

CS 610: GPU Architecture and CUDA Programming

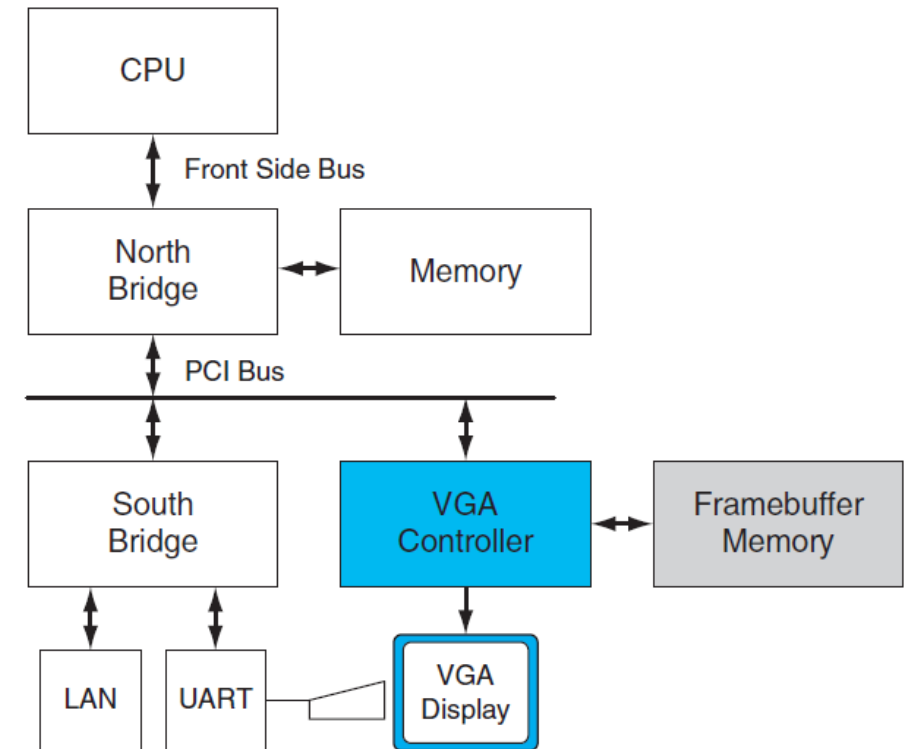
Swarnendu Biswas

Semester 2022-2023-I
CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Rise of GPU Computing

- Popularity of graphical OS in late 80s created a market for a new compute device
 - 2D display accelerators offered hardware-assisted bitmap operations
- Silicon Graphics popularized use of 3D graphics
 - Released OpenGL as a programming interface to its hardware
- Popularity of first-person games in mid-90s was the final push

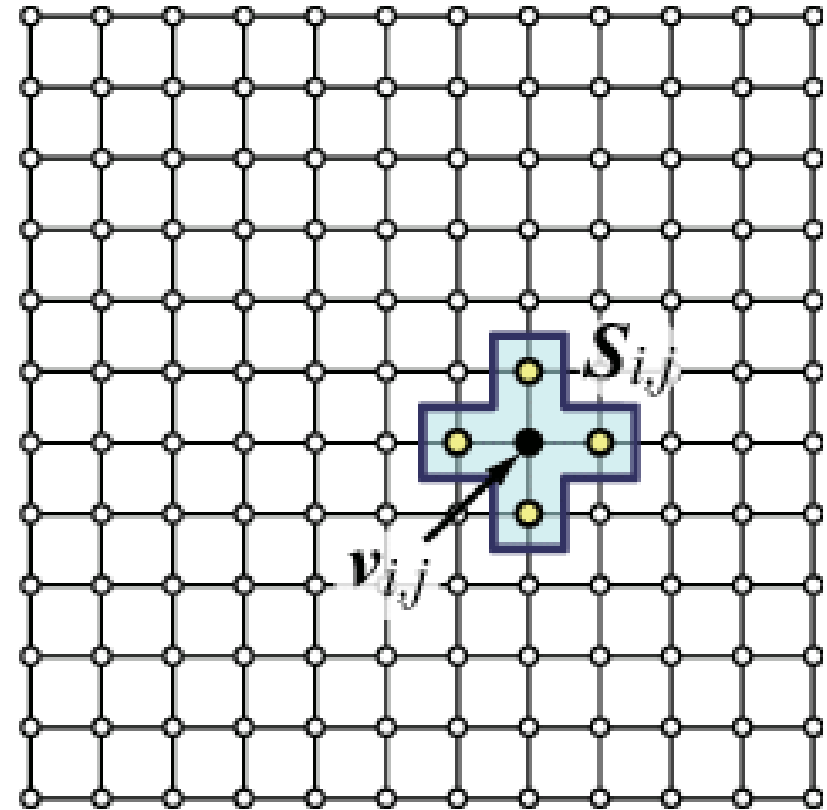


Rise of GPU Computing

- Pixel shaders were used to produce a color for a pixel on screen
 - It uses the (x,y) coordinates, input colors, texture coordinates and other attributes as inputs
 - Input is a set of vertices, an input assembler forms shapes (i.e., lines, triangles, and polygons), which are processed by shaders to decide on illumination
- NVIDIA's GeForce 3 series in 2001 implemented the DirectX 8.0 standard from Microsoft
 - Hardware implementations for fixed-point functions

Need for GPU Computing Support

- Many real-world applications are compute-intensive and data-parallel
 - They need to process a lot of data, mostly floating-point operations
 - For example, real-time high-definition graphics applications such as your favorite video games
 - Iterative kernels which update elements according to some **fixed pattern called a stencil**



Rise of GPU Computing

- Researchers tricked GPUs to perform non-rendering computations
- Programming initial GPU devices for other purposes was very convoluted
- Programming model was very restrictive
 - Limited input colors and texture units, writes to arbitrary locations, floating-point computations
- This spurred the need for a highly-parallel computational device with high computational power and memory bandwidth
 - CPUs are more complex devices catering to a wider audience

Enter NVIDIA and CUDA

- NVIDIA released GeForce 8800 GTX in 2006 with CUDA architecture
 - General-purpose ALU and instruction set for general-purpose computation
 - IEEE compliance for single-precision floating-point arithmetic
 - Allowed arbitrary reads and writes to shared memory
- Introduced CUDA C and the toolchain for ease of development with the CUDA architecture

Rise of GPU Computing

- GPUs are now used in different applications
 - Game effects, computational science simulations, image processing and machine learning, linear algebra
- Several GPU vendors like NVIDIA, AMD, Intel, Qualcomm, and ARM

GPU Architecture

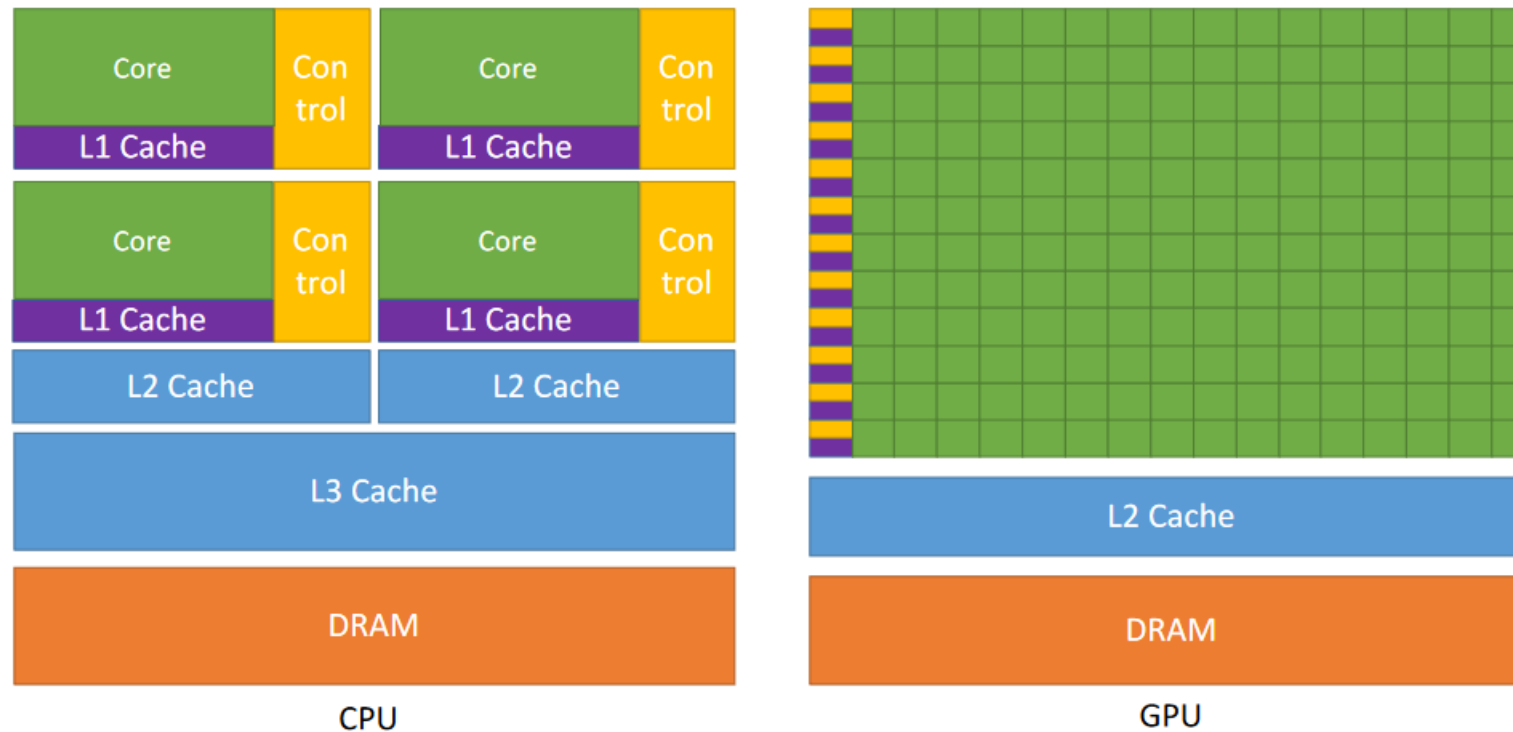
Philosophy and design goals

Key Insights in GPU Architecture

- GPUs are suited for compute-intensive data-parallel applications
 - The same program is executed for each data element
 - Less complex control flow
- Multi-core chip
 - SIMD execution within a single core (many ALUs performing the same instruction)
 - Multi-threaded execution on a single core (multiple threads executed concurrently by a core)

Key Insights in GPU Architecture

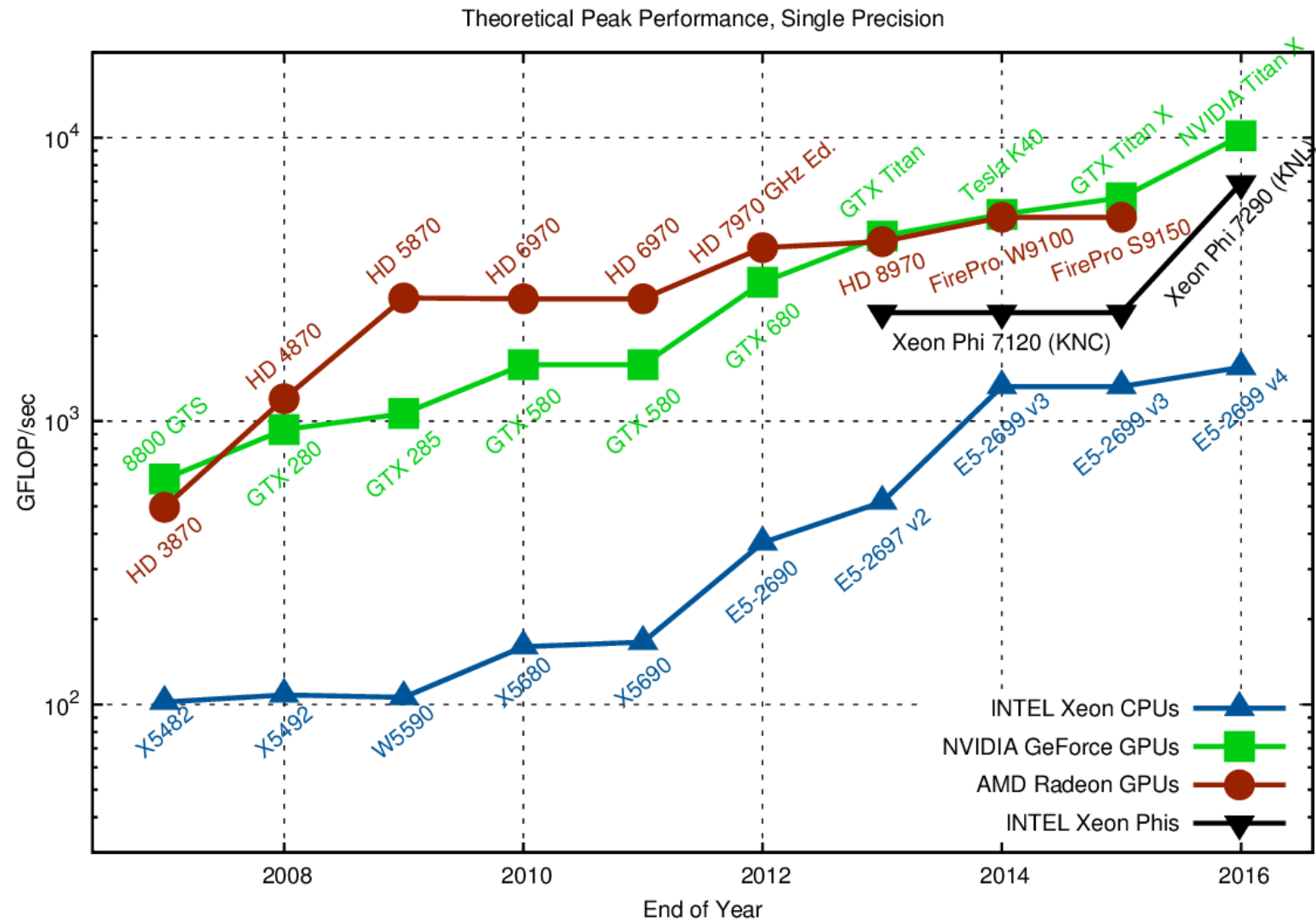
- Much more transistors or real-estate is devoted to computation rather than data caching and control flow



Key Insights in GPU Architecture

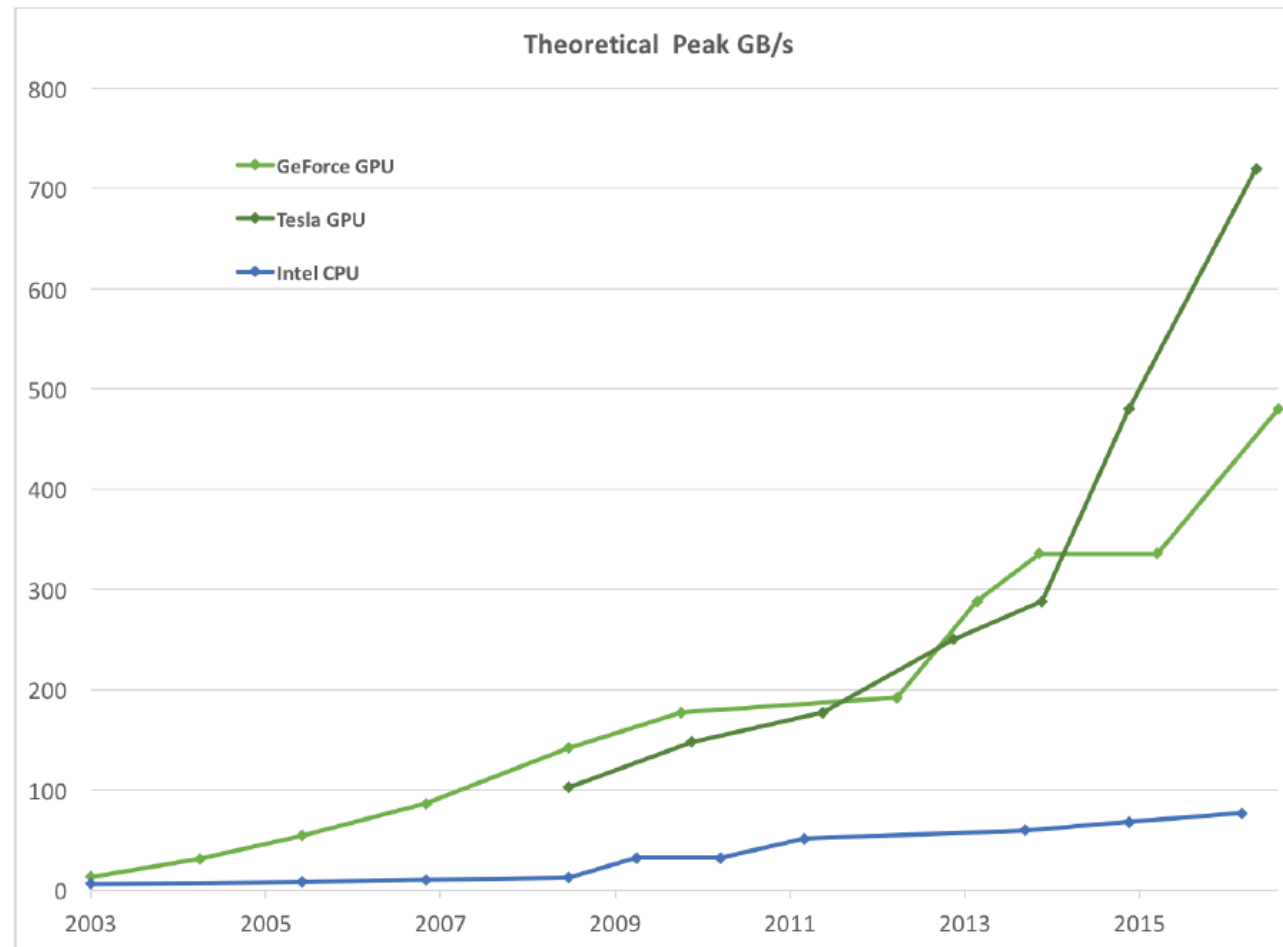
- GPUs do not reduce latency, they aim to **hide latency**
- The focus is on overall computing throughput rather than on the speed of an individual core
 - High arithmetic intensity to hide latency of memory accesses
 - Large number of schedulable units

Floating-Point Operations per Second for the CPU and GPU



<https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

Memory Bandwidth for CPU and GPU



High-end CPU-GPU Comparison

| | Xeon 8180M | Titan V |
|-----------------------|-------------------|----------------|
| Cores | 28 | 5120 (+ 640) |
| Active threads | 2 per core | 32 per core |
| Frequency | 2.5 (3.8) GHz | 1.2 (1.45) GHz |
| Peak performance (SP) | 4.1 TFlop/s | 13.8 TFlop/s |
| Peak mem. bandwidth | 119 GB/s | 653 GB/s |
| Maximum power | 205 W | 250 W |
| Launch price | \$13,000 | \$3000 |

Release dates
Xeon: Q3'17
Titan V: Q4'17



Compare GPU to CPU Architecture

- CPUs aim to reduce memory latency with increasingly large and complex memory hierarchy
- Disadvantages
 - The Intel I7-920 processor has some 8 MB of internal L3 cache, almost 30% of the size of the chip
 - Larger cache structures increases the physical size of the processor
 - Implies more expensive manufacturing costs and increases likelihood of manufacturing defects
- Effect of larger, progressively more inefficient caches ultimately results in higher costs to the end user

Advantages of a GPU



- Performance of Xeon 8180M and Titan V (based on peak values)
 - 3.4X operations executed per second compared to the CPU
- Main memory bandwidth
 - 5.5X bytes transferred per second compared to the CPU
- Cost- and energy-efficiency
 - 15X as much performance per dollar
 - 2.8X as much performance per watt
- GPU's higher performance and energy efficiency are due to different allocation of chip area
 - High degree of SIMD parallelism, simple in-order cores, less control/sync. logic, less cache/scratchpad capacity
 - SIMD is more energy-efficient than MIMD since a single instruction can launch many data operations
 - Simpler pipeline with no support for restartable instructions and precise exceptions

From FLOPS to FLOPS/Watt

- Exploiting hardware specialization can improve energy efficiency
- Moving to vector hardware, such as that found in GPUs, may yield up to 10X gain in efficiency by eliminating overheads of instruction processing
- For example, Apple A8 application processor devotes more die area to its integrated GPU than to central processor unit (CPU) cores
- Most energy-efficient supercomputers are now based on GPUs instead of only-CPU

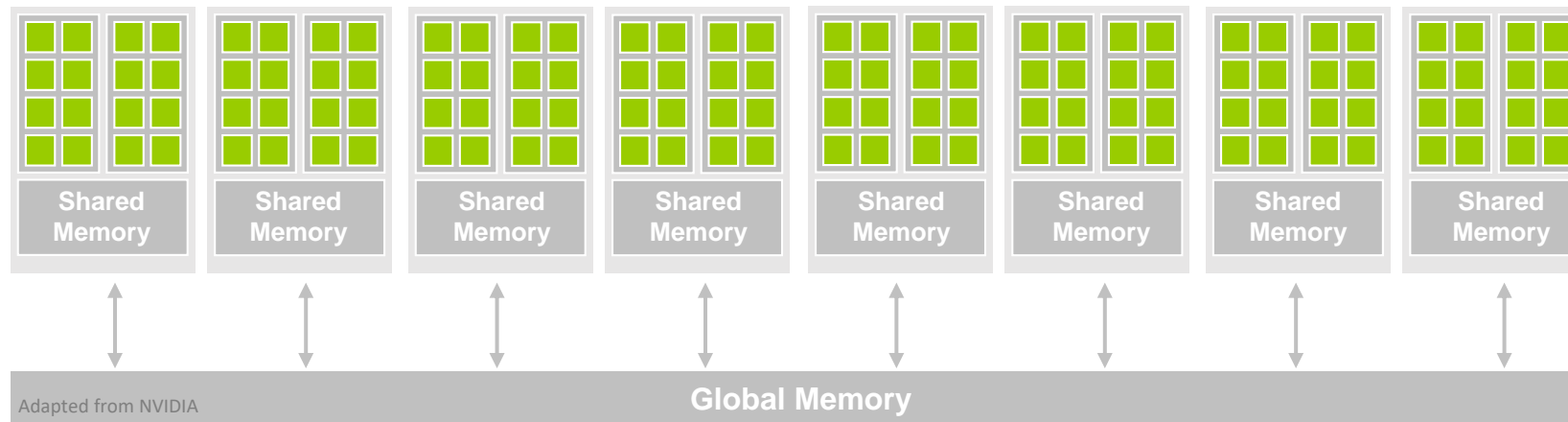


GPU Disadvantages

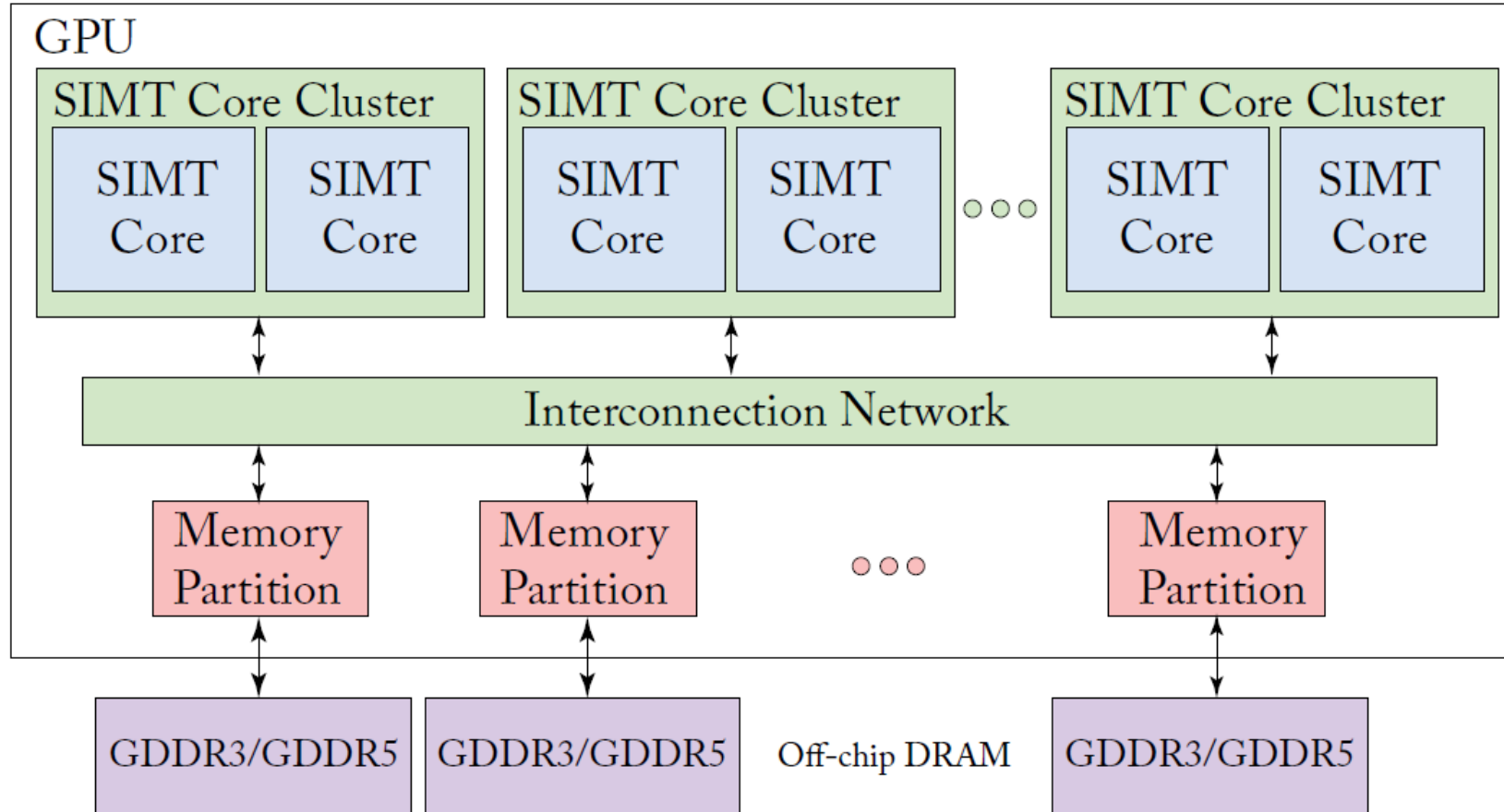
- Clearly, we should be using GPUs all the time??
- GPUs can only execute some types of code fast
 - SIMD parallelism is not well suited for all algorithms
 - Need lots of data parallelism, data reuse, & regularity
- GPUs are harder to program and tune than CPUs because of their architecture
 - Fewer tools and libraries exist

GPU Architecture

- GPUs consist of Streaming Multiprocessors (SMs)
 - NVIDIA calls these streaming multiprocessors and AMD calls them compute units
- SMs contain Streaming Processors (SPs) or Processing Elements (PEs)
 - Each core contains one or more ALUs and FPUs
- GPU can be thought of as a multi-multicore (manycore) system



A Generic Modern GPU Architecture



Ampere Architecture



NVIDIA Ampere GA102 GPU Architecture

Ampere Architecture

- GA102 GPU includes 28.3 billion transistors with a die size of 628.4 mm²
- A full GA102 GPU includes 7 Graphics Processing Clusters (GPCs), 42 (7*6) Texture Processing Clusters (TPCs), and 84 (7*12) Streaming Multiprocessors (SMs)
 - Each SM in GA10x GPUs contain 128 CUDA Cores for a total of $84 * 128 = 10752$ CUDA cores
 - Each SM has 256 (4*16384*32 bits) KB register file
 - 84 RT Cores, and 336 Tensor Cores, 168 FP64 units (two per SM)
- Includes PCIe Gen4 providing up to up to 16 Gigatransfers/second bit rate
- The memory subsystem consists of twelve 32-bit memory controllers (384-bit total)
- 512 KB of L2 cache is paired with each 32-bit memory controller, for a total of 6144 KB

| NVIDIA GPU Microarchitecture | Release Year | Remarks |
|-------------------------------------|---------------------|---|
| Tesla | 2006 | Unified shader model |
| Fermi | 2010 | Improved double precision performance, support for FMA |
| Kepler | 2012 | Focused on energy efficiency, shuffle instructions, dynamic parallelism |
| Maxwell | 2014 | Focused on energy efficiency, larger L2 cache |
| Pascal | 2016 | Unified memory, half-precision floating-point |
| Volta | 2017 | Features tensor cores for deep learning workloads |
| Turing | 2018 | Features tensor cores for deep learning workloads and real-time ray tracing. Gaming version of Volta. |
| Ampere | 2020 | New generation tensor and ray-tracing cores |
| Hopper | 2022 | |

Compute Capability

- When programming with CUDA, it is very important to be aware of the differences among different versions of hardware
- In CUDA, compute capability refers to architecture features
 - For example, number of registers and cores, cache and memory size, supported arithmetic instructions
- For example, compute capability 1.x devices have 16KB local memory per thread, and 2.x and 3.x devices have 512KB local memory per thread

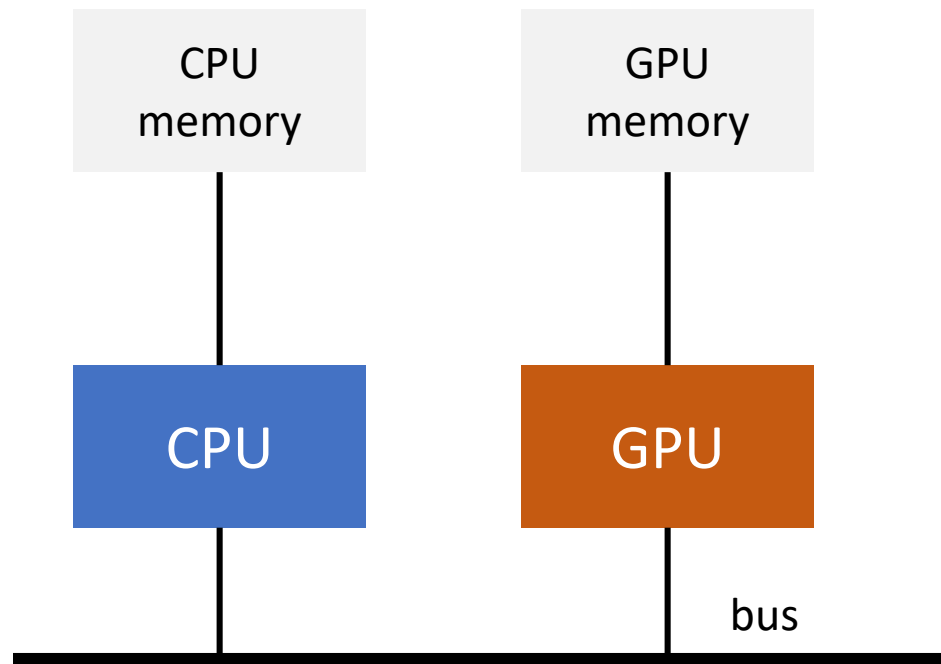
https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications

