



GPU Teaching Kit

Accelerated Computing



Lecture 1.1 – Course Introduction

Course Introduction and Overview


Course Goals

- Learn how to program heterogeneous parallel computing systems and achieve
 - High performance and energy-efficiency
 - Functionality and maintainability
 - Scalability across future generations
 - Portability across vendor devices
- Technical subjects
 - Parallel programming API, tools and techniques
 - Principles and patterns of parallel algorithms
 - Processor architecture features and constraints

People

- Wen-mei Hwu (University of Illinois, NVIDIA)
- David Kirk (NVIDIA)
- Joe Bungo (NVIDIA)
- Mark Ebersole (formerly NVIDIA)
- Abdul Dakkak (Microsoft, formerly University of Illinois)
- Izzat El Hajj (American University of Beirut, formerly University of Illinois)
- Andy Schuh (University of Illinois)
- John Stratton (Whitman College)
- Isaac Gelado (NVIDIA)
- John Stone (NVIDIA, formerly University of Illinois)
- Javier Cabezas (AMD, formerly NVIDIA)
- Michael Garland (NVIDIA)

Course Content

Module 1: Course Introduction	<ul style="list-style-type: none">• 1.1 - Course Introduction and Overview• 1.2 - Introduction to Heterogeneous Parallel Computing• 1.3 - Portability and Scalability in Heterogeneous Parallel Computing
Module 2: Introduction to CUDA C	<ul style="list-style-type: none">• 2.1 - CUDA C vs. CUDA Libs vs. OpenACC• 2.2 - Memory Allocation and Data Movement API Functions• 2.3 – Threads and Kernel Functions• 2.4 - Introduction to CUDA Toolkit• 2.5 – Nsight Compute and Nsight Systems• 2.6 – Unified Memory
Module 3: CUDA Parallelism Model	<ul style="list-style-type: none">• 3.1 - Kernel-Based SPMD Parallel Programming• 3.2 - Multidimensional Kernel Configuration• 3.3 - Color-to-Greyscale Image Processing Example• 3.4 - Blur Image Processing Example• 3.5 - Thread Scheduling
Module 4: Memory Model and Locality	<ul style="list-style-type: none">• 4.1 - CUDA Memories• 4.2 - Tiled Parallel Algorithms• 4.3 - Tiled Matrix Multiplication• 4.4 - Tiled Matrix Multiplication Kernel• 4.5 - Handling Arbitrary Matrix Sizes in Tiled Algorithms
Module 5: Thread Execution Efficiency	<ul style="list-style-type: none">• 5.1 - Warps and SIMD Hardware• 5.2 - Performance Impact of Control Divergence 

Course Content

Module 6: Memory Access Performance	<ul style="list-style-type: none">• 6.1 - DRAM Bandwidth• 6.2 - Memory Coalescing in CUDA
Module 7: Parallel Computation Patterns (Histogram)	<ul style="list-style-type: none">• 7.1 - Histogramming• 7.2 - Introduction to Data Races• 7.3 - Atomic Operations in CUDA• 7.4 - Atomic Operation Performance• 7.5 - Privatization Technique for Improved Throughput
Module 8: Parallel Computation Patterns (Stencil)	<ul style="list-style-type: none">• 8.1 - Convolution• 8.2 - Tiled Convolution• 8.3 - Tile Boundary Conditions• 8.4 - Analyzing Data Reuse in Tiled Convolution
Module 9: Parallel Computation Patterns (Reduction)	<ul style="list-style-type: none">• 9.1 - Parallel Reduction• 9.2 - A Basic Reduction Kernel• 9.3 - A Better Reduction Kernel
Module 10: Parallel Computation Patterns (Scan)	<ul style="list-style-type: none">• 10.1 - Prefix Sum• 10.2 - A Work-inefficient Scan Kernel• 10.3 - A Work-Efficient Parallel Scan Kernel• 10.4 - More on Parallel Scan



Course Content

Module 11: Breadth-First (BFS) Queue	<ul style="list-style-type: none">• 11.1 – Breadth-First (BFS) Queue
Module 12: Floating Point Considerations	<ul style="list-style-type: none">• 12.1 - Floating Point Precision Considerations• 12.2 - Numerical Stability
Module 13: GPU as part of the PC Architecture	<ul style="list-style-type: none">• 13.1 - GPU as part of the PC Architecture
Module 14: Efficient Host-Device Data Transfer	<ul style="list-style-type: none">• 14.1 - Pinned Host Memory• 14.2 - Task Parallelism in CUDA• 14.3 - Overlapping Data Transfer with Computation• 14.4 - CUDA Unified Memory
Module 15: Application Case Study: Advanced MRI Reconstruction	<ul style="list-style-type: none">• 15.1 - Advanced MRI Reconstruction• 15.2 - Kernel Optimizations
Module 16: Application Case Study: Electrostatic Potential Calculation	<ul style="list-style-type: none">• 16.1 - Electrostatic Potential Calculation (Part 1)• 16.2 - Electrostatic Potential Calculation (part 2)



Course Content

Module 17: Computational Thinking for Parallel Programming	<ul style="list-style-type: none">• 17.1 - Introduction to Computational Thinking
Module 18: Related Programming Models: MPI	<ul style="list-style-type: none">• 18.1 - Introduction to Heterogeneous Supercomputing and MPI
Module 19: CUDA Python Using Numba	<ul style="list-style-type: none">• 19.1 - CUDA Python using Numba
Module 20: Related Programming Models: OpenCL	<ul style="list-style-type: none">• 20.1 - OpenCL Data Parallelism Model• 20.2 - OpenCL Device Architecture• 20.3 - OpenCL Host Code (Part 1)
Module 21: Related Programming Models: OpenACC	<ul style="list-style-type: none">• 21.1 - Introduction to OpenACC• 21.2 - OpenACC Subtleties
Module 22: Related Programming Models: OpenGL	<ul style="list-style-type: none">• <i>Module scheduled for a future release of the teaching kit</i>



Course Content

Module 23: Dynamic Parallelism	<ul style="list-style-type: none">• 23.1 - Dynamic Parallelism
Module 24: Multi-GPU	<ul style="list-style-type: none">• 24.1 - OpenMP• 24.2 - Multi-GPU Introduction I• 24.3 - Multi-GPU Introduction II• 24.4 - OpenMP and Cooperative Groups• 24.5 - Multi-GPU Heat Equation
Module 25: Using CUDA Libraries	<ul style="list-style-type: none">• 25.1 - cuBLAS• 25.2 - cuSOLVER• 25.3 - cuFFT• 25.4 - Thrust
Module 26: Advanced Thrust	<ul style="list-style-type: none">• <i>Module scheduled for a future release of the teaching kit</i>





GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



Lecture 1.2 – Course Introduction

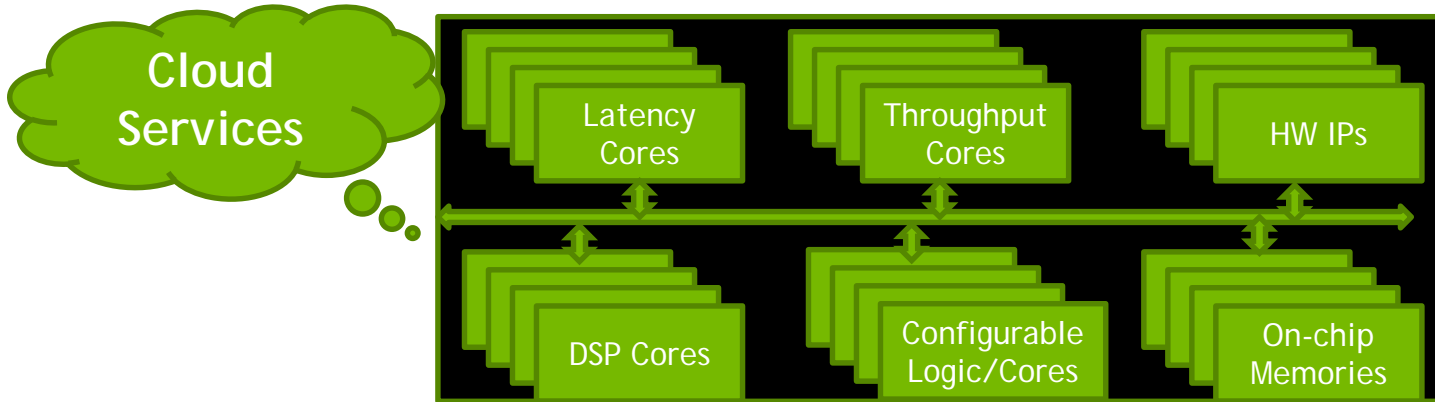
Introduction to Heterogeneous Parallel Computing

Objectives

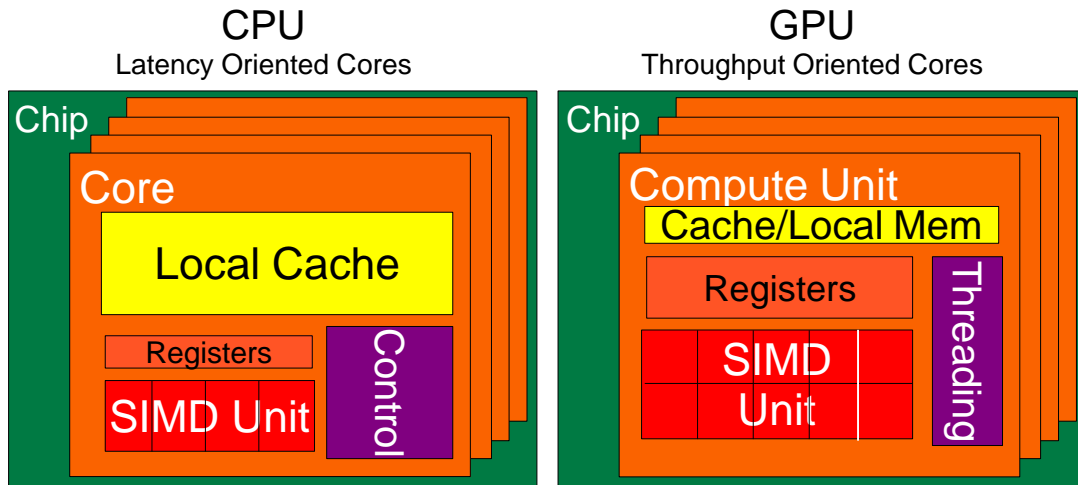
- To learn the major differences between latency devices (CPU cores) and throughput devices (GPU cores)
- To understand why winning applications increasingly use both types of devices

Heterogeneous Parallel Computing

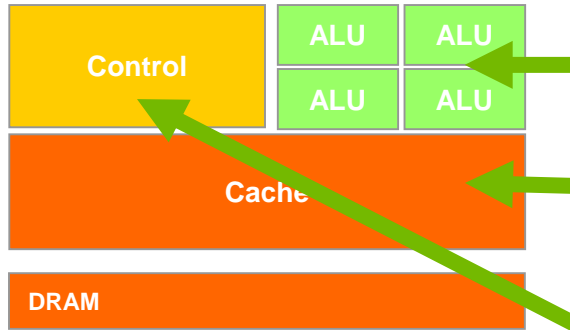
- Use the best match for the job (heterogeneity in mobile SOC)



CPU and GPU are designed very differently

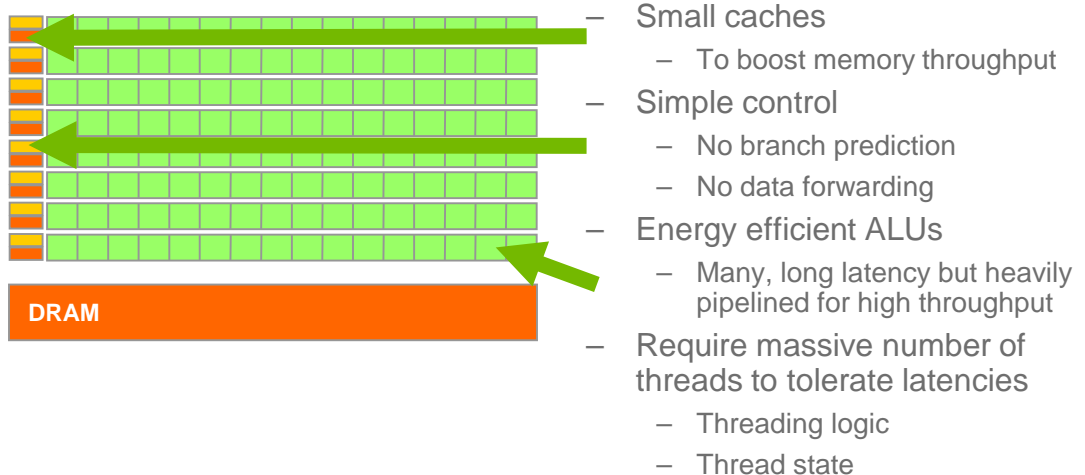


CPUs: Latency Oriented Design



- Powerful ALU
 - Reduced operation latency
- Large caches
 - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
 - Branch prediction for reduced branch latency
 - Data forwarding for reduced data latency

