2048, 2D=(16384,16384) x 1
  Total amount of constant memory:            65536 bytes
  Total amount of shared memory per block:    49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                 32
  Maximum number of threads per multiprocessor: 1536
  Maximum number of threads per block:        1024
  Maximum sizes of each dimension of a block: 1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:  65535 x 65535 x 65535
```



# GPU Compute Capability

- Programming ability of a GPU device

Feature support (unlisted features are supported for all compute capabilities)	Compute capability (version)							
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	5.0
Integer atomic functions operating on 32-bit words in global memory	No	Yes						
atomicExch() operating on 32-bit floating point values in global memory								
Integer atomic functions operating on 32-bit words in shared memory	No	Yes						
atomicExch() operating on 32-bit floating point values in shared memory								
Integer atomic functions operating on 64-bit words in global memory								
Warp vote functions								

Technical specifications	Compute capability (version)						
	1.0	1.1	1.2	1.3	2.x 3.0	3.5	5.0
Maximum dimensionality of grid of thread blocks	2				3		
Maximum x-, y-, or z-dimension of a grid of thread blocks	65535				$2^{31}-1$		
Maximum dimensionality of thread block	3						
Maximum x- or y-dimension of a block	512				1024		
Maximum z-dimension of a block	64						
Maximum number of threads per block	512				1024		
Warp size	32						

# CUDA Toolkits

- Software Development Kit(SDK) for CUDA Programming
  - The CUDA-C and CUDA-C++ compiler, nvcc
  - Tools: IDE, Debugger, Profilers, Utilities
  - Library: BLAS, CUDA Device Runtime, FFT, ...
  - Sample Code
  - Documentation

CUDA SDK Version	Compute Capability	Architecture
6.5	1.X	Tesla ~ Maxwell
7.5	2.0-5.x	Fermi ~ Maxwell
8.0	2.0-6.x	Fermi ~ Pascal
9.0	3.0-7.x	Kepler ~ Volta
12.2	5.0-9.0	>= Maxwell

# Outline

- Heterogeneous Computing & GPU Intro
- GPU Memory Hierarchy & Execution Model
- CUDA Programming Model & API
- Coding Example & Share Memory Optimization



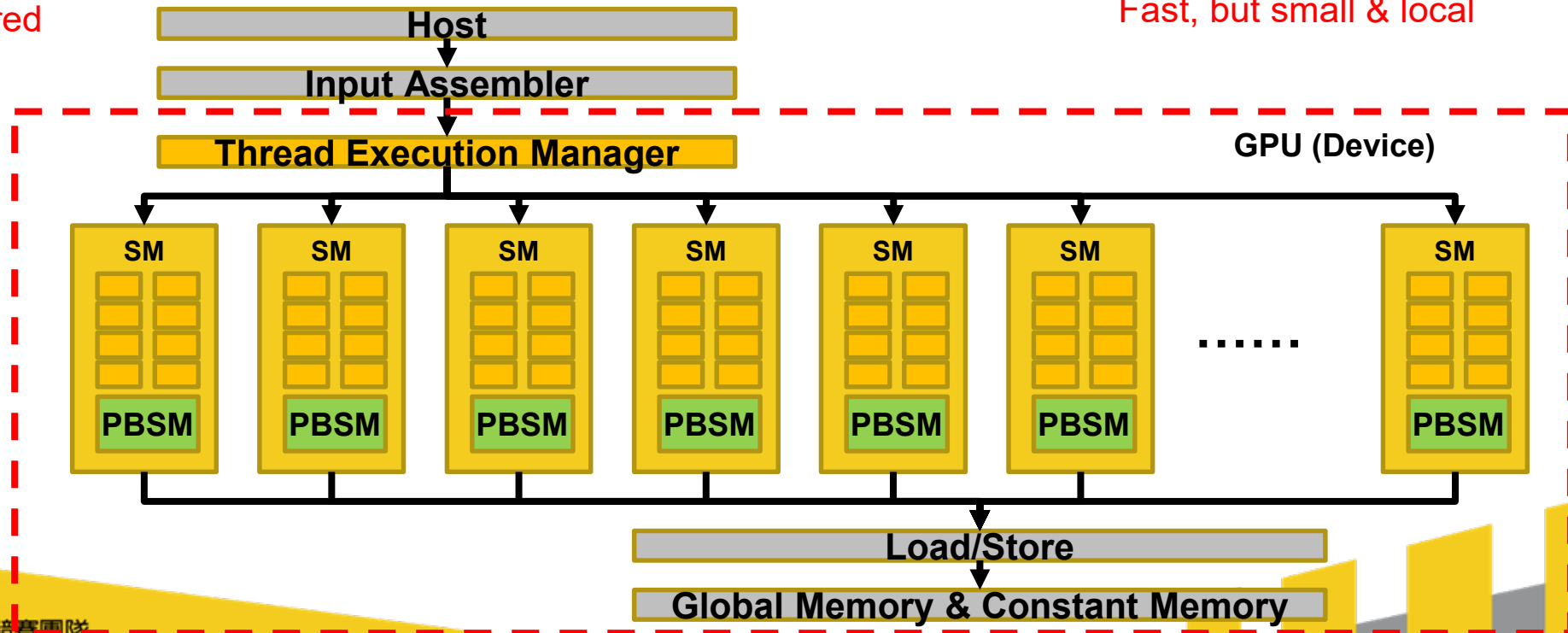


# GPU Architecture

- Consist of multiple stream multi-processors (SM)
- Memory hierarchic:
  - global memory → PBSM/shared memory → local register

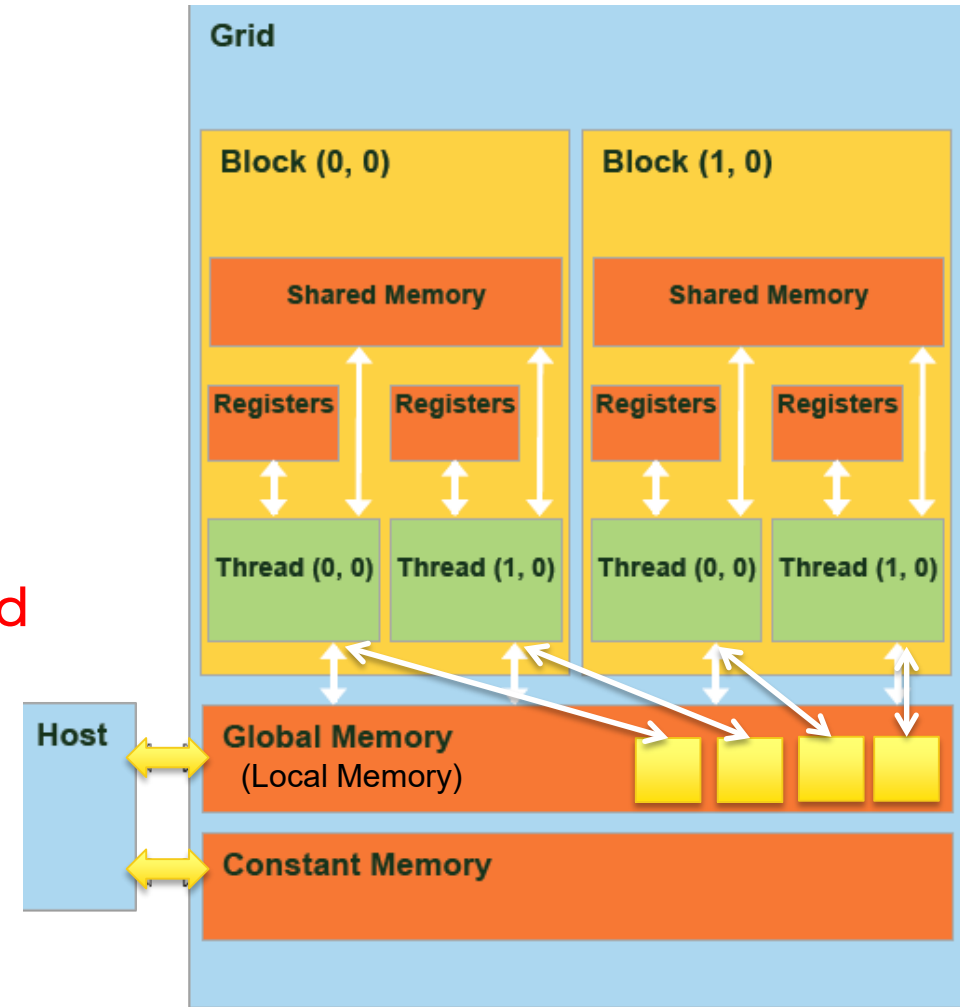
Slow, but large & shared

Fast, but small & local



# GPU Memory Hierarchy

- Registers
  - Read/write per-thread
  - **Low latency & High BW**
- Shared memory
  - Read/write **per-block**
  - Similar to register performance
- Global/Local memory (DRAM)