Role of CPUs

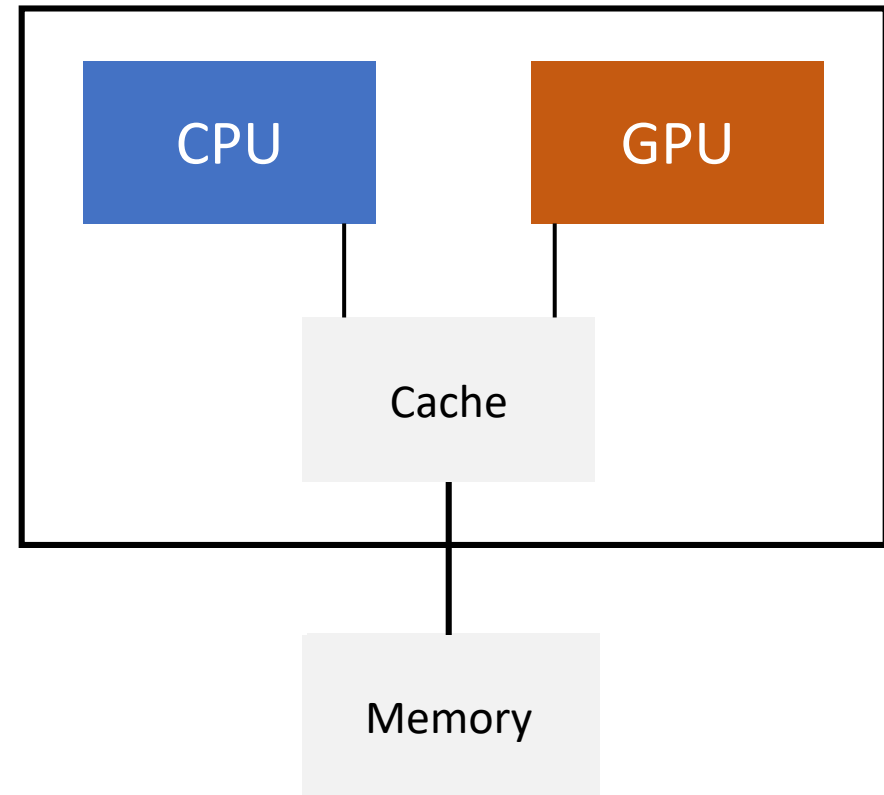
- CPU is responsible for initiating computation on the GPU and transferring data to and from the GPU
- Beginning and end of the computation typically require access to input/output (I/O) devices
- There are ongoing efforts to develop APIs providing I/O services directly on the GPU
 - GPUs are not standalone yet, assumes the existence of a CPU

Discrete and Integrated GPUs

Discrete



Integrated



CPUs vs GPUs

CPUs

- Designed for running a small number of potentially complex tasks
 - Tasks may be unconnected
 - Suitable to run system software like the OS and applications
- Small number of registers per core private to a task
 - Context switch between tasks is expensive in terms of time
 - Register set must be saved to memory and the next one restored from memory

GPUs

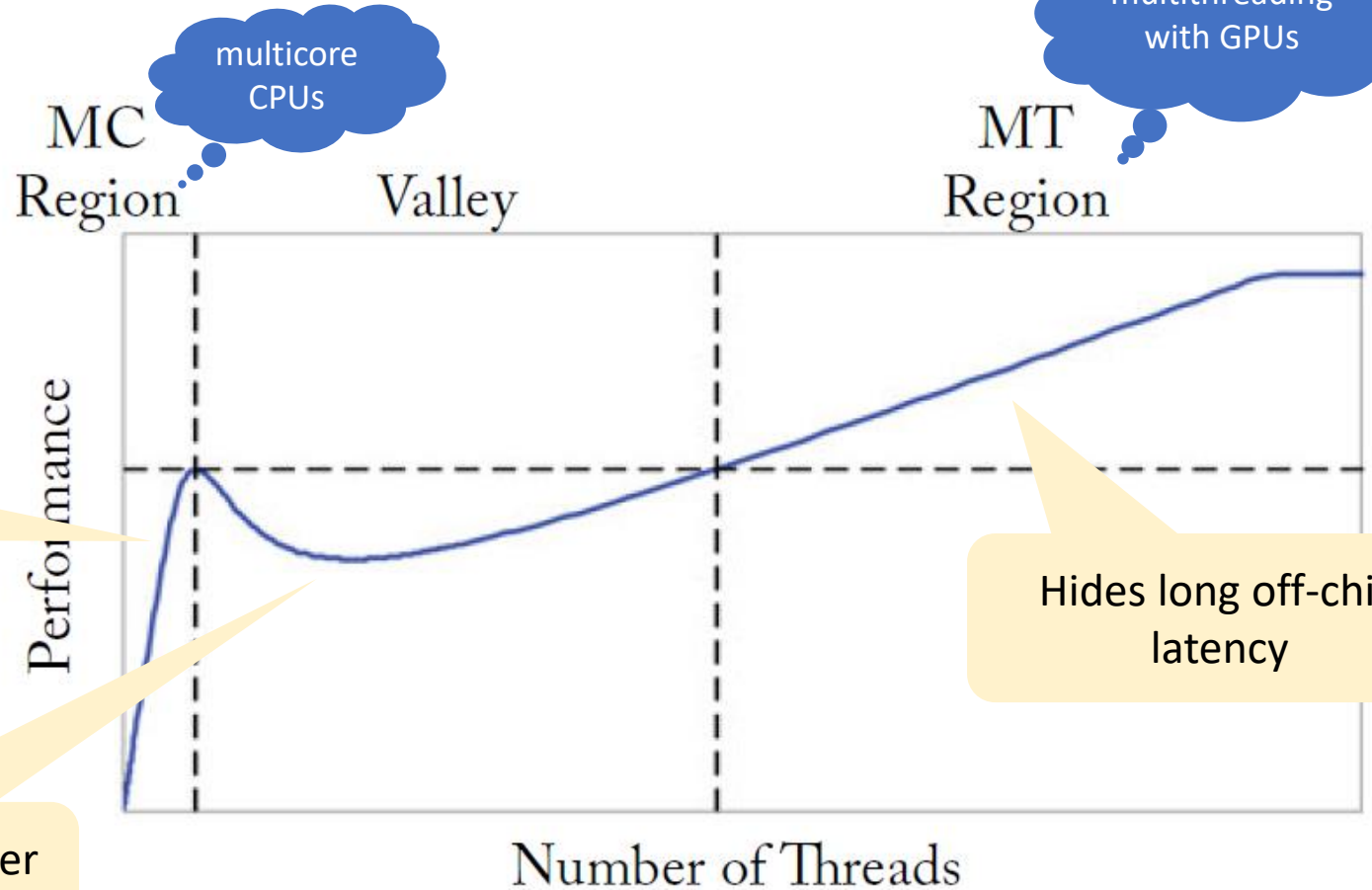
- Designed for running large number of simple tasks
 - Suitable for data-parallelism
- Have a single set of registers but with multiple banks
 - A context switch involves setting a bank selector to switch in and out the current set of registers
 - Orders of magnitude faster than having to save to RAM

An Analytical Model-based Analysis

Simple cache model where threads do not share data and there is infinite off-chip memory bandwidth

Large cache shared among few threads

Working set no longer fits in the cache



CUDA Programming

Programming API for NVIDIA GPUs

What is CUDA?

- It is general purpose **parallel computing platform** and programming model that leverages the parallel compute engine in NVIDIA GPUs
 - Introduced in 2007 with NVIDIA Tesla architecture
 - CUDA C, C++, Fortran, PyCUDA are language systems built on top of CUDA
- **Three key abstractions** in CUDA
 - Hierarchy of thread groups
 - Shared memories
 - Barrier synchronization

CUDA Philosophy

SIMT philosophy

- Single Instruction Multiple Thread

Computationally intensive

- The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory

Massively parallel

- The computations can be broken down into hundreds or thousands of independent units of work

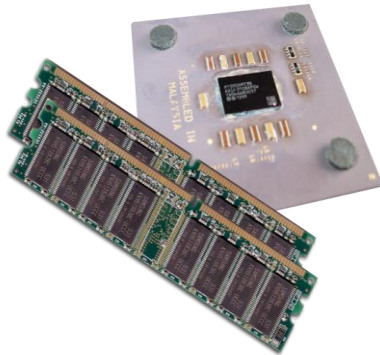
CUDA Programming Model

- Allows fine-grained data parallelism and thread parallelism nested within coarse-grained data parallelism and task parallelism
1. Partition the problem into coarse sub-problems that can be solved independently
 2. Assign each sub-problem to a “block” of threads to be solved in parallel
 3. Each sub-problem is also decomposed into finer work items that are solved in parallel by all threads within the “block”

Heterogeneous Computing

Host

- CPU and its memory (host memory)



Device

- GPU and its memory (device memory)



Heterogeneous Computing

```
#include <iostream>
#include <algorithm>
using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int index = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[gindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[index - RADIUS] = in[index - RADIUS];
        temp[index + BLOCK_SIZE] = in[index + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[index + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out; // host copies of a, b, c
    int *d_in, *d_out; // device copies of a, b, c
    int size = (N + 2 * RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2 * RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2 * RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<n/BLOCK_SIZE, BLOCK_SIZE>>>(d_in + RADIUS,
    d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

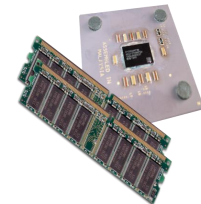
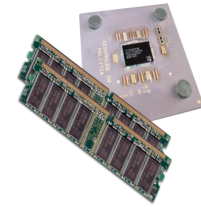
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

serial code

parallel code

serial code



Heterogeneous Computing

```
#include <iostream>
#include <algorithm>
using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out; // host copies of a, b, c
    int *d_in, *d_out; // device copies of a, b, c
    int size = (N + 2 * RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2 * RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2 * RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **) &d_in, size);
    cudaMalloc((void **) &d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE, BLOCK_SIZE>>>(d_in + RADIUS,
    d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

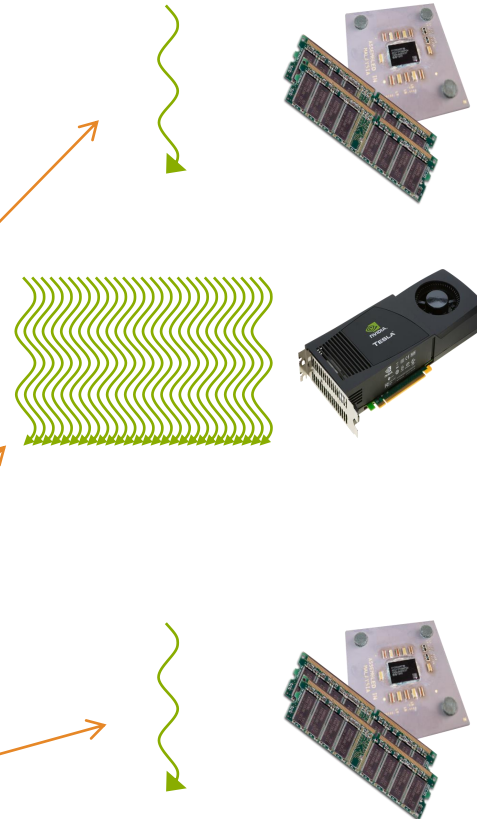
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

serial code

parallel code

serial code



Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
    printf("Hello world!\n");
}

int main() {
    hwkernel<<<1, 1>>>();
}
```

```
$ nvcc hello-world.cu

$ ./a.out

$
```

Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
    printf("Hello world!\n");
}

int main() {
    hwkernel<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

```
$ nvcc hello-world.cu
```

```
$/a.out
```

```
Hello world!
```

```
$
```

Program returns immediately after launching the kernel. To prevent program to finish before kernel is completed, we call `cudaDeviceSynchronize()`.

Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
    printf("Hello world!\n");
}

int main() {
    hwkernel<<<1, 32>>>();
    cudaThreadSynchronize();
}
```

```
$ nvcc hello-world.cu

$ ./a.out
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
...
...
$
```

Function Declarations in CUDA

| | Executed on | Callable from |
|--|-------------|---------------|
| <code>__device__ float deviceFunc()</code> | Device | Device |
| <code>__global__ void kernelFunc()</code> | Device | Host* |
| <code>__host__ float hostFunc()</code> | Host | Host |

- `__global__` define a kernel function, must return void
- `__device__` functions can have return values
- `__host__` is default, and can be omitted
- Prepending `__host__` `__device__` causes the system to compile separate host and device versions of the function

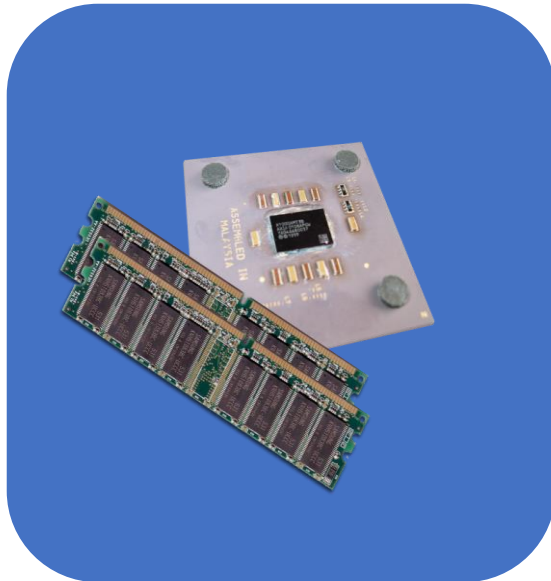
*A kernel function can also be called from the device if dynamic parallelism is enabled.

Dynamic Parallelism

- It is possible to launch kernels from other kernels
- Calling `__global__` functions from the device is referred to as dynamic parallelism
 - Requires CUDA devices of compute capability 3.5 and CUDA 5.0 or higher

Execution Model

Host
(serial execution)



Serial code on host



Parallel kernel on device



Serial code on host

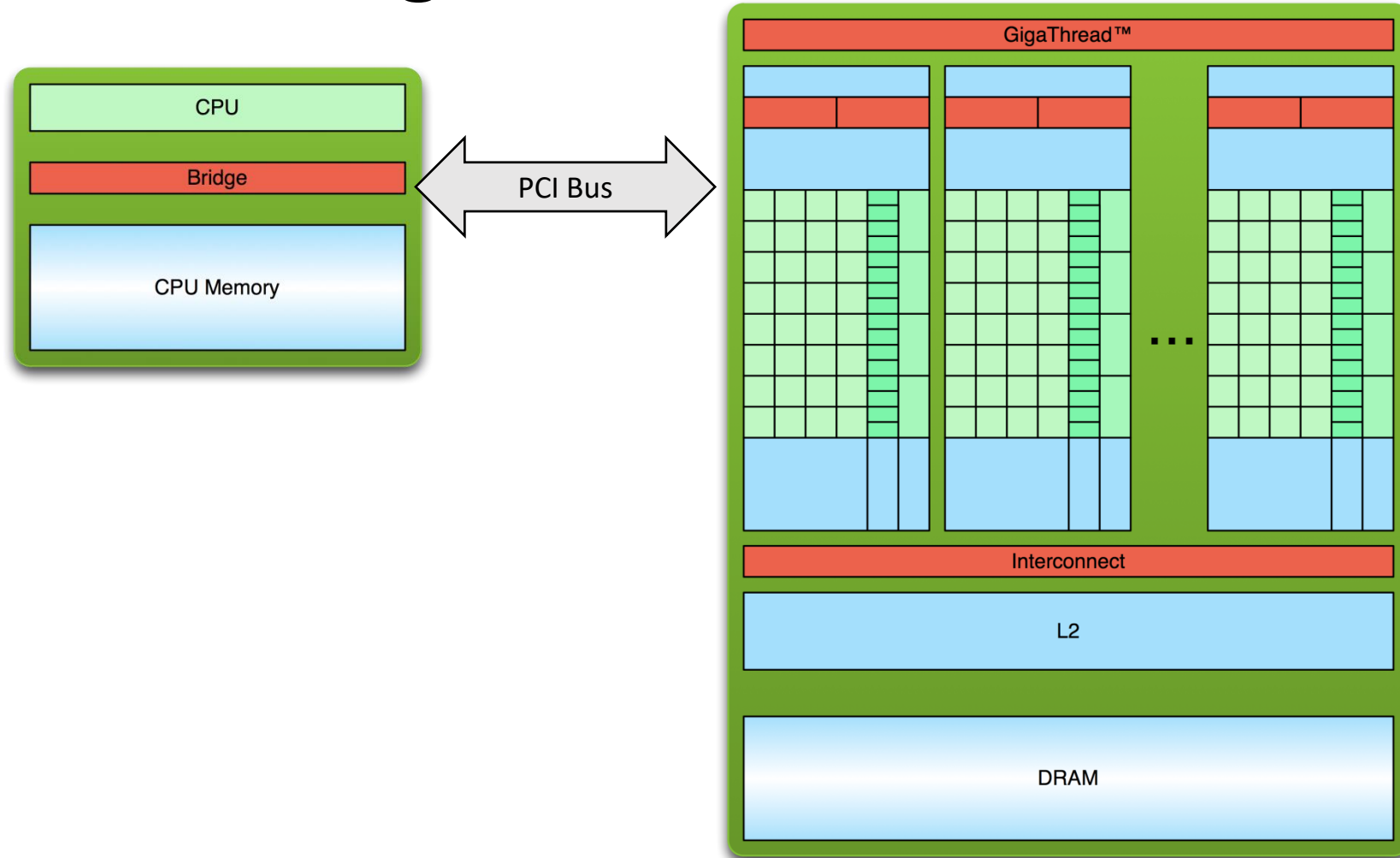


Parallel kernel on device

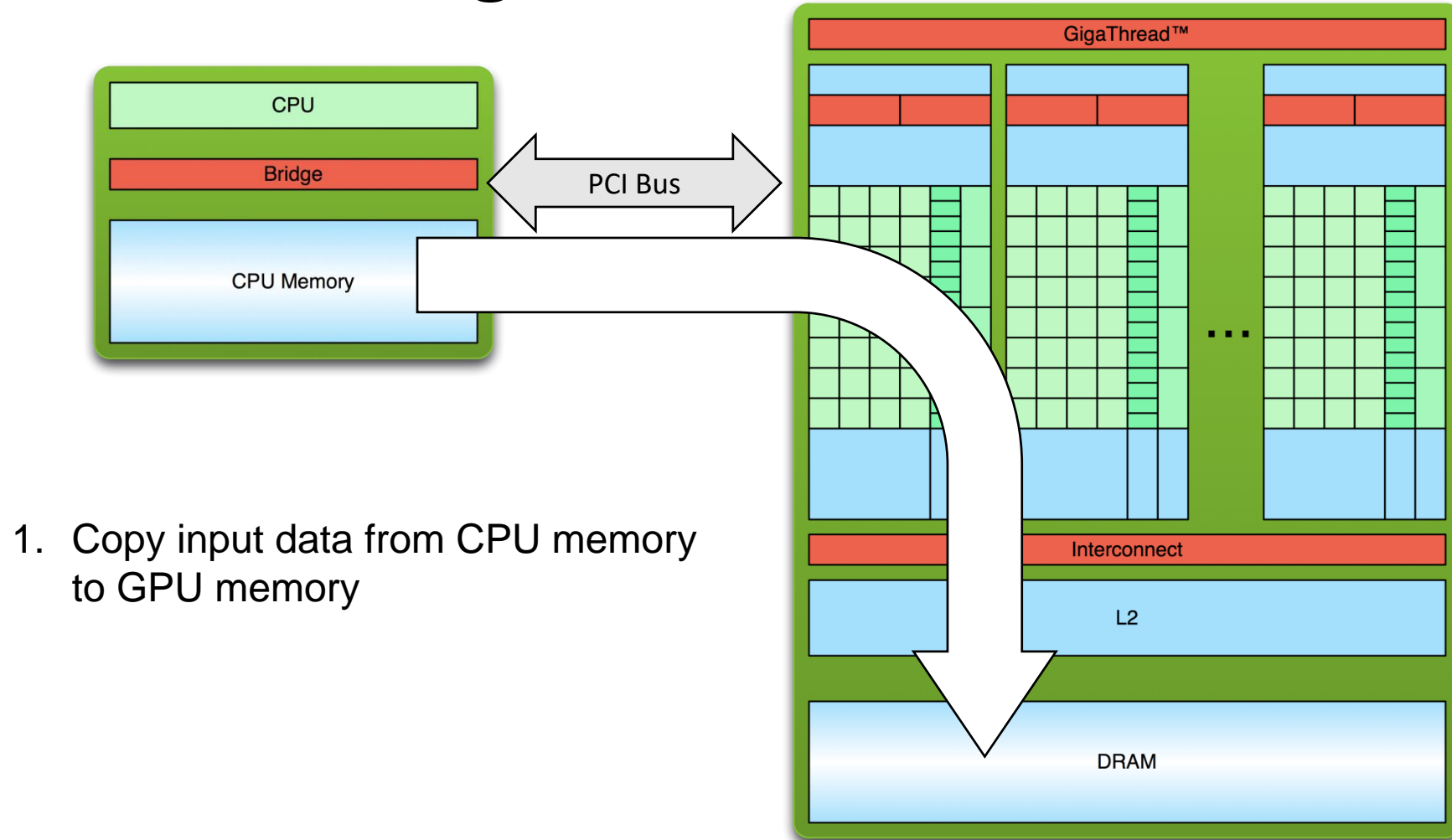
Device
(Parallel execution)



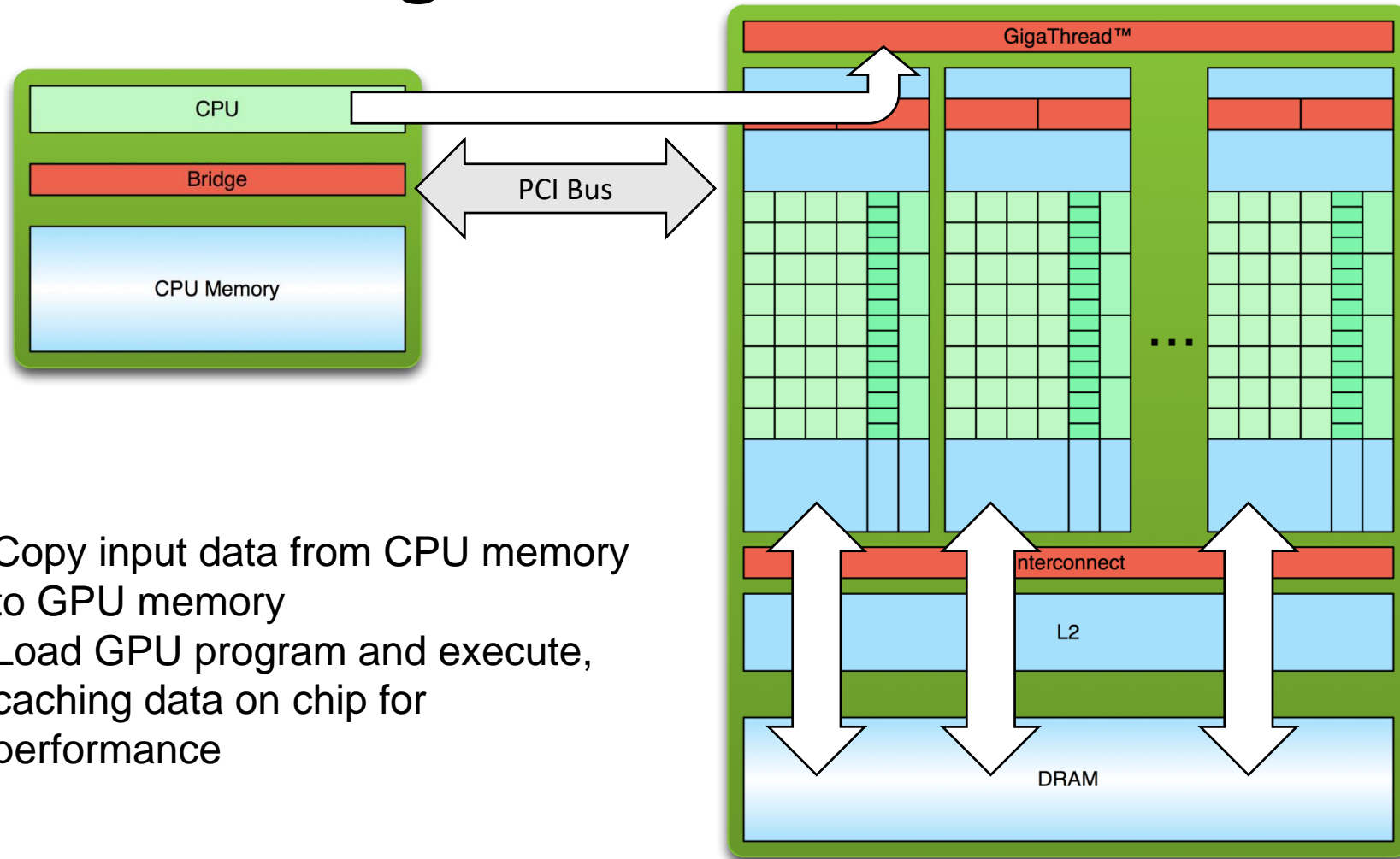
Simple Processing Flow



Simple Processing Flow

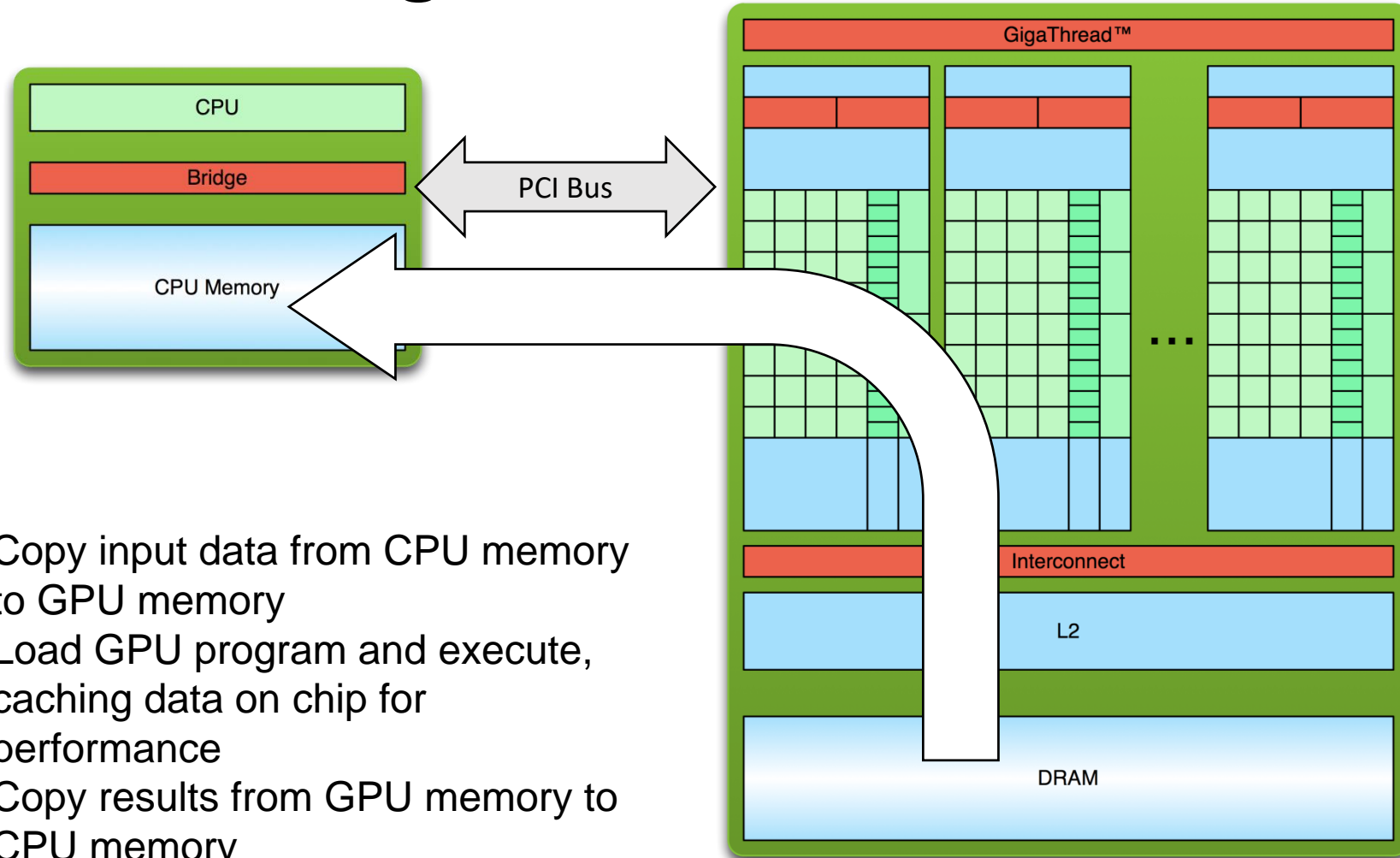


Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Vector Addition Example

```
__global__ void VecAdd(float* A, float* B,  
    float* C, int N) {  
    int i = blockDim.x * blockIdx.x +  
        threadIdx.x;  
    if (i < N)  
        C[i] = A[i] + B[i];  
}
```

```
int main() {  
    ...  
    float* h_A = (float*)malloc(size);  
    float* h_B = (float*)malloc(size);  
    float* h_C = (float*)malloc(size);
```

```
float* d_A;  
cudaMalloc(&d_A, size);  
float* d_B;  
cudaMalloc(&d_B, size);  
float* d_C;  
cudaMalloc(&d_C, size);
```

```
// Copy vectors from host memory to  
// device memory  
cudaMemcpy(d_A, h_A, size,  
           cudaMemcpyHostToDevice);  
cudaMemcpy(d_B, h_B, size,  
           cudaMemcpyHostToDevice);
```