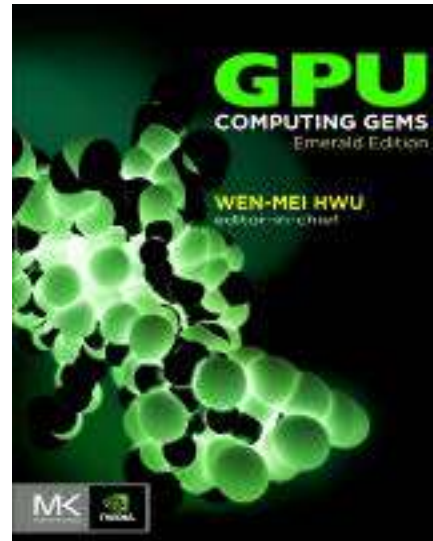
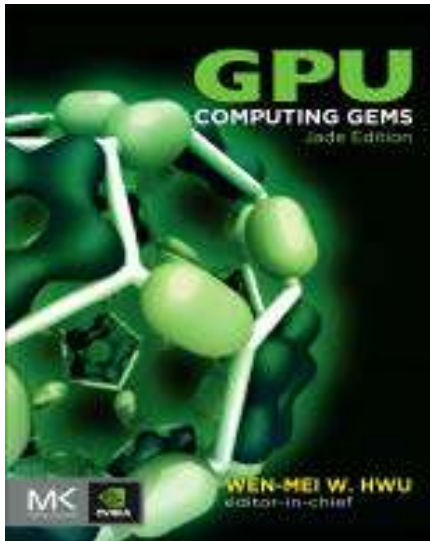
GPUs: Throughput Oriented Design



Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
 - CPUs can be 10X+ faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
 - GPUs can be 10X+ faster than CPUs for parallel code

GPU computing reading resources



90 articles in two volumes

Heterogeneous Parallel Computing in Many Disciplines

Financial
Analysis

Scientific
Simulation

Engineering
Simulation

Data
Intensive
Analytics

Medical
Imaging

Digital Audio
Processing

Digital Video
Processing

Computer
Vision

Biomedical
Informatics

Electronic
Design
Automation

Statistical
Modeling

Numerical
Methods

Ray Tracing
Rendering

Interactive
Physics



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



Lecture 1.3 – Course Introduction

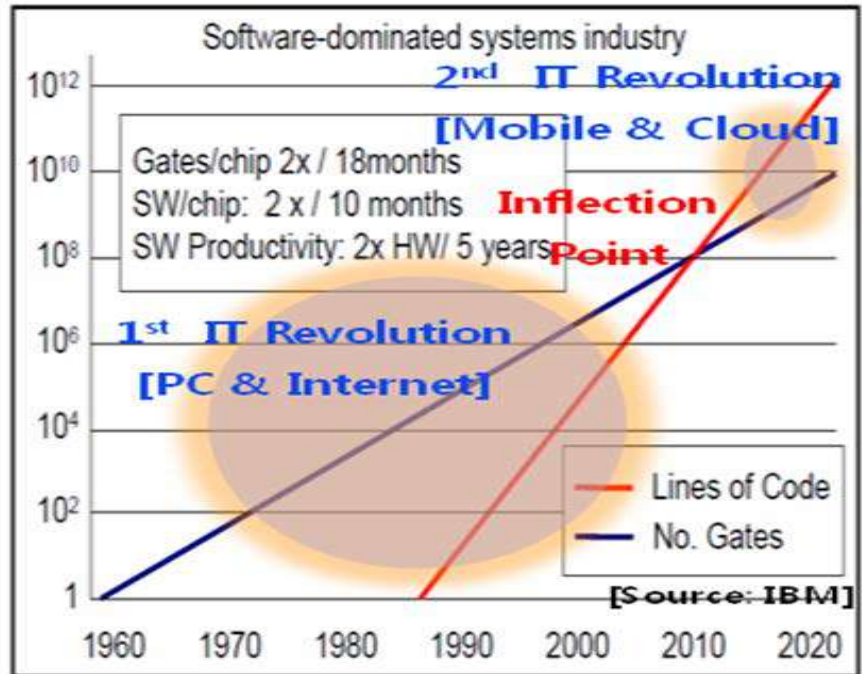
Portability and Scalability in Heterogeneous Parallel Computing

Objectives

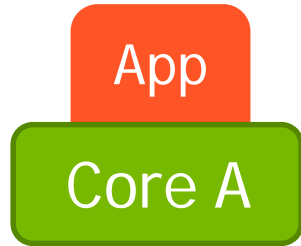
- To understand the importance and nature of scalability and portability in parallel programming

Software Dominates System Cost

- SW lines per chip increases at 2x/10 months
- HW gates per chip increases at 2x/18 months
- Future systems must minimize software redevelopment



Keys to Software Cost Control



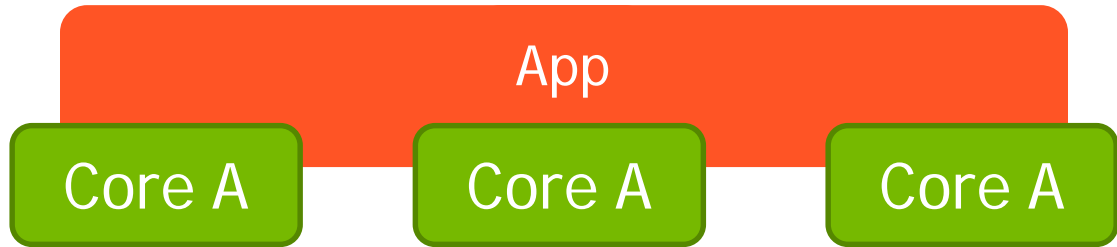
- Scalability

Keys to Software Cost Control



- Scalability
 - The same application runs efficiently on new generations of cores

Keys to Software Cost Control



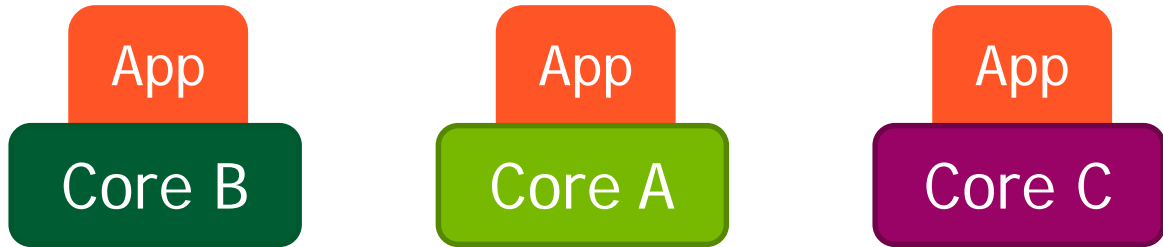
– Scalability

- The same application runs efficiently on new generations of cores
- **The same application runs efficiently on more of the same cores**

More on Scalability

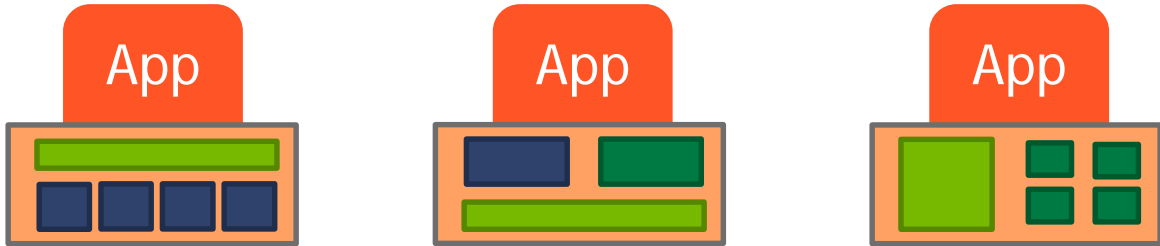
- Performance growth with HW generations
 - Increasing number of compute units (cores)
 - Increasing number of threads
 - Increasing vector length
 - Increasing pipeline depth
 - Increasing DRAM burst size
 - Increasing number of DRAM channels
 - Increasing data movement latency

Keys to Software Cost Control



- Scalability
- **Portability**
 - The same application runs efficiently on different types of cores

Keys to Software Cost Control



- Scalability
- Portability
 - The same application runs efficiently on different types of cores
 - The same application runs efficiently on systems with different organizations and interfaces

More on Portability

- Portability across many different HW types
 - Across ISAs (Instruction Set Architectures) - X86 vs. ARM, etc.
 - Latency oriented CPUs vs. throughput oriented GPUs
 - Across parallelism models - VLIW vs. SIMD vs. threading
 - Across memory models - Shared memory vs. distributed memory



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Lecture 2.1 - Introduction to CUDA C

CUDA C vs. Thrust vs. CUDA Libraries

Objective

- To learn the main venues and developer resources for GPU computing
 - Where CUDA C fits in the big picture

3 Ways to Accelerate Applications

Applications

Libraries

Easy to use
Most Performance

Compiler
Directives

Easy to use
Portable code

Programming
Languages

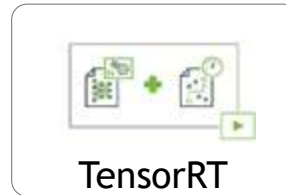
Most Performance
Most Flexibility

Libraries: Easy, High-Quality Acceleration

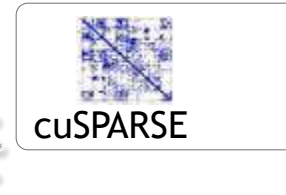
- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications

NVIDIA GPU Accelerated Libraries

DEEP LEARNING



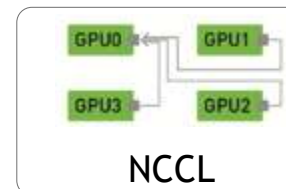
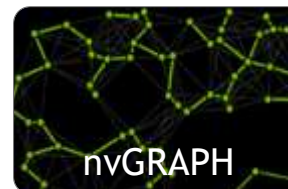
LINEAR ALGEBRA



SIGNAL, IMAGE, VIDEO



PARALLEL ALGORITHMS



Vector Addition in Thrust

```
#include <thrust/device_vector.h>
#include <thrust/copy.h>
```

```
int main(void) {
    size_t inputLength = 500;
    thrust::host_vector<float> hostInput1(inputLength);
    thrust::host_vector<float> hostInput2(inputLength);
    thrust::device_vector<float> deviceInput1(inputLength);
    thrust::device_vector<float> deviceInput2(inputLength);
    thrust::device_vector<float> deviceOutput(inputLength);
```

```
    thrust::copy(hostInput1.begin(), hostInput1.end(), deviceInput1.begin());
    thrust::copy(hostInput2.begin(), hostInput2.end(), deviceInput2.begin());
```

```
    thrust::transform(deviceInput1.begin(), deviceInput1.end(),
                      deviceInput2.begin(), deviceOutput.begin(),
                      thrust::plus<float>());
```

```
}
```

Compiler Directives: Easy, Portable Acceleration

- **Ease of use:** Compiler takes care of details of parallelism management and data movement
- **Portable:** The code is generic, not specific to any type of hardware and can be deployed into multiple languages
- **Uncertain:** Performance of code can vary across compiler versions

OpenACC

- Compiler directives for C, C++, and FORTRAN

```
#pragma acc parallel loop  
copyin(input1[0:inputLength],input2[0:inputLength]),  
copyout(output[0:inputLength])  
for(i = 0; i < inputLength; ++i) {  
    output[i] = input1[i] + input2[i];  
}
```

Programming Languages: Most Performance and Flexible Acceleration

- **Performance:** Programmer has best control of parallelism and data movement
- **Flexible:** The computation does not need to fit into a limited set of library patterns or directive types
- **Verbose:** The programmer often needs to express more details