  - **Global is per-grid & Local is per-thread**
  - High latency & Low BW
  - **Not cached**
- Constant memory
  - **Read only** per-grid
  - **Cached**



# Execution Model

Threads are executed by scalar processor

Thread blocks are executed on SM

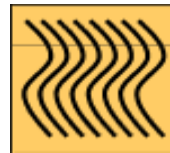
Several concurrent thread block can reside on one SM

A kernel is launched as a grid of thread blocks

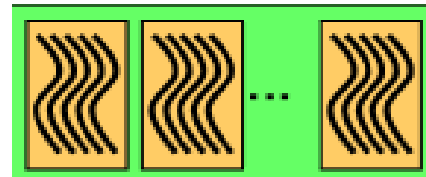
## Software



Thread



Thread block



Grid

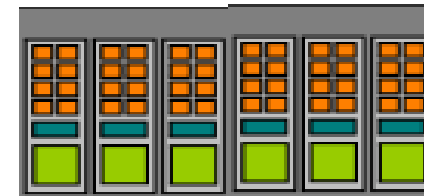
## Hardware



Scalar processor

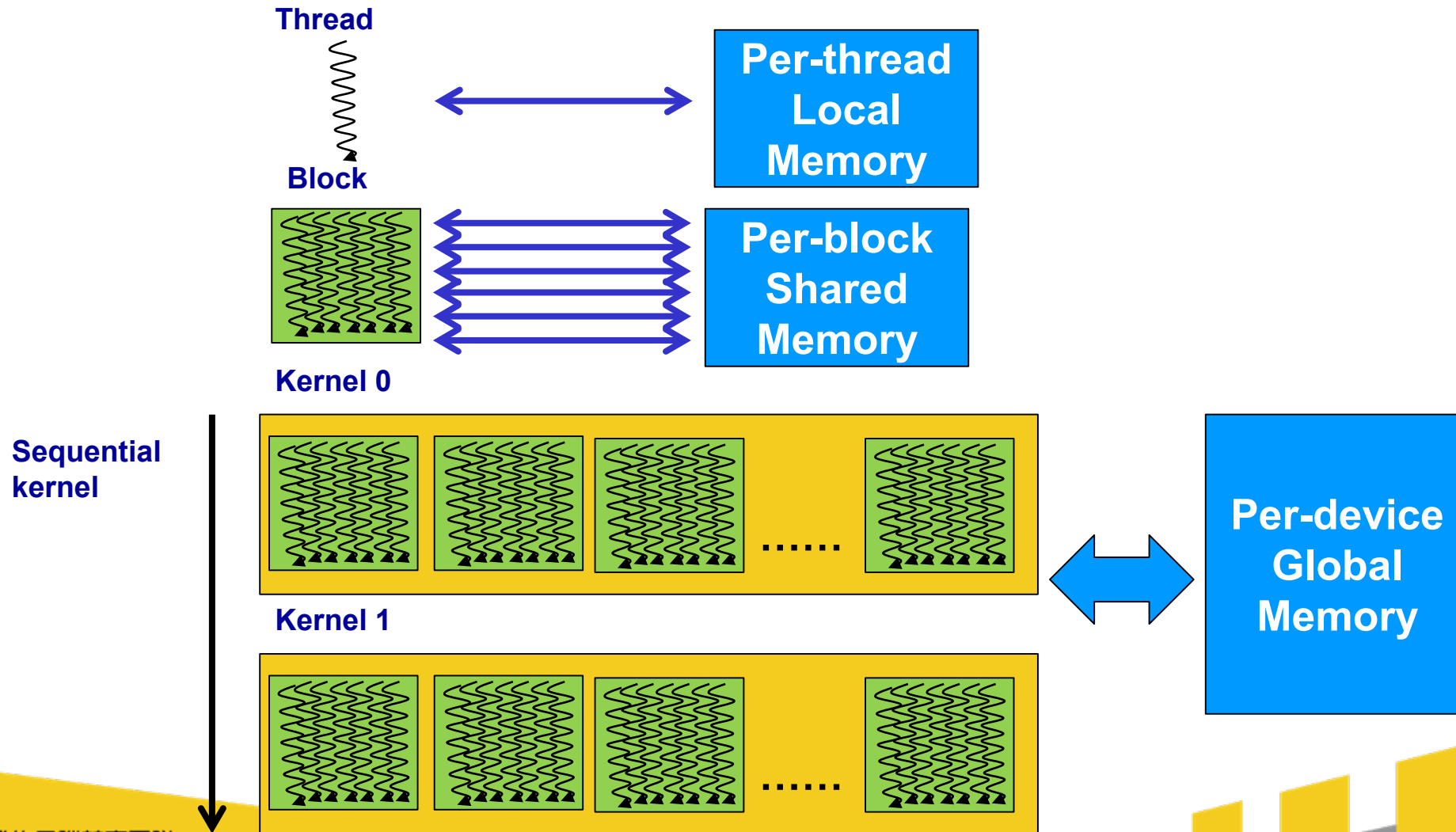


Stream Processor (SM)



GPU device

# Memory Hierarchy

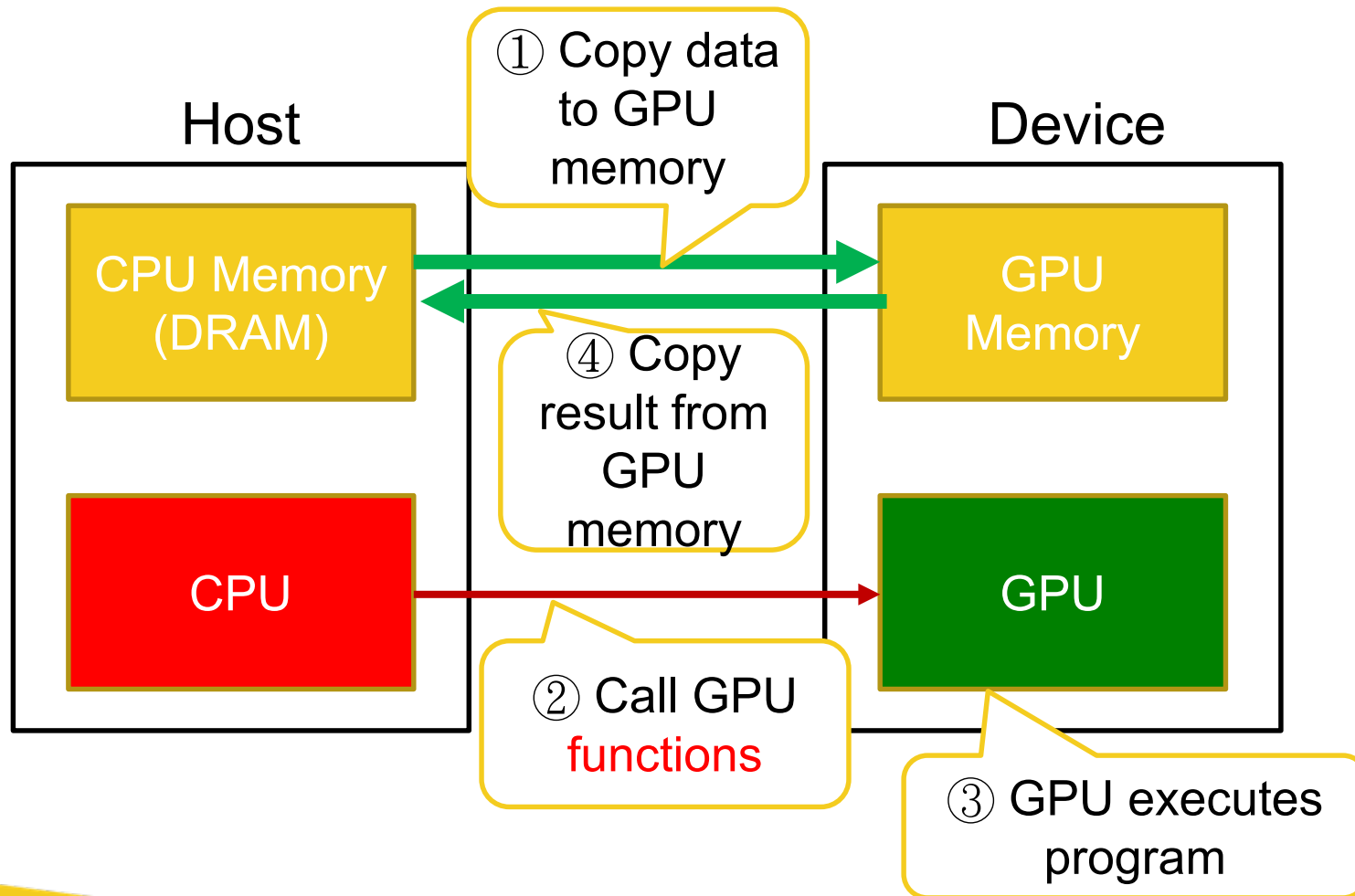


# Outline

- Heterogeneous Computing & GPU Intro
- GPU Memory Hierarchy & Execution Model
- **CUDA Programming Model & API**
- Coding Example & Share Memory Optimization

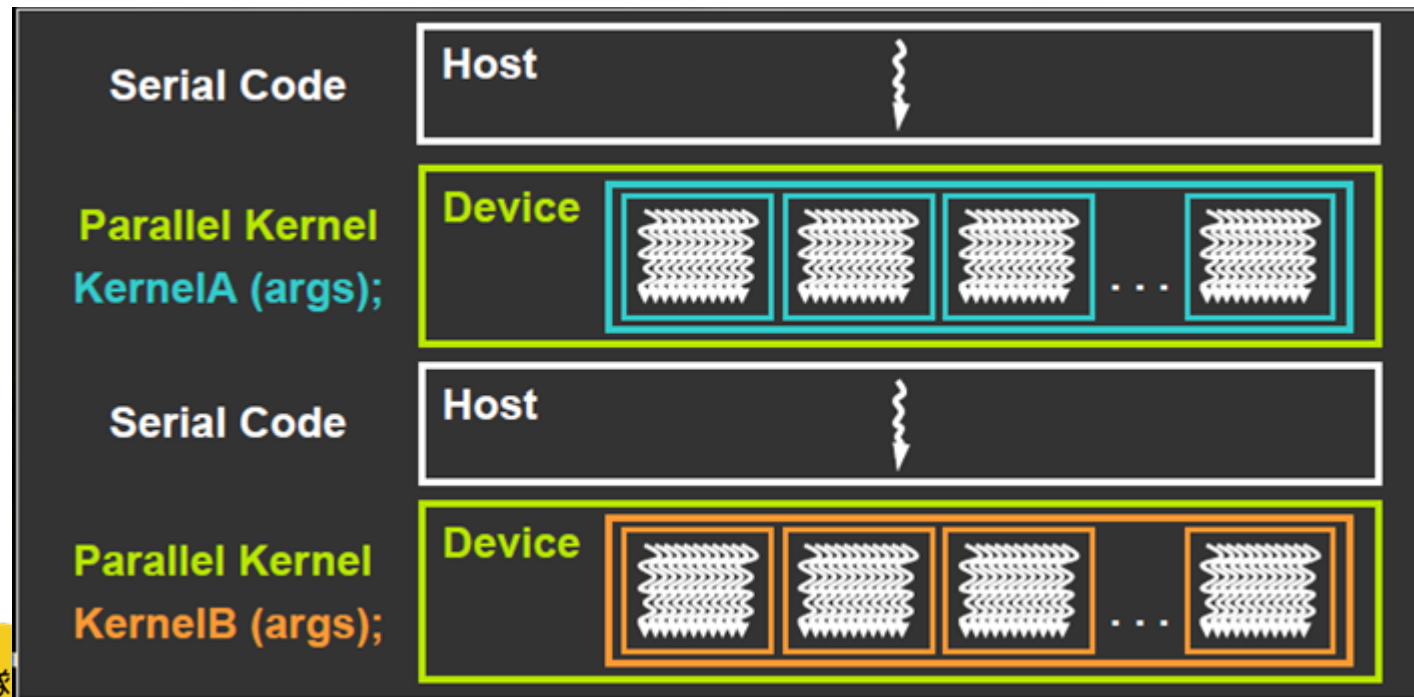


# GPU program flow



# CUDA Programming Model

- CUDA = serial program with parallel kernels, all in C
  - Serial C code executes in a host thread (i.e. CPU thread)
  - Parallel kernel C code executes in many device threads across multiple processing elements (i.e. GPU threads)



# CUDA Program Framework

GPU code  
(parallel)

CPU code  
(serial or  
parallel if  
p-thread/  
OpenMP/T  
BB/MPI is  
used.)