Vector Addition Example

```
// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = N/threadsPerBlock
;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```

```
// Copy result from device memory to
// host memory
cudaMemcpy(h_C, d_C, size,
           cudaMemcpyDeviceToHost);

...
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);

...
}
```

Typical CUDA Program Flow

1. Load data into CPU memory
 - `fread/rand`
2. Copy data from CPU to GPU memory
 - `cudaMemcpy(..., cudaMemcpyHostToDevice)`
3. Call GPU kernel
 - `yourkernel<<<x, y>>>(...)`
4. Copy results from GPU to CPU memory.
 - `cudaMemcpy(..., cudaMemcpyDeviceToHost)`
5. Use results on CPU

CUDA Extensions for C/C++

- Kernel launch
 - Calling functions on GPU
- Memory management
 - GPU memory allocation, copying data to/from GPU
- Declaration qualifiers
 - `__device__`, `__shared`, `__local`, `__global__`, `__host__`
- Special instructions
 - Barriers, fences, etc.
- Keywords
 - `threadIdx`, `blockIdx`, `blockDim`

C++11 Support from CUDA 7.5+

Supported Features

- auto
- lambdas
- constexpr
- rvalue references
- range-based for loops

Unsupported Features

- Standard library
 - You cannot use `std::cout` in device code

Kernels

- Special functions that a CPU can call to execute on the GPU
 - Executed N times in parallel by N different CUDA threads
 - Cannot return a value
- Each thread will execute VecAdd()
- Each thread has a unique thread ID that is accessible within the kernel through the built-in threadIdx variable

```
// Kernel definition
__global__ void VecAdd(float* A,
float* B, float* C) {
    int i = threadIdx.x;
    ...
}

int main() {
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

Kernels

- GPU spawns m blocks with n threads (i.e., $m*n$ threads total) that run a copy of the same function

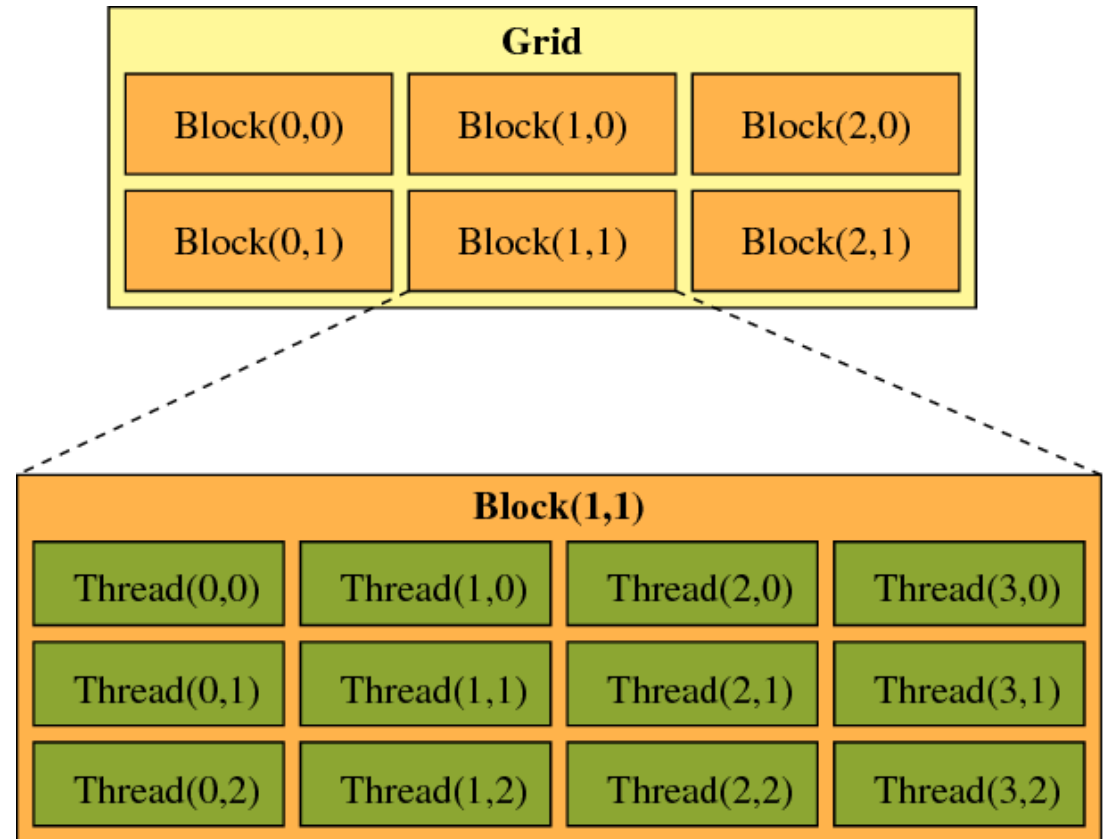
```
KernelName<<<m, n>>>(arg1, arg2, ...)
```

- CPU can continue processing while GPU runs kernel
- Kernel call returns when all threads have terminated

```
kernel1<<<X,Y>>>( ... ); // kernel starts execution, CPU continues to next statement  
kernel2<<<X,Y>>>( ... ); // kernel2 placed in queue, will start after kernel1 finishes,  
                          // CPU continues  
cudaMemcpy( ... ); // CPU blocks until memory is copied, memory copy starts after all  
                  // preceding CUDA calls finish
```

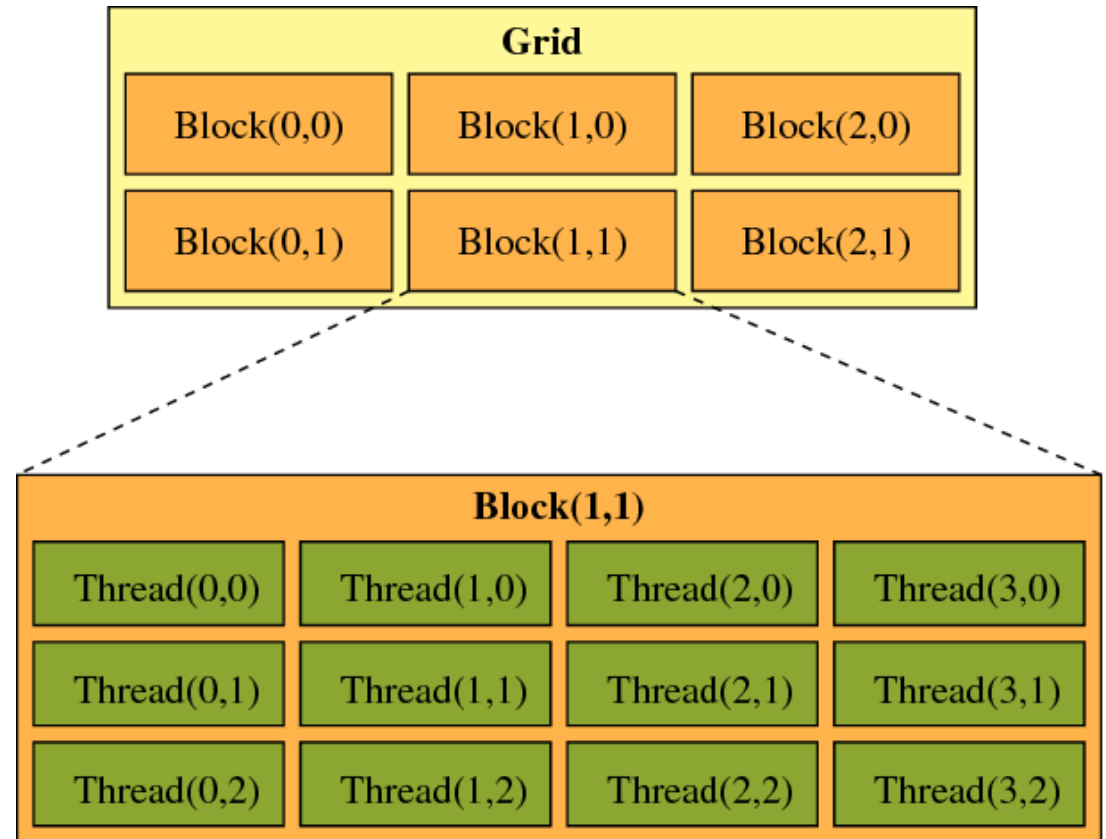
Thread Hierarchy

- A kernel executes in parallel across a set of parallel threads
- All threads that are generated by a kernel launch are collectively called a grid
- Threads are organized in thread blocks, and blocks are organized in to grids

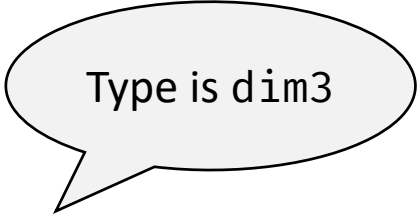


Thread Hierarchy

- A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory
- A grid is an array of thread blocks that execute the same kernel
 - Read inputs to and write results to global memory
 - Synchronize between dependent kernel calls



Dimension and Index Variables



Type is dim3

Dimension

- `gridDim` specifies the number of blocks in the grid
- `blockDim` specifies the number of threads in each block

Index

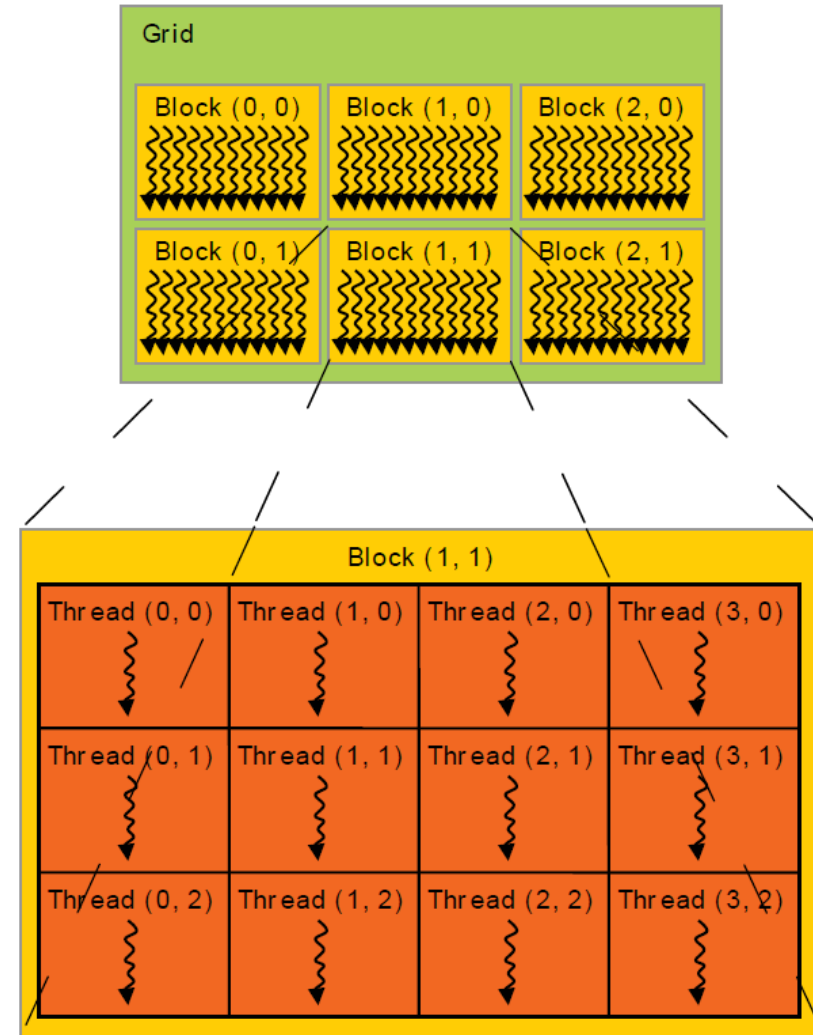
- `blockIdx` gives the index of the block in the grid
- `threadIdx` gives the index of the thread within the block

Thread Hierarchy

- `threadIdx` is a 3-component vector
 - Thread index can be 1D, 2D, or 3D
 - Thread blocks as a result can be 1D, 2D, or 3D
- How to find out the relation between thread ids and `threadIdx`?
 - 1D: `tid = threadIdx.x`
 - 2D block of size (D_x, D_y) : thread ID of a thread of index (x, y) is $(x + yD_x)$
 - 3D block of size (D_x, D_y, D_z) : thread ID of a thread of index (x, y, z) is $(x + yD_x + zD_xD_y)$

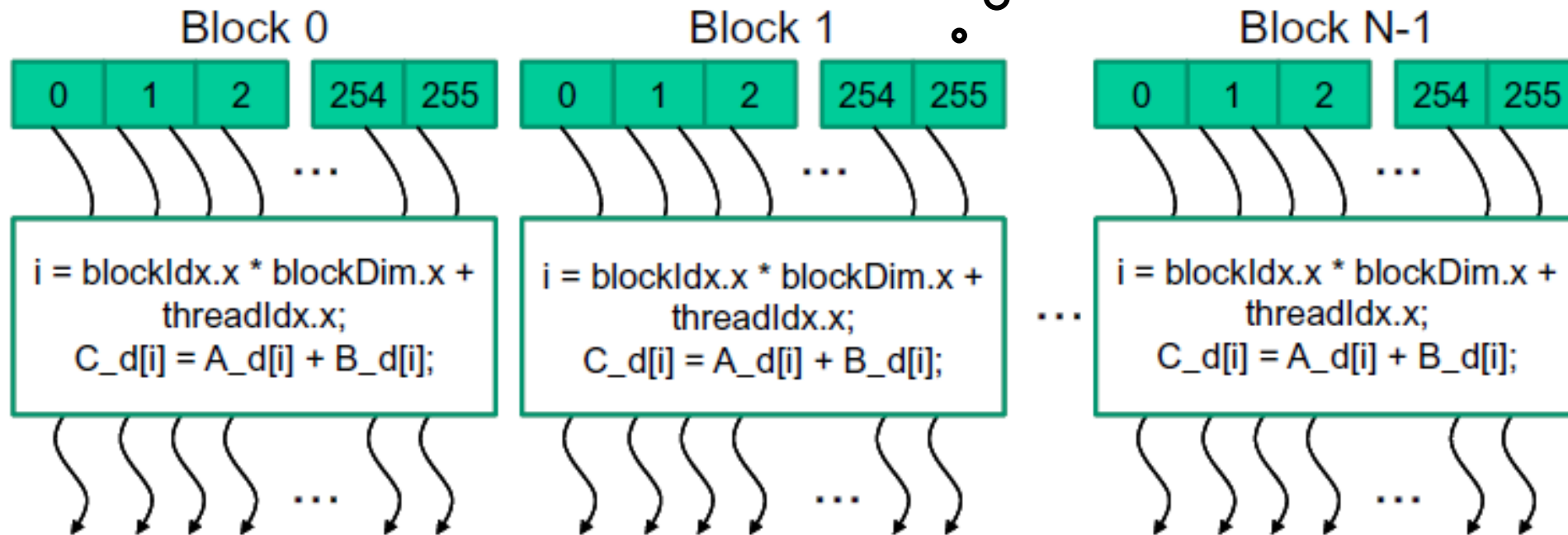
Thread Hierarchy

- Threads in a block reside on the same core, max 1024 threads in a block
- Thread blocks are organized into 1D, 2D, or 3D grids
 - Also called cooperative thread array
 - Grid dimension is given by `gridDim` variable
- Identify block within a grid with the `blockIdx` variable
 - Block dimension is given by `blockDim` variable



Finding Thread IDs

i is local to each thread



Determining Block Dimensions

- Assume a block with a maximum of 1024 allowed threads

| Variable blockDim | Valid/Invalid |
|-------------------|---------------|
| (512,1,1) | ✓ |
| (8, 16, 4) | ✓ |
| (32, 16, 2) | ✓ |
| (32, 32, 32) | ✗ |

Find Device Information

```
int count;
cudaError_t err =
    cudaGetDeviceCount(&count);
if (err != cudaSuccess) {
    cerr << cudaGetErrorString(err) << endl;
}

cudaDeviceProp Props;
for (int i = 0; i < count; i++) {
    err = cudaGetDeviceProperties(&Props, i);
}
```

Device number: 3

Device name: GeForce GTX 1080 Ti

Integrated or discrete GPU? discrete

Clock rate: 1544 MHz

Compute capability: 6.1

Number of SMs: 28

Total number of CUDA cores: 3584

Max threads per SM: 2048

Max threads per block: 1024

Warp size: 32

Max grid size (i.e., max number of blocks): [2147483647,65535,65535]

Max block dimension: [1024,1024,64]

Total global memory: 11172 MB

Shared memory per SM: 96 KB

32-bit registers per SM: 65536

Shared mem per block: 48 KB

Registers per block: 65536

Total const mem: 64 KB

L2 cache size: 2816 KB

Device Management

- Application can query and select GPUs
 - `cudaGetDeviceCount(int *count)`
 - `cudaSetDevice(int device)`
 - `cudaGetDevice(int *device)`
 - `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`
- Multiple host threads can share a device
- A single host thread can manage multiple devices
 - `cudaSetDevice(i)` to select current device
 - `cudaMemcpy(...)` for peer-to-peer copies

Launching Kernels

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
    int i = threadIdx.y;
    int j = threadIdx.x;
    C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<numBlocks, threadsPerBlock>>(A, B, C);
    ...
}
```

Type dim3 is not
required if 1D layout

Execution Configuration


- Assume data is of length N , and say the kernel execution configuration is $\langle\langle\langle N/TPB, TPB \rangle\rangle\rangle$
 - Each block has TPB threads
 - There are N/TPB blocks
- Suppose $N = 64$ and $TPB = 32$
 - Implies there are 2 blocks of 32 threads
- Dimension variables are vectors of integral type

Launching Kernels

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
int main() {
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```


Execution Configuration Uses Integer Arithmetic

- Assume data is of length N , and say the kernel execution configuration is $\langle\langle\langle N/TPB, TPB \rangle\rangle\rangle$
 - Each block has TPB threads
 - There are N/TPB blocks
- Suppose $N = 64$ and $TPB = 32$
 - Implies there are 2 blocks of 32 threads
- Dimension variables are vectors of integral type
- Now assume $N = 65$



So now
what?

Execution Configuration Uses Integer Arithmetic

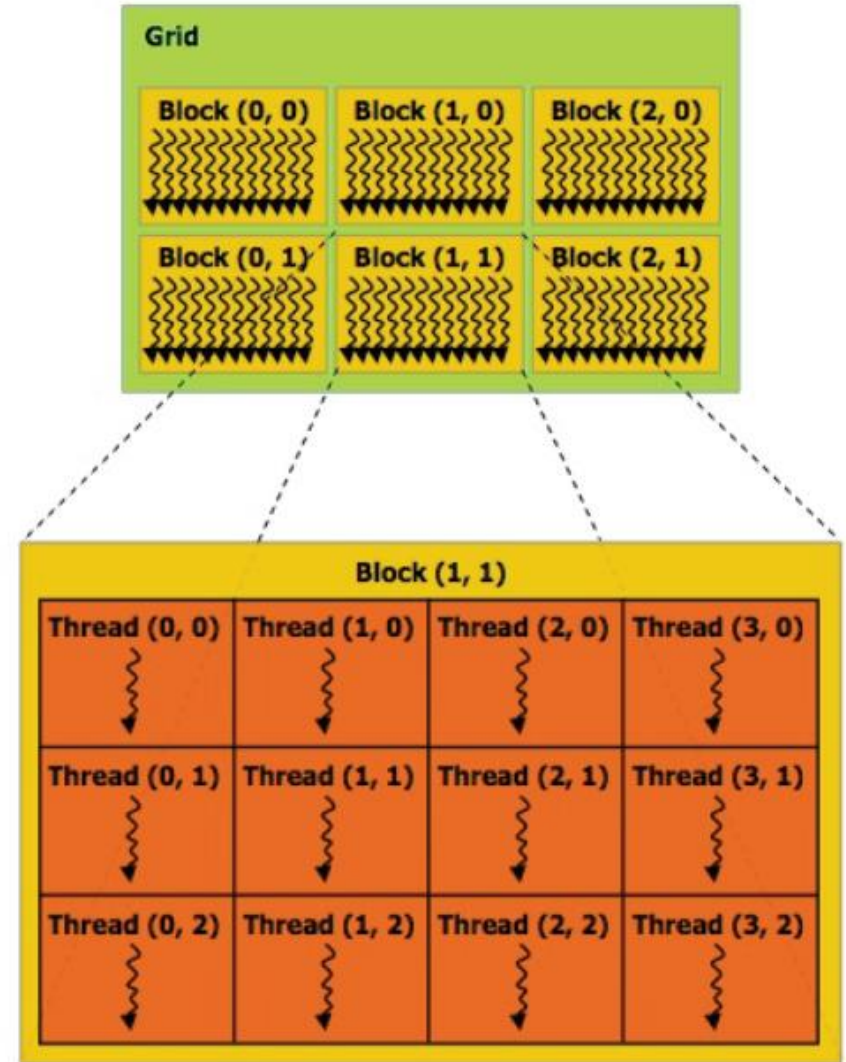
- Ensure that the grid covers the array length
- One strategy is to change the number of blocks from N/TPB to $(N+TPB-1)/TPB$ to ensure rounding up
- This means that a thread index can exceed the maximum array index
- Many examples use a control statement in the kernel to check for such corner cases

What should be numBlocks?

```
const int Nx = 11; // not a multiple of threadsPerBlock.x
const int Ny = 5; // not a multiple of threadsPerBlock.y

////////////////////////////////////
dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(x, y, z);

// assume A, B, C are allocated Nx x Ny float arrays
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

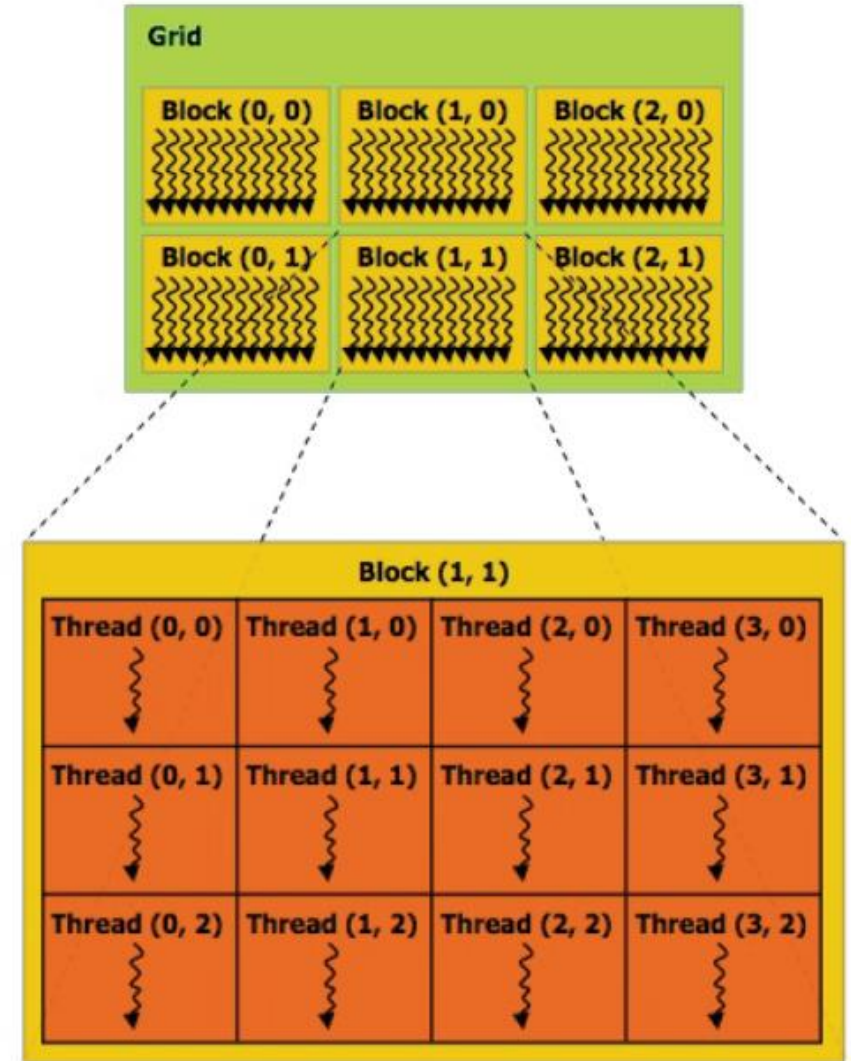


What should be numBlocks?

```
const int Nx = 11; // not a multiple of threadsPerBlock.x
const int Ny = 5; // not a multiple of threadsPerBlock.y

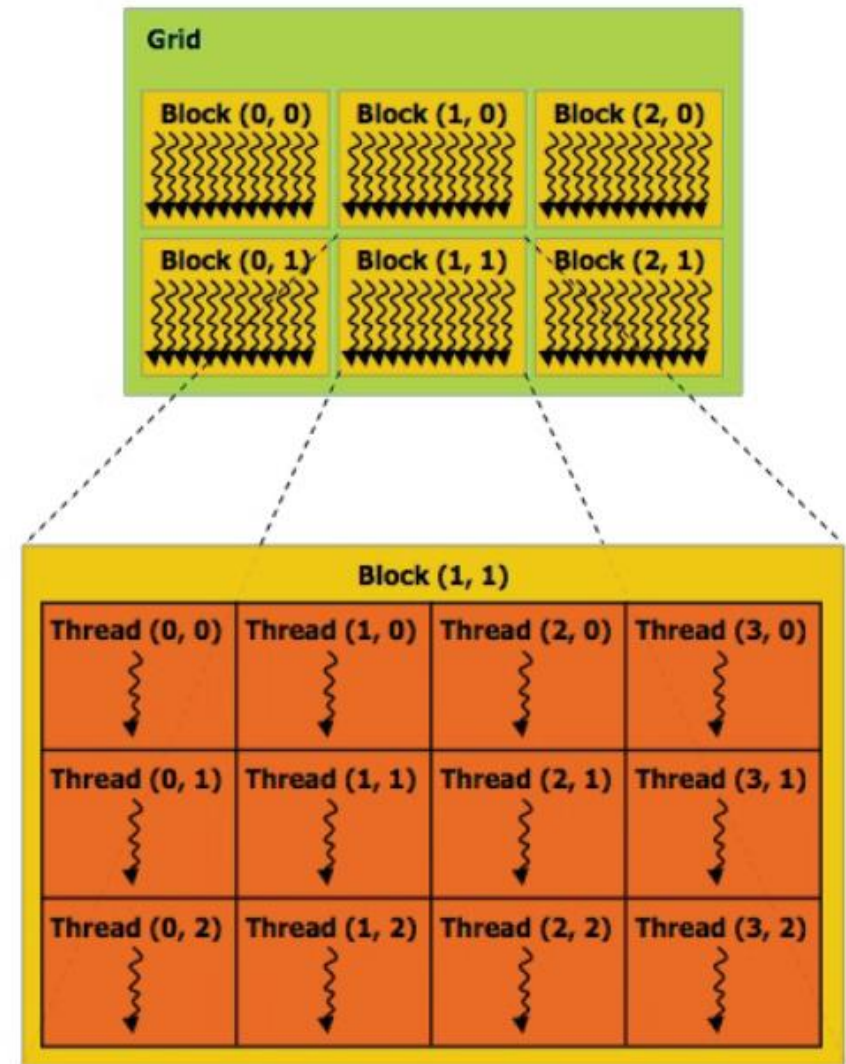
////////////////////////////////////
dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks((Nx+threadsPerBlock.x-1)/threadsPerBlock.x,
(Ny+threadsPerBlock.y-1)/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays
// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```



Example

```
__global__ void matrixAdd(float* A,  
                          float* B, float* C) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    int j = blockIdx.y * blockDim.y + threadIdx.y;  
    // Guard against out of bounds array access  
    if (i < N && j < N)  
        C[i+N*j] = A[i+N*j] + B[i+N*j];  
}
```



Matrix Multiplication Example

```
int main() {
    int SIZE = N * N;
    cudaError_t status;

    float *hostA, *hostB, *hostC;
    hostA = (float*)malloc(SIZE * sizeof(
float));
    hostB = (float*)malloc(SIZE * sizeof(
float));
    hostC = (float*)malloc(SIZE * sizeof(
float));
```

```
    float *deviceA, *deviceB, *deviceC;
    status = cudaMalloc((void**)&deviceA,
SIZE * sizeof(float));
    if (status != cudaSuccess) {
        cerr << cudaGetErrorString(status)
<< endl;
    }

    status = cudaMalloc((void**)&deviceB,
SIZE * sizeof(float));
    status = cudaMalloc((void**)&deviceC,
SIZE * sizeof(float));
```

Matrix Multiplication Example

```
status = cudaMemcpy(deviceA, hostA, SIZE * sizeof(float), cudaMemcpyHostToDevice);
```

```
status = cudaMemcpy(deviceB, hostB, SIZE * sizeof(float), cudaMemcpyHostToDevice);
```

```
dim3 blocksPerGrid(1, 1);
```

```
dim3 threadsPerBlock(N, N);
```

```
matmulKernel<<<blocksPerGrid, threadsPerBlock>>>(deviceA, deviceB, deviceC);
```

```
cudaMemcpy(hostC, deviceC, SIZE * sizeof(float), cudaMemcpyDeviceToHost);
```