GPU Programming Languages

Numerical analytics ►

MATLAB, Mathematica, LabVIEW

Python ►

PyCUDA, Numba

Fortran ►

CUDA Fortran, OpenACC

C ►

CUDA C, OpenACC

C++ ►

CUDA C++, Thrust

C# ►

Hybridizer

CUDA - C

Applications

Libraries

Compiler
Directives

Programming
Languages

Easy to use
Most Performance

Easy to use
Portable code

Most Performance
Most Flexibility



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

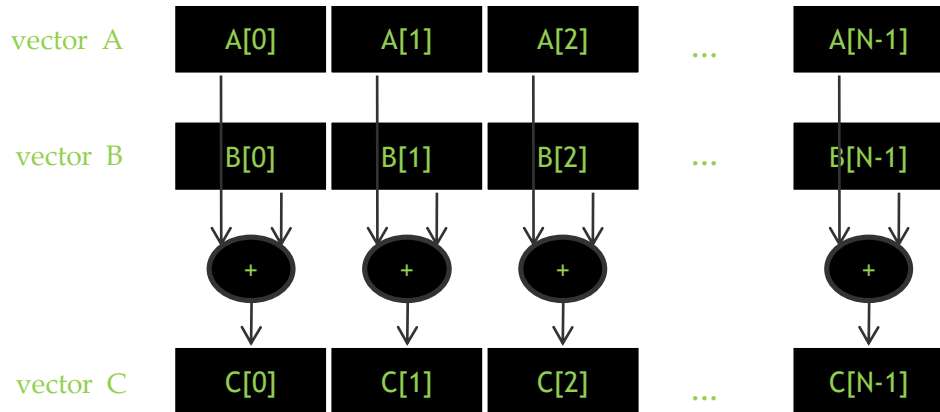
Lecture 2.2 - Introduction to CUDA C

Memory Allocation and Data Movement API Functions

Objective

- To learn the basic API functions in CUDA host code
 - Device Memory Allocation
 - Host-Device Data Transfer

Data Parallelism - Vector Addition Example

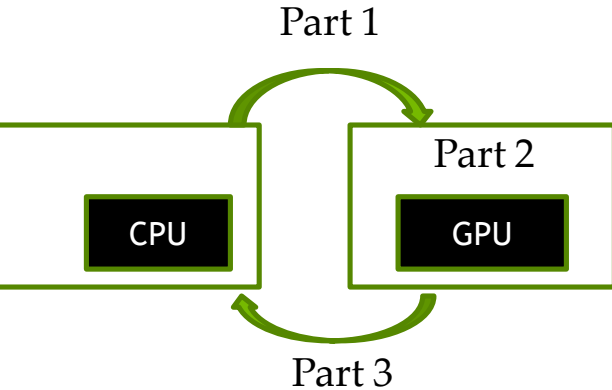


Vector Addition – Traditional C Code

```
// Compute vector sum  $C = A + B$ 
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

Heterogeneous Computing vecAdd CUDA Host Code

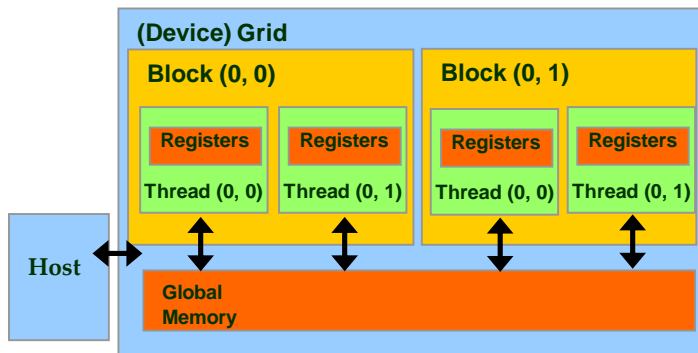


```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;
    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

    // Part 2
    // Kernel launch code – the device performs the actual vector addition

    // Part 3
    // copy C from the device memory
    // Free device vectors
}
```

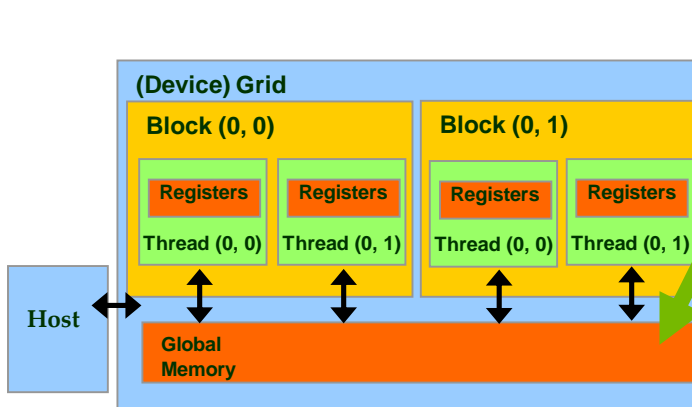
Partial Overview of CUDA Memories



- Device code can:
 - R/W per-thread **registers**
 - R/W all-shared **global memory**
- Host code can
 - Transfer data to/from per grid **global memory**

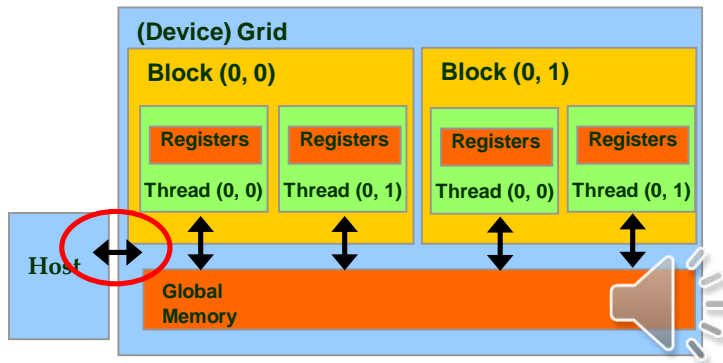
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- `cudaMalloc()`
 - Allocates an object in the device global memory
 - Two parameters
 - **Address of a pointer** to the allocated object
 - **Size of** allocated object in terms of bytes
- `cudaFree()`
 - Frees object from device global memory
 - One parameter
 - **Pointer** to freed object

Host-Device Data Transfer API functions



– cudaMemcpy()

- memory data transfer
- Requires four parameters
 - Pointer to destination
 - Pointer to source
 - Number of bytes copied
 - Type/Direction of transfer
- Transfer to device is synchronous with respect to the host

Vector Addition, Explicit Memory Management

... Allocate h_A , h_B , h_C ...

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;
```

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

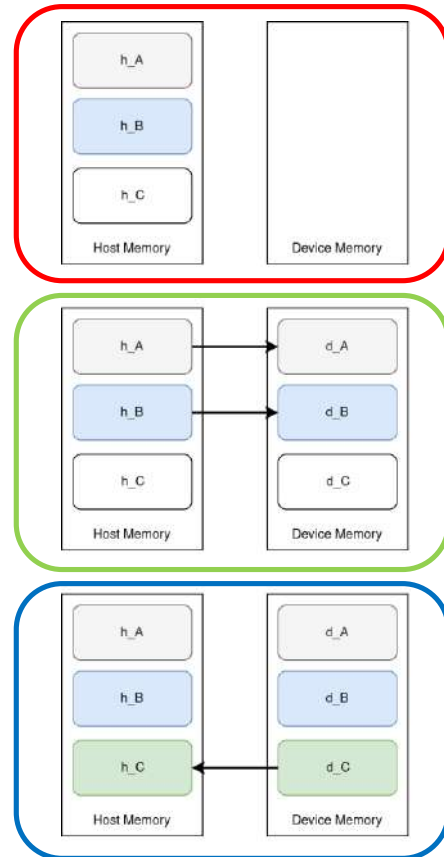


```
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

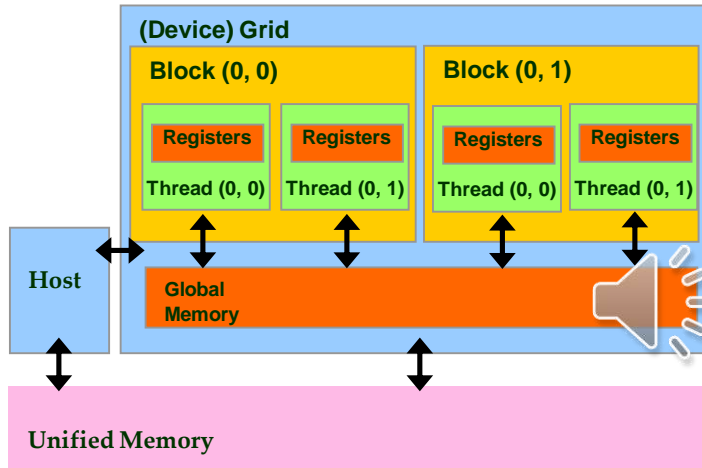
// Kernel invocation code – to be shown later

```
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

... Free h_A , h_B , h_C ...



Unified Memory



– `cudaMallocManaged(void** ptr, size_t size)`

- Single memory space for all CPUs/GPUs
- Maintain single copy of data
- CUDA-managed data
- On-demand page migration
- Compatible with `cudaMalloc()`, `cudaFree()`
- Can be optimized
 - `cudaMemAdvise()`,
 - `cudaMemPrefetchAsync()`,
 - `cudaMemcpyAsync()`

Vector Addition, Unified Memory

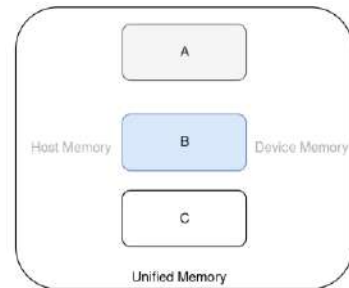
```
float *A, *B, *C  
cudaMallocManaged(&A, n * sizeof(float));  
cudaMallocManaged(&B, n * sizeof(float));  
cudaMallocManaged(&C, n * sizeof(float));
```

```
// Initialize A, B
```

```
void vecAdd(float *A, float *B, float *C, int n)  
{  
    // Kernel invocation code – to be shown later  
}
```



```
cudaFree(A);  
cudaFree(B);  
cudaFree(C);
```



In Practice, Check for API Errors in Host Code

```
cudaError_t err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__,
        __LINE__);
    exit(EXIT_FAILURE);
}
```



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit
Accelerated Computing



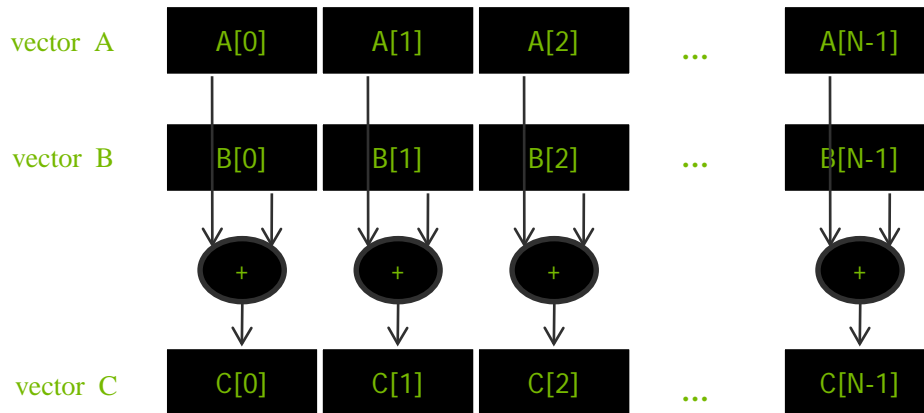
Lecture 2.3 – Introduction to CUDA C

Threads and Kernel Functions

Objective

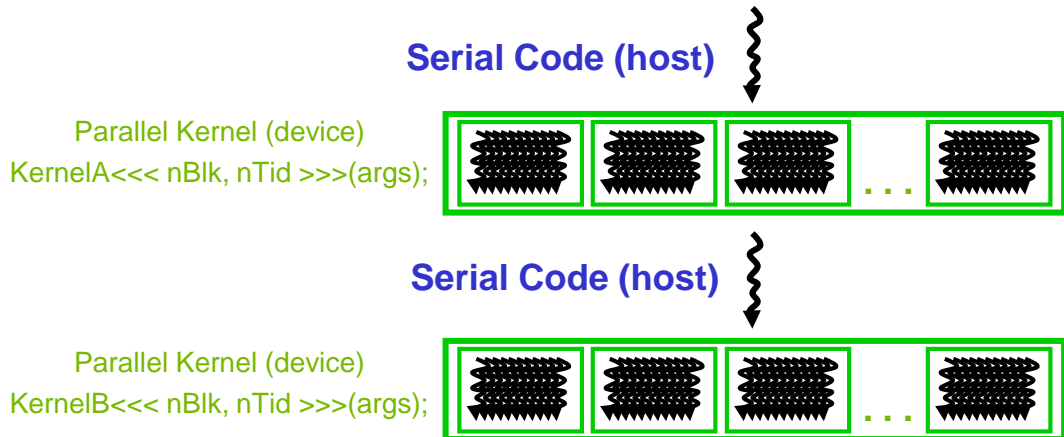
- To learn about CUDA threads, the main mechanism for exploiting of data parallelism
 - Hierarchical thread organization
 - Launching parallel execution
 - Thread index to data index mapping