```
#include <cuda_runtime.h>

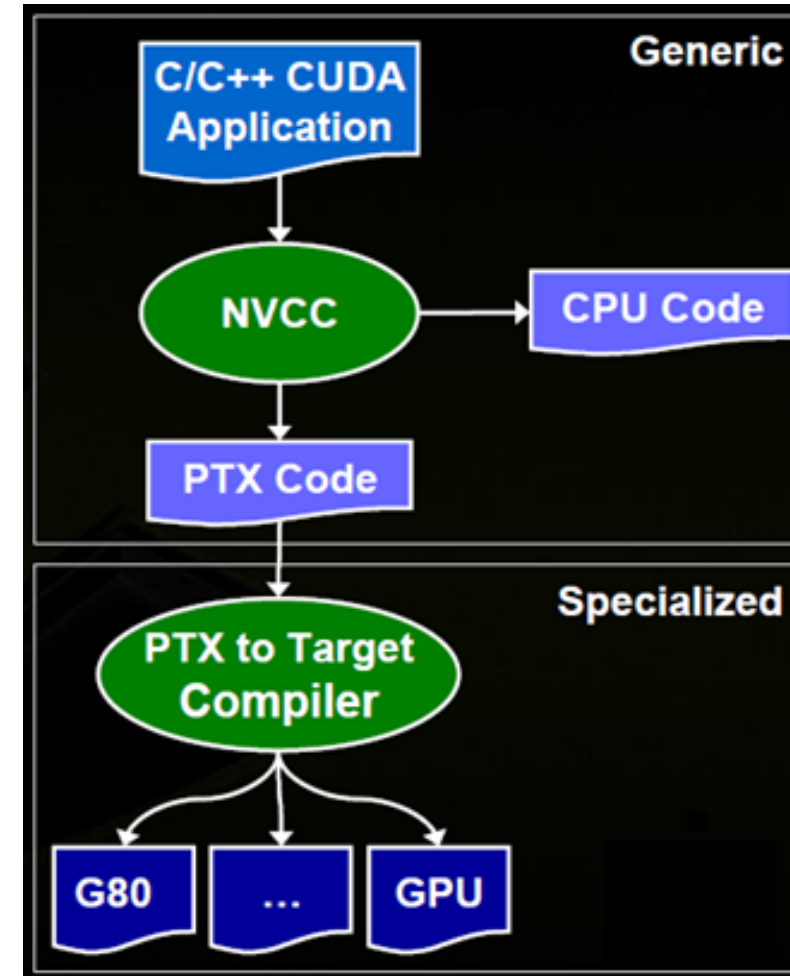
__global__ void my_kernel(...) {
    ...
}

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



# Program Compilation

- Any source file containing CUDA language must be compiled with NVCC
  - NVCC separates code running on the host from code running on the device
- Two-stage compilation:
  - Virtual ISA
    - PTX: Parallel Threads eXecutions
  - Device-specific binary object



# Global (Device) memory operations

- Three functions:
    - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
    - Similar to the C's `malloc()`, `free()`, `memcpy()`
1. `cudaMalloc(void **devPtr, size_t size)`
    - `devPtr`: return the address of the allocated device memory
    - `size`: the allocated memory size (bytes)
  2. `cudaFree (void *devPtr)`
  3. `cudaMemcpy( void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)`
    - `count`: size in bytes to copy

# cudaMemcpyKind

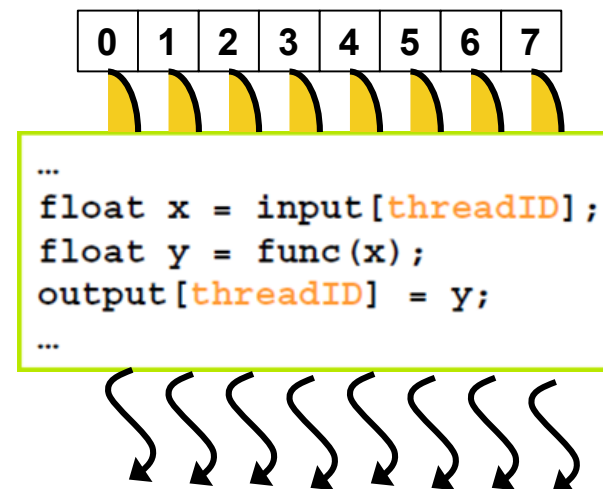
- one of the following four values

cudaMemcpyKind	Meaning	dst	src
cudaMemcpyHostToHost	Host → Host	host	host
cudaMemcpyHostToDevice	Host → Device	device	host
cudaMemcpyDeviceToHost	Device → Host	host	device
cudaMemcpyDeviceToDevice	Device → Device	device	device

host to host has the same effect as memcpy()

# Kernel = Many Concurrent Threads

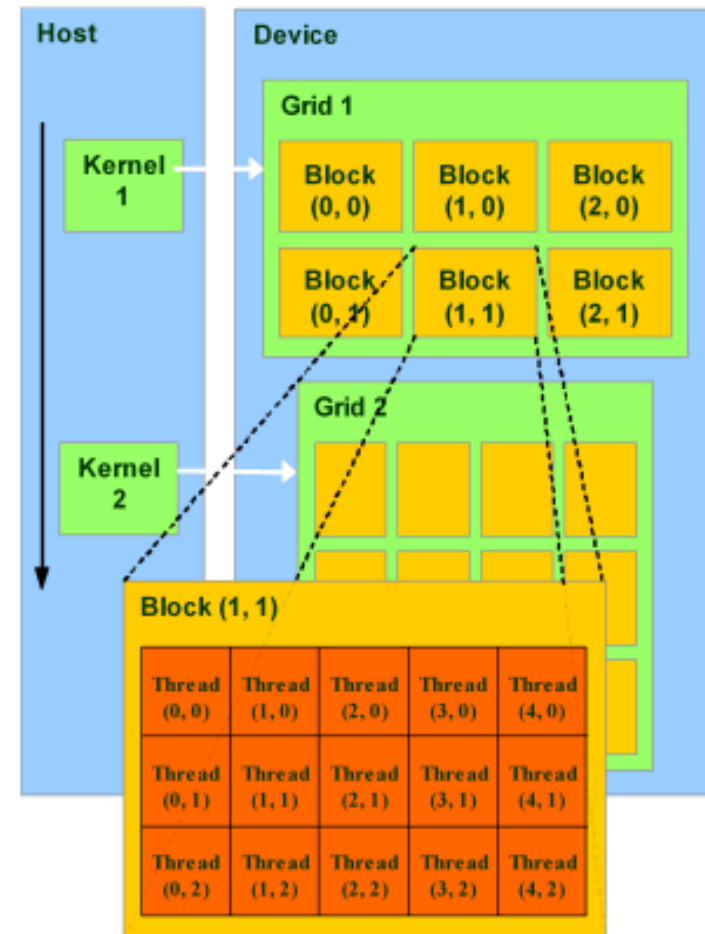
- One kernel is executed at a time on the device
- Many thread execute each kernel
  - Each thread executes the same code
  - ... on the different data based on its threadID
- CUDA thread might be
  - **Physical threads**
    - As on NVIDIA GPUs
    - GPU thread creation and context switching are essentially free
  - Or virtual threads
    - E.g. 1 CPU core might execute multiple CUDA threads