```
...
```

```
cudaFree(deviceA);
```

```
cudaFree(deviceB);
```

```
cudaFree(deviceC);
```

```
free(hostA);
```

```
free(hostB);
```

```
...
```

```
}
```

Matrix Multiplication Example

```
__global__ void matmulKernel(float* A, float* B, float* C) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    float tmp = 0;
    if (i < N && j < N) {
        // Each thread computes one element of the matrix
        for (int k = 0; k < N; k++) {
            tmp += A[i * N + k] * B[k * N + j];
        }
    }
    C[i * N + j] = tmp;
}
```


Choosing Optimal Execution Configuration

- The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system
 - It is okay to have a much greater number of threads
- No fixed rule, needs exploration and experimentation
- Choose number of threads in a block to be some multiple of 32

Timing a CUDA Kernel

```
float memsettime;
cudaEvent_t start, stop;
// initialize CUDA timer
cudaEventCreate(&start);  cudaEventCreate(&stop);
cudaEventRecord(start,0);

// CUDA Kernel
...

cudaEventRecord(stop,0); // stop CUDA timer
cudaEventSynchronize(stop);
cudaEventElapsedTime(&memsettime,start,stop); // in milliseconds
std::cout << "Kernel execution time: " << memsettime << "\n";
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
 - Error in the API call itself or error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error
`cudaError_t cudaGetLastError(void)`
- Get a string to describe the error:
`char *cudaGetErrorString(cudaError_t)`

Mapping Blocks and Threads

- A GPU executes one or more kernel grids
- When a CUDA kernel is launched, the thread blocks are enumerated and distributed to SMs
 - Potentially >1 block per SM
- An SM executes one or more thread blocks
 - Each GPU has a limit on the number of blocks that can be assigned to each SM
 - For example, a CUDA device may allow up to eight blocks to be assigned to each SM
 - Multiple thread blocks can execute concurrently on one SM

Mapping Blocks and Threads

- The threads of a block execute concurrently on one SM
 - CUDA cores in the SM execute threads
- A block begins execution only when it has secured all execution resources necessary for all the threads
- As thread blocks terminate, new blocks are launched on the vacated multiprocessors
- Blocks are mostly not supposed to synchronize with each other
 - Allows for simple hardware support for data parallelism

Mapping Blocks and Threads

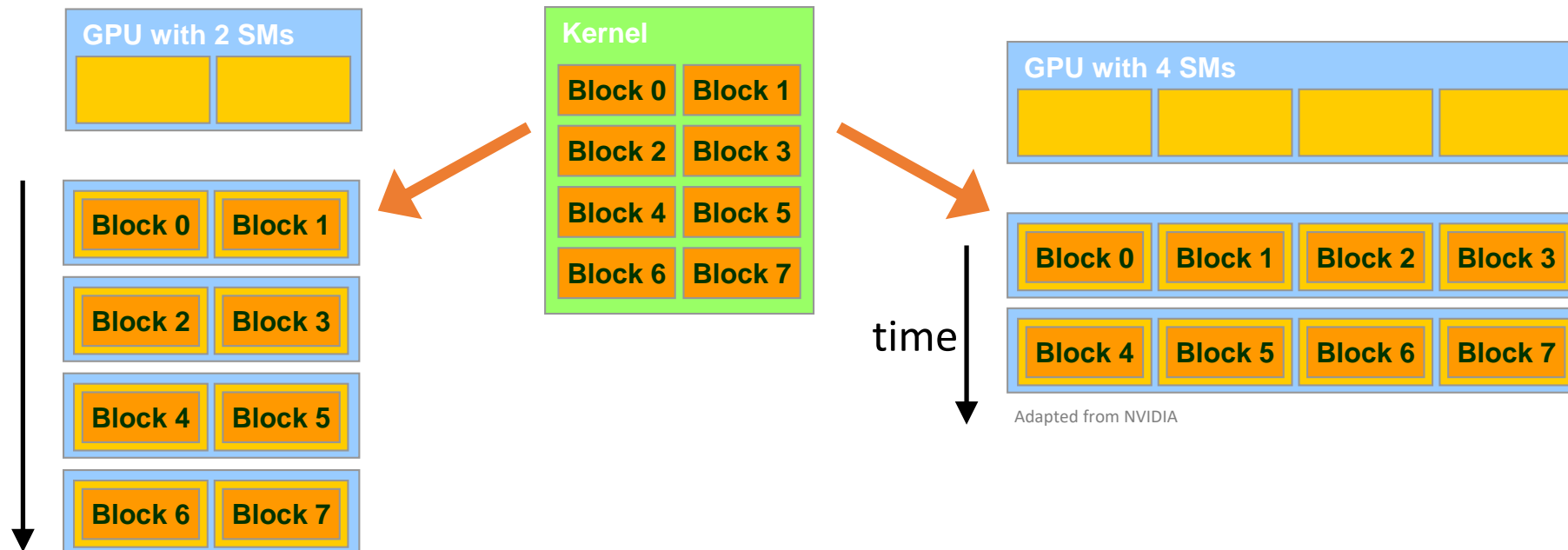
- The threads of a block execute concurrently on one SM
 - CUDA cores in the SM execute threads
- A block begins execution only when it has secured all execution resources
- As multiple blocks are executed simultaneously, **CUDA runtime can execute blocks in any order** needed
- Blocks are mostly not supposed to synchronize with each other
 - Allows for simple hardware support for data parallelism

Scheduling Blocks

- Number of threads that can be simultaneously tracked and scheduled is bounded
 - Requires resources for an SM to maintain block and thread indices and their execution status
- Up to 2048 threads can be assigned to each SM on recent CUDA devices
 - For example, 8 blocks of 256 threads, or 4 blocks of 512 threads
- Assume a CUDA device with 28 SMs
 - Each SM can accommodate up to 2048 threads
 - The device can have up to 57344 threads simultaneously residing in the device for execution

Block Scalability

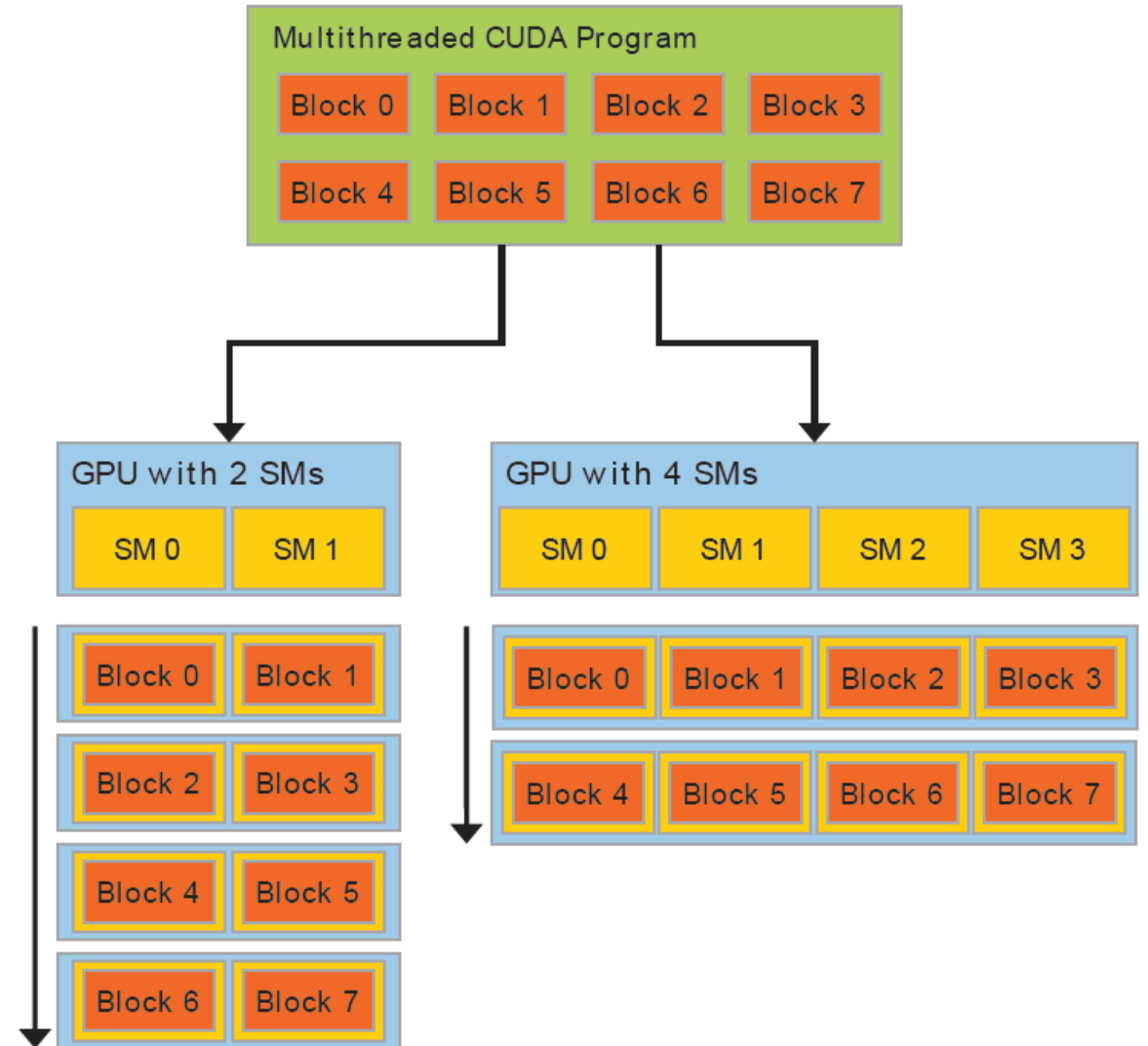
- Hardware can assign blocks to SMs in any order
 - A kernel with enough blocks scales across GPUs
 - Not all blocks may be resident at the same time



Scalability of GPU Architecture

A multithreaded program is partitioned into blocks of threads that execute independently from each other.

A GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

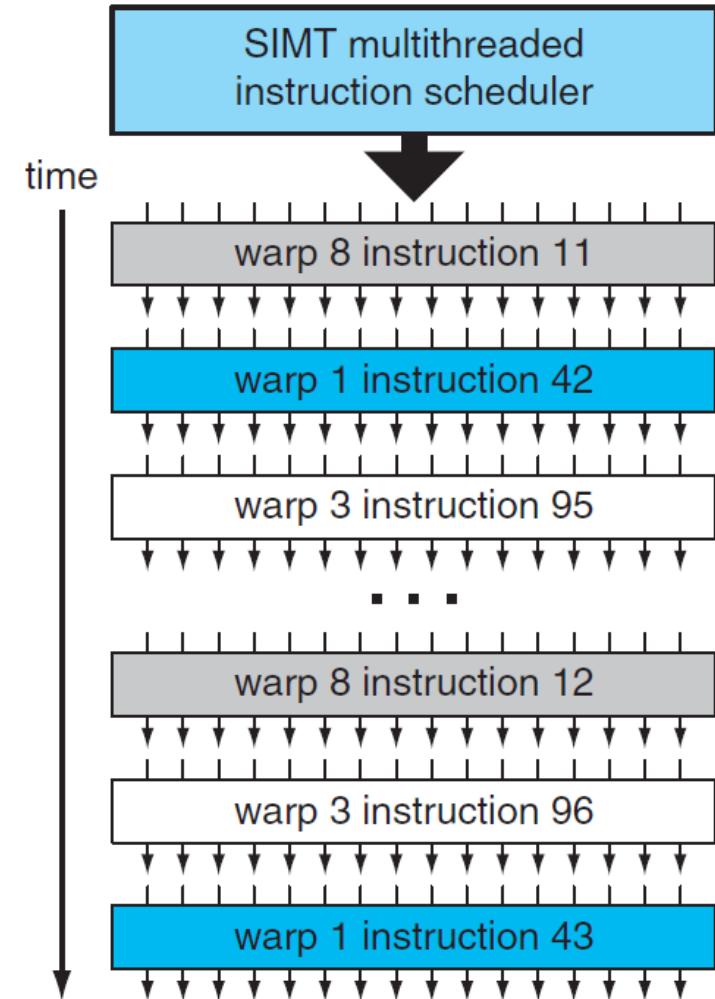


Thread Warps

- Conceptually, threads in a block can execute in any order
- Sharing a control unit among compute units reduce hardware complexity, cost, and power consumption
- A set of consecutive threads (currently 32) that execute in SIMD fashion is called a warp
 - These are called wavefront (with 64 threads) on AMD
- Warps are scheduling units in an SM
 - Part of the implementation in NVIDIA, not the programming model

Thread Warps

- All threads in a warp run in lockstep
 - Warps share an instruction stream
 - Same instruction is fetched for all threads in a warp during the instruction fetch cycle
 - Prior to Volta, warps used a single shared program counter
 - In the execution phase, each thread will either execute the instruction or will execute nothing
 - Individual threads in a warp have their own instruction address counter and register state



Thread Warps

- Warp threads are fully synchronized
 - There is an implicit barrier after each step/instruction
- If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there in an SM?
 - Each Block is divided into $256/32 = 8$ warps
 - There are $8 * 3 = 24$ warps

Thread Divergence

- If some threads take the `if` branch and other threads take the `else` branch, they cannot operate in lockstep
 - Some threads must wait for the others to execute, renders code at that point to be serial rather than parallel
- The programming model does not prevent thread divergence
 - Divergence occurs only within a warp, so it is a performance problem at the warp level

Thread Divergence

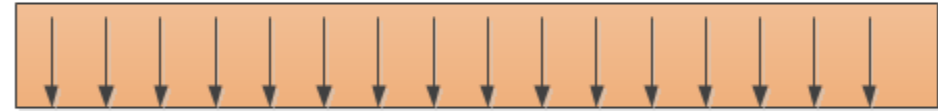
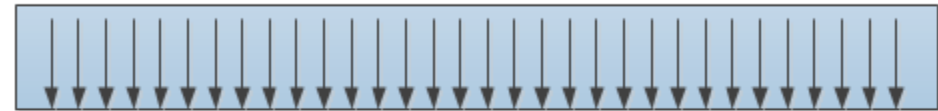
```
unsigned int index = ( blockDim.x * blockIdx.x ) + threadIdx.x;  
float value = 0.0f;
```



```
value = PathA( src );
```

```
value = PathB( src );
```

```
dst[index] = value;
```



Scheduling Thread Warps

- Each SM launches warps of threads, and executes warps on a timesharing basis
 - Timesharing is implemented in hardware, not software
- SM schedules and executes warps that are ready to run
 - Warps run for fixed-length time slices like processes
 - Warps whose next instruction has its operands ready for consumption are eligible for execution
 - Selection of ready warps for execution does not introduce any idle time into the execution timeline, called zero-overhead scheduling
 - If more than one warp is ready for execution, a priority mechanism is used to select one for execution

Scheduling Thread Warps

- Suppose an instruction executed by a warp has to wait for the result of a previously initiated long-latency operation
 - The warp is not selected for execution, another warp that is not waiting for results is selected for execution
- Goal is to have enough threads and warps around to utilize hardware in spite of long-latency operations
 - GPU hardware will likely find a warp to execute at any point in time
 - Hides latency of long operations with work from other threads, called latency tolerance or latency hiding

Scheduling Thread Warps

- Thread blocks execute on an SM, thread instructions execute on a core
- CUDA virtualizes the physical hardware
 - Thread is a virtualized scalar processor (registers, PC, state)
 - Block is a virtualized multiprocessor (threads, shared memory)
- As warps and thread blocks complete, resources are freed

Question

- Assume that a CUDA device allows up to 8 blocks and 1024 threads per SM, whichever becomes a limitation first
 - It allows up to 512 threads in each block
- Say for the matrix-matrix multiplication kernel, should we use 8x8, 16x16, or 32x32 thread blocks?

SIMT Architecture

- GPUs employ SIMD hardware to exploit the data-level parallelism
 - In SIMD, you program with the vector width in mind
 - In vectorization, users program the SIMD hardware directly, or uses auto-vectorization or intrinsics
- SIMT can be thought of as SIMD with multithreading
 - Software analog compared to the hardware perspective of SIMD
 - For e.g., we rarely need to know the number of cores with CUDA

SIMT Architecture

- CUDA also features a MIMD-like programming model
 - Launch large number of threads
 - Each thread can have its own execution path and access arbitrary memory locations
- This execution model is called single-instruction multiple-thread (SIMT)
- Two levels of parallelism
 - Independent grids (i.e., kernels) or concurrent thread blocks represent coarse-grained data parallelism or task parallelism
 - Concurrent threads/warps represent fine-grained data parallelism or thread parallelism

SIMD vs SPMD

SIMD

- Processing units are executing the same instruction at any instant

SPMD

- Parallel processing units execute the same program on multiple parts of the data
- All the processing units may not execute the same instruction at the same time

Memory Hierarchy

Memory Access Efficiency

- Compute to global memory access ratio
 - Number of floating-point operations performed for each access to global memory

```
for (int i = 0; i < N; i++)  
    tmp += A[i*N+K]*B[k*N+j];
```

- Assume a GPU device with 1 TB/s global memory bandwidth and peak single-precision performance of 12 TFLOPS
 - What is the performance we expect with an access ratio of 1?
 - We can do 1000/4 GFLOPS, which is only ~2% of the peak performance

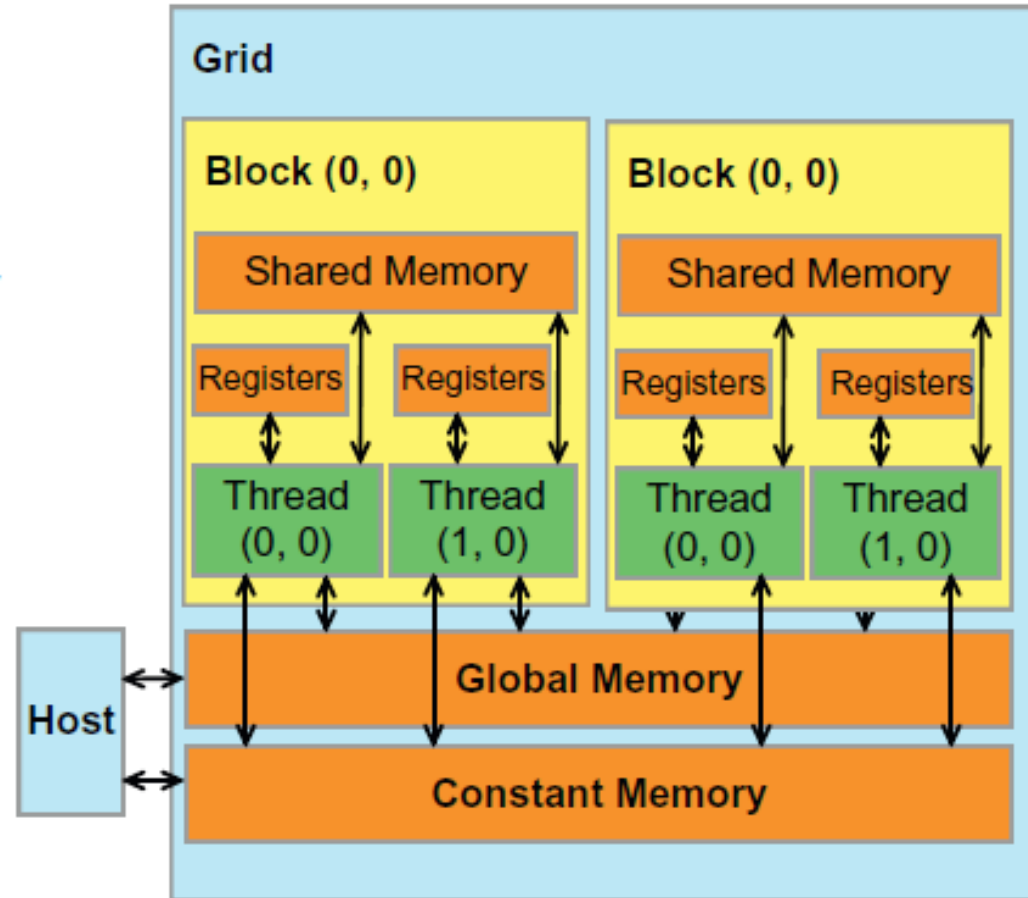
Memory Hierarchy in CUDA

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories



Variable Type Qualifiers in CUDA

| | Memory | Scope | Lifetime |
|---|----------|--------|-------------|
| <code>int localVar</code> | Register | Thread | Kernel |
| <code>__device__ __local__ int localVar</code> | Local | Thread | Kernel |
| <code>__device__ __shared__ int sharedVar</code> | Shared | Block | Kernel |
| <code>__device__ int globalVar</code> | Global | Grid | Application |
| <code>__device__ __constant__ int constVar</code> | Constant | Grid | Application |

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
 - Except arrays that reside in local memory
- Pointers can only point to memory allocated or declared in global memory

Memory Organization

- Host and device maintain their own separate memory spaces
 - A variable in CPU memory may not be accessed directly in a GPU kernel
- It is programmer's responsibility to keep them in sync
 - A programmer needs to maintain copies of variables

Registers

- 64K 32-bit registers per SM
 - So 256KB register file per SM, a CPU in contrast has a few (1-2 KB) per core
- Up to 255 registers per thread (compute capability 3.5+)
- If a code uses the maximum number of registers per thread (255) and an SM has 64K registers, then the SM can support a maximum of 256 threads
- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread

Registers

- 64K 32-bit registers per SM
 - So 256KB register file per SM, a CPU in contrast has a few (1-2 KB) per core
- Up to 255 registers per thread
- If a code uses more than 255 registers, it will not run on an SM that has a maximum of 256 threads
- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread

What if each thread uses 33 registers?

Registers

- If we use the maximum allowable number of threads per SM (2048), then each thread can use at most 32 registers per thread
- What if each thread uses 33 registers?
 - Fewer threads => fewer warps
- There is a big difference between “fat” threads which use lots of registers, and “thin” threads that require very few!

Shared Memory

- Shared memory aims to bridge the gap in memory speed and access
 - Also called scratchpad memory
 - Usually 16-64KB of storage that can be accessed efficiently by all threads in a block
- Primary mechanism in CUDA for efficiently supporting thread cooperation
- Each SM contains a single shared memory
 - Resides adjacent to an SM, on-chip
 - The space is shared among all blocks running on that SM

Shared Memory

- Variable in shared memory is allocated using the `__shared__` specifier
 - Faster than global memory
 - Can be accessed only by threads within a block
- **Amount of shared memory per block limits occupancy**

- Say an SM with 4 thread blocks has 16 KB of shared memory

```
__shared__ float min[256];  
__shared__ float max[256];  
__shared__ float avg[256];  
__shared__ float stdev[256];
```

Registers vs Shared Memory

Registers

- Faster than shared memory
- Private to a thread

Shared Memory

- On-chip memory space, requires load/store operations
- Visible to all threads in a block

Global Variables

- Variable lock can be accessed by both kernels
 - Resides in global memory space
 - Can be both read and modified by all threads

```
__device__ int lock=0;

__global__ void kernel1(...) {
    // Kernel code
}

__global__ void kernel2(...) {
    // Kernel code
}
```

Global Memory

- On-device memory accessed via 32, 64, or 128 B transactions
- A warp executes an instruction that accesses global memory
 - The addresses are coalesced into transactions
 - Number of transactions depend on the access size and distribution of memory addresses
 - More transactions mean less throughput
 - For example, if 32 B transaction is needed for a thread's 4 B access, throughput is essentially $1/8^{\text{th}}$

Constant Memory

- Used for data that will not change during kernel execution
 - Constant memory is 64KB
- Constant memory is cached
 - Each SM has a read-only constant cache that is shared by all cores in the SM
 - Used to speed up reads from the constant memory space which resides in device memory
 - Read from constant memory incurs a memory latency on a miss
 - Otherwise, it is a read from constant cache, which is almost as fast as registers

Constant Variables

- Constant variables cannot be modified by kernels
 - Reside in constant memory
 - Accessible from all threads within a grid
- They are defined with global scope within the kernel using the prefix `__constant__`
- Host code can access via `cudaMemcpyToSymbol()` and `cudaMemcpyFromSymbol()`

Local Memory

- Local memory is off-chip memory
 - More like thread-local global memory, so it requires memory transactions and consumes bandwidth
- Automatic variables are placed in local memory
 - Arrays for which it is not known whether indices are constant quantities
 - Large structures or arrays that consume too much register space
 - In case of register spilling
- Inspect PTX assembly code (compile with `-ptx`)
 - Check for `ld.local` and `st.local` mnemonic

Device Memory Management

- Global device memory can be allocated with `cudaMalloc()`
- Freed by `cudaFree()`
- Data transfer between host and device is with `cudaMemcpy()`
- Initialize memory with `cudaMemset()`

- There are asynchronous versions

GPU Caches

- GPUs have L1 and L2 data caches on devices with CC 2.x and higher
 - Texture and constant cache are available on all devices
- L1 cache is write-through, and per SM
 - Shared memory is partitioned out of unified data cache and its size can be configured, remaining portion is the L1 cache
 - Can be configured as 48 KB of shared memory and 16 KB of L1 cache, or 16 KB of shared memory and 48 KB of L1 cache, or 32 KB each
 - L1 caches are 16-48 KB
- L2 cache is shared by all SMs
- L1 cache lines are 128 B wide in Fermi onward, while L2 lines are 32 B

CPU Caches vs GPU caches

CPU

- Data is automatically moved by hardware between caches
 - Association between threads and cache does not have to be exposed to programming model
- Caches are generally coherent

GPU

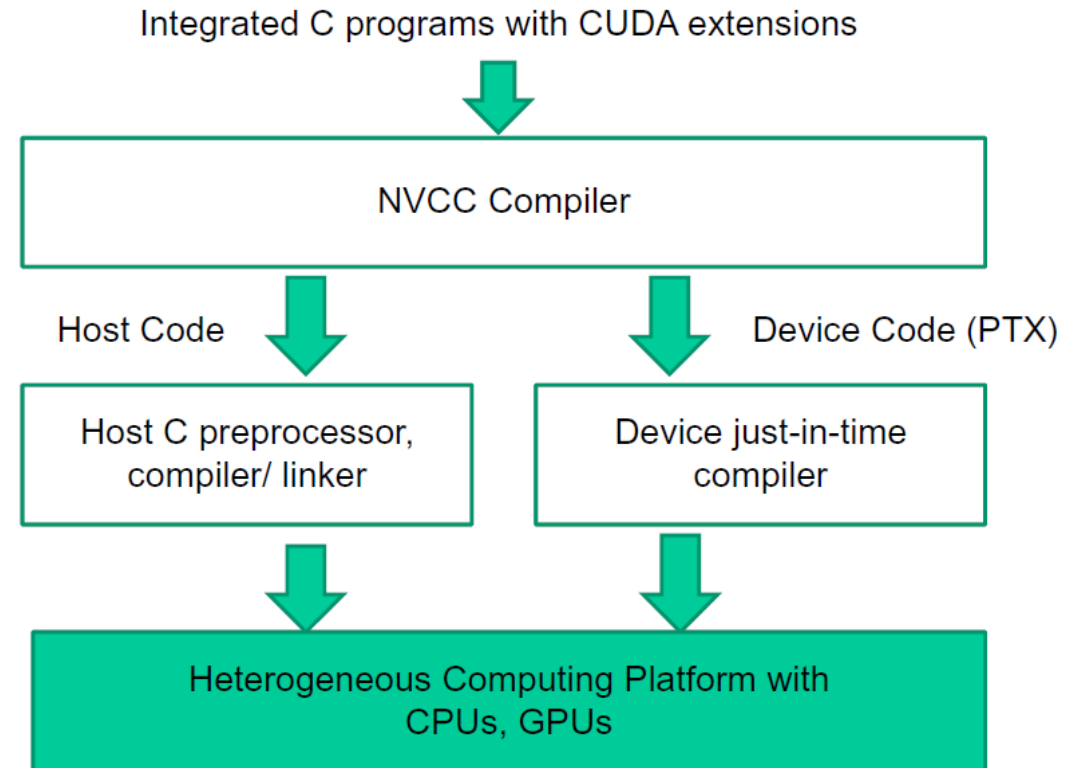
- Data movement must be orchestrated by programmer
 - Association between threads and storage is exposed to programming model
- L1 cache is not coherent, L2 cache is coherent

CUDA Compilation

Binary compatibility of GPU applications is not guaranteed across different generations

How NVCC works?