Data Parallelism - Vector Addition Example

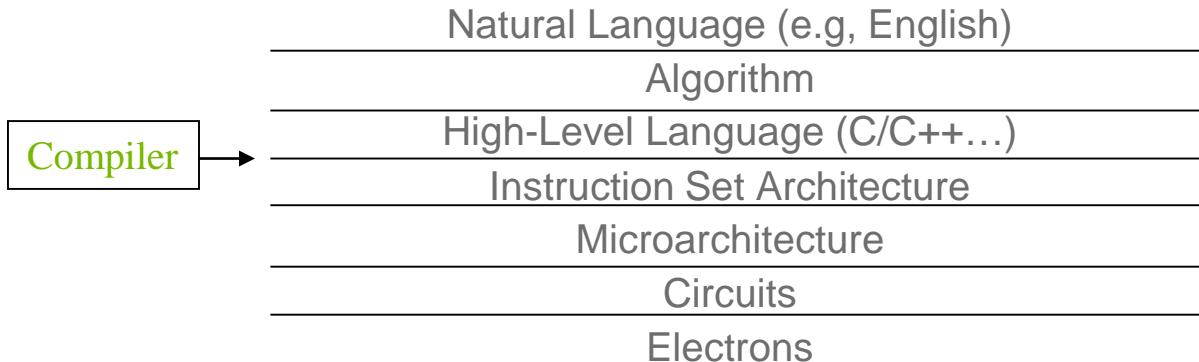


CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
 - Serial parts in **host** C code
 - Parallel parts in **device** SPMD kernel code



From Natural Language to Electrons



©Yale Patt and Sanjay Patel, *From bits and bytes to gates and beyond*

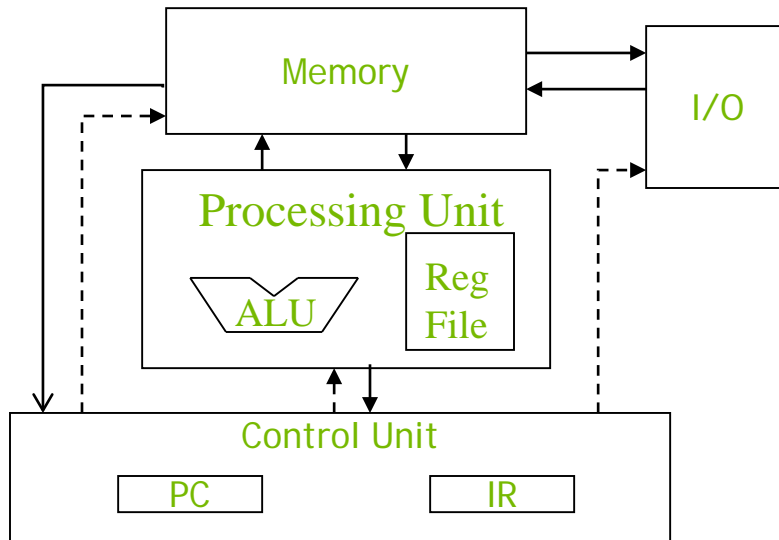
A program at the ISA level

- A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.
 - Both CPUs and GPUs are designed based on (different) instruction sets
- Program instructions operate on data stored in memory and/or registers.

A Thread as a Von-Neumann Processor

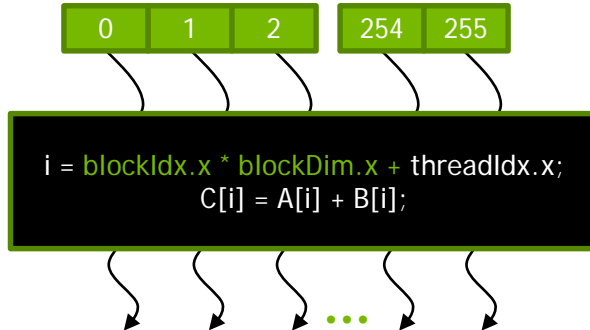
A thread is a “virtualized” or
“abstracted”

Von-Neumann Processor

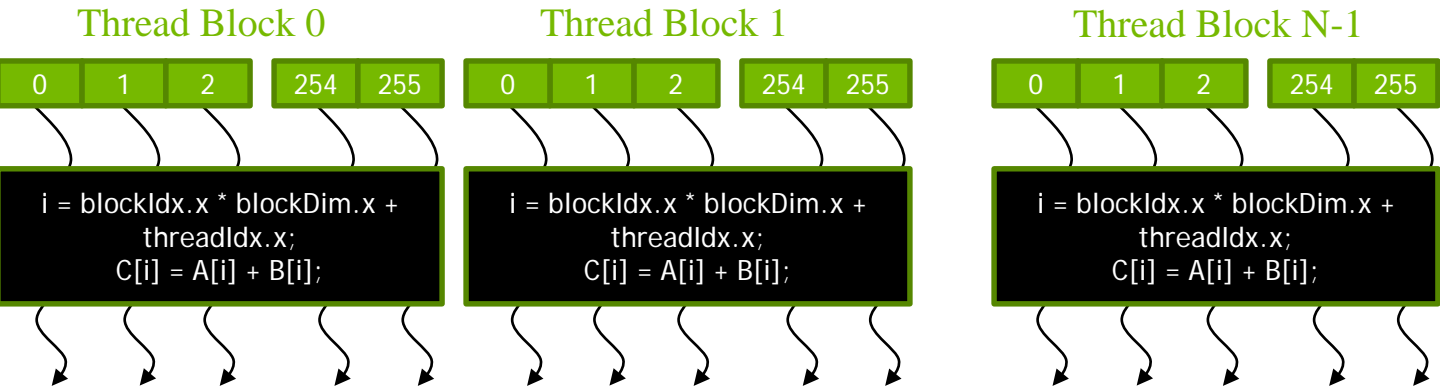


Arrays of Parallel Threads

- A CUDA kernel is executed by a **grid** (array) of threads
 - All threads in a grid run the same kernel code (Single Program Multiple Data)
 - Each thread has indexes that it uses to compute memory addresses and make control decisions



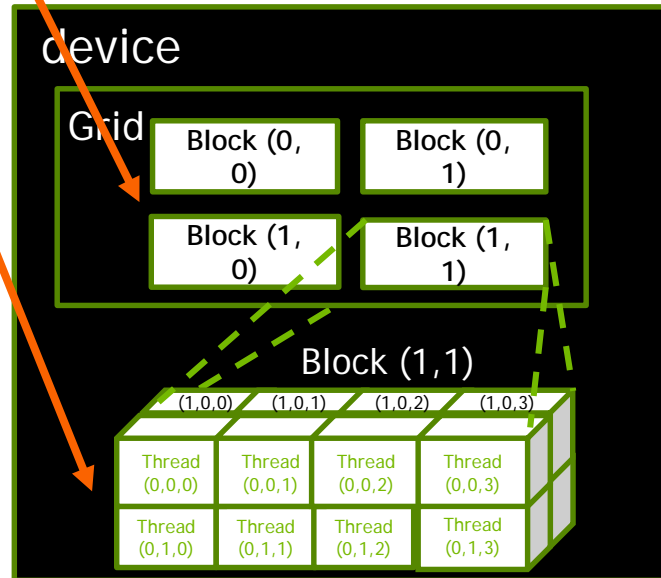
Thread Blocks: Scalable Cooperation



- Divide thread array into multiple blocks
 - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
 - Threads in different blocks do not interact

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
 - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
 - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
 - Image processing
 - Solving PDEs on volumes
 - ...





GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



Lecture 2.4 – Introduction to CUDA C

Introduction to the CUDA Toolkit

Objective

- To become familiar with some valuable tools and resources from the CUDA Toolkit
 - Compiler flags
 - Debuggers
 - Profilers

GPU Programming Languages

Numerical analytics ► MATLAB, Mathematica, LabVIEW

Python ► PyCUDA, Numba

Fortran ► CUDA Fortran, OpenACC

C ► CUDA C, OpenACC

C++ ► CUDA C++, Thrust

C# ► Hybridizer



CUDA - C

Applications

Libraries

Easy to use
Most Performance

Compiler
Directives

Easy to use
Portable code

Programming
Languages

Most Performance
Most Flexibility

NVCC Compiler

- NVIDIA provides a CUDA-C compiler
 - `nvcc`
- NVCC compiles device code then forwards code on to the host compiler (e.g. `g++`)
- Can be used to compile & link host only applications

Example 1: Hello World

```
#include <stdio>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

Instructions:

1. Build and run the hello world code
2. Modify Makefile to use nvcc instead of g++
3. Rebuild and run

CUDA Example 1: Hello World

```
#include <stdio>

__global__ void mykernel(void) {

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Instructions:

1. Add kernel and kernel launch to main.cc
2. Try to build

CUDA Example 1: Build Considerations

- Build failed
 - Nvcc only parses .cu files for CUDA
- Fixes:
 - Rename main.cc to main.cu
 - OR
 - `nvcc -x cu`
 - Treat all input files as .cu files

Instructions:

1. Rename main.cc to main.cu
2. Rebuild and Run

Hello World! with Device Code

```
#include <stdio>

__global__ void mykernel(void) {

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

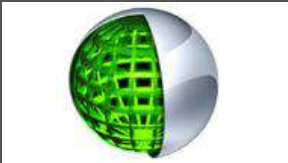
Output:

```
$ nvcc main.cu
$ ./a.out
Hello World!
```

- `mykernel` (does nothing, somewhat anticlimactic!)

Developer Tools - Debuggers

Nsight



**Nsight
Systems**



CUDA-GDB



**CUDA
MEMCHECK**



NVIDIA Provided

**arm
FORGE**

TotalView®

3rd Party

<https://developer.nvidia.com/debugging-solutions>

Compiler Flags

- Remember there are two compilers being used
 - NVCC: Device code
 - Host Compiler: C/C++ code
- NVCC supports some host compiler flags
 - If flag is unsupported, use `-Xcompiler` to forward to host
 - e.g. `-Xcompiler -fopenmp`
- Debugging Flags
 - `-g`: Include host debugging symbols
 - `-G`: Include device debugging symbols
 - `-lineinfo`: Include line information with symbols

CUDA-MEMCHECK

- Memory debugging tool
 - No recompilation necessary
 - `%> cuda-memcheck ./exe`
- Can detect the following errors
 - Memory leaks
 - Memory errors (OOB, misaligned access, illegal instruction, etc)
 - Race conditions
 - Illegal Barriers
 - Uninitialized Memory
- For line numbers use the following compiler flags:
 - `-Xcompiler -rdynamic -lineinfo`

<http://docs.nvidia.com/cuda/cuda-memcheck>

Example 2: CUDA-MEMCHECK

Instructions:

1. Build & Run Example 2
Output should be the numbers 0-9
Do you get the correct results?
2. Run with cuda-memcheck
`%> cuda-memcheck ./a.out`
3. Add nvcc flags “-Xcompiler -rdynamic -lineinfo”
4. Rebuild & Run with cuda-memcheck
5. Fix the illegal write

<http://docs.nvidia.com/cuda/cuda-memcheck>

CUDA-GDB

- cuda-gdb is an extension of GDB
 - Provides seamless debugging of CUDA and CPU code
- Works on Linux and Macintosh
 - For a Windows debugger use NVIDIA Nsight Eclipse Edition or Visual Studio Edition

<http://docs.nvidia.com/cuda/cuda-gdb>

Example 3: cuda-gdb

Instructions:

1. Run exercise 3 in cuda-gdb

```
%> cuda-gdb --args ./a.out
```

2. Run a few cuda-gdb commands:

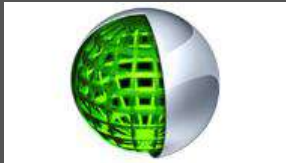
```
(cuda-gdb) b main           //set break point at main
(cuda-gdb) r                 //run application
(cuda-gdb) l                 //print line context
(cuda-gdb) b foo             //break at kernel foo
(cuda-gdb) c                 //continue
(cuda-gdb) cuda thread       //print current thread
(cuda-gdb) cuda thread 10    //switch to thread 10
(cuda-gdb) cuda block        //print current block
(cuda-gdb) cuda block 1      //switch to block 1
(cuda-gdb) d                 //delete all break points
(cuda-gdb) set cuda memcheck on //turn on cuda memcheck
(cuda-gdb) r                 //run from the beginning
```

3. Fix Bug

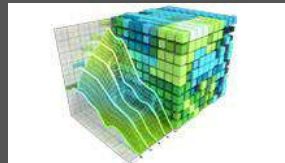
<http://docs.nvidia.com/cuda/cuda-gdb>

Developer Tools - Profilers

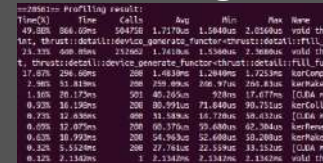
NSIGHT



NVVP



NVPROF