# Thread and Block IDs

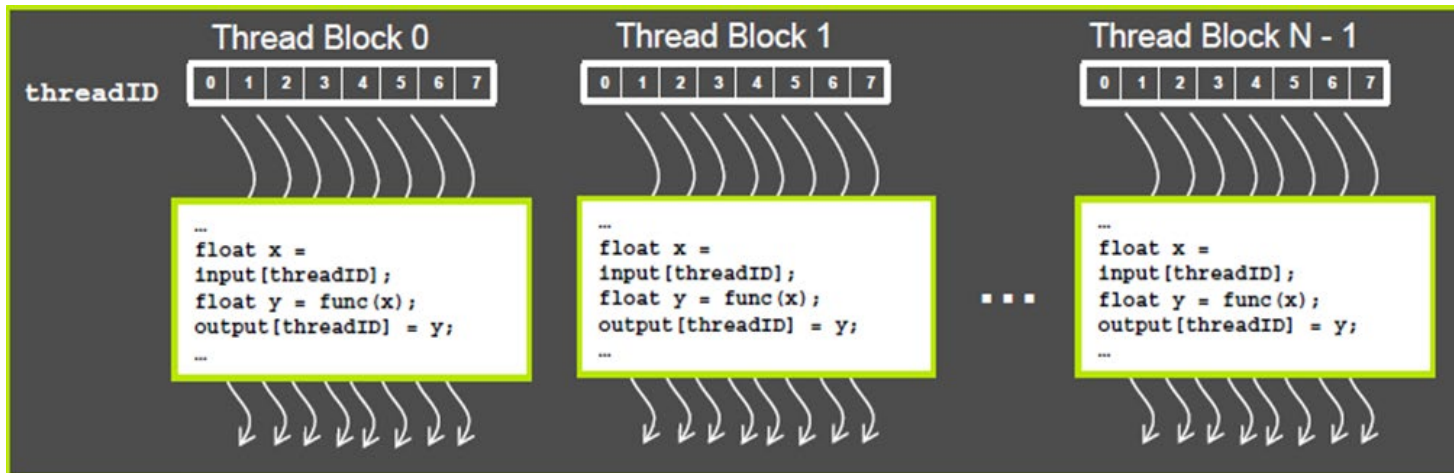
- Build-in device variables
  - `threadIdx`; `blockIdx`; `blockDim`; `gridDim`
- The index of threads and blocks can be denoted by a 3 dimensional struct
  - `dim3` defined in `vector_types.h`  

```
struct dim3 { x; y; z; };
```
- Example:
  - `dim3 grid(3, 2);`
  - `dim3 blk(5, 3); // n x n x 1`
  - `my_kernel<<< grid, blk >>>();`
- Each thread can be uniquely identified by a tuple of index (x,y) or (x,y,z)



# Hierarchy of Concurrent Threads

- Threads are grouped into thread blocks
  - Kernel = grid of thread blocks

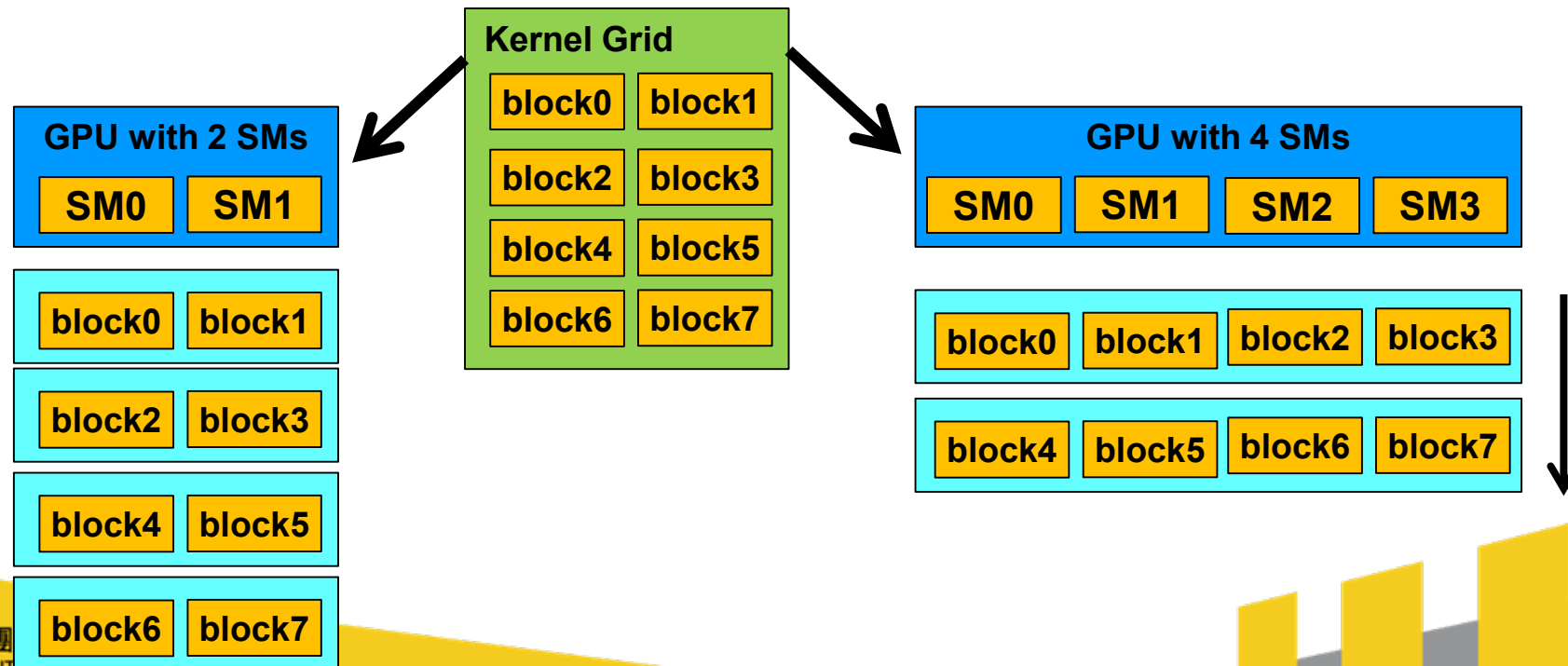


- By definition, threads in the same block may be synchronized with barriers, but not between blocks

```
scratch[threadID] = begin[threadID];
__syncthreads();
int left = scratch[threadID - 1];
```