- Nvcc is a driver program based on LLVM
 - Compiles and links all input files
 - Requires a general-purpose C/C++ host compiler
 - Uses gcc and g++ by default on Linux platforms
 - `nvcc --version`



<https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html>

NVCC Details

| Important Options | Description |
|---|--|
| <code>-std {c++03 c++11 c++14}</code> | Select a particular C++ dialect |
| <code>-m {32 64}</code> | Specify the architecture |
| <code>-arch ARCH</code> | Specify the class of the virtual GPU architecture |
| <code>-code CODE</code> | Specify the name of the GPU to assemble and optimize for |

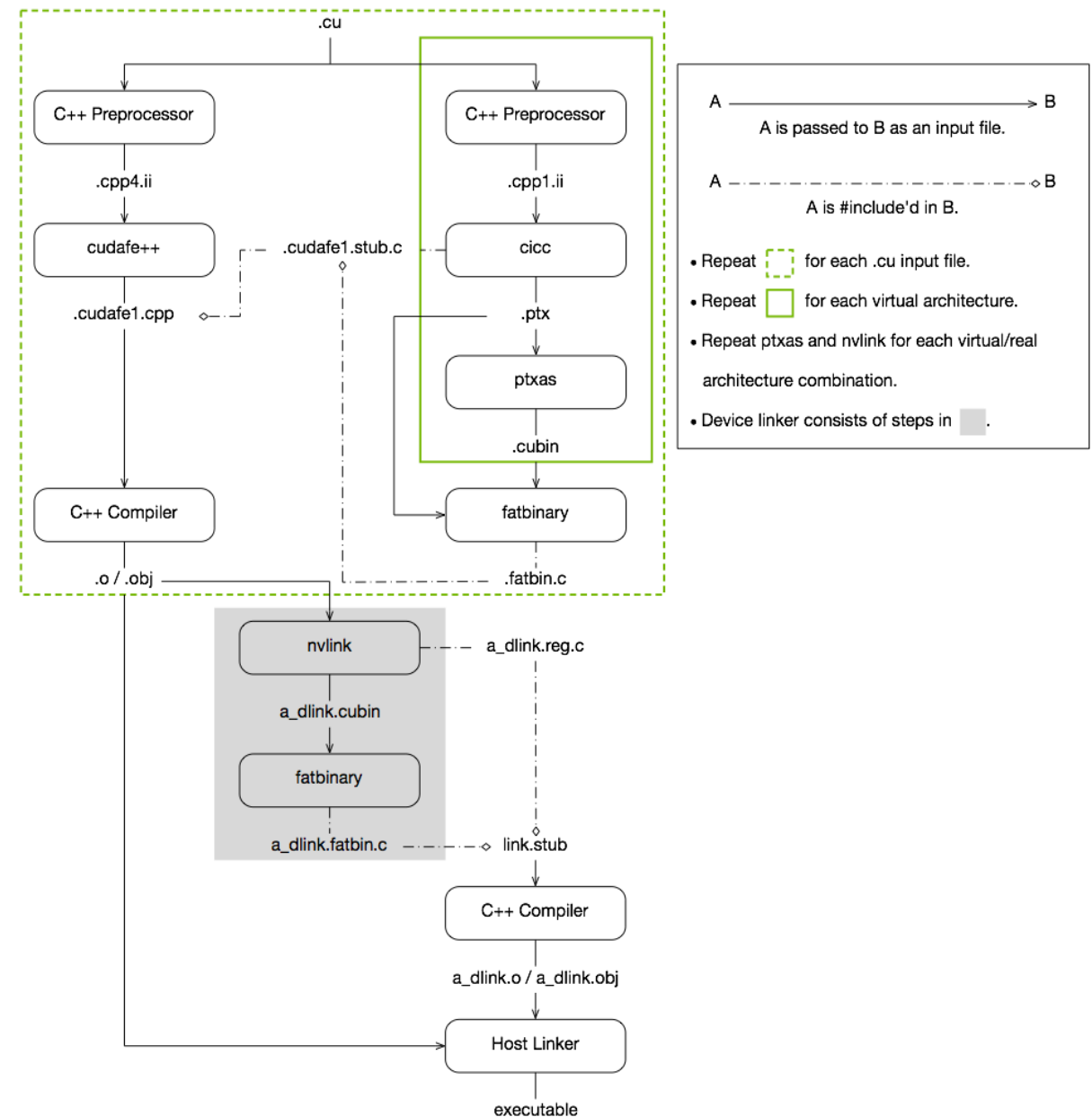
| Input File Type | Description |
|----------------------------------|--|
| <code>.cu</code> | CUDA source file |
| <code>.c, .cpp, .cxx, .cc</code> | C/C++ source files |
| <code>.ptx</code> | PTX intermediate assembly |
| <code>.cubin</code> | CUDA device binary code for a single GPU architecture |
| <code>.fatbin</code> | CUDA fat binary file that may contain multiple PTX and CUBIN files |
| <code>.a, .so, .lib</code> | ... |

CUDA Compilation Trajectory

- Conceptually, the flow is as follows
 - Input program is preprocessed for device compilation
 - It is compiled to a CUDA binary (.cubin) and/or PTX (Parallel Thread Execution) intermediate code which are encoded in a fatbinary
 - Input program is processed for compilation of the host code
 - CUDA-specific C++ constructs are transformed to standard C++ code
 - Synthesized host code and the embedded fatbinary are linked together to generate the executable

CUDA Compilation Trajectory

- A compiled CUDA device binary includes
 - Program text (instructions)
 - Information about the resources required
 - N threads per block
 - X bytes of local data per thread
 - M bytes of shared space per block



NVCC Details

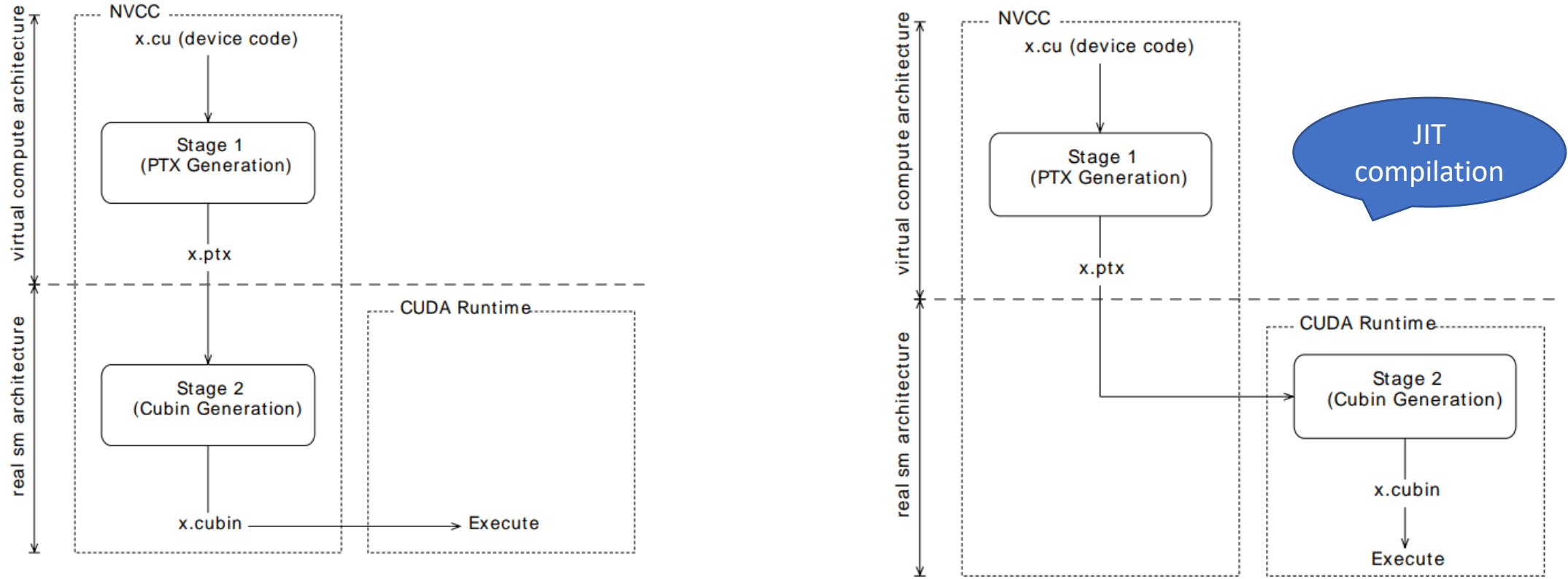
| Important Options | Description | Input File Type | Description |
|-------------------|--|-----------------|--|
| -std {c++} | Select a particular C++ | .cu | CUDA source file |
| -m | | | |
| -arch | | | |
| | architecture | | |
| -code CODE | Specify the name of the GPU to assemble and optimize for | .fatbin | CUDA fat binary file that may contain multiple PTX and CUBIN files |
| | | .a, .so, .lib | ... |

```
nvcc -arch=compute_30 -code=sm_52 hello-world.cu
```

```
nvcc -arch=compute_30 -code=sm_30,sm_52 hello-world.cu
```

NVIDIA. CUDA Compiler Driver NVCC. v11.8.

Two-Stage Compilation with nvcc



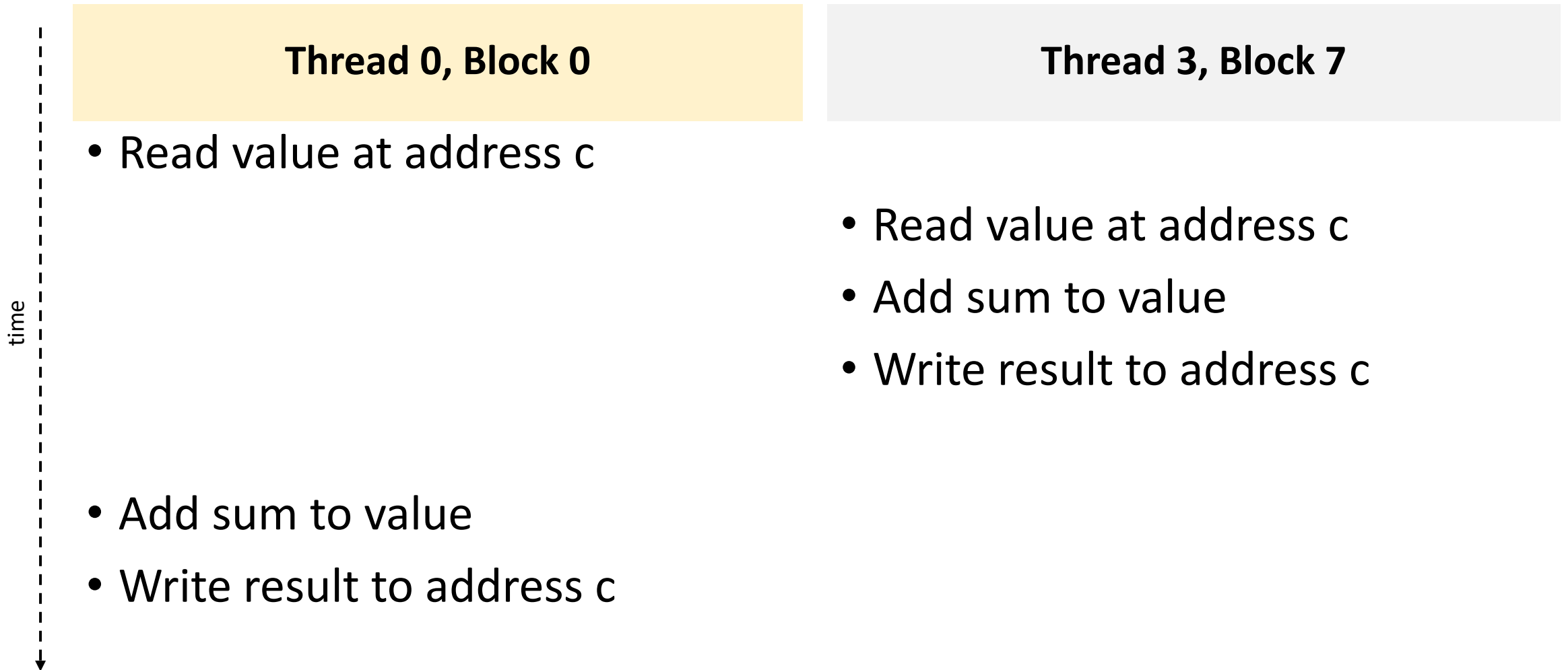
- `nvcc -arch=compute_50 -code=compute_50 hello-world.cu`
- `nvcc -arch=compute_50 -code=compute_50,sm_50,sm_52 hello-world.cu`

Synchronization in CUDA

Race Conditions and Data Races

- A race condition occurs when program behavior depends upon relative timing of two (or more) event sequences
- Execute: `*c += sum;`
 - Read value at address `c`
 - Add `sum` to value
 - Write result to address `c`
- There can be intra-warp, inter-warp, and inter-block races

Be Careful to Avoid Race Conditions!



Synchronization Constructs in CUDA

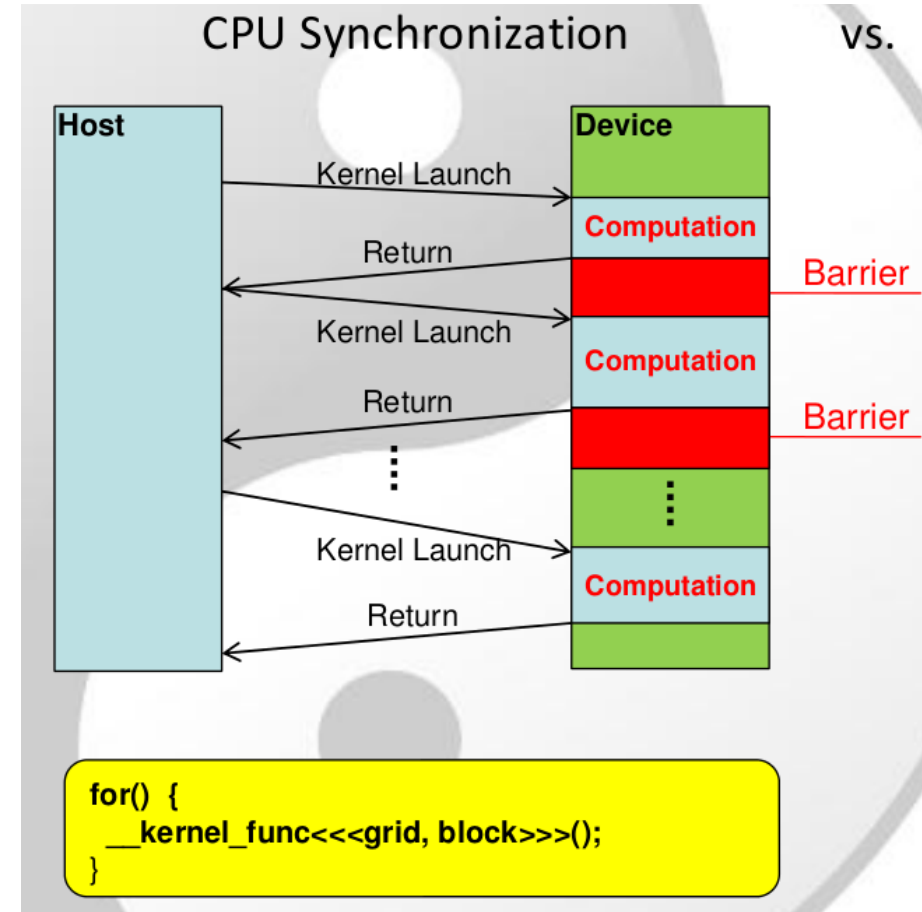
1. `__syncThreads()` synchronizes threads within a block
2. `cudaDeviceSynchronize()` synchronizes all threads in a grid
 - There are other variants
3. Atomic operations prevent conflicts associated with multiple threads concurrently accessing a variable
 - Atomic operations on both global memory and shared memory variables
 - For e.g., `float atomicAdd(float* addr, float amount)`

__syncthreads()

- A `__syncthreads()` statement must be executed by all threads in a block
- `__syncthreads()` is in an `if` statement
 - Either all threads in the block execute the path that includes the `__syncthreads()` or none of them does
- `__syncthreads()` statement is in each path of an `if-then-else` statement
 - Either all threads in a block execute the `__syncthreads()` on the then path or all of them execute the else path
 - The two `__syncthreads()` are different barrier synchronization points

Synchronization Between Grids

- For threads from different grids, system ensures writes from kernel happen before reads from subsequent grid launches



Atomic Operations

- Perform read-modify-write (RMW) atomic operations on data residing in global or shared memory
 - `atomicAdd()`, `atomicSub()`, `atomicMin()`, `atomicMax()`,
`atomicInc()`, `atomicDec()`, `atomicExch()`, `atomicCAS()`
- Predictable result when simultaneous access to memory required

Concurrency and CUDA Streams

Overlap host and device computation with data transfers

Classic Copy-then-Execute Model

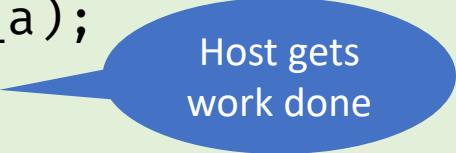
```
1. cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);  
2. kernel1<<<1,N>>>(d_a);  
3. cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- Data transfer on line 1 is blocking or synchronous
 - Host thread cannot launch the kernel until the copy is done
- Kernel launch is asynchronous
- Data transfer on line 3 cannot begin due to the device-side ordering (i.e., until the kernel completes)

Overlap Host and Device Computation

```
cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);  
kernel1<<<1,N>>>(d_a);  
cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```

```
cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);  
kernel1<<<1,N>>>(d_a);  
h_func(h_b);  
cudaMemcpy(h_res, d_a, numBytes, cudaMemcpyDeviceToHost);
```



Host gets work done

Goal is to Utilize GPU Hardware

- Overlap kernel execution with memory copy between host and device
- Overlap execution of multiple kernels if there are enough resources

- Depends on whether the GPU architecture supports overlapped execution

CUDA Streams

- Sequence of operations that execute on the device in the order in which they were issued by the host
 - Operations across streams can interleave and run concurrently
- All GPU device operations run in a stream
 - The default “null” stream is used if no custom stream is specified, the default stream is synchronizing
 - No operation in the default stream will begin until all previously issued operations in any stream have completed
 - An operation in the default stream must complete before any other operation in any stream will begin

Using a Non-default Stream

- Manipulate non-default streams from the host

```
cudaStream_t stream1;  
cudaError_t result;  
result = cudaStreamCreate(&stream1);  
result = cudaStreamDestroy(stream1);
```

- Issue a data transfer to a non-default stream

```
result = cudaMemcpyAsync(d_a, a, N,  
                        cudaMemcpyHostToDevice, stream1);
```

- Specifying a stream during kernel launch is optional

```
kernel1<<<blocks, threads, bytes>>>(); // default/NULL stream  
kernel2<<<blocks, threads, bytes, stream1>>>();
```

Non-Default Streams

- Operations in a non-default stream are non-blocking with host
- Use `cudaDeviceSynchronize()`
 - Blocks host until all previously issued operations on the device have completed
- Cheaper alternatives
 - `cudaStreamSynchronize()`,
`cudaEventSynchronize()`, ...

```
cudaStream_t stream1;
cudaError_t res;
res = cudaStreamCreate(&stream1);
res = cudaMemcpyAsync(d_a, a, N,
    cudaMemcpyHostToDevice, stream1);
increment<<<1,N,0,stream1>>>(d_a);
// Block the host thread
cudaStreamSynchronize(stream1);
res = cudaStreamDestroy(&stream1);
```

Why Use CUDA Streams?

- Memory copy and kernel execution can be overlapped if they occur in different, non-default streams
 - Most recent GPUs are capable of “concurrent copy and execution”, can be queried from the `deviceOverlap/asyncEngineCount` field of the `cudaDeviceProp` struct
- Individual kernels can overlap if there are enough resources on the GPU

Overlapping Kernel Execution and Data Transfers

```
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;

    cudaMemcpyAsync(&d_a[offset], &h_a[offset], streamBytes,
                  cudaMemcpyHostToDevice, stream[i]);

    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);

    cudaMemcpyAsync(&h_a[offset], &d_a[offset], streamBytes,
                  cudaMemcpyDeviceToHost, stream[i]);
}
```

<https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/>

Overlapping Kernel Execution and Data Transfers

```
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &h_a[offset], streamBytes,
                  cudaMemcpyHostToDevice, stream[i]);
}

for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}

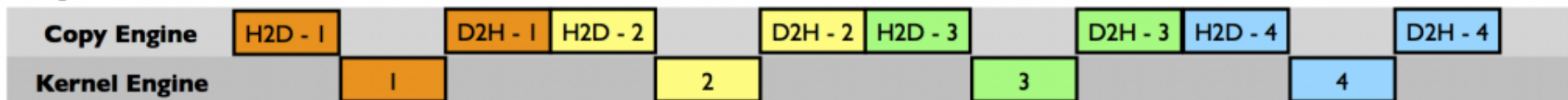
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&h_a[offset], &d_a[offset], streamBytes,
                  cudaMemcpyDeviceToHost, stream[i]);
}
```


CI060 Execution Time Lines

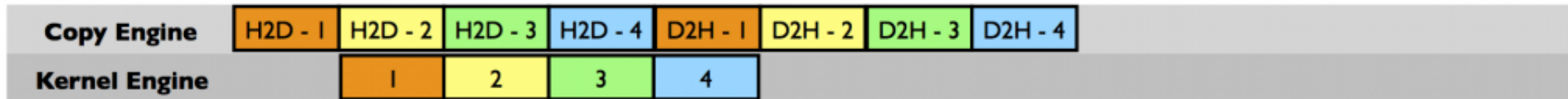
Sequential Version



Asynchronous Version 1



Asynchronous Version 2



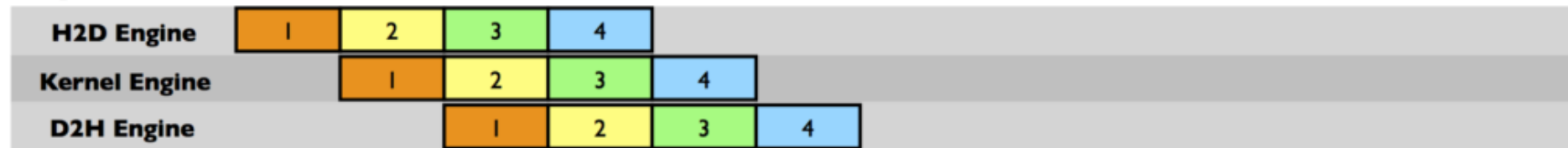
Time →

C2050 Execution Time Lines

Sequential Version



Asynchronous Version 1



Asynchronous Version 2



Time →

Concurrent Host Execution

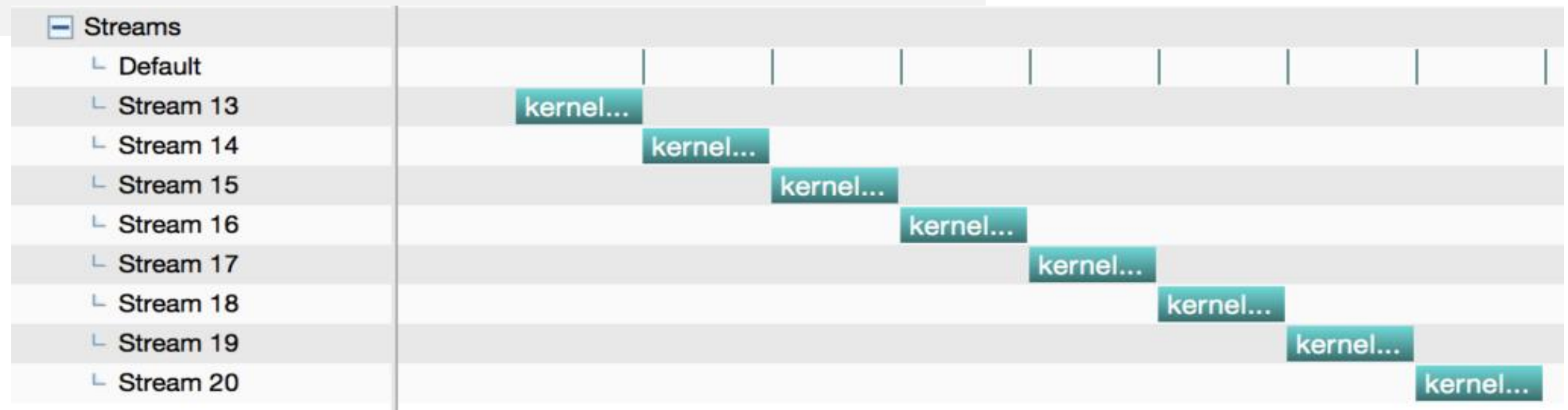
- Asynchronous functions are nonblocking
 - kernel launches
 - memory copies from host to device of a memory block of 64 KB or less;
 - memory copies performed by functions that are suffixed with Async

Streams and Concurrency in CUDA 7+

- Prior to CUDA 7, all host threads shared the default stream
 - Implied synchronization
- CUDA 7+ provides an option to have a per-host-thread default stream
 - Commands issued to the default stream by different host threads can run concurrently
 - Commands in the default stream may run concurrently with commands in non-default streams

Multi-Stream Example: Legacy Behavior

```
for (int i = 0; i < num_streams; i++) {  
    cudaStreamCreate(&streams[i]);  
  
    cudaMalloc(&data[i], N * sizeof(float));  
  
    // launch one worker kernel per stream  
    kernel<<<1, 64, 0, streams[i]>>>(data[i], N);  
  
    // launch a dummy kernel on the default stream  
    kernel<<<1, 1>>>(0, 0);  
}
```



Multi-Stream Example: Per-Thread Default Stream

```
nvcc --default-stream per-thread <file.cu>
```

| Streams | |
|-------------|---------------------|
| └ Stream 13 | kernel(float*, int) |
| └ Stream 14 | |
| └ Stream 15 | kernel(float*, int) |
| └ Stream 16 | kernel(float*, int) |
| └ Stream 17 | kernel(float*, int) |
| └ Stream 18 | kernel(float*, int) |
| └ Stream 19 | kernel(float*, int) |
| └ Stream 20 | kernel(float*, int) |
| └ Stream 21 | kernel(float*, int) |

Performance Bottlenecks with CUDA

Differences between Host and Device

Host

- Limited amount of concurrent threads
- Context switches of threads are heavyweight
- Designed to minimize latency

Device

- Massive number of concurrently active threads
- Context switches are lightweight
 - Resources stay allocated to a thread till it completes
- Designed to maximize throughput

Desired Application Characteristics for Device Execution

- Large data-parallel computation
- Complex computation kernel to justify the data movement costs
 - Think of matrix addition versus matrix multiplication
 - Keep data on the device to avoid repeated transfers

Key Ideas for Performance

- Try and reduce resource consumption
- Exploit SIMT, reduce thread divergence in a warp
- Strive for good locality, use tiling to exploit shared memory
 - Improve throughput by reducing global memory traffic
 - Copy blocks of data from global memory to shared memory and operate on them (e.g., matrix multiplication kernel)
- Memory access optimization
 - Global memory: memory coalescing
 - Shared memory: avoid bank conflicts

What can we say about this code?

```
__global__ void dkernel(float *vector, int vectorsize) {  
    int id = blockIdx.x * blockDim.x + threadIdx.x;  
    switch (id) {  
        case 0: vector[id] = 0; break;  
        case 1: vector[id] = vector[id] * 10; break;  
        case 2: vector[id] = vector[id - 2]; break;  
        case 3: vector[id] = vector[id + 3]; break;  
        ...  
        case 31: vector[id] = vector[id] * 9; break;  
    }  
}
```

Deal with Thread Divergence

- Thread divergence renders execution sequential
 - SIMD hardware takes multiple passes through the divergent paths
- Condition evaluating to different truth values is not bad

```
if (threadIdx.x / WARP_SIZE > 2) {}
```

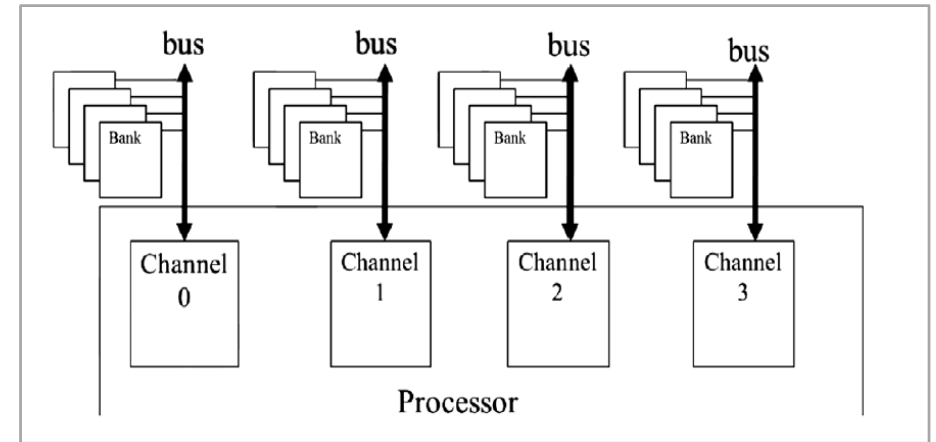
- Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
- Conditions evaluating to different truth-values for threads in a warp is bad

```
if (threadIdx.x > 2) {}
```

- Creates two different control paths for threads in a block; branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp

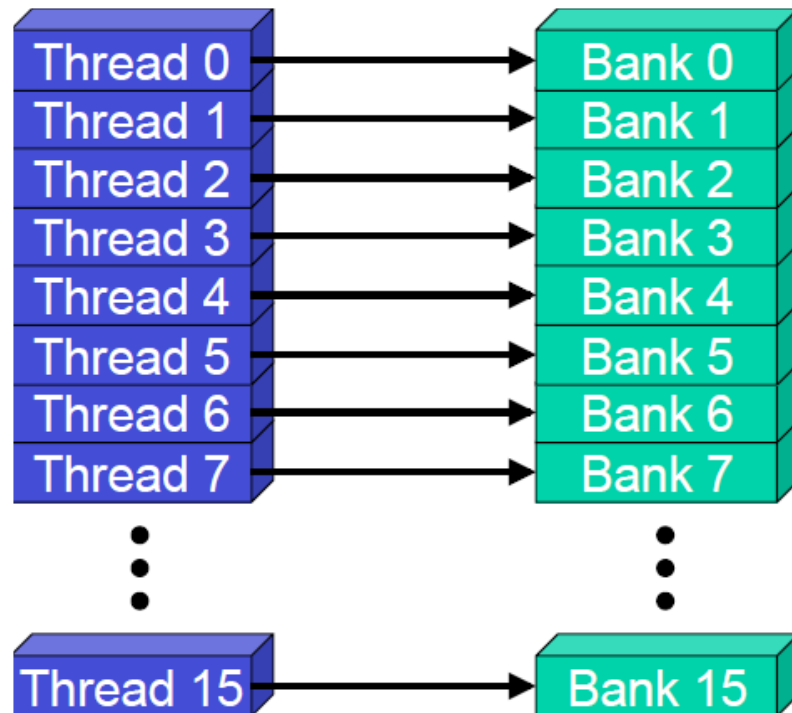
Parallel Memory Architecture

- In a parallel machine, many threads access memory
- Memory is divided into banks to achieve high bandwidth
 - Each bank can service one address per cycle
 - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
 - Conflicting accesses are serialized