```
--2006100 Profiling result:
Time(s)   time    calls      Avg    Min    Max  Name
49.328  266.69ms  56472    1.727ms  1.544ms  2.026ms  void th
tat_thrust::detail::device_generate_function_cuda::Fill
23.333  698.89ms  75260    1.741ms  1.536ms  2.308ms  void th
t::device::detail::device_generate_function_cuda::Fill_Fu
17.878  296.69ms  200     1.483ms  1.284ms  1.723ms  horComp
2.985  55.81ms   200     258.89ms  208.97ms  756.83ms  horMat
1.808  76.17ms   201     40.35ms   30ms      17.477ms  [CMA m
0.928  16.150ms  200     80.991ms  71.640ms  90.771ms  horCol
0.735  12.450ms   600     31.588ms  15.778ms  58.433ms  [CMA m
0.408  12.497ms  200     60.376ms  39.448ms  82.364ms  horPer
0.628  10.993ms  200     54.963ms  32.608ms  58.280ms  horPer
0.328  5.552ms    200     27.781ms  22.559ms  33.152ms  [CMA m
0.128  2.144ms    1       2.144ms   2.144ms   2.144ms   void th
```

NVIDIA Provided

TAU



VampirTrace



3rd Party

<https://developer.nvidia.com/performance-analysis-tools>

NVPROF

Command Line Profiler

- Compute time in each kernel
- Compute memory transfer time
- Collect metrics and events
- Support complex process hierarchy's
- Collect profiles for NVIDIA Visual Profiler
- No need to recompile

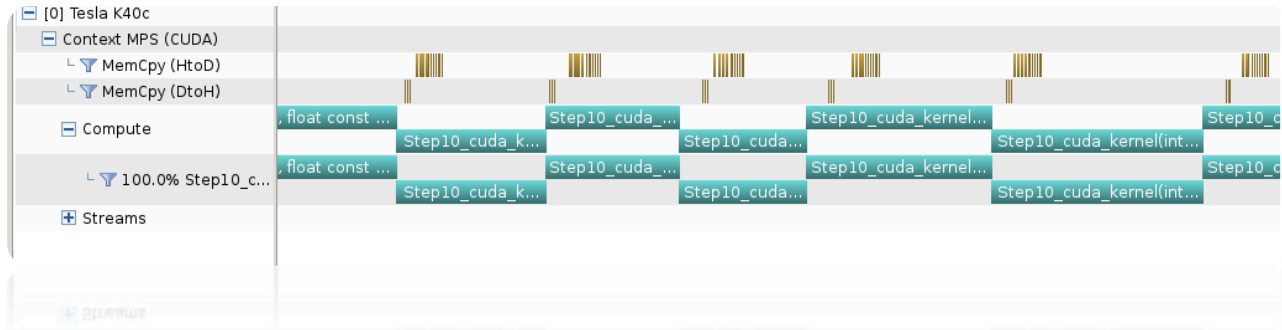
Example 4: nvprof

Instructions:

1. Collect profile information for the matrix add example
`%> nvprof ./a.out`
2. How much faster is add_v2 than add_v1?
3. View available metrics
`%> nvprof --query-metrics`
4. View global load/store efficiency
`%> nvprof --metrics
gld_efficiency,gst_efficiency ./a.out`
5. Store a timeline to load in NVVP
`%> nvprof -o profile.timeline ./a.out`
6. Store analysis metrics to load in NVVP
`%> nvprof -o profile.metrics --analysis-metrics
./a.out`

NVIDIA's Visual Profiler (NVVP)

Timeline



Guided System

1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10_cuda_kernel" is most likely limited by Compute.

Perform Compute Analysis

The most likely bottleneck to performance for this kernel is compute, so you should first perform compute analysis to determine how it is limiting performance.

Perform Latency Analysis

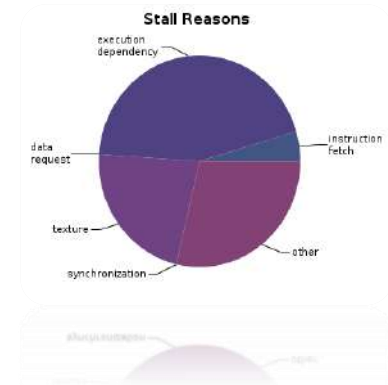
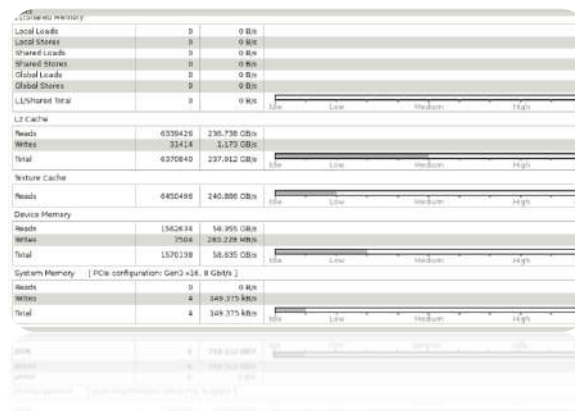
Perform Memory Bandwidth Analysis

Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Analysis



Example 4: NVVP

Instructions:

1. Import nvprof profile into NVVP

Launch nvvp

Click File/ Import/ Nvprof/ Next/ Single process/ Next / Browse

Select profile.timeline

Add Metrics to timeline

Click on 2nd Browse

Select profile.metrics

Click Finish

2. Explore Timeline

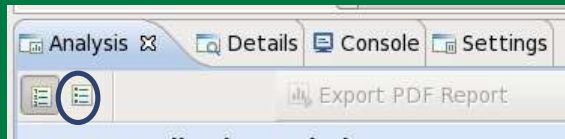
Control + mouse drag in timeline to zoom in

Control + mouse drag in measure bar (on top) to measure time

Example 4: NVVP

Instructions:

1. Click on a kernel
2. On Analysis tab click on the unguided analysis



2. Click Analyze All
Explore metrics and properties
What differences do you see between the two kernels?

Note:

If kernel order is non-deterministic you can only load the timeline or the metrics but not both.

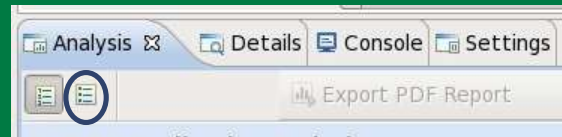
If you load just metrics the timeline looks odd but metrics are correct.

Example 4: NVVP

Let's now generate the same data within NVVP

1. Click File / New Session / Browse
Select Example 4/a.out
Click Next / Finish

2. Click on a kernel
Select Unguided Analysis
Click Analyze All

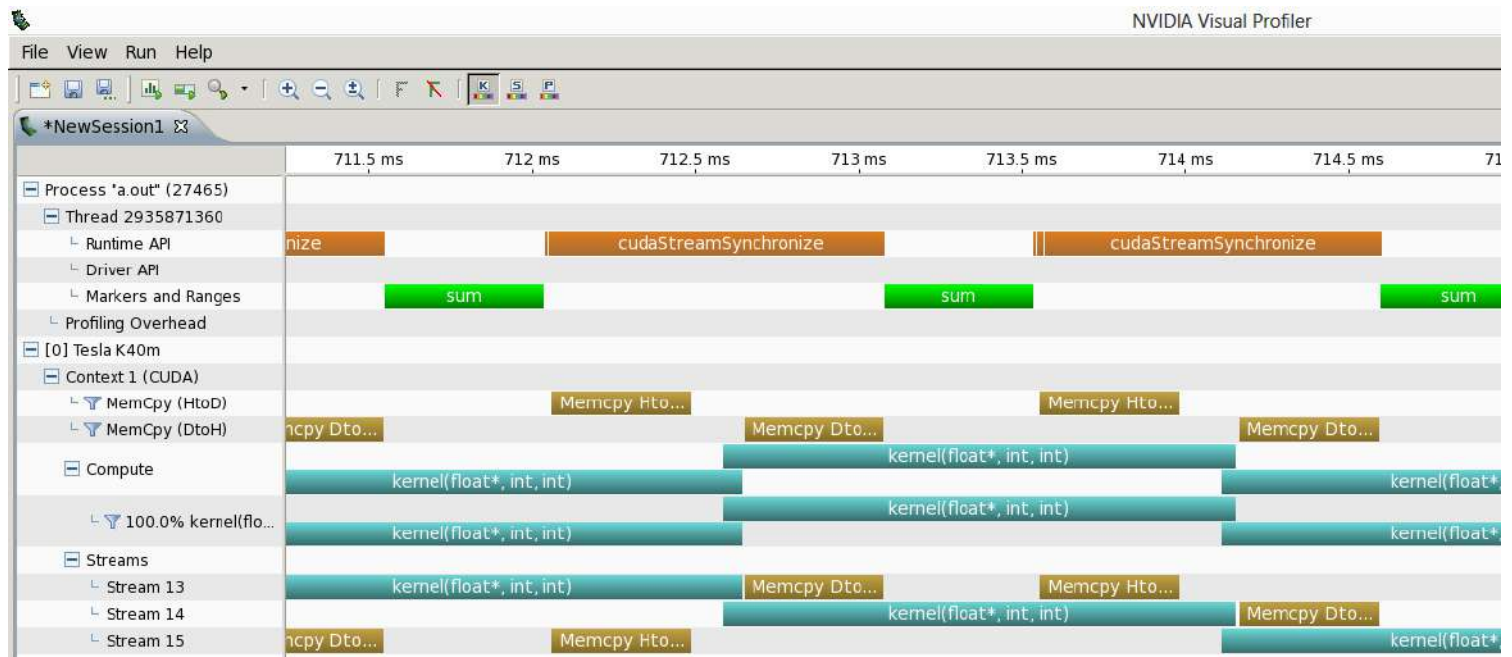


NVTX

- Our current tools only profile API calls on the host
 - What if we want to understand better what the host is doing?
- The NVTX library allows us to annotate profiles with ranges
 - Add: `#include <nvToolsExt.h>`
 - Link with: `-lnvToolsExt`
- Mark the start of a range
 - `nvtxRangePushA("description");`
- Mark the end of a range
 - `nvtxRangePop();`
- Ranges are allowed to overlap

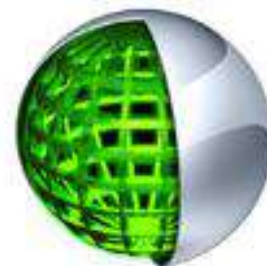
<http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/>

NVTX Profile



NSIGHT

- CUDA enabled Integrated Development Environment
 - Source code editor: syntax highlighting, code refactoring, etc
 - Build Manger
 - Visual Debugger
 - Visual Profiler
- Linux/Macintosh
 - Editor = Eclipse
 - Debugger = cuda-gdb with a visual wrapper
 - Profiler = NVVP
- Windows
 - Integrates directly into Visual Studio
 - Profiler is NSIGHT VSE



Example 4: NSIGHT

Let's import an existing Makefile project into NSIGHT

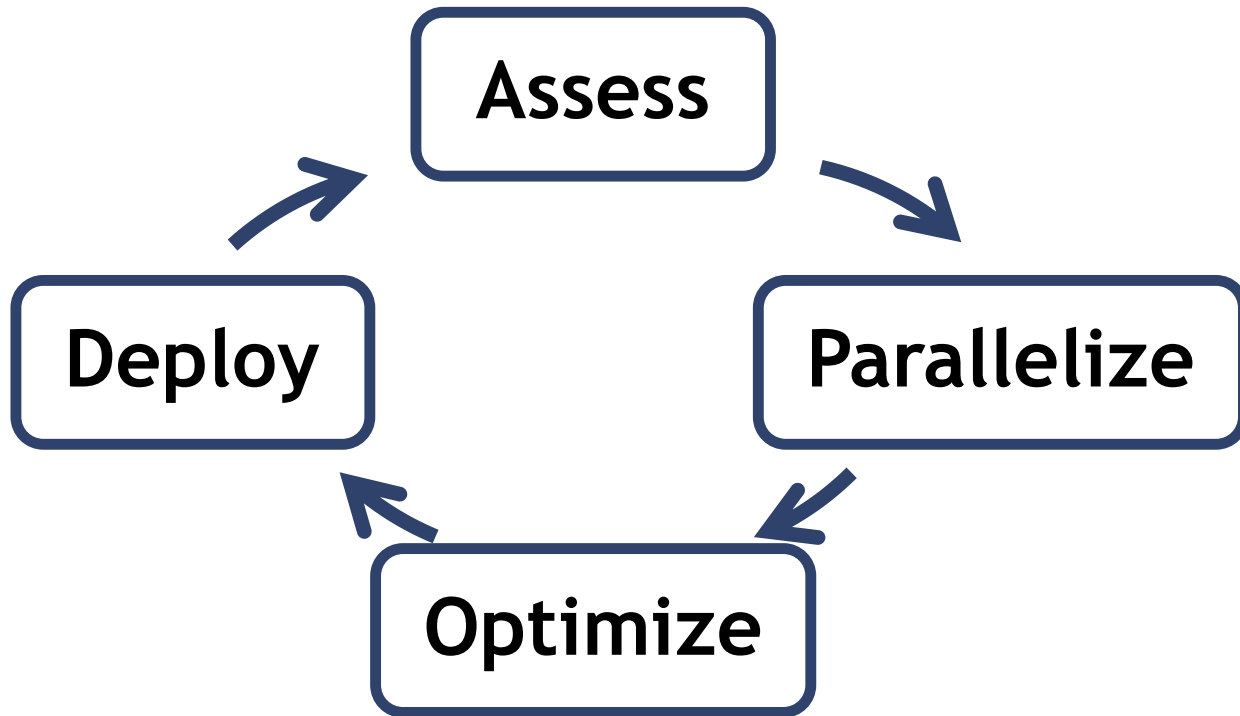
Instructions:

1. Run nsight
 - Select default workspace
2. Click File / New / Makefile Project With Existing CodeTest
3. Enter Project Name and select the Example15 directory
4. Click Finish
5. Right Click On Project / Properties / Run Settings / New / C++ Application
6. Browse for Example 4/a.out
7. In Project Explorer double click on main.cu and explore source
8. Click on the build icon
9. Click on the run icon
10. Click on the profile icon

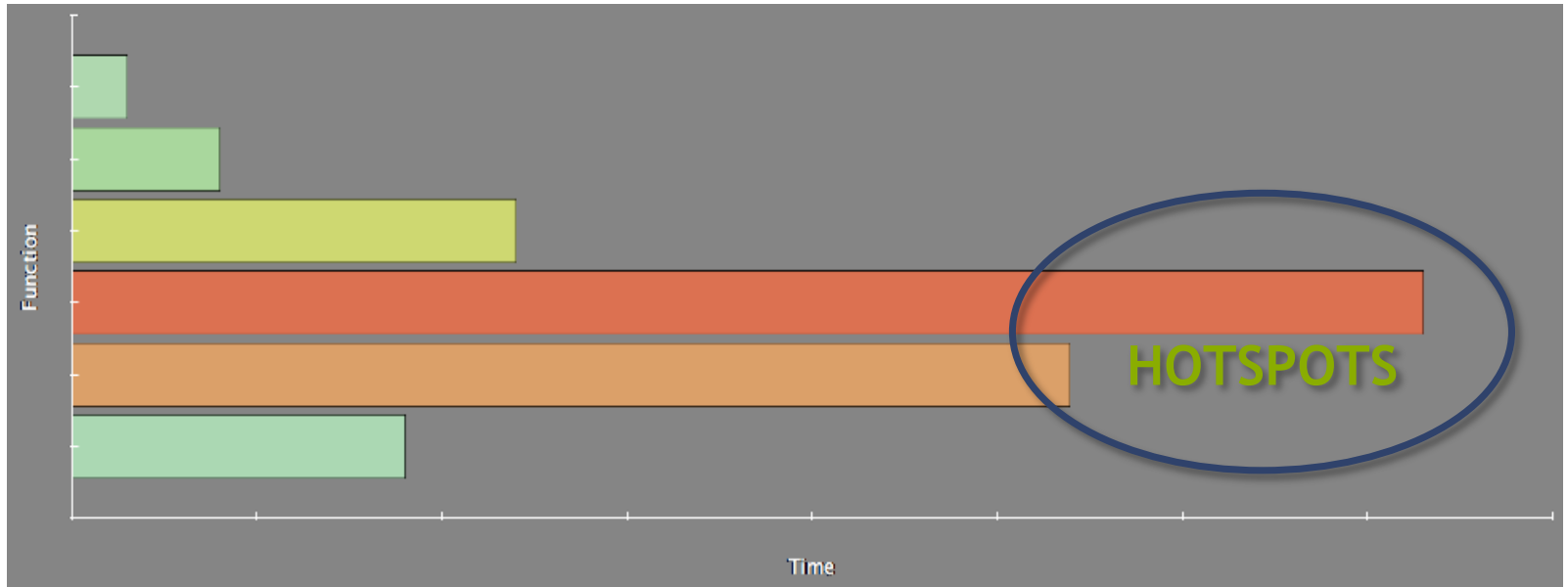
Profiler Summary

- Many profile tools are available