# Block Level Scheduling

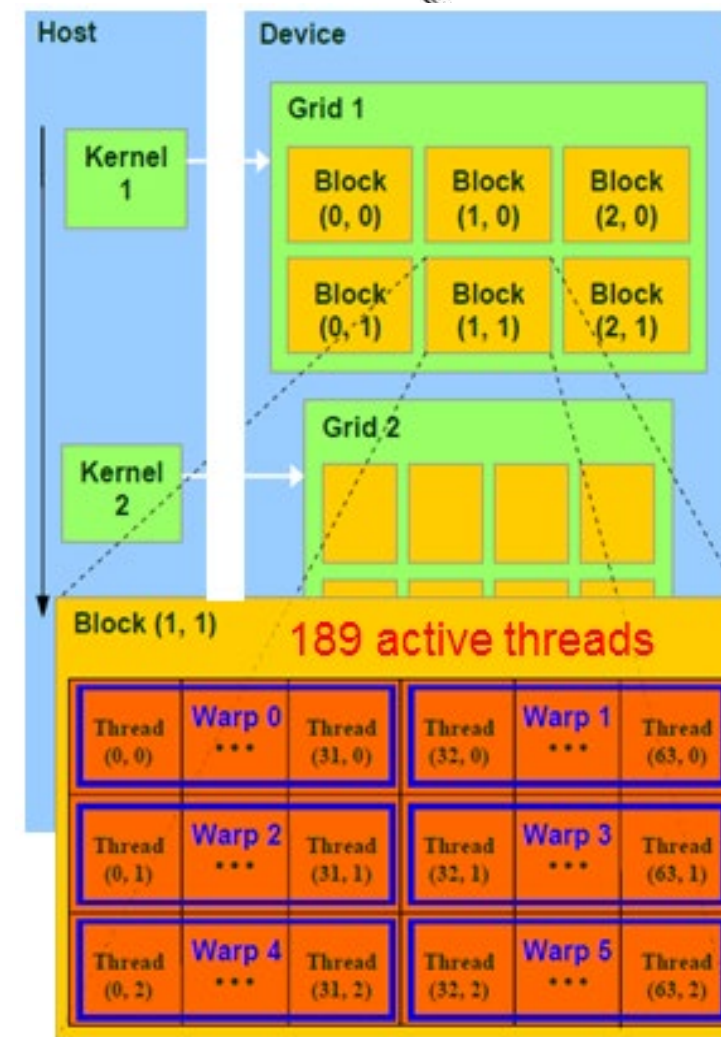
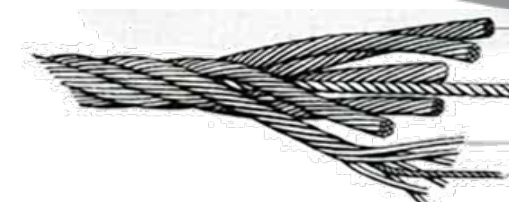
- Blocks are independent to each other to give scalability
  - A kernel scales across any number of parallel cores by scheduling blocks to SMs
- No global synchronization among blocks





# Warp

- Inside the SM, threads are launched in groups of 32, called **warps**
    - Warps share the control part (**warp scheduler**)
    - At any time, **only one warp is executed per SM**
    - **Threads in a warp will be executing the same instruction (SIMD)**
  - In other words ...
    - Threads in a wrap execute **physically in parallel**
    - Warps and blocks execute **logically in parallel**
- ➔ **Kernel needs to sync threads within a block**





# Warp Scheduler



SM multithreaded  
Warp scheduler

time

warp 8 instruction 11

warp 1 instruction 42

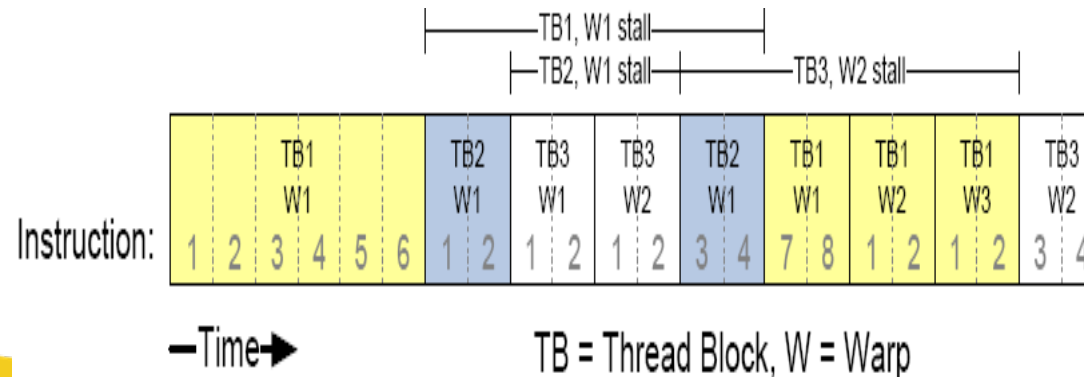
warp 3 instruction 95

⋮

warp 8 instruction 12

warp 3 instruction 96

- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Warps are switched when memory stalls
  - Eligible Warps are selected for execution on prioritized scheduling
  - All threads in a Warp execute the same instruction when selected



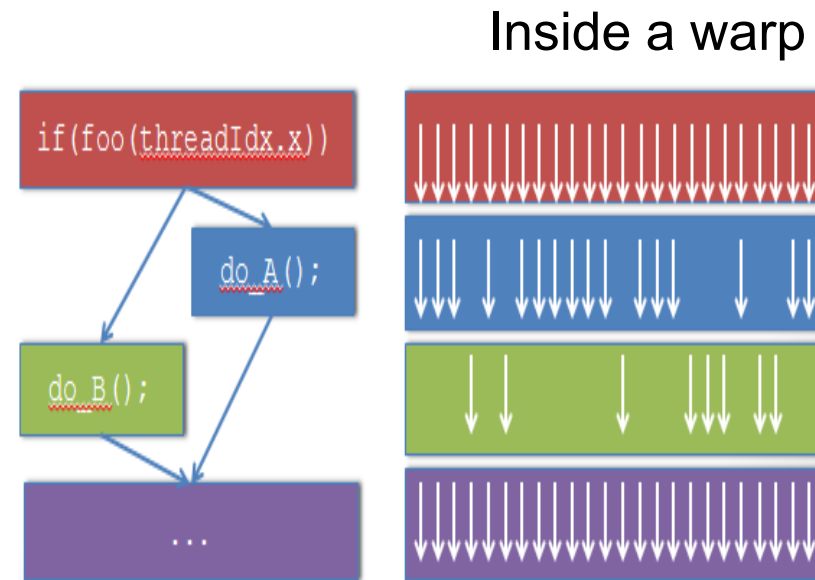
# Warp Divergence

- What if different threads **in a warp** need to do different things:
  - Including any flow control instruction (if, switch, **do, for, while**)

```
if (foo(threadIdx.x)) {  
    do_A();  
} else {  
    do_B();  
}
```