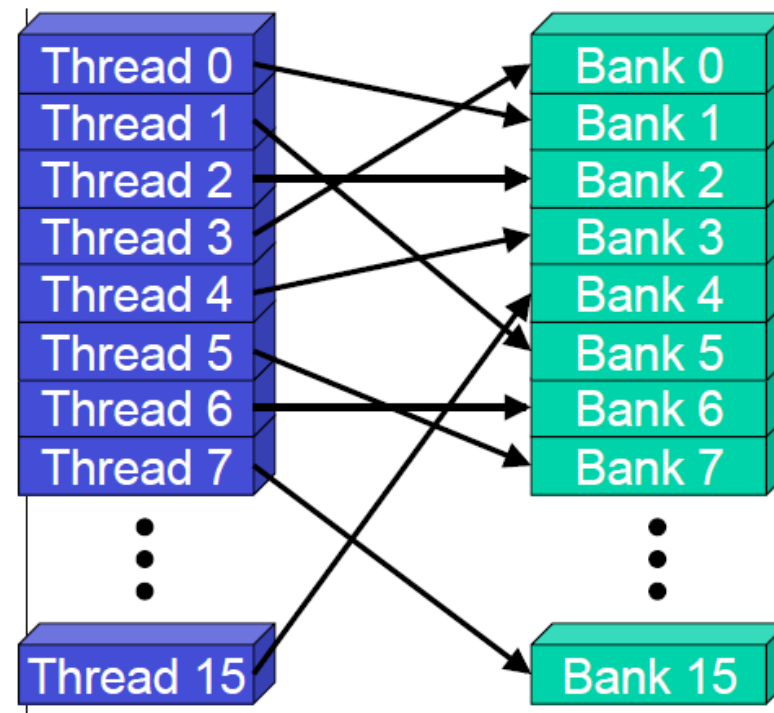


Example of Bank Addressing

- No bank conflicts
 - Linear addressing, stride=1



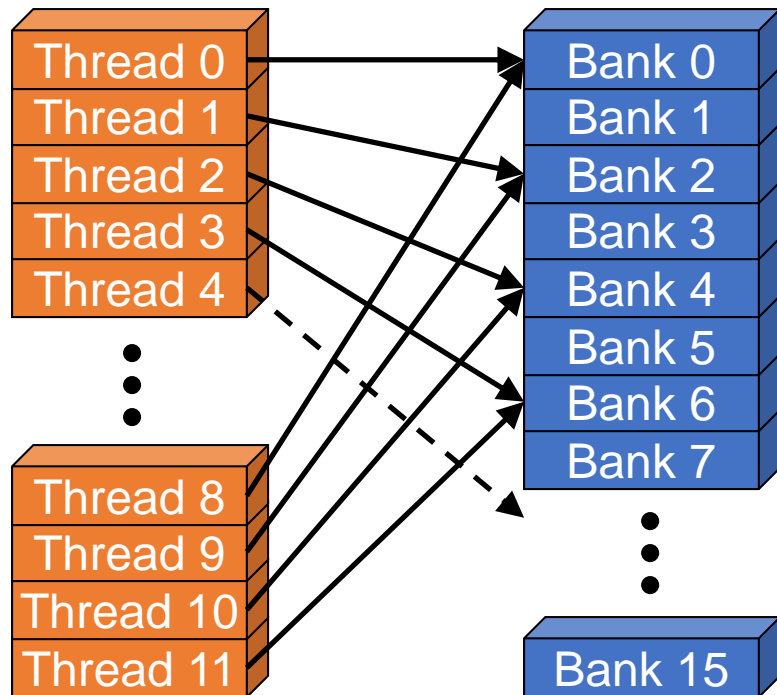
- No bank conflicts
 - Random permutation



Example of Bank Addressing

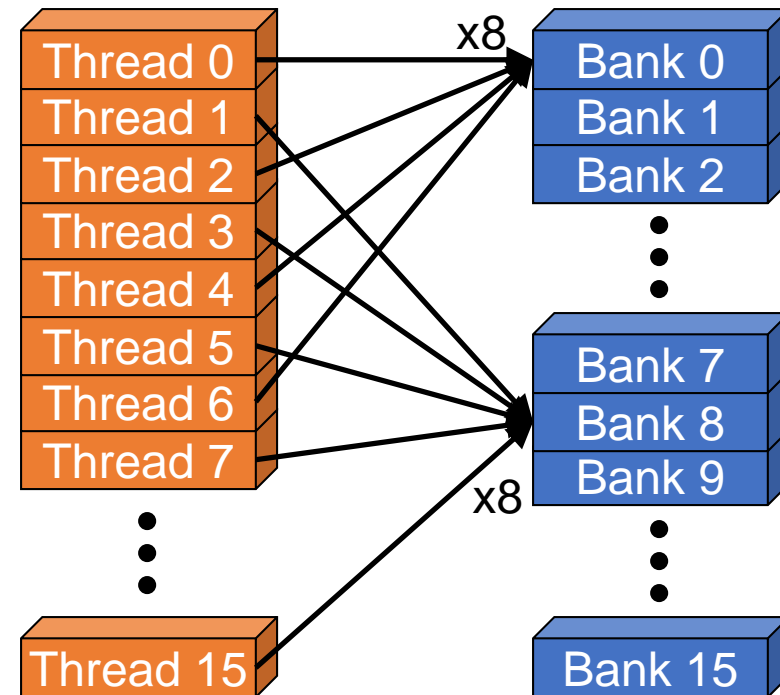
- 2-way Bank Conflicts

- Linear addressing, stride = 2



- 8-way Bank Conflicts

- Linear addressing, stride = 8

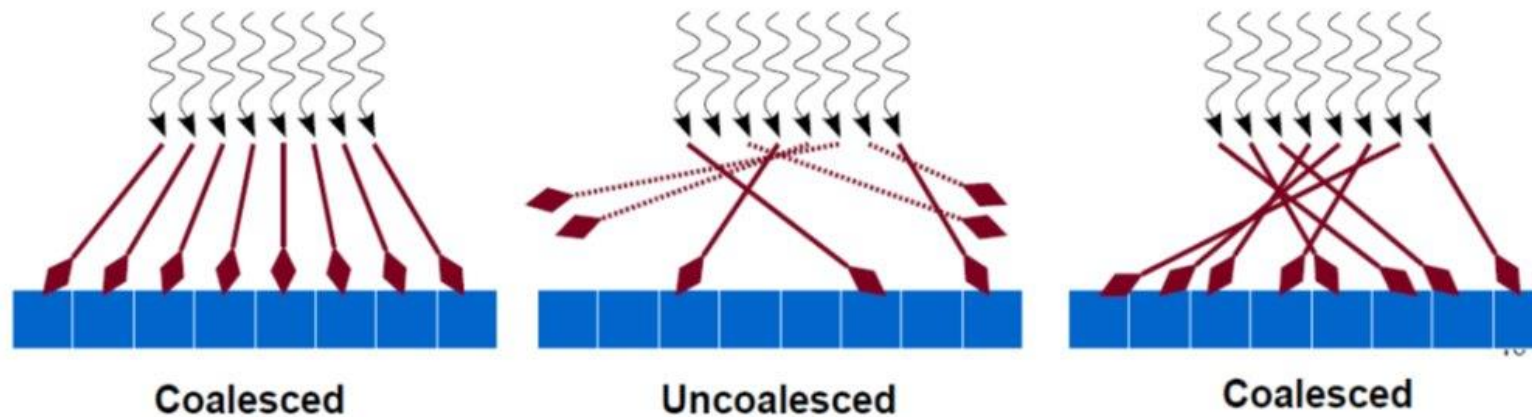


Shared Memory Bank Conflicts

- Shared memory is as fast as registers if there are **no bank conflicts**
- Fast case
 - If all threads of a warp access different banks, there is no bank conflict
 - If all threads of a warp access the identical address, there is no bank conflict (broadcast)
- Slow case
 - Bank Conflict: multiple threads in the same half-warp access (?) the same bank
 - Must serialize the accesses
 - Cost = max # of simultaneous accesses to a single bank
- Give low priority to fix low-degree bank conflicts since resolving it will increase instructions

Memory Coalescing

- Coalesced memory access
 - A warp of threads access adjacent data in a cache line
 - In the best case, this results in one memory transaction (best bandwidth)
- Uncoalesced memory access
 - A warp of threads access scattered data all in different cache lines
 - This may result in 32 different memory transactions (poor bandwidth)



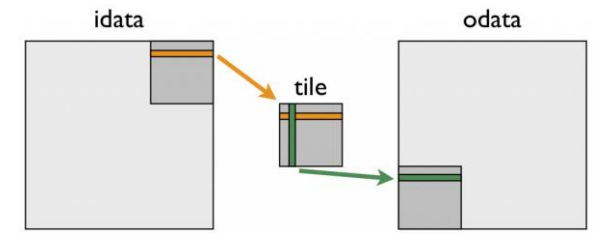
Matrix Transpose

```
__global__ void copy(float *odata, const float *idata) {  
    int x = blockIdx.x * TILE_DIM + threadIdx.x;  
    int y = blockIdx.y * TILE_DIM + threadIdx.y;  
    int width = blockDim.x * TILE_DIM;  
  
    for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)  
        odata[(y+j)*width + x] = idata[(y+j)*width + x];  
}
```

```
__global__ void transposeNaive(float *odata, const float *idata) {  
    int x = blockIdx.x * TILE_DIM + threadIdx.x;  
    int y = blockIdx.y * TILE_DIM + threadIdx.y;  
    int width = blockDim.x * TILE_DIM;  
  
    for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)  
        odata[x*width + (y+j)] = idata[(y+j)*width + x];  
}
```

reads from idata are coalesced,
but writes to odata have a stride
of 1024

Optimizing Matrix Transpose



```
__global__ void transposeCoalesced(float *odata, const float *idata) {
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = gridDim.x * TILE_DIM;

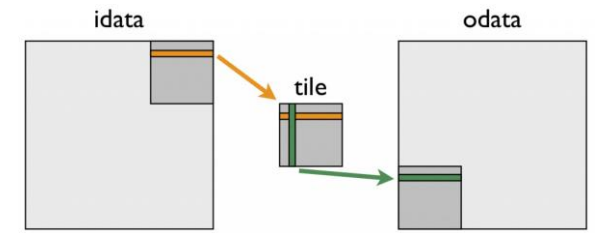
    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

    __syncthreads();

    x = blockIdx.y * TILE_DIM + threadIdx.x; // transpose block offset
    y = blockIdx.x * TILE_DIM + threadIdx.y;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

Optimizing Matrix Transpose



```
__global__ void transposeCoalesced(float *odata, const float *idata) {  
__sh $ ./bin/cuda-matrix-transpose-nvidia
```

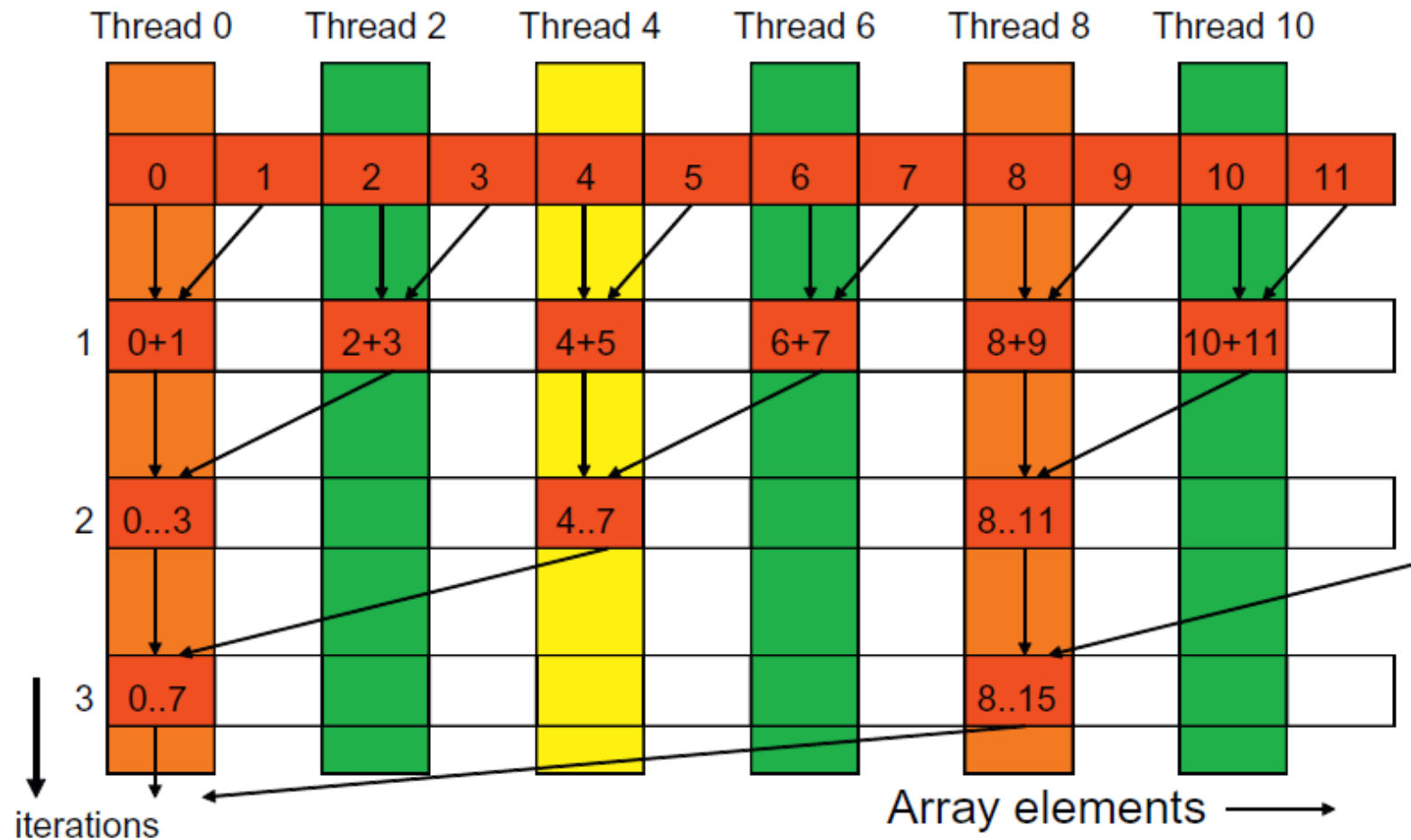
```
int Device : Quadro RTX 5000  
int Matrix size: 1024 1024, Block size: 64 8, Tile size: 64 64  
int dimGrid: 16 16 1. dimBlock: 64 8 1
```

| | Routine | Bandwidth (GB/s) |
|------|-------------------------|------------------|
| for | copy | 288.31 |
| | shared memory copy | 196.17 |
| | naive transpose | 124.92 |
| __sy | coalesced transpose | 131.81 |
| | conflict-free transpose | 145.65 |

```
x = blockIdx.y * TILE_DIM + threadIdx.x; // transpose block offset  
y = blockIdx.x * TILE_DIM + threadIdx.y;
```

```
for (int j = 0; j < TILE_DIM; j++)  
    odata[(y * TILE_DIM + j) * TILE_DIM + x] = idata[x * TILE_DIM + j];  
__shared__ float tile[TILE_DIM][TILE_DIM+1];  
}
```

Implement a Reduction Kernel in CUDA



Reduction Kernel

```
__shared__ float partialSum[];  
partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x+threadIdx.x];  
__syncthreads();  
  
unsigned int t = threadIdx.x;  
for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {  
    if (t % (2*stride) == 0)  
        partialSum[t] += partialSum[t+stride];  
    __syncthreads();  
}
```

only even threads
are active

Is this kernel
enough?

Sequence of Optimizations on the Reduction Kernel

- i. basic implementation with modulo operator
- ii. strided access starting with each thread accessing two adjacent locations
- iii. strided access with reversed loop index
- iv. halve the number of threads
- v. unroll the last few loop iterations and avoid synchronization
- vi. ...
- vii. ...



NVIDIA[®]

Optimizing Parallel Reduction in CUDA

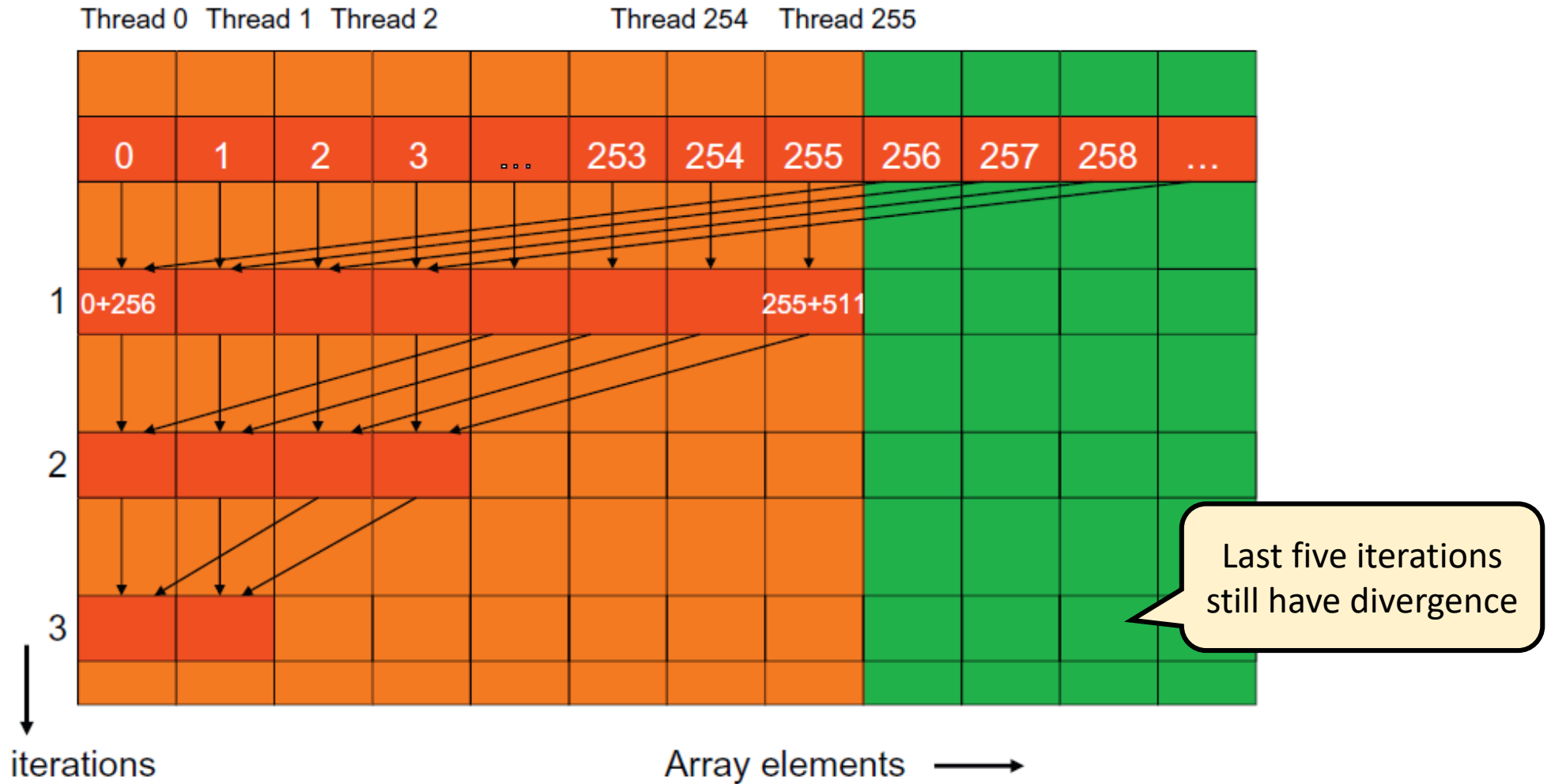
Mark Harris
NVIDIA Developer Technology

Reduction Kernel

```
__shared__ float partialSum[];
partialSum[threadIdx.x] = X[blockDim.x*blockDim.x+threadIdx.x];
__syncthreads();

unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2) {
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
    __syncthreads();
}
```


Execution of the Revised Kernel



Avoid Divergence in Last Few Iterations

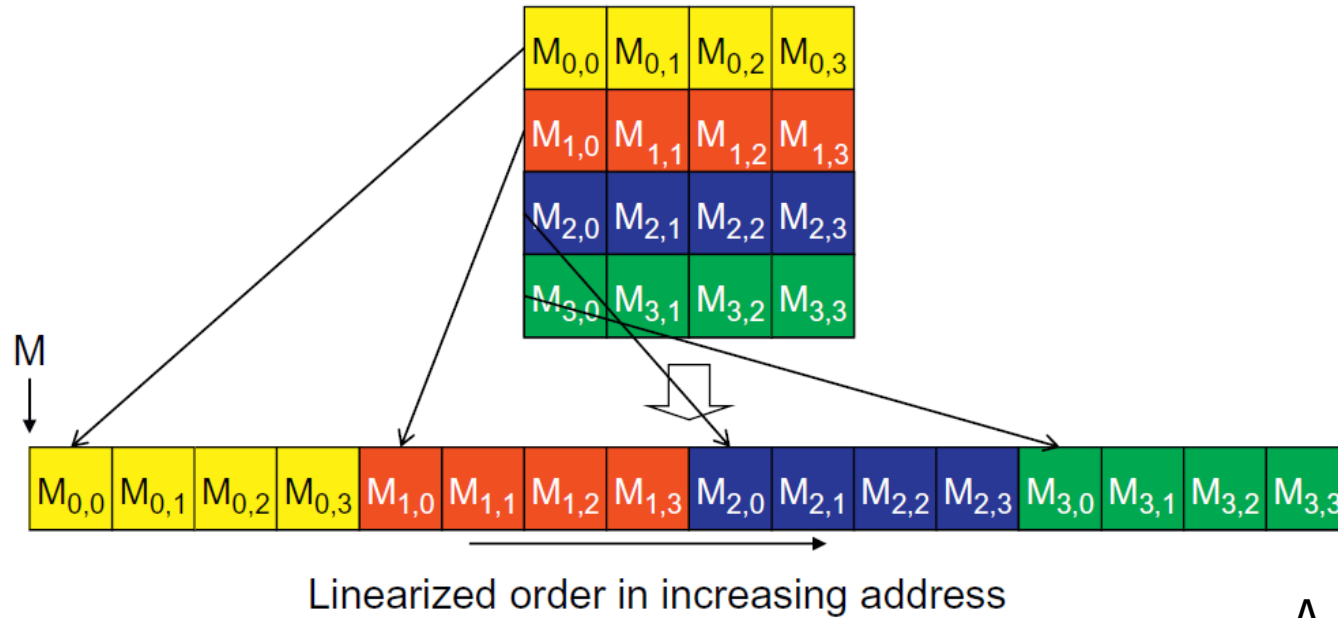
```
for (unsigned int stride = blockDim.x/2; stride > 32; stride /= 2) {
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
    __syncthreads();
}
if (tid < 32) {
    partialSum[tid] += partialSum[tid+32];
    partialSum[tid] += partialSum[tid+16];
    partialSum[tid] += partialSum[tid+8];
    partialSum[tid] += partialSum[tid+4];
    partialSum[tid] += partialSum[tid+2];
    partialSum[tid] += partialSum[tid+1];
}
```

Avoid Divergence in Last Few Iterations

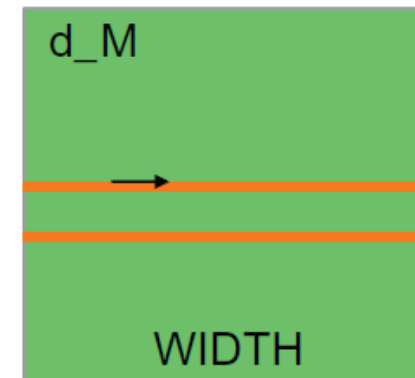
```
for (unsigned int stride = blockDim.x/2; stride > 32; stride /= 2) {  
    if (t < stride)  
        partialSum[t] += partialSum[t+stride];
```

```
$ ./a.out  
} Host reduction: 2048      Reduction1: 2048      Reduction2:  
i 2048      Reduction3: 2048      Reduction4: 2048  
Kernel1 time (ms): 0.02048 // interleaved w/ modulo  
Kernel2 time (ms): 0.00819 // interleaved, even threads  
Kernel3 time (ms): 0.00614 // sequential  
Kernel4 time (ms): 0.00544 // sequential w/ unrolling  
partialSum[tid] += partialSum[tid+4];  
partialSum[tid] += partialSum[tid+2];  
partialSum[tid] += partialSum[tid+1];  
}
```

Memory Access Patterns in 2D Arrays

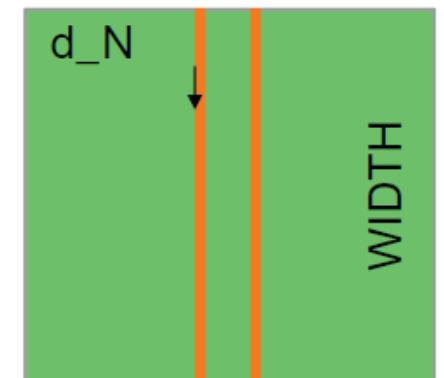


A not coalesced

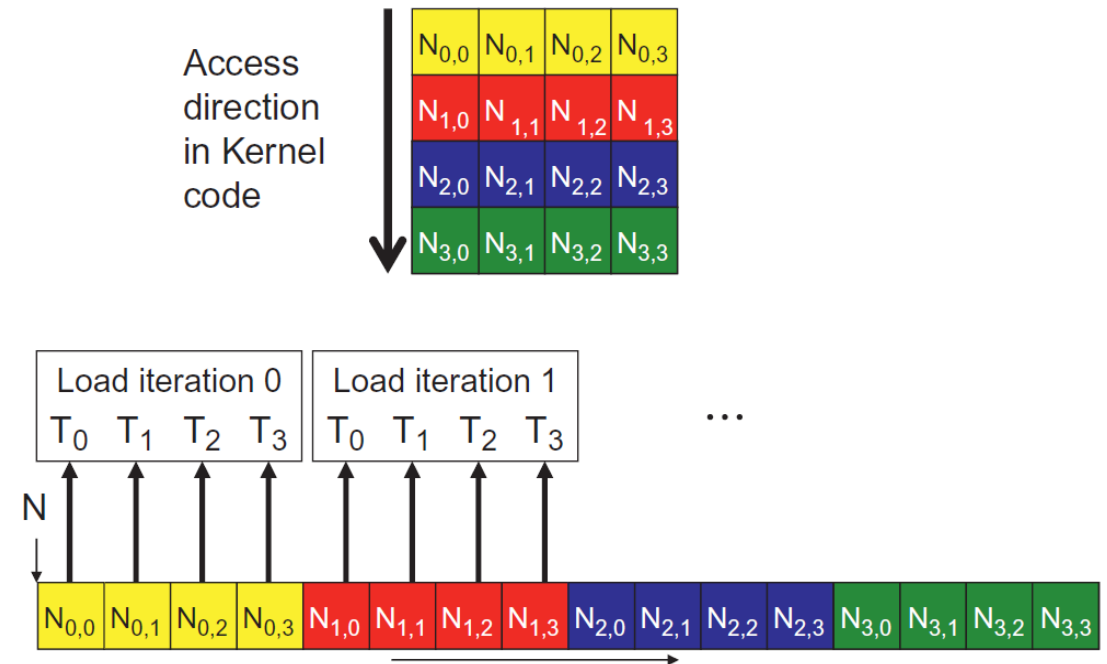
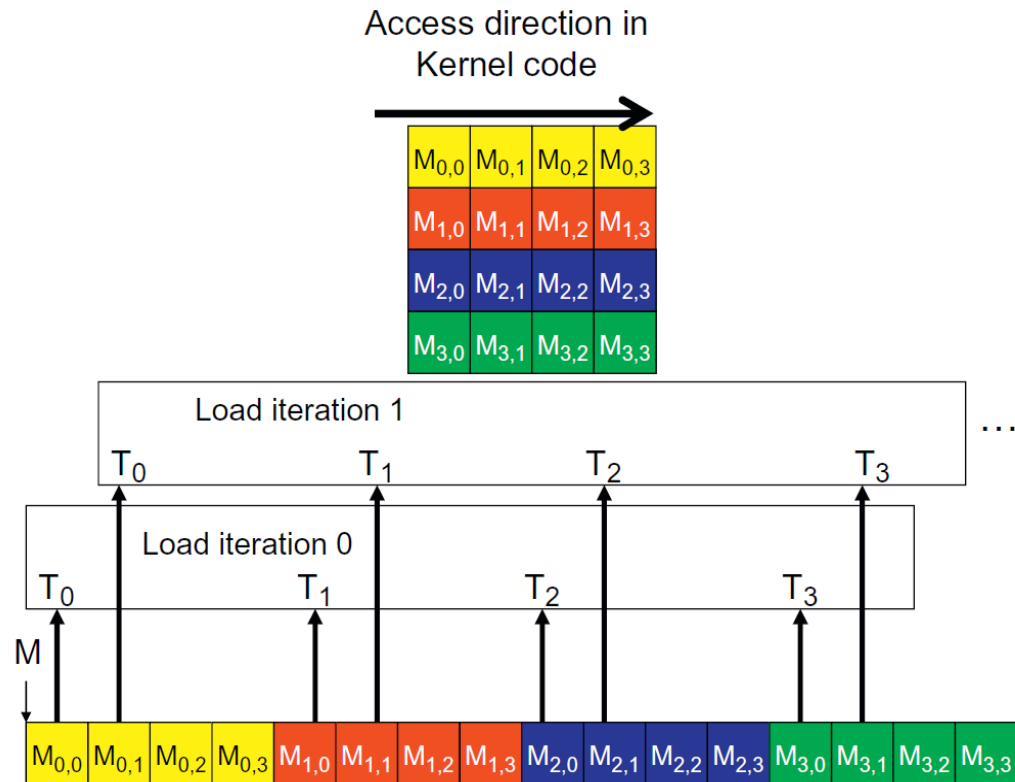


Thread 1
Thread 2

B coalesced

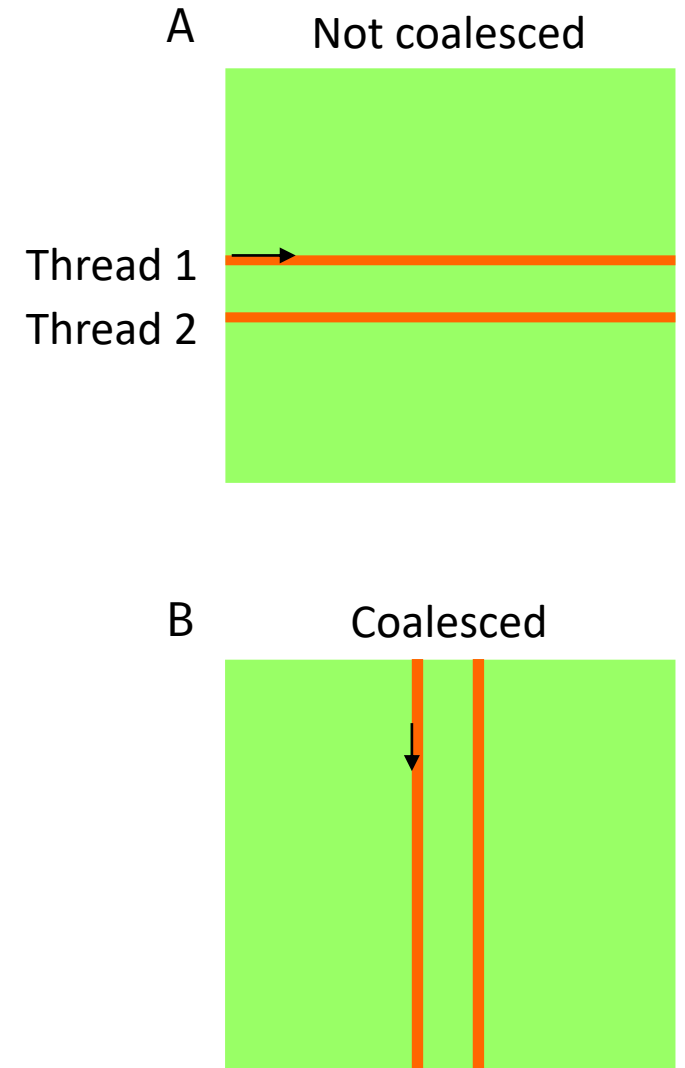


Memory Access Patterns in C 2D Arrays



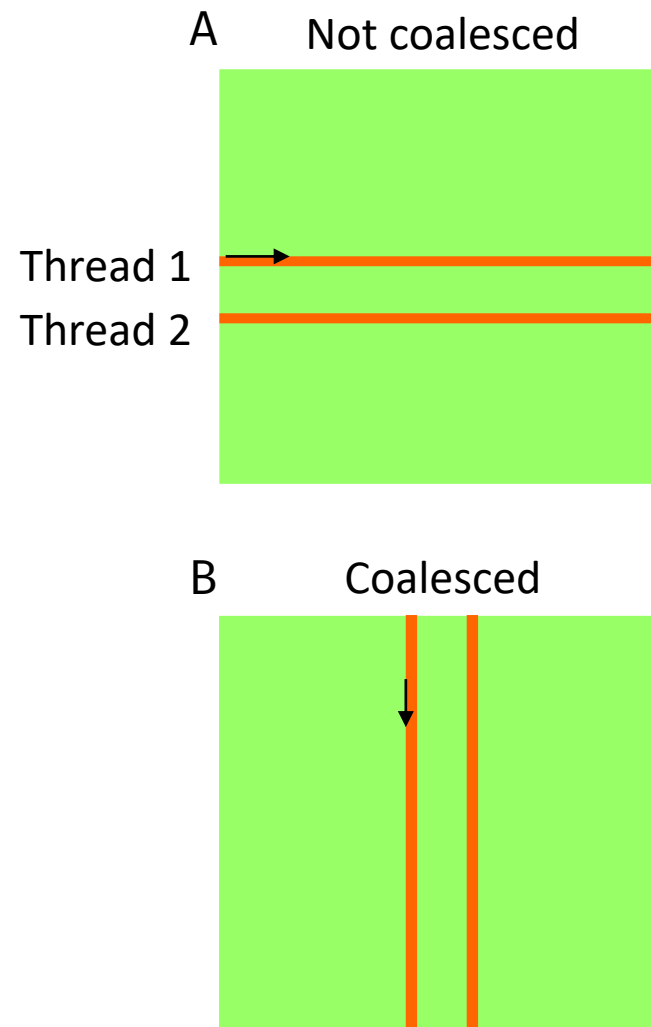
Matrix Multiplication Example