- NVIDIA Provided
 - NVPROF: Command Line
 - NVVP: Visual profiler
 - NSIGHT: IDE (Visual Studio and Eclipse)
- 3rd Party
 - TAU
 - VAMPIR

Optimization



Assess



- Profile the code, find the hotspot(s)
- Focus your attention where it will give the most benefit

Parallelize

Applications

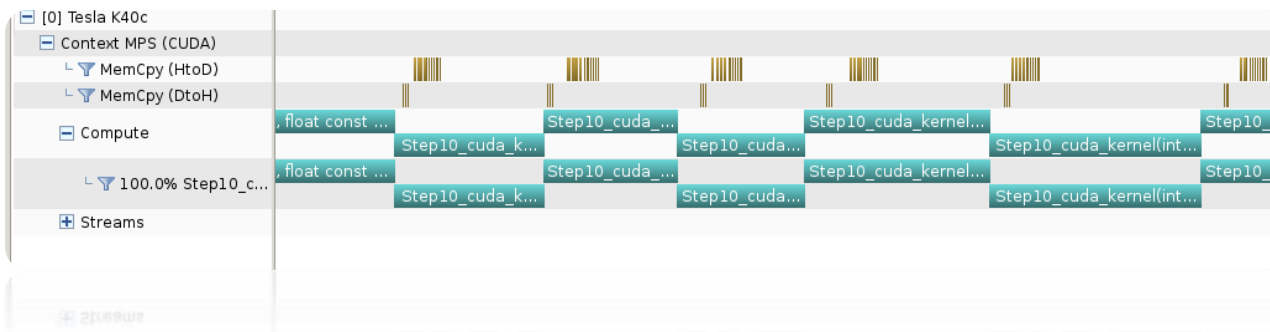
Libraries

Compiler
Directives

Programming
Languages

Optimize

Timeline



Guided System

1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10_cuda_kernel" is most likely limited by compute.

 Perform Compute Analysis

The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

- Perform Latency Analysis

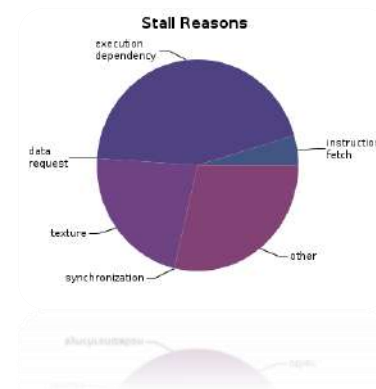
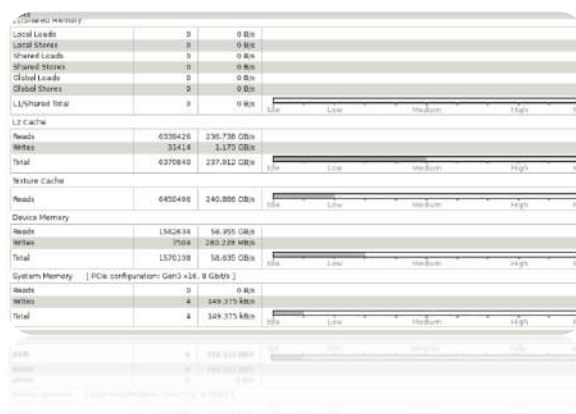
Perform Memory Bandwidth Analysis

Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

Run Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Analysis



Bottleneck Analysis

- Don't assume an optimization was wrong
- Verify if it was wrong with the profiler

129 GB/s ➡ 84 GB/s

L1/Shared Memory

Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	2097152	1,351.979 GB/s
Shared Stores	131072	84.499 GB/s
Global Loads	131072	42.249 GB/s
Global Stores	131072	42.249 GB/s
Atomic	0	0 B/s
L1/Shared Total	2490368	1,520.977 GB/s



gpuTranspose_kernel(int, int, float const *, float*)	
Start	547.303 ms [5]
End	547.716 ms [5]
Duration	413.872 µs
Grid Size	[64,64,1]
Block Size	[32,32,1]
Registers/Thread	10
Shared Memory/Block	4 KiB
▼ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	5.9%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficiency	97.1%
▼ Occupancy	
Achieved	86.7%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB

⚠ Shared Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each access pattern of the memory access.

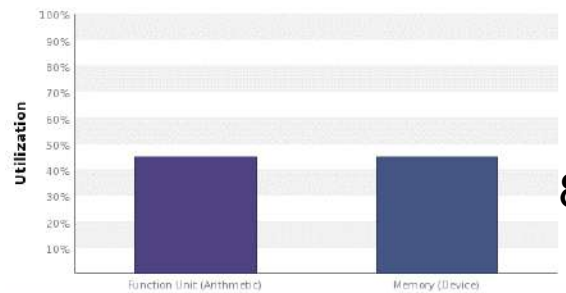
▼ Line / File | main.cu - /home/jluitjens/code/CudaHandsOn/Example19

49 | Shared Load Transactions/Access = 16, Ideal Transactions/Access = 1 [2097152 transactions for 131072 total executions]

Performance Analysis

gpuTranspose_kernel(int, int, float const *, float)

Start	770.067
End	770.324
Duration	256.714
Grid Size	[64,64,1
Block Size	[32,32,1
Registers/Thread	10
Shared Memory/Block	4.125 KiB
▼ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	50%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficiency	97.1%
▼ Occupancy	
Achieved	87.7%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB



84 GB/s ➔ 137 GB/s

L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	131072	138.433 GB/s	
Shared Stores	131720	139.118 GB/s	
Global Loads	131072	69.217 GB/s	
Global Stores	131072	69.217 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	524936	415.984 GB/s	
L2 Cache			
L1 Reads	524288	69.217 GB/s	
L1 Writes	524288	69.217 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Total	1048576	138.433 GB/s	
Texture Cache			
Reads	0	0 B/s	
Device Memory			
Reads	524968	69.306 GB/s	
Writes	524289	69.217 GB/s	
Total	1049257	138.523 GB/s	



GPU Teaching Kit



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

GPU Teaching Kit

Accelerated Computing

Lecture 2.6 - Introduction to CUDA C Unified Memory

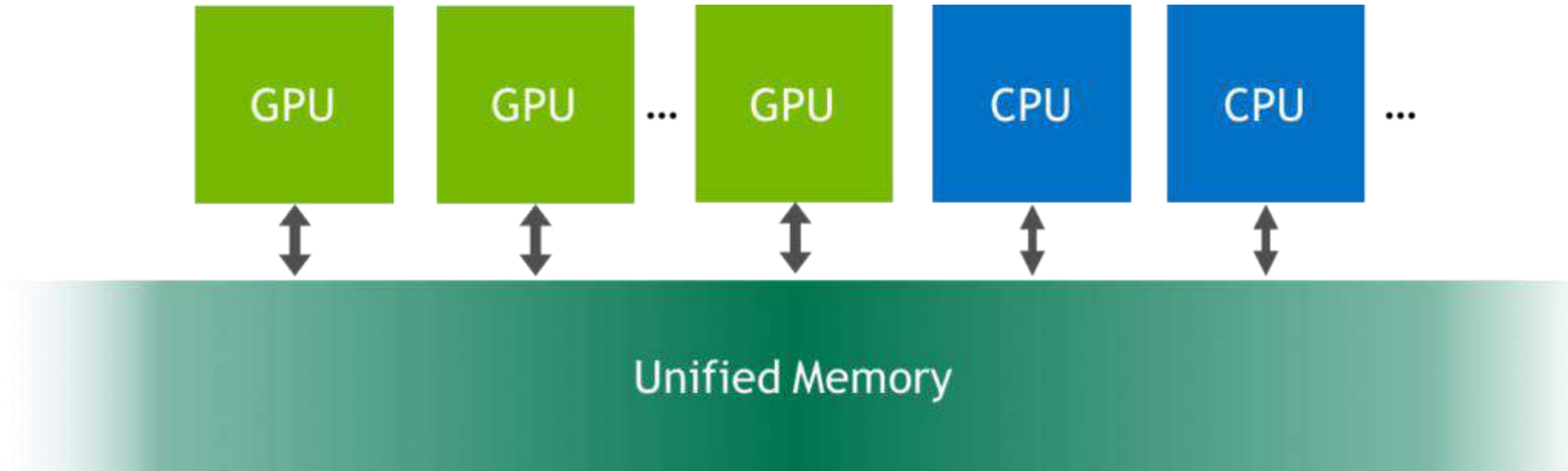


Objective

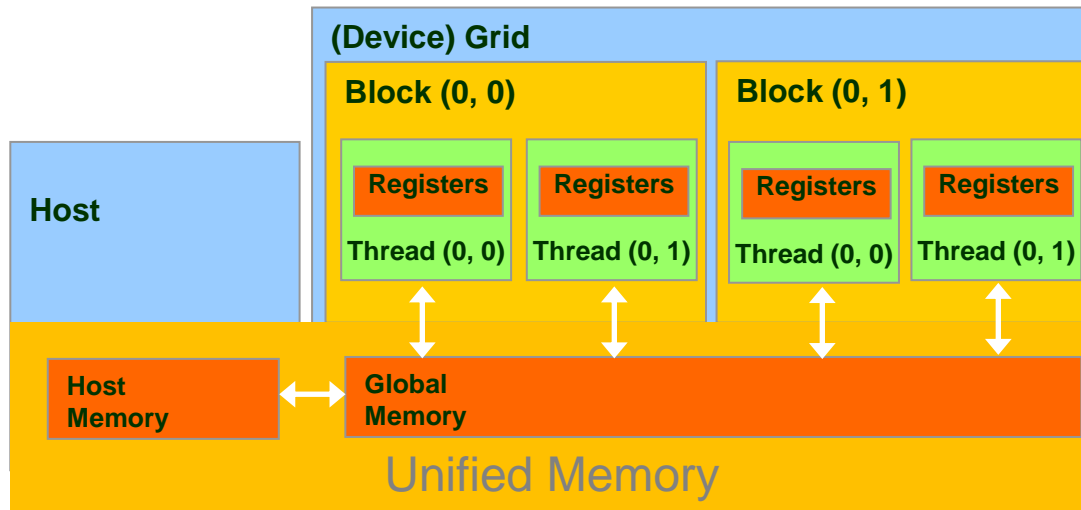
- To learn the basic API functions in CUDA host code for CUDA Unified Memory
 - Unified Memory Allocation
 - Data Transfer in Unified Memory

CUDA Unified Memory (UM)

- Is a single memory address space accessible both from the host and from the device.
- The hardware/software handles automatically the data migration between the host and the device maintaining consistency between them.

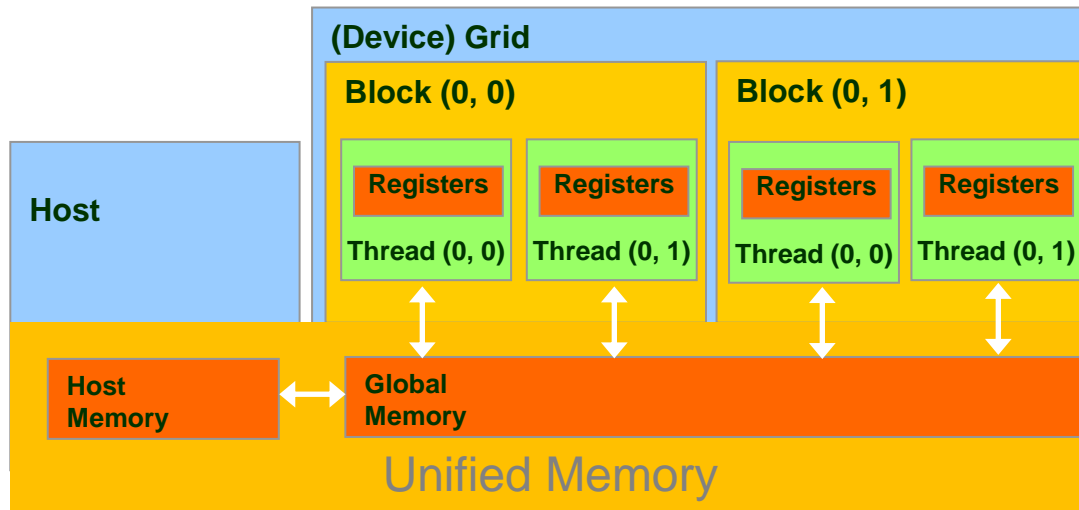


Partial Overview of CUDA Memories



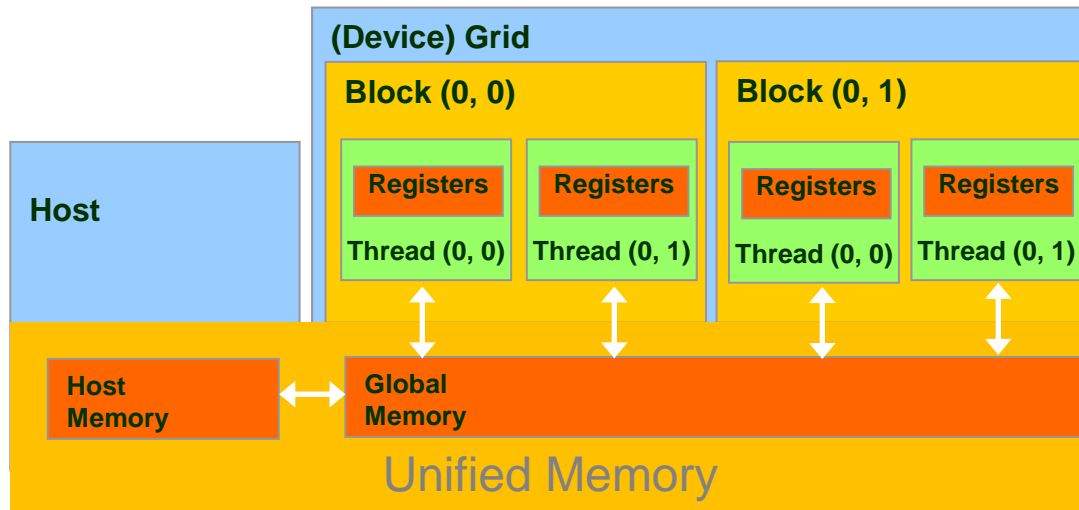
- Device code can:
 - R/W per-thread registers
 - R/W all-shared global memory
 - R/W managed memory (Unified Memory)
- Host code can
 - Transfer data to/from per grid global memory
 - R/W managed memory

Partial Overview of CUDA Memories



- `cudaMallocManaged()`
 - Allocates an object in the Unified Memory address space.
 - Two parameters, with an optional third parameter.
 - Address of a pointer to the allocated object
 - Size of the allocated object in terms of bytes
 - [Optional] Flag indicating if memory can be accessed from any device or stream
- `cudaFree()`
 - Frees object from unified memory.
 - One parameter
 - Pointer to freed object

Partial Overview of CUDA Memories



- `cudaMemcpy()`
 - Memory data transfer
 - Requires four parameters
 - Pointer to destination
 - Pointer to source
 - Number of bytes copied
 - Type/Direction of transfer
 - Depending on the transfer type, the driver may decide to use the memory on the host or the device.
 - In Unified Memory this function is utilized to copy data between different arrays, regardless of position.

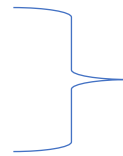
Putting it all together, vecAdd CUDA host code using Unified Memory