- Different execution paths within a warp are serialized
  - Predicated instructions which are carried out only if logical flag is true
  - All threads compute the logical predicate and two predicated instructions/statements

➔ **Potential large lost of performance**




# Outline

- Heterogeneous Computing & GPU Intro
- GPU Memory Hierarchy & Execution Model
- CUDA Programming Model & API
- Coding Example & Share Memory Optimization



# Example: add 2 numbers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}  
  
int main(void) {  
    int ha=1,hb=2,hc;  
    add<<<1,1>>>(&ha, &hb, &hc);  
    printf("c=%d\n",hc);  
    return 0;  
}
```



- This does not work!!
- `int ha, hb, hc` are in the host memory (DRAM), which cannot be used by device (GPU).
- We need to allocate variables in “device memory”.

# The correct main()

```
int main(void) {  
    int a=1, b=2, c; // host copies of a, b, c  
    int *d_a, *d_b, *d_c; // device copies of a, b, c  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **)&d_a, sizeof(int));  
    cudaMalloc((void **)&d_b, sizeof(int));  
    cudaMalloc((void **)&d_c, sizeof(int));  
    // Copy inputs to device  
    cudaMemcpy(d_a, &a, sizeof(int), cudaMemcpyHostToDevice);  
    cudaMemcpy(d_b, &b, sizeof(int), cudaMemcpyHostToDevice);  
    // Launch add() kernel on GPU  
    add<<<1,1>>>(d_a, d_b, d_c);  
    // Copy result back to host  
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);  
    // Cleanup  
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);  
    return 0;  
}
```

# Example: add 2 vectors

- Let's first look at the sequential code!

```
// function definition
void VecAdd(int N, float* A, float* B, float* C)
{
    for(int i = 0; i<N; i++)
        C[i] = A[i] + B[i];
}

int main()
{ ...
  VecAdd (N, Ah, Bh, Ch);
  ...
}
```

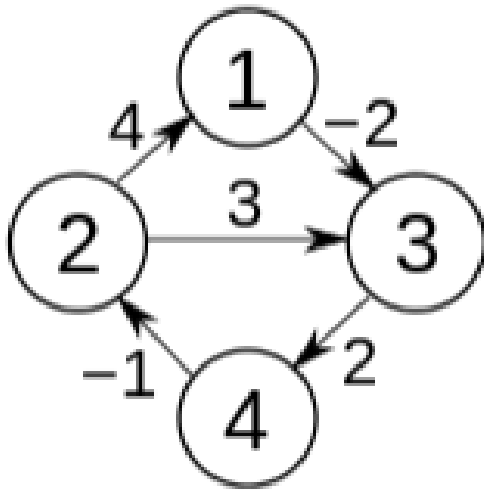
# Parallel CUDA code

- Use `threadIdx.x` as the index of the arrays
  - Each thread processes 1 addition, for the elements indexed at `threadIdx.x`.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{ ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(Ah, Bh, Ch); ...
}
```

# Example: All-Pair-Shortest-Path

- Given a weighted directed graph  $G(V, E, W)$ , where  $|V| = n$ ,  $|W|=m$ , and  $W>0$ , find the shortest path of all pairs of vertices  $(v_i, v_j)$ .
- Example:



0	INF	-2	INF
4	0	3	INF
INF	INF	0	2
INF	-1	INF	0

Initial  
weight

0	-1	-2	0
4	0	2	4
5	1	0	2
3	-1	1	0

Final  
result



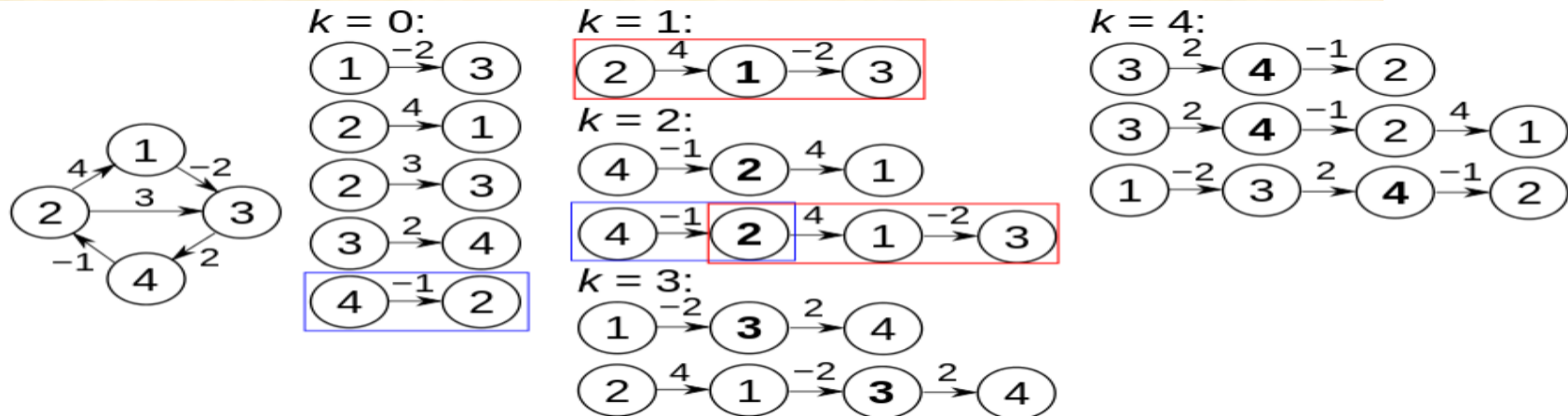
# Floyd-Warshall (Sequential code)

Floyd-Warshall (G, W)

```

{  n ← |V|
   D(0) ← W
   for k = 1 to n do
     for i = 1 to n do
       for j = 1 to n do
         if D(k-1)[i, j] > D(k-1)[i, k] + D(k-1)[k, j]
           then D(k)[i, j] ← D(k-1)[i, k] + D(k-1)[k, j]
         else D(k)[i, j] ← D(k-1)[i, j]
   return D(n)
}
    
```

How to parallelize it?



# Implementation 1

- 1 block and  $n$  threads.
- Thread  $i$  updates the SP for vertex  $i$ .

```
__global__ void FW_APSP(int k, int D[n][n]) {  
    int i = threadIdx.x;  
    for (int j = 0; j < n; j++)  
        if (D[i][j] > D[i][k] + D[k][j])  
            D[i][j] = D[i][k] + D[k][j];  
}  
  
int main() { ...  
    for (int k = 0; k < n; k++)  
        FW_APSP<<<1, n>>>(k, D);  
}
```

Simple! But can it be faster ?