```
__global__ void matmulKernel(float* A, float* B, float* C) {  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
  
    float tmp = 0;  
    if (row < N && col < N) {  
        // Each thread computes one element of the matrix  
        for (int k = 0; k < N; k++) {  
            tmp += A[row * N + k] * B[k * N + col];  
        }  
    }  
    C[row * N + col] = tmp;  
}
```



Optimizing Global Memory Accesses

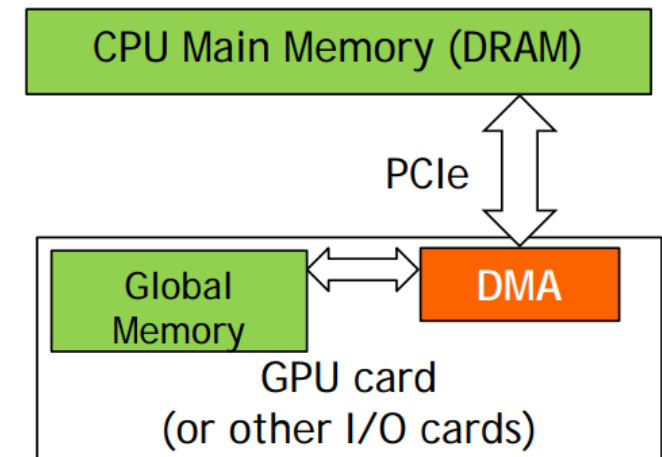
- Try to ensure that memory requests from a warp can be coalesced
 - Using optimizations like tiling to make use of the faster shared memory
 - Stride-one access across threads in a warp is good
 - Use structure of arrays rather than array of structures



Efficient Data Management

Data Transfer Between CPU and GPU

- DMA (Direct Memory Access) hardware is used by `cudaMemcpy()`
 - DMA is a hardware unit specialized to **transfer bytes between physical memory address spaces**
 - Uses system interconnect, typically PCIe in today's systems



Challenges with Virtual Memory

- Virtual memory support complicates data transfer
 - Pages in virtual address space are **mapped into and out** of the physical memory
 - The presence of a data (i.e., page) in the physical memory is checked during address translation
- `cudaMemcpy()` copies data as one or more DMA transfers
 - Address is translated and page presence is checked for the entire source and destination regions at the beginning of each DMA transfer
- The OS could accidentally page-out the data that is being accessed by a DMA and page-in another virtual page into the same physical location

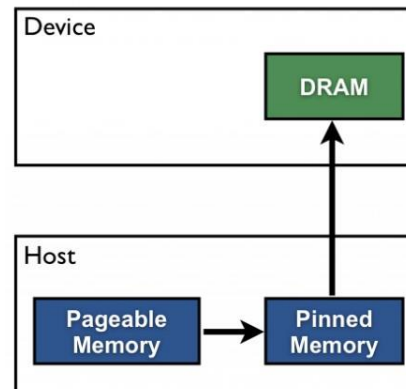
Pinned Memory

- **Pinned memory** are virtual memory pages that are specially marked so that they **cannot be paged out**
 - Also called Page Locked Memory, Locked Pages, etc.
- CPU memory that serves as the source or destination of a DMA transfer must be allocated as pinned memory
- If the source or destination of `cudaMemcpy()` in the host is not pinned, it needs to be **first copied to a pinned memory** leading to extra overhead
 - `cudaMemcpy()` is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed

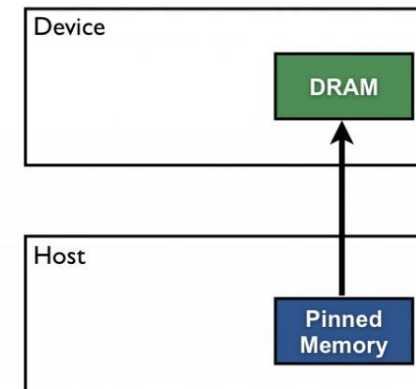
Pinned Memory

- Allocate and free pinned memory with `cudaHostAlloc()` and `cudaFreeHost()`
 - Use the option `cudaHostAllocDefault`
- Pinned memory is a limited resource – over-subscription can have serious consequences

Pageable Data Transfer



Pinned Data Transfer



cudaMemCpy() with malloc()

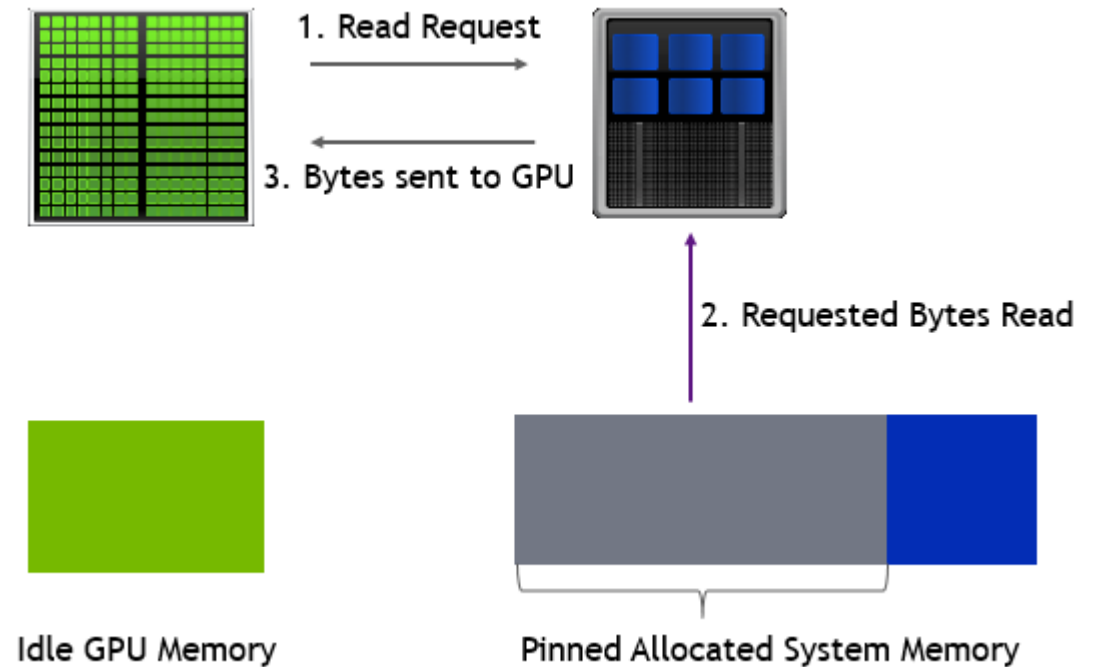
```
$ nvprof ./bin/cuda-vector-addition
==1472722== NVPROF is profiling process 1472722, command: ./bin/cuda-vector-
addition
Time taken (ms): h2d: 251.084 Kernel: 8.40627 d2h: 113.932
Type  Time(%)      Time      Calls      Avg      Min      Max  Name
GPU activities:  80.42%  501.62ms          2  250.81ms  250.79ms  250.84ms  [CUDA
memcpy HtoD]
                18.24%  113.77ms          1  113.77ms  113.77ms  113.77ms  [CUDA
memcpy DtoH]
                1.34%   8.3746ms          1   8.3746ms  8.3746ms  8.3746ms
vecAdd(float const *, float const *, float*, int)
  API calls:    67.15%  616.00ms          3  205.33ms  113.92ms  251.05ms
cudaMemcpy
                31.25%  286.69ms          3   95.562ms  919.48us  284.85ms
cudaMalloc
                0.92%   8.4529ms          3   2.8176ms  2.7130us  8.3720ms
cudaEventSynchronize
                0.65%   5.9759ms          3   1.9920ms  987.27us  2.9258ms
```

cudaMemCpy() with cudaHostAlloc(Default)

```
$ nvprof ./bin/cuda-vector-addition-hostalloc-default
==1472614== NVPROF is profiling process 1472614, command: ./bin/cuda-vector-
addition-hostalloc-default
Time taken (ms): h2d: 87.2246 Kernel: 8.42 d2h: 81.3472
Type  Time(%)      Time      Calls      Avg      Min      Max  Name
GPU activities:  66.04%  174.40ms          2  87.201ms  87.190ms  87.212ms  [CUDA
memcpy HtoD]
                30.79%  81.328ms          1  81.328ms  81.328ms  81.328ms  [CUDA
memcpy DtoH]
                3.17%  8.3739ms          1  8.3739ms  8.3739ms  8.3739ms
vecAdd(float const *, float const *, float*, int)
API calls:      68.72%  1.70102s          3  567.01ms  453.20ms  787.02ms
cudaHostAlloc
                20.23%  500.75ms          3  166.92ms  155.30ms  181.85ms
cudaFreeHost
                10.33%  255.79ms          3  85.263ms  81.344ms  87.244ms
cudaMemcpy
                0.34%  8.3757ms          3  2.7919ms  3.3700us  8.3687ms
```

Zero-Copy Memory (i)

- Zero copy memory is pinned memory that is mapped into the device address space
 - Both host and device have fine-grained direct access
 - Can leverage host memory when there is insufficient device memory
 - Avoids explicit data transfers between host and device
 - Should be used for occasional accesses when the data is read-only or the GPU memory is really scarce



Zero-Copy Memory (ii)

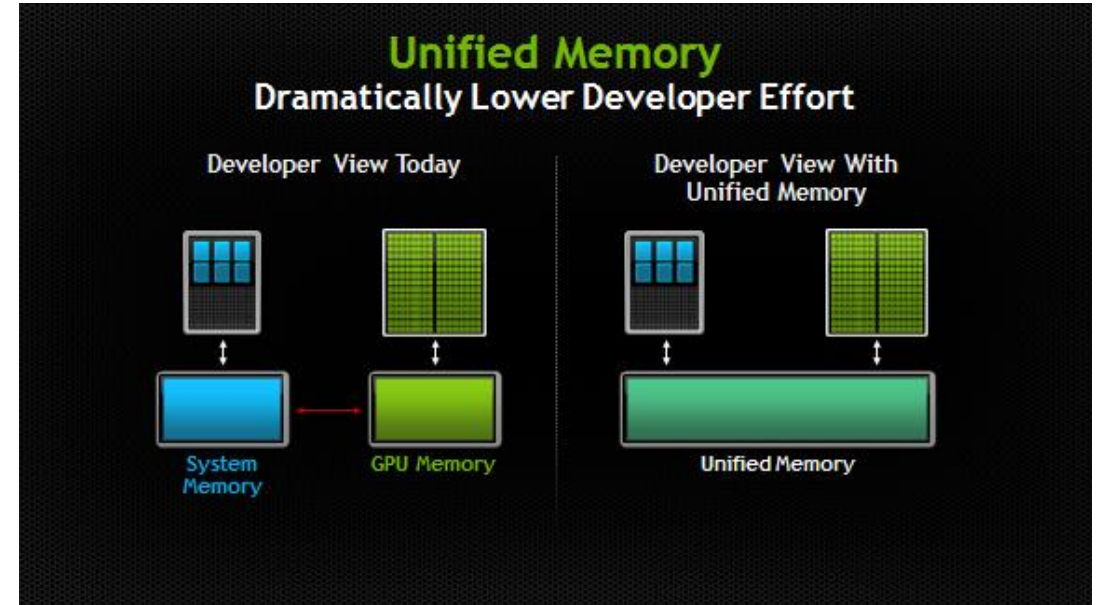
- Speed is limited by the interconnect (PCIe or NVLink) and it is not possible to take advantage of data locality
- Pinned memory does not imply zero-copy memory (GPU may not be able to access it)
- Unified virtual addressing (UVA) enables zero-copy memory
 - Provides a single virtual memory address space for all memory in the system, enables pointers to be accessed from GPU code

cudaMemCpy() with cudaHostAlloc(Mapped)

```
$ nvprof ./bin/cuda-vector-addition-hostalloc-mapped
==1472835== NVPROF is profiling process 1472835, command: ./bin/cuda-vector-
addition-hostalloc-mapped
Time taken (ms): kernel: 10.9529
==1472835== Profiling application: ./bin/cuda-vector-addition-hostalloc-mapped
==1472835== Profiling result:
          Type  Time(%)   Time         Calls          Avg           Min           Max    Name
GPU activities: 100.00%  10.914ms         1    10.914ms    10.914ms    10.914ms
vecAdd(float*, float*, float*, int)
  API calls:    67.19%  278.63ms         1    278.63ms    278.63ms    278.63ms
  cudaSetDeviceFlags
                20.90%  86.674ms         3    28.891ms    28.629ms    29.043ms
  cudaHostAlloc
                9.20%  38.156ms         3    12.719ms    12.374ms    13.003ms
  cudaFreeHost
                2.63%  10.909ms         1    10.909ms    10.909ms    10.909ms
  cudaEventSynchronize
                0.04%   149.57us        101     1.4800us      165ns    68.497us
```

Managed Memory (i)

- Unified Virtual Memory (UVM) provides a single memory space accessible by all GPUs and CPUs in the system
- Use `cudaMallocManaged()` to allocate data in unified memory
 - Returns a pointer that can be accessed from both host and device code
 - Or use `__managed__` keyword in the global scope



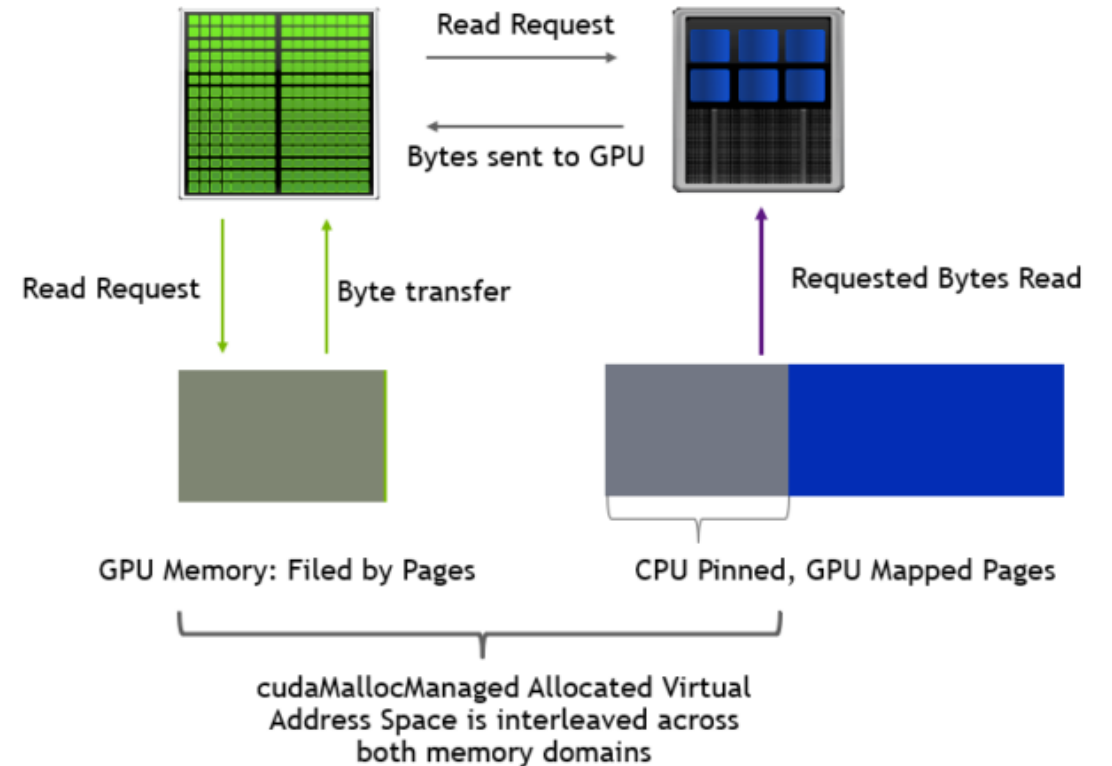
[Unified Memory in CUDA 6 | NVIDIA Technical Blog](#)

[An Even Easier Introduction to CUDA | NVIDIA Technical Blog](#)

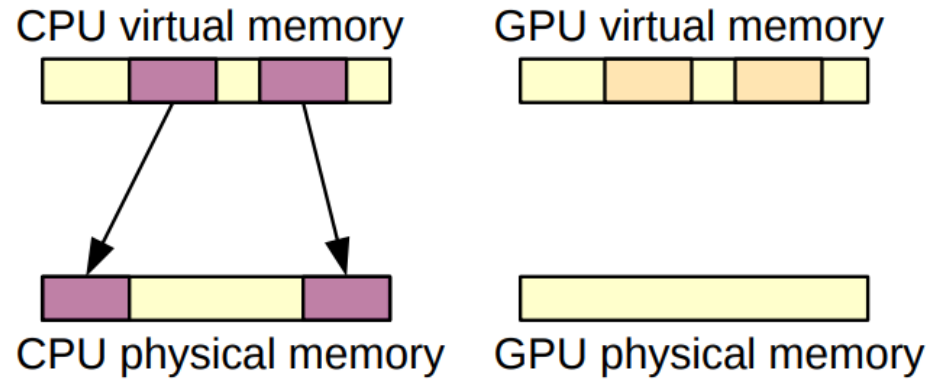
[Unified Memory for CUDA Beginners | NVIDIA Technical Blog](#)

Managed Memory (ii)

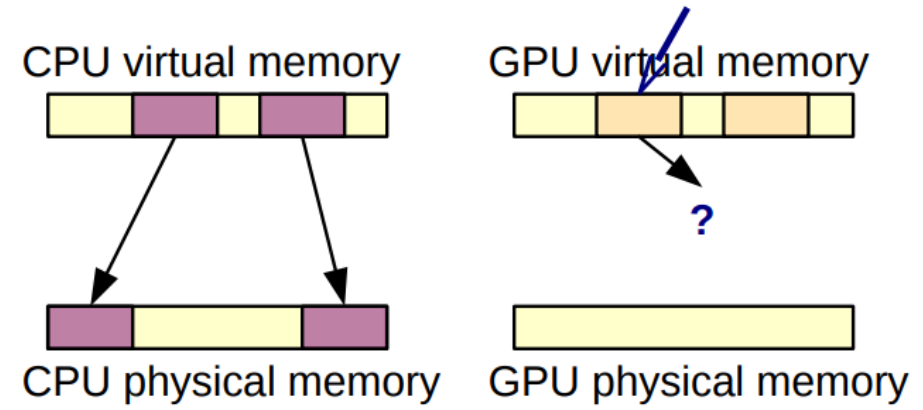
- UVM features have been evolving starting from CUDA 6+
 - 6.x: use a **single pointer** in both CPU functions and GPU kernels
 - 8.x: added **49-bit virtual addressing** and **on-demand page migration**
- CUDA runtime **automatically migrates data** allocated in Unified Memory between host and device, different from UVA



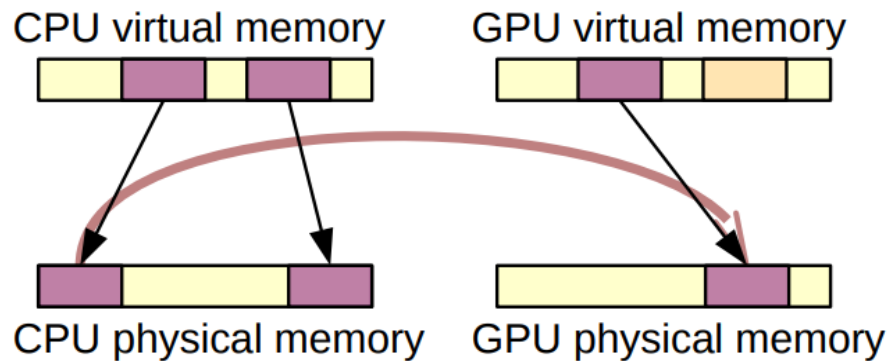
On-demand Paging



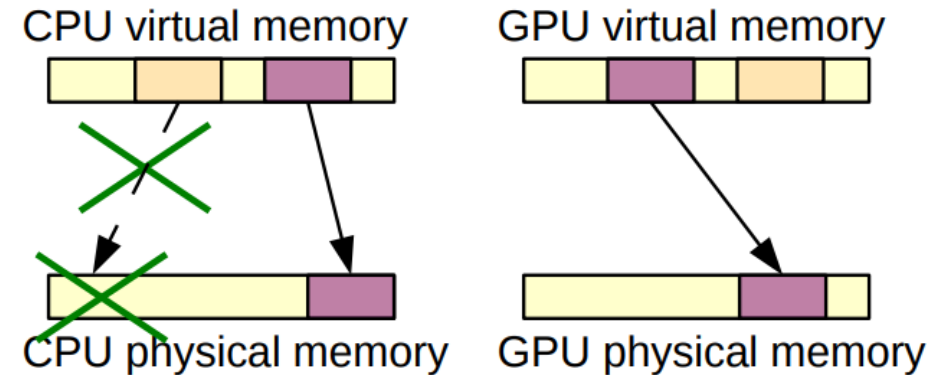
1 Data allocated in CPU memory



2 GPU touches unallocated page, triggers page fault



3 Page fault handler allocates page in GPU mem, copies contents



4 If GPU modifies page contents, invalidate CPU copy. Next CPU access will cause data to be copied back from GPU mem.

cudaMemCpy() with cudaMallocManaged()

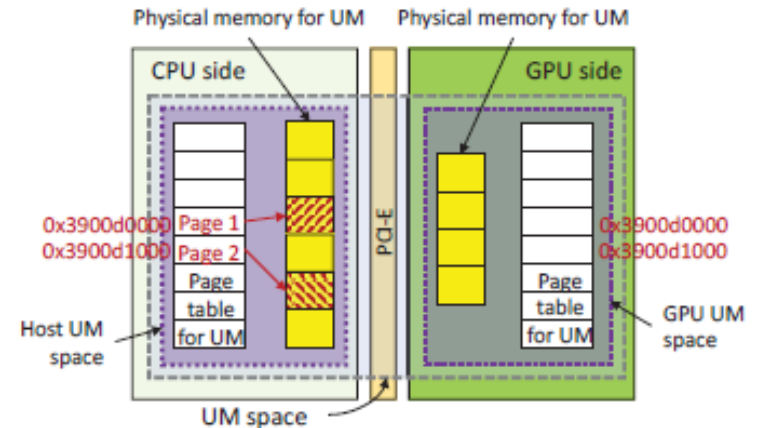
```
$ nvprof ./bin/cuda-vector-addition-managed
==1472948== NVPROF is profiling process 1472948, command: ./bin/cuda-vector-addition-managed
Time taken (ms): kernel: 906.52
Type  Time(%)      Time      Calls      Avg      Min      Max  Name
GPU activities: 100.00%  906.36ms      1  906.36ms  906.36ms  906.36ms  vecAdd(float
const *, float const *, float*, int)
  API calls: 63.76%  906.30ms      1  906.30ms  906.30ms  906.30ms
  cudaEventSynchronize
    21.24%  301.91ms      3  100.64ms  16.895us  301.84ms  cudaFree
    0.02%  247.87us      1  247.87us  247.87us  247.87us
==1472948== Unified Memory profiling result:
Device "Quadro RTX 5000 (0)"
  Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  56858  54.592KB  4.0000KB  0.9961MB  2.960228GB  357.1731ms  Host To Device
   6142  170.71KB  4.0000KB  0.9961MB  0.999939GB  87.60043ms  Device To Host
   4872      -      -      -      -      839.8422ms  Gpu page fault groups
Total CPU Page faults: 12288
```

cudaMemCpy() with cudaMallocManaged()

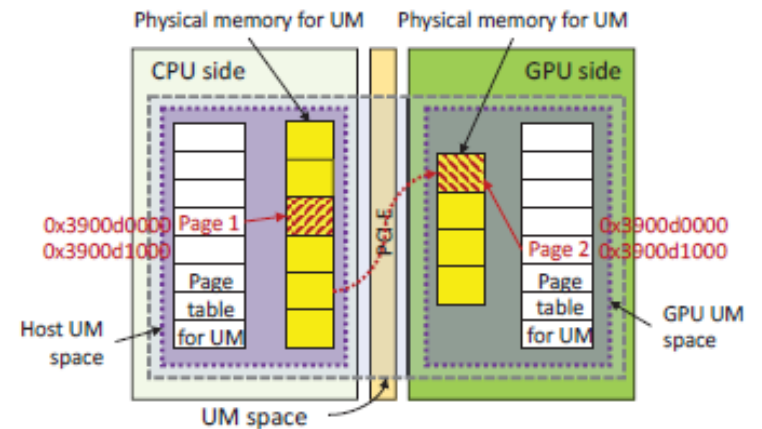
```
$ nvprof --print-gpu-trace ./bin/cuda-vector-addition-managed 2>&1 | tee output.txt
==1474233== Profiling application: ./bin/cuda-vector-addition-managed
==1474233== Profiling result:
   Start   Duration      Grid Size      Block Size      Regs*      SSMem*      DSMem*           Device      Context
Stream   Unified Memory  Virtual Address  Name
492.94ms  -                -                -                -                -                -                -                -
-         PC 0x9119d76f    0x7fab02000000  [Unified Memory CPU page faults]
493.05ms  -                -                -                -                -                -                -                -
-         PC 0x9119d76f    0x7fab02010000  [Unified Memory CPU page faults]
493.14ms  -                -                -                -                -                -                -                -
-         PC 0x9119d76f    0x7fab02020000  [Unified Memory CPU page faults]
493.28ms  -                -                -                -                -                -                -                -
-         PC 0x9119d76f    0x7fab02040000  [Unified Memory CPU page faults]
494.01ms  -                -                -                -                -                -                -                -
-         PC 0x9119d76f    0x7fab02080000  [Unified Memory CPU page faults]
4.72717s  -                -                -                -                -                -                -                -
-         PC 0x9119d76f    0x7faac1f00000  [Unified Memory CPU page faults]
.....
5.96964s  847.69ms        (1048576 1 1)    (256 1 1)      16           0B           0B  Quadro RTX 5000  1
7         -                -                vecAdd(float const *, float const *, float*, int) [117]
5.96964s  979.83us        -                -                -                -                -  Quadro RTX 5000  -
-         11 0x7fab02000000  [Unified Memory GPU page faults]
5.97056s  10.432us        -                -                -                -                -  Quadro RTX 5000  -
-         100.000000KB  0x7fab02000000  [Unified Memory Memcpy HtoD]
5.97057s  4.3840us        -                -                -                -                -  Quadro RTX 5000  -
-         28.000000KB  0x7fab02019000  [Unified Memory Memcpy HtoD]
.....
```

Unified Virtual Memory (UVM)

- On pre-Pascal GPUs, `cudaMallocManaged()` allocated managed space on the device that was **active** at the time of the call
- Pascal onward, `cudaMallocManaged()` does not immediately allocate physical memory
 - Pages and page table entries (PTEs) are not created **until the first access** by the GPU or the CPU
- Hardware supports page faulting and migration
 - The GPU **stalls the accessor threads** when they access absent pages
 - The Page Migration Engine migrates pages to the device before resuming the threads



(a) After the host has accessed page 1 and page 2.



(b) After the GPU has accessed page 2.

Pre-Pascal Behavior of `cudaMallocManaged()`