```
int main() {
```

```
    float *m_A, float *m_B, float *m_C, int n;
```

```
    int size = n * sizeof(float);
```

```
    cudaMallocManaged((void**) &m_A, size);  
    cudaMallocManaged((void**) &m_B, size);  
    cudaMallocManaged((void**) &m_C, size);
```



Allocation of Managed Memory

```
    // Memory initialization on the Host
```



m_A, m_B gets initialized on the host

```
    // Kernel invocation code - to be shown later
```



The device performs the actual vector addition

```
    cudaFree(m_A); cudaFree(m_B); cudaFree(m_C);
```

```
}
```

CUDA Unified Memory for different architectures

Prior to compute capability 6.x

- There is no specialized hardware units to improve UM efficiency.
- For data migration the full memory block needs to be copied synchronically by the driver.
- No memory oversubscription.

Compute capability 6.x onwards

- There are specialized hardware units managing page faulting.
- Data is migrated on demand, meaning that data gets copied only on page fault.
- Possibility to oversubscribe memory, enabling larger arrays than the device memory size.

GPU Teaching Kit

Accelerated Computing

The GPU Teaching Kit is licensed by NVIDIA under the [Creative Commons Attribution-NonCommercial 4.0 International License](#).





GPU Teaching Kit
Accelerated Computing



Module 3.1 - CUDA Parallelism Model

Kernel-Based SPMD Parallel Programming

Objective

- To learn the basic concepts involved in a simple CUDA kernel function
 - Declaration
 - Built-in variables
 - Thread index to data index mapping

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition
```

```
__global__
```

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

Example: Vector Addition Kernel Launch (Host Code)

Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    // d_A, d_B, d_C allocations and copies omitted
    // Run ceil(n/256.0) blocks of 256 threads each
    vecAddKernel<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
}
```

The ceiling function makes sure that there are enough threads to cover all elements.

More on Kernel Launch (Host Code)

Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    dim3 DimGrid((n-1)/256 + 1, 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

This is an equivalent way to express the ceiling function.

Kernel execution in a nutshell