# Implementation 2

- Each thread updates one pair of vertices
  - Increase parallelism from  $n$  to  $n^2$

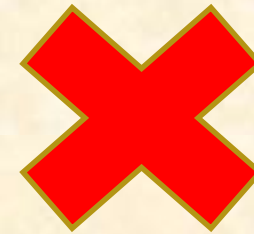
```
__global__ void FW_APSP(int k, int D[n][n]) {  
    int i = threadIdx.x;  
    int j = threadIdx.y;  
    if (D[i][j] > D[i][k] + D[k][j])  
        D[i][j] = D[i][k] + D[k][j];  
}  
  
int main() { ...  
    dim3 threadsPerBlock(n, n);  
    for (int k = 0; k < n; k++)  
        FW_APSP<<<1, threadsPerBlock >>>(k, D);  
}
```

How about the for-loop of k?

# Implementation 3

- It is a synchronous computation
  - There are data dependency on k...

```
__global__ void FW_APSP(int D[n][n]) {  
    int i = threadIdx.x;  
    int j = threadIdx.y;  
    for (int k = 0; k < n; k++)  
        if (D[i][j] > D[i][k] + D[k][j])  
            D[i][j] = D[i][k] + D[k][j];  
}  
int main() { ...  
    dim3 threadsPerBlock(n, n);  
    FW_APSP<<<1, threadsPerBlock >>>(D);  
}
```

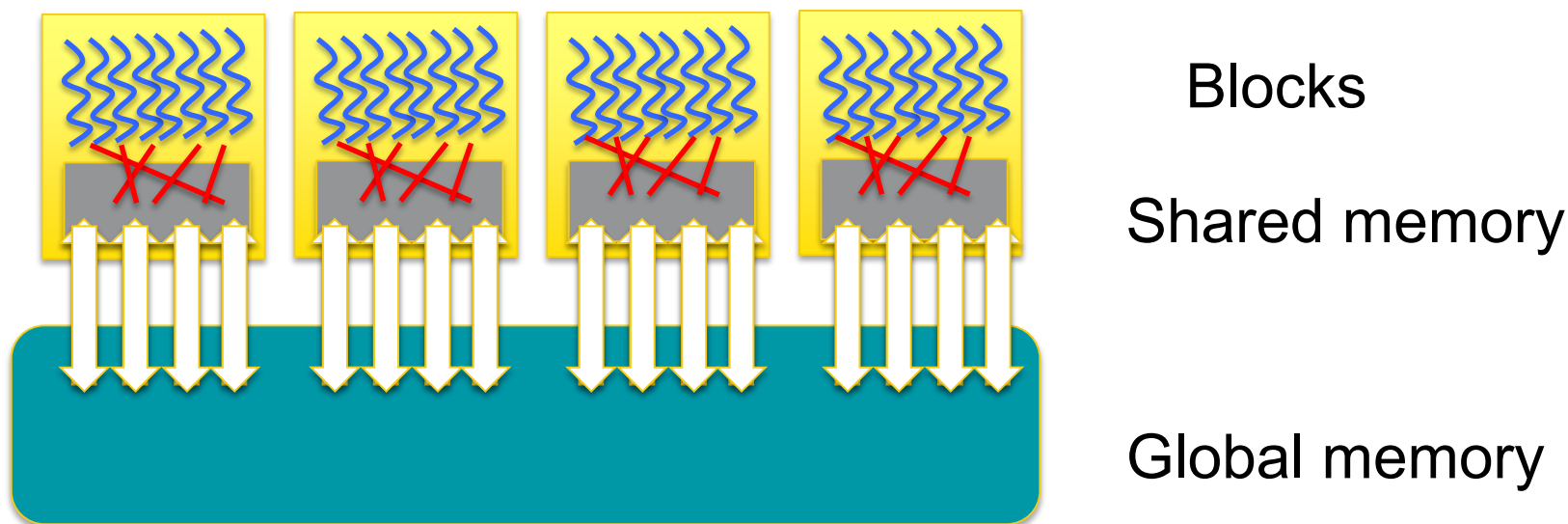


# Shared Memory: Programmable Cache

- Programmable cache!!
  - Almost as fast as registers
- Scope: shared by all the threads in a block.
  - The threads in the same block can communicate with each other through the shared memory.
  - Threads in different blocks can only communicate with each other through global memory.
- Limited size: 64kB or 96kB per thread block

# General Strategy

1. Load data from global memory to shared memory
2. Process data in the shared memory
3. Write data back from shared memory to global memory



# APSP Parallel Implementation Revisit

- Use  $n \times n$  threads.
- Each updates the shortest path of one pair vertices
- Use global memory to store the matrix D.

```
__global__ void FW_APSP(int k, int D[n][n]) {  
    int i = threadIdx.x;  
    int j = threadIdx.y;  
    if (D[i][j] > D[i][k] + D[k][j])  
        D[i][j] = D[i][k] + D[k][j];  
}  
int main() { ...  
    dim3 threadsPerBlock(n, n);  
    for (int k = 0; k < n; k++)  
        FW_APSP<<<1, threadsPerBlock >>>(k, D);  
}
```

6 global  
memory  
access

# Using Shared Memory Optimization

- This way of using shared memory is called **dynamic allocation of shared memory**, whose **size is specified in the kernel launcher**.

```
FW_APSP<<<1, n*n, n*n*sizeof(int)>>> (...);
```

- The third parameter is the size of shared memory.

```
extern __shared__ int S[][];  
__global__ void FW_APSP(int k, int D[n][n]) {  
    int i = threadIdx.x;  
    int j = threadIdx.y;  
    S[i][j]=D[i][j]; // move data to shared memory  
    __syncthreads();  
    // do computation  
    if (S[i][j]>S[i][k]+S[k][j])  
        D[i][j]= S[i][k]+S[k][j];  
}
```

**ONLY 2**  
global mem  
access



# Load Everything to Shared Memory

- `Matrix_Mul<<<1, N, 2*N*N>>>(A, B, C, N);`
  - The third parameter is the size of shared memory.

```
extern __shared__ int S[];
inline int Addr(int matrixIdx, int i, int j, int N) {
    return (N*N*matrixIdx + i*N+ j);
}
__global__ void Matrix_Mul(int* A, int* B, int* C, int* N) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    //move data to shared memory
    S[Addr(0, i, j, N)]=A[Addr(0, i, j, N)];
    S[Addr(1, i, j, N)]=B[Addr(0, i, j, N)];
    __syncthreads();
    // do computation
    for(int k=0; k<N; k++)
        C[Addr(1, i, j, N)]=S[Addr(0, j, k, N)]*S[Addr(0, k, j, N)];
}
```

# Reference

- Nvidia
  - [NVIDIA, CUDA C++ Best Practices Guide](#)
  - [CUDA Tutorial](#)
    - [NVIDIA CUDA Library Documentation](#)
    - [NVIDIA Advanced CUDA Webinar Memory Optimizations](#)
    - [Mark Harris, NVIDIA Developer Technology](#)
- Parallel Prog. Class [Recorded video](#)
  - Chap6: Heterogeneous Computing
  - Chpa7: CUDA Intro
  - Chap8: GPU Architecture
  - Chap9: CUDA Optimization
    - With a complete optimization example at the end