```
__global__ void add(int n, float* x, float* y) {  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    for (int i = index; i < n; i += stride)  
        y[i] = x[i] + y[i];  
}  
  
int main(void) {  
    int N = 1 << 20;  
    float *x, *y;  
    cudaMallocManaged(&x, N * sizeof(float));  
    cudaMallocManaged(&y, N * sizeof(float));  
    for (int i = 0; i < N; i++) {  
        x[i] = 1.0f; y[i] = 2.0f;  
    }  
    int blockSize = 256;  
    int numBlocks = (N + blockSize - 1) / blockSize;  
    add<<<numBlocks, blockSize>>>(N, x, y);  
    cudaDeviceSynchronize();  
    ...  
}
```

x and y are allocated on GPU memory, driver sets up page table entries

Page fault on CPU, x and y are copied to CPU memory

Lacked support for page faults, so **all data is copied to GPU before kernel launch**

Moves data back to CPU memory

Pre-Pascal Behavior of `cudaMallocManaged()`

No concurrent access

No on-demand migration to GPU

No oversubscription

... allocated on GPU memory,
... page table entries

Page fault on CPU, x and y are
copied to CPU memory

... support for page faults, so all data
... GPU before kernel launch

Moves data back to CPU memory

```
-- glo  
int  
int  
for  
y[i] = x[i] + y[i];  
}  
int main(void) {  
int  
flo  
cu  
cu  
for  
x[i] = 1.0f; y[i] = 2.0f;  
}  
int  
int  
add  
cu  
...  
}
```

Behavior of `cudaMallocManaged()` for Pascal+

```
__global__ void add(int n, float* x, float* y) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}

int main(void) {
    int N = 1 << 20;
    float *x, *y;
    cudaMallocManaged(&x, N * sizeof(float));
    cudaMallocManaged(&y, N * sizeof(float));
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f; y[i] = 2.0f;
    }
    int blockSize = 256;
    int numBlocks = (N + blockSize - 1) / blockSize;
    add<<<numBlocks, blockSize>>>(N, x, y);
    cudaDeviceSynchronize();
    ...
}
```

Page migration cost is now part of the kernel execution time

Physical memory for x and y are not immediately allocated, allocated only on first access or prefetch

Page fault on CPU, x and y are allocated on CPU memory

Supports page faults, so no data migration overhead before kernel launch

Profiling with nvprof on Turing GPU

```
==1459082== NVPROF is profiling process 1459082, command: ./bin/cuda-add-grid
```

```
...
Type  Time(%)    Time      Calls      Avg      Min      Max  Name
GPU activities: 100.00%  4.0563ms      1  4.0563ms  4.0563ms  4.0563ms  add(int, float*,
float*)
  API calls:    98.37%  310.90ms      2  155.45ms  46.172us  310.86ms  cudaMallocManaged
                1.28%  4.0568ms      1  4.0568ms  4.0568ms  4.0568ms
cudaDeviceSynchronize
                0.27%  853.93us      2  426.97us  411.16us  442.77us  cudaFree
                0.05%  148.86us     101  1.4730us   165ns    66.807us
cuDeviceGetAttribute
                0.01%  44.596us      1  44.596us  44.596us  44.596us  cuDeviceGetName
                0.01%  43.138us      1  43.138us  43.138us  43.138us  cudaLaunchKernel
                0.00%  9.0230us      1  9.0230us  9.0230us  9.0230us
cuDeviceGetPCIBusId
```

```
...
==1459082== Unified Memory profiling result:
```

```
Device "Quadro RTX 5000 (0)"
```

| Count | Avg Size | Min Size | Max Size | Total Size | Total Time | Name |
|-------|----------|----------|----------|------------|------------|-----------------------|
| 93 | 88.086KB | 4.0000KB | 992.00KB | 8.000000MB | 849.3010us | Host To Device |
| 24 | 170.67KB | 4.0000KB | 0.9961MB | 4.000000MB | 342.6530us | Device To Host |
| 12 | - | - | - | - | 4.268824ms | Gpu page fault groups |

```
Total CPU Page faults: 36
```

How to reduce overheads?

Page Migration: High-Level Mechanism

- Assume a Pascal+ GPU accesses a page is not present in the local GPU memory
- Address translation for the faulting page generates a fault message and locks the TLBs for the corresponding SM
 - On some architectures, two SMs share a TLB, so both are locked
 - Locking implies outstanding translations can proceed but new translations will be stalled until all faults are resolved
- GPU can generate many faults concurrently for the same page
- UVM driver processes the faults, remove duplicates, updates mappings and transfers the data
- Fault handling adds significant overhead to streaming performance of UVM

Reduce Page Migration Overhead (i)

- Initialize data on the device

```
__global__ void init(int n, float *x, float *y) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    int stride = blockDim.x * gridDim.x;  
    for (int i = index; i < n; i += stride) {  
        x[i] = 1.0f;  
        y[i] = 2.0f;  
    }  
}
```

==1460828== NVPROF is profiling process 1460828, command: ./bin/cuda-add-grid-init

==1460828== Profiling result:

| Type | Time(%) | Time | Calls | Avg | Min | Max | Name |
|-----------------|---------|----------|-------|----------|----------|----------|---------------------------|
| GPU activities: | 97.93% | 1.6654ms | 1 | 1.6654ms | 1.6654ms | 1.6654ms | init(int, float*, float*) |
| | 2.07% | 35.263us | 1 | 35.263us | 35.263us | 35.263us | add(int, float*, float*) |
| API calls: | 99.19% | 299.71ms | 2 | 149.86ms | 33.526us | 299.68ms | cudaMallocManaged |

==1460828== Unified Memory profiling result:

| Count | Avg Size | Min Size | Max Size | Total Size | Total Time | Name |
|-------|----------|----------|----------|------------|------------|-----------------------|
| 24 | 170.67KB | 4.0000KB | 0.9961MB | 4.000000MB | 341.6910us | Device To Host |
| 11 | - | - | - | - | 1.644405ms | Gpu page fault groups |

Total CPU Page faults: 12

Reduce Page Migration Overhead (ii)

- Use warm-up iterations or take the average of kernel multiple runs
- Prefetch data on the GPU with `cudaMemPrefetchAsync()`

```
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(x, N*sizeof(float),
                    device, NULL);
cudaMemPrefetchAsync(y, N*sizeof(float),
                    device, NULL);
```

```
==1461431== NVPROF is profiling process 1461431, command: ./bin/cuda-add-grid-prefetch
Type  Time(%)   Time      Calls      Avg      Min      Max      Name
GPU activities: 100.00% 29.600us   1         29.600us 29.600us 29.600us add(int, float*,
float*)
    API calls:  99.19% 301.89ms   2         150.94ms 38.488us 301.85ms cudaMallocManaged
                0.34% 1.0467ms   1         1.0467ms 1.0467ms 1.0467ms
cudaDeviceSynchronize
                0.21% 638.19us   2         319.09us 304.82us 333.37us cudaFree
                0.17% 504.87us   2         252.43us 188.92us 315.95us
cudaMemPrefetchAsync
==1461431== Unified Memory profiling result:
Device "Quadro RTX 5000 (0)"
  Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    4    2.0000MB 2.0000MB 2.0000MB 8.000000MB 710.9370us Host To Device
   24   170.67KB 4.0000KB 0.9961MB 4.000000MB 341.5960us Device To Host
Total CPU Page faults: 36
```

No GPU
page faults

Explicit Memory Hints in UVM

- Advise runtime on expected memory access behaviors

```
cudaMemAdvise(ptr, count, hint, device)
```

- Possible hints:
 - `cudaMemAdviseSetReadMostly`: Specify read duplication
 - `cudaMemAdviseSetPreferredLocation`: Suggest best location
 - `cudaMemAdviseSetAccessedBy`: Suggest mapping
- Hints do not trigger data movement by themselves

cudaMemAdviseSetReadMostly

- Data will usually be read-only
- UM system will make a “local” copy of the data for each processor that touches it
- If a processor writes to it, this invalidates all copies except the one written

cudaMemAdviseSetPreferredLocation

- Suggests which processor is the best location for the data
- Does not automatically cause migration
- Data will be migrated to the preferred processor on-demand (or if prefetched)
- If possible, data mappings will be provided when other processors touch it
- If mapping is not possible, data is migrated

cudaMemAdviseSetAccessedBy

- Does not cause movement or affect location of data
- Indicated processor receives a mapping to the data
- If the data is migrated, mapping is updated
- Objective: provide access without incurring page faults

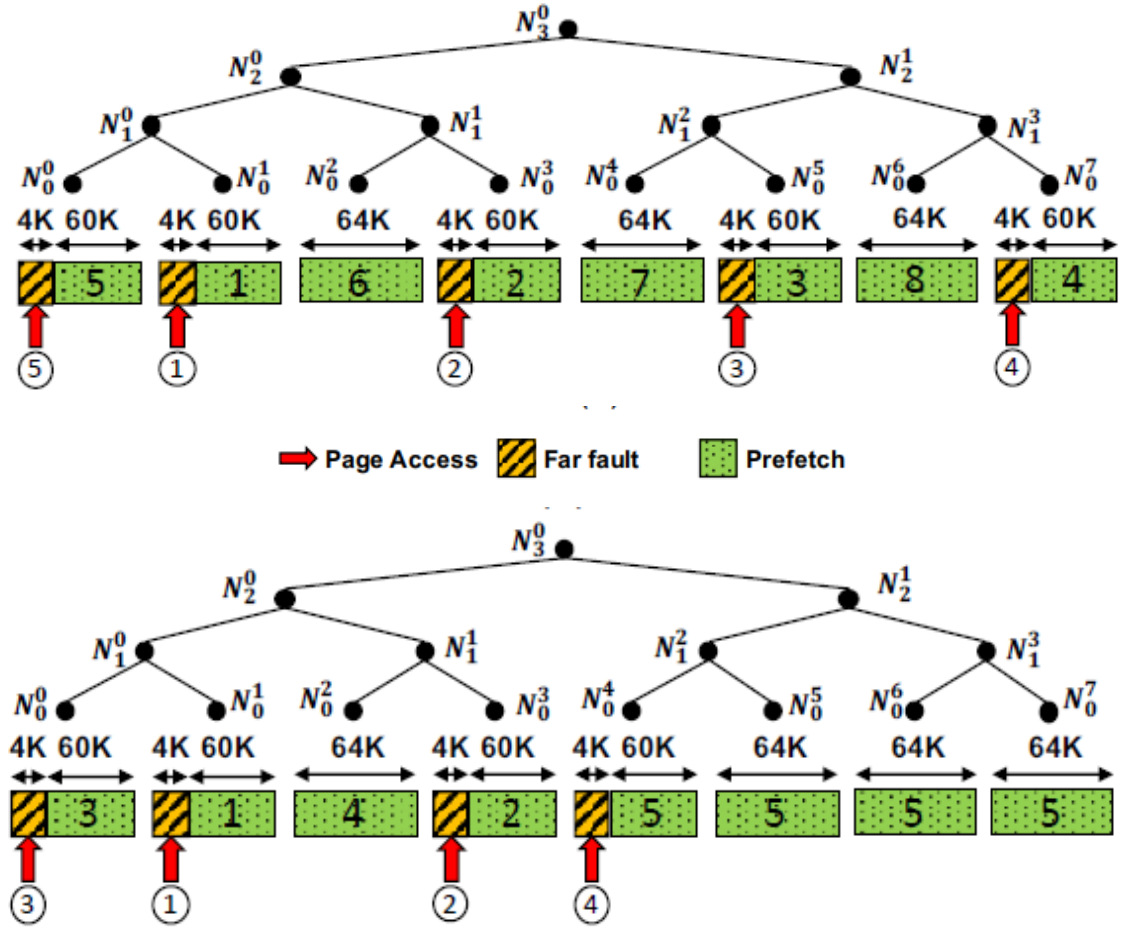
Hardware Prefetching

- Providing user hints can be complicated and error-prone
- Hardware can implement different prefetch policies: (i) random, (ii) sequential, or (iii) locality-aware
 - E.g., “random” prefetches a random 4KB page from the 2MB large page boundary along with the 4KB faulting page
- Nvidia implements a locality-aware tree-based neighborhood prefetcher GeForce GTX 1080ti (reverse engineered*)
 - Migrate multiples of 64KB basic blocks contiguous in the virtual address space grouped in a single transfer
 - All pages being prefetched are local to the current faulty pages and are within 2MB large page boundary

*D. Ganguly et al. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. ISCA'19.

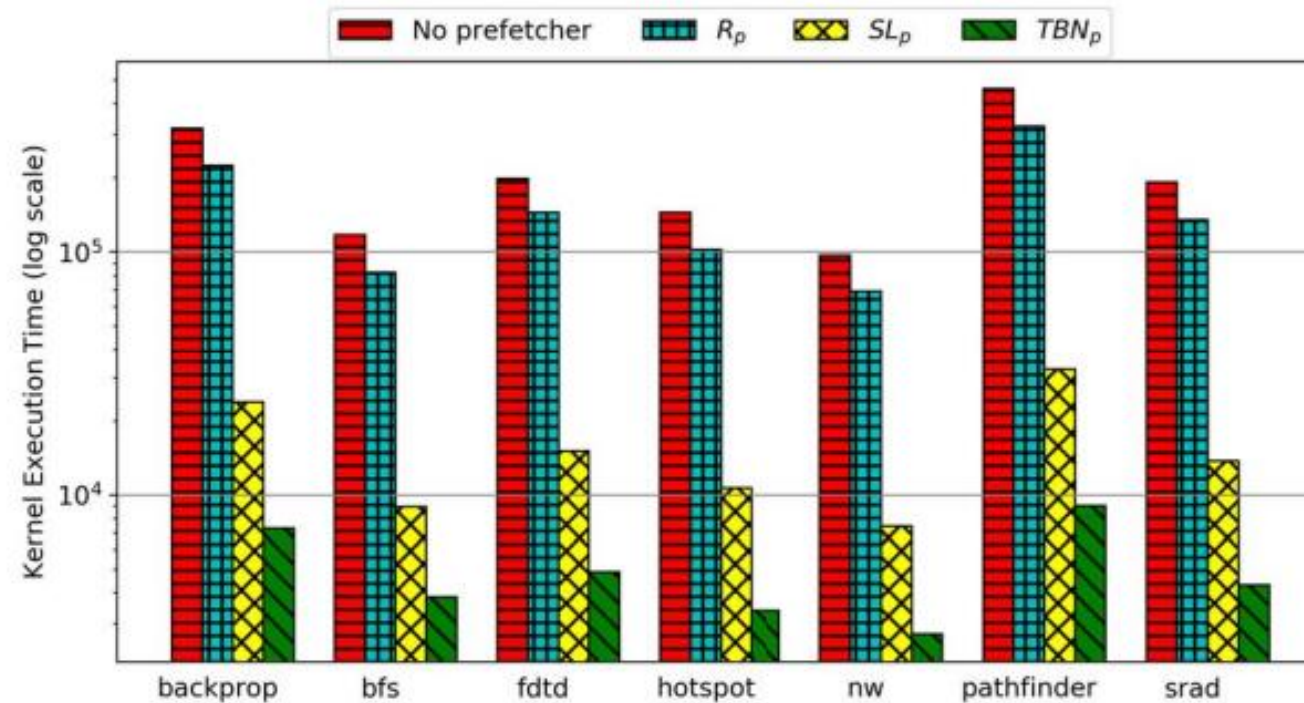
Nvidia's Tree-based Neighborhood Prefetcher

- Allocation with `cudaMallocManaged()` is logically divided into 2MB large pages
- The 2MB pages are further divided into logical 64KB basic blocks to create a full binary tree
- If the user-specified allocation request is not a multiple of 2MB, the remainder allocation is rounded up to the next $2^i * 64KB$
 - If the requests are for 4MB and 192KB, then two 2MB trees and 1 256KB trees are created



D. Ganguly et al. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. ISCA'19.

Effectiveness of Prefetching without Oversubscription



D. Ganguly et al. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. ISCA'19.

Note about UVM

- Primary goal for UVM is to **improve programmer productivity**
 - Code is less verbose, makes it easy to work with nested data structures (think of C++ classes with dynamically allocated attributes)
- UVM kernels may have poorer performance

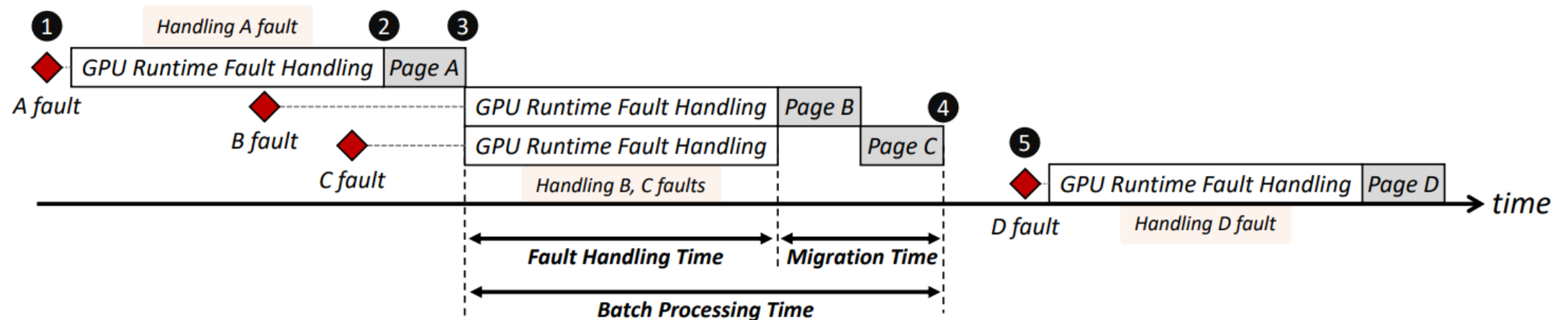
```
struct dataElem {  
    int prop1;  
    int prop2;  
    char *name;  
}
```

Note about UVM

- UVM provides a coherent view of a single virtual address space between CPUs and GPUs with automatic data migration via demand paging
 - Allows GPUs to access a page that resides in the CPU memory as if it were in the GPU memory
 - Allows applications to run without worrying about the device memory capacity
- Negative is the substantial cost of address translation overhead and demand paging

Handling GPU Page Faults

- When a GPU tries to access a physical memory page that is not currently resident in device memory, a page fault is raised and GPU runtime migrates the requested page to the GPU memory
- Page fault handling is expensive because it requires long latency communications between the CPU and GPU over the PCIe bus
 - The GPU runtime processes a group of page faults together to amortize overhead



H. Kim et al. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. ASPLOS'20.

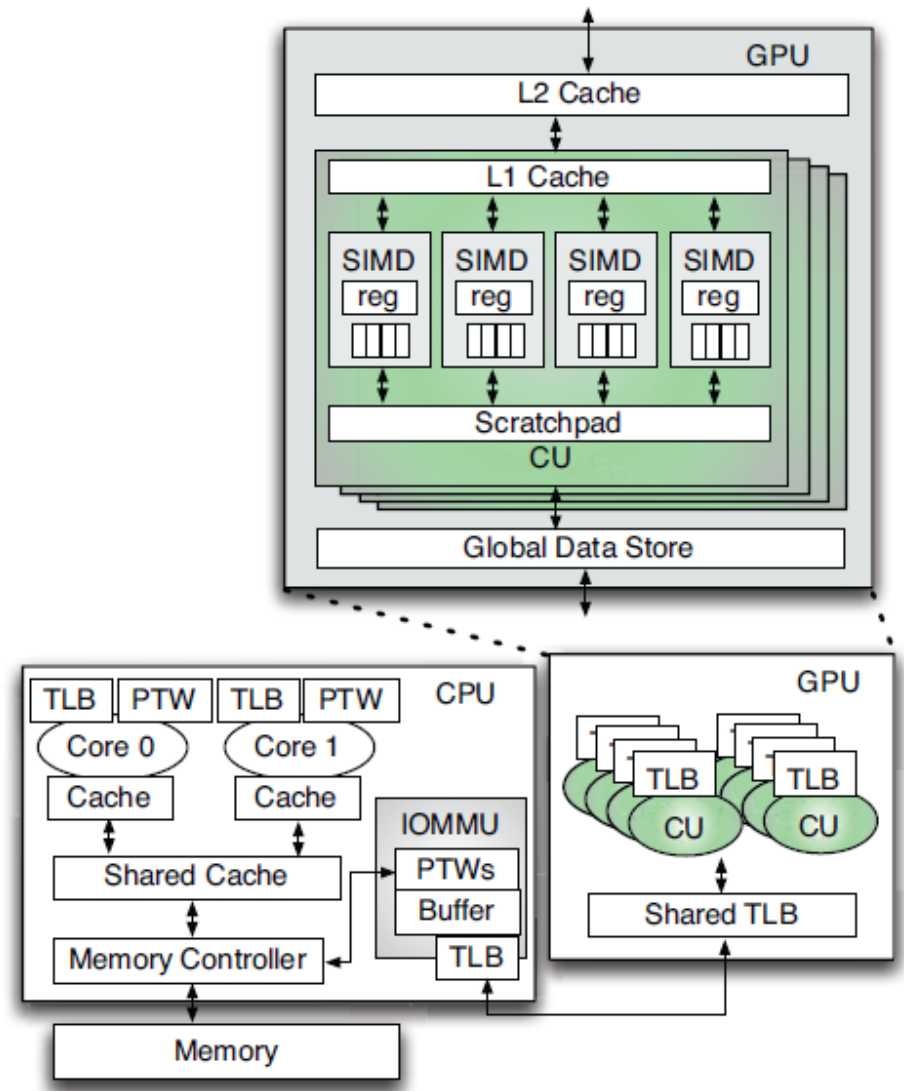
Memory Access Divergence

- An SIMD memory instruction cannot complete until data for all threads are available
 - Problematic for irregular applications with little scope for coalescing
 - Execution of one instruction requires multiple cache accesses when accesses fall on distinct cache lines and multiple virtual-to-physical address translations when accesses fall on distinct pages
- Negative impact from divergence can impact address translation more than cache access
 - Irregular memory accesses can lead from 1 to warp size (32/64) address translation requests, most will miss in the TLB
 - A page table walk on a TLB miss can take up to four memory accesses, for a total of 128-256 memory accesses per instruction
 - GPUs employ physical caches which makes address translation the bottleneck

Address Translation Request

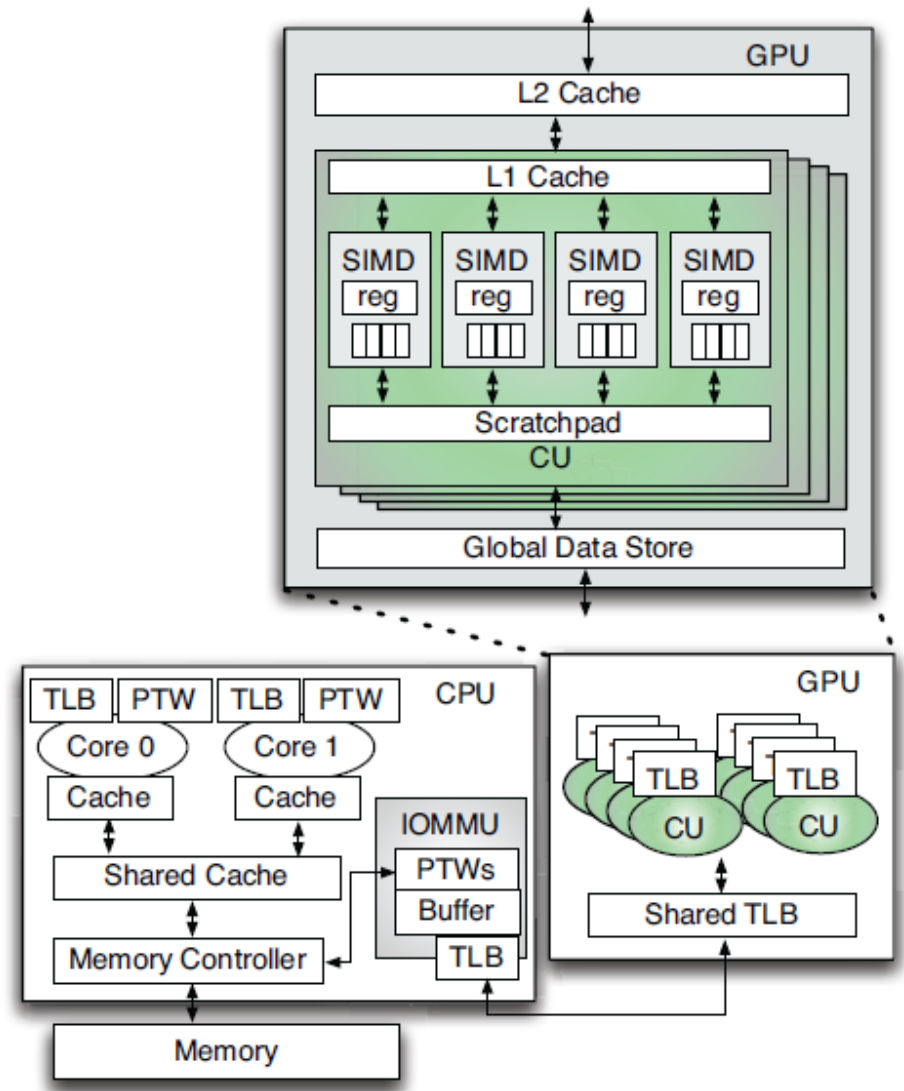
- IOMMU is a hardware component on the CPU that services address translation requests for accesses to the DRAM by any accelerator (e.g., GPU)
- IOMMU supports multiple page table walkers (e.g., 8-16) to concurrently service multiple page table walk requests (TLB misses)
- IOMMU employs small page walk caches (PWC) for the first three levels of the page tables

- GPU multiprocessors (compute units) share a private L1 TLB across SIMD units
- The GPU's L1 TLBs are backed by a larger L2 TLB that is shared across all the CUs in the GPU



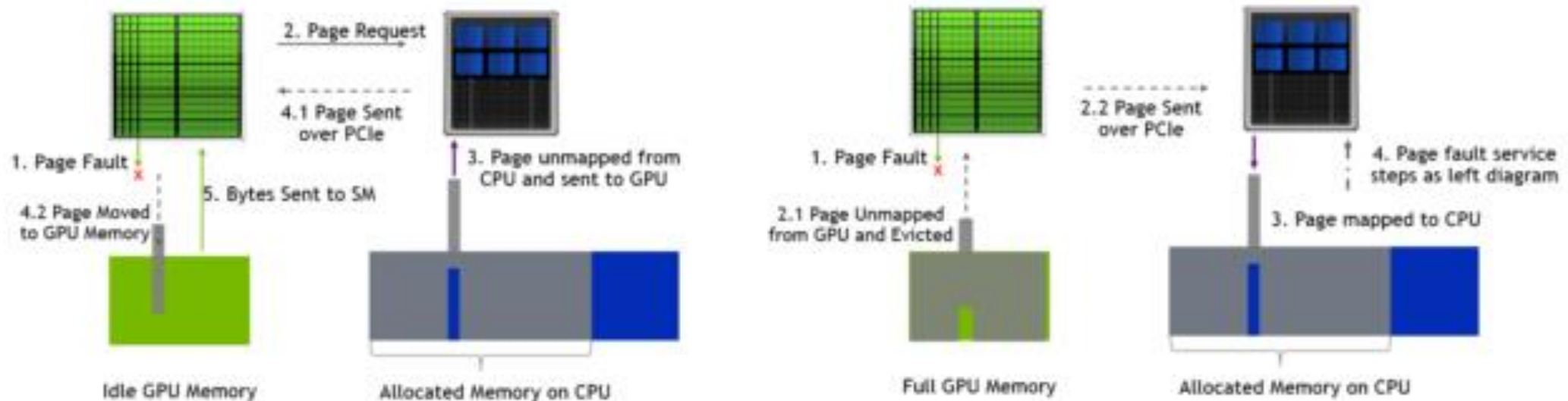
Address Translation Request

- i. An address translation request is generated by a SIMD load/store instruction
- ii. A hardware coalescer merges multiple requests to the same page generated by the same SIMD instruction
- iii. The coalesced translation request looks up the GPU's L1 TLB and then the GPU's shared L2
- iv. On a miss in the GPU's L2 TLB, the request is sent to the IOMMU
- v. At the IOMMU, the request looks up the IOMMU's TLBs
- vi. On a miss, the request queues up as a page walk request in the IOMMU buffer
- vii. When an IOMMU's page table walker becomes free, it selects a pending request from the IOMMU buffer in some order
- viii. The page table walker first performs a PWC lookup and then completes the walk of the page table, generating one to four memory accesses
- ix. On finishing a walk, the desired translation is returned to the TLBs and ultimately to the GPU SIMD unit that requested it



Memory Oversubscription

- UVM support allows GPUs to **oversubscribe memory**
- At oversubscription, a memory page is first evicted from GPU memory to system memory, followed by transfer of requested memory from CPU to GPU



Prefetching with Memory Oversubscription

- Aggressive prefetching under memory constraint can be counter-productive, may cause displacement of heavily-accessed pages
- CUDA drivers implement LRU 4KB page replacement policy
- Penalty of faults are greater under oversubscription
 - Threads need to be stalled for **writing back pages** along with the latency to migrate new pages
- An option is to disable prefetching on oversubscription
 - Active area of research

References

- NVIDIA – [CUDA C++ Programming Guide v11.8](#).
- NVIDIA – [CUDA C++ Best Practices Guide v11.8](#).
- D. Kirk and W. Hwu – Programming Massively Parallel Processors, 3rd edition.
- N. Matloff – Programming on Parallel Machines.
- Shane Cook – CUDA Programming: A Developer's Guide to Parallel Computing with GPUs.
- T. Aamodt et al. – General-Purpose Graphics Processor Architecture.
- J. Sanders and E. Kandrot – CUDA By Example: An Introduction to General-Purpose GPU Programming.