```
__host__  
void vecAdd(...)
```

```
{  
    dim3 DimGrid(ceil(n/256.0),1,1);  
    dim3 DimBlock(256,1,1);
```

```
    vecAddKernel<<<DimGrid,DimBlock>>>(d_A,d_B  
    ,d_C,n);
```

```
}
```

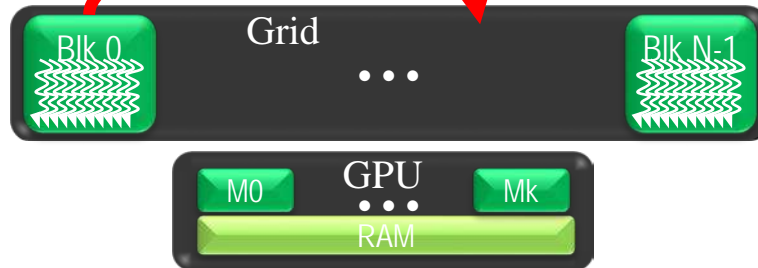
```
__global__
```

```
void vecAddKernel(float *A,  
    float *B, float *C, int n)
```

```
{  
    int i = blockIdx.x * blockDim.x  
        + threadIdx.x;
```

```
    if( i<n ) C[i] = A[i]+B[i];
```

```
}
```

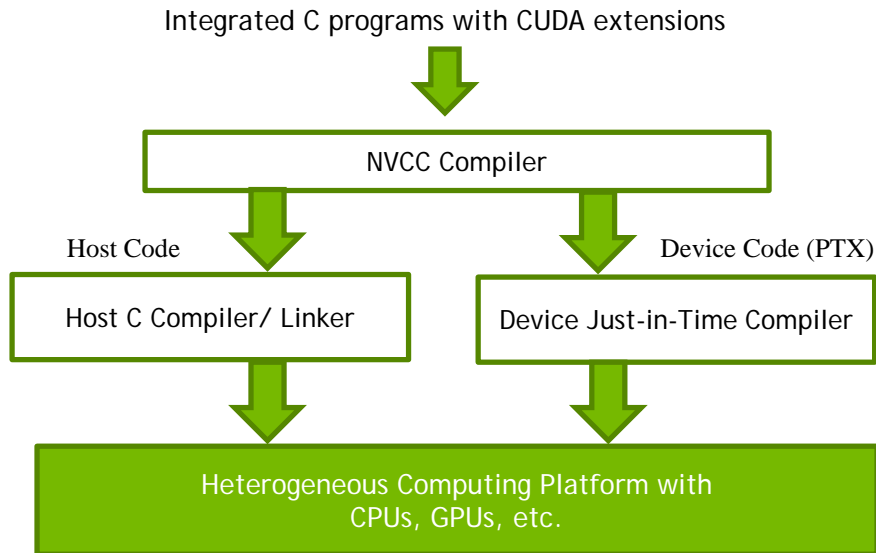


More on CUDA Function Declarations

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` defines a kernel function
 - Each “__” consists of two underscore characters
 - A kernel function must return `void`
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

Compiling A CUDA Program





GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit
Accelerated Computing



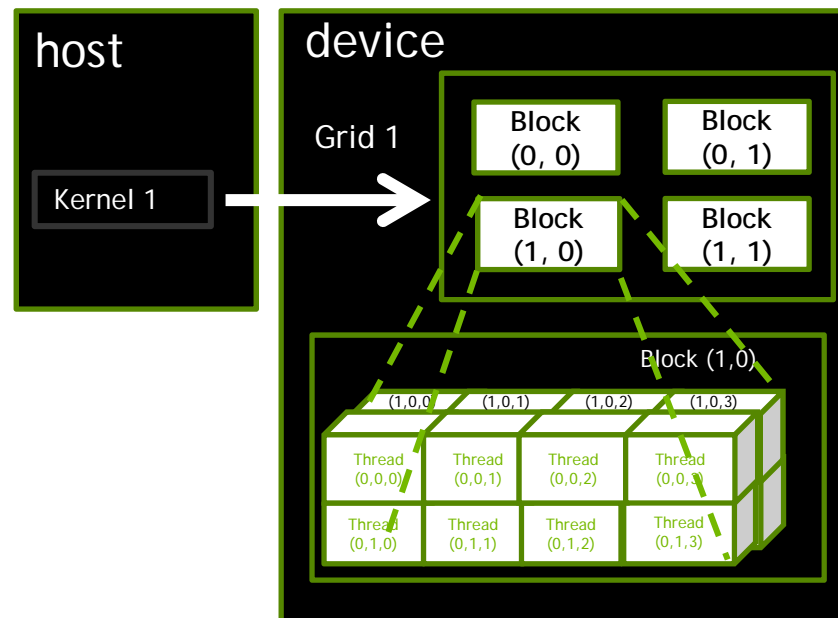
Lecture 3.2 – CUDA Parallelism Model

Multidimensional Kernel Configuration

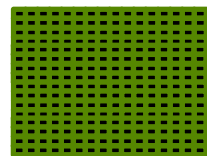
Objective

- To understand multidimensional Grids
 - Multi-dimensional block and thread indices
 - Mapping block/thread indices to data indices

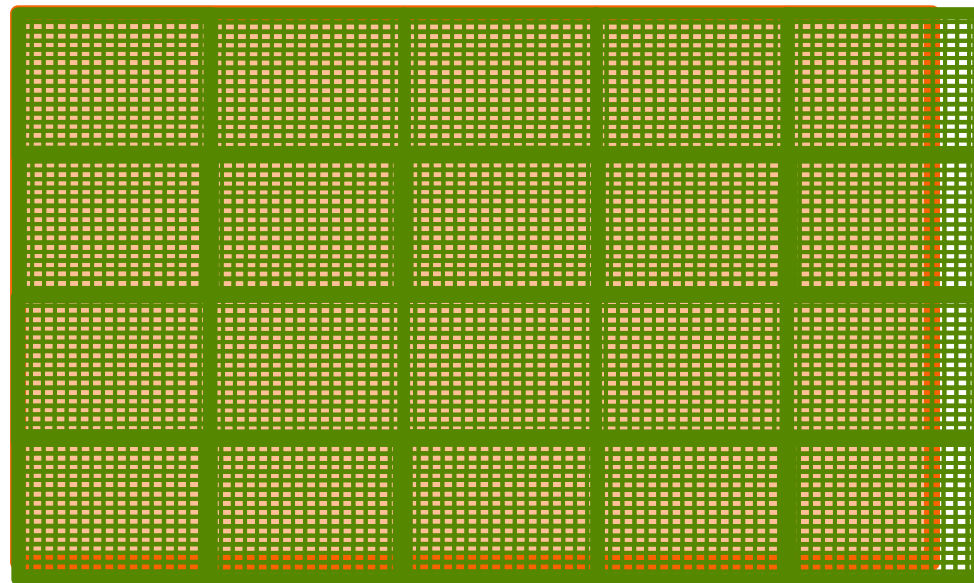
A Multi-Dimensional Grid Example



Processing a Picture with a 2D Grid

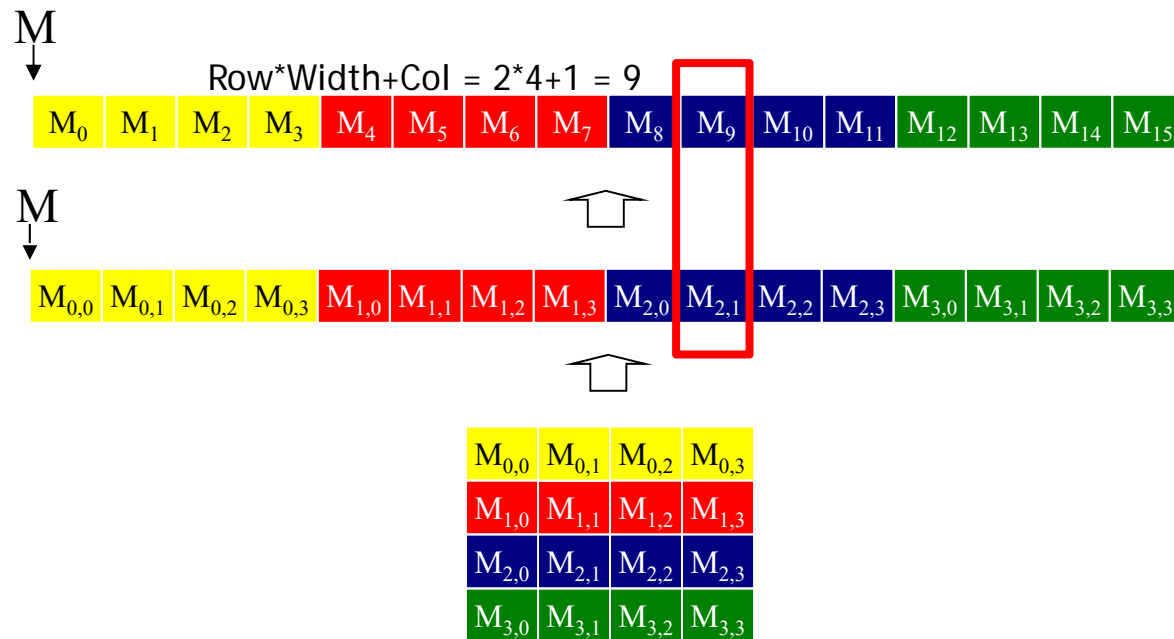


16×16 blocks



62×76 picture

Row-Major Layout in C/C++



Source Code of a PictureKernel

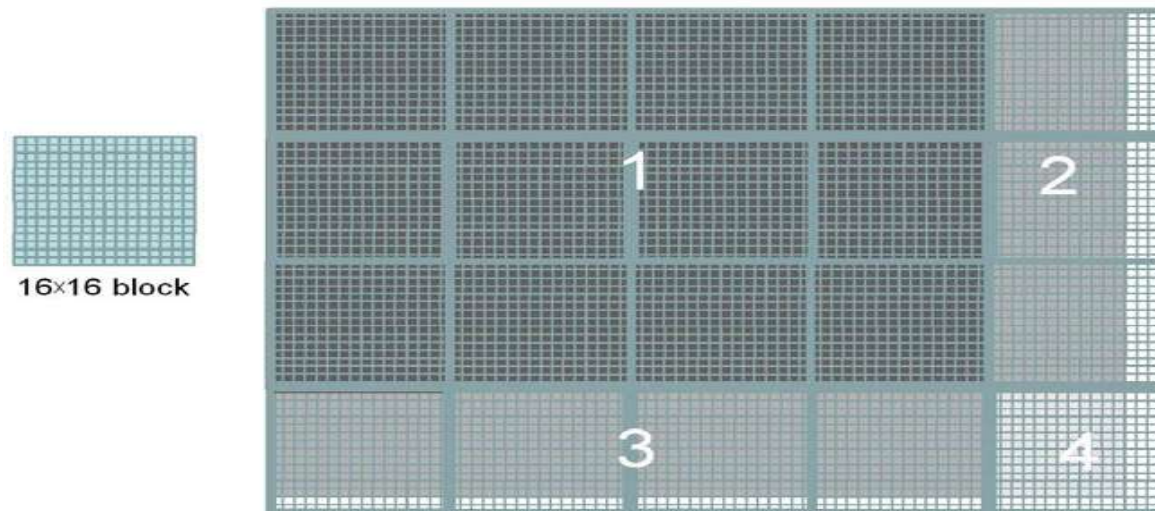
```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,  
                             int height, int width)  
{  
  
    // Calculate the row # of the d_Pin and d_Pout element  
    int Row = blockIdx.y*blockDim.y + threadIdx.y;  
  
    // Calculate the column # of the d_Pin and d_Pout element  
    int Col = blockIdx.x*blockDim.x + threadIdx.x;  
  
    // each thread computes one element of d_Pout if in range  
    if ((Row < height) && (Col < width)) {  
        d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];  
    }  
}
```

Scale every pixel value by 2.0

Host Code for Launching PictureKernel

```
// assume that the picture is m × n,  
// m pixels in y dimension and n pixels in x dimension  
// input d_Pin has been allocated on and copied to device  
// output d_Pout has been allocated on device  
...  
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);  
dim3 DimBlock(16, 16, 1);  
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);  
...
```

Covering a 62×76 Picture with 16×16 Blocks



Not all threads in a Block will follow the same control flow path.



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit
Accelerated Computing



Lecture 3.3 – CUDA Parallelism Model

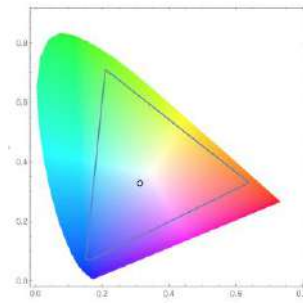
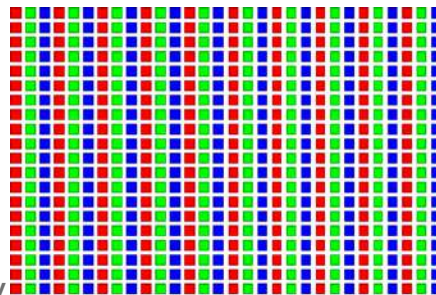
Color-to-Grayscale Image Processing Example

Objective

- To gain deeper understanding of multi-dimensional grid kernel configurations through a real-world use case

RGB Color Image Representation

- Each pixel in an image is an RGB value
- The format of an image's row is (r g b) (r g b) ... (r g b)
- RGB ranges are not distributed uniformly
- Many different color spaces, here we show the constants to convert to AdobeRGB color space
 - The vertical axis (y value) and horizontal axis (x value) show the fraction of the pixel intensity that should be allocated to G and B. The remaining fraction ($1-y-x$) of the pixel intensity that should be assigned to R
 - The triangle contains all the representable colors in this color space



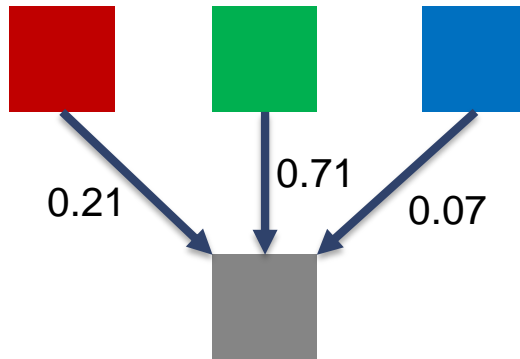
RGB to Grayscale Conversion



A grayscale digital image is an image in which the value of each pixel carries only intensity information.

Color Calculating Formula

- For each pixel (r g b) at (I, J) do:
 $\text{grayPixel}[I,J] = 0.21*r + 0.71*g + 0.07*b$
- This is just a dot product $\langle [r,g,b], [0.21,0.71,0.07] \rangle$ with the constants being specific to input RGB space



RGB to Grayscale Conversion Code

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

    if (x < width && y < height) {

    }
}
```

RGB to Grayscale Conversion Code

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

    if (x < width && y < height) {
        // get 1D coordinate for the grayscale image
        int grayOffset = y*width + x;
        // one can think of the RGB image having
        // CHANNEL times columns than the gray scale image
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = rgbImage[rgbOffset]; // red value for pixel
        unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
        unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
    }
}
```

RGB to Grayscale Conversion Code

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

    if (x < width && y < height) {
        // get 1D coordinate for the grayscale image
        int grayOffset = y*width + x;
        // one can think of the RGB image having
        // CHANNEL times columns than the gray scale image
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = rgbImage[rgbOffset]; // red value for pixel
        unsigned char g = rgbImage[rgbOffset + 2]; // green value for pixel
        unsigned char b = rgbImage[rgbOffset + 3]; // blue value for pixel
        // perform the rescaling and store it
        // We multiply by floating point constants
        grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit
Accelerated Computing



Lecture 3.4 – CUDA Parallelism Model

Image Blur Example

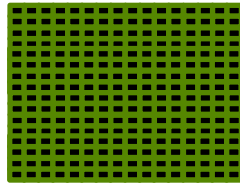
Objective

- To learn a 2D kernel with more complex computation and memory access patterns

Image Blurring



Blurring Box



Pixels
processed
by a thread
block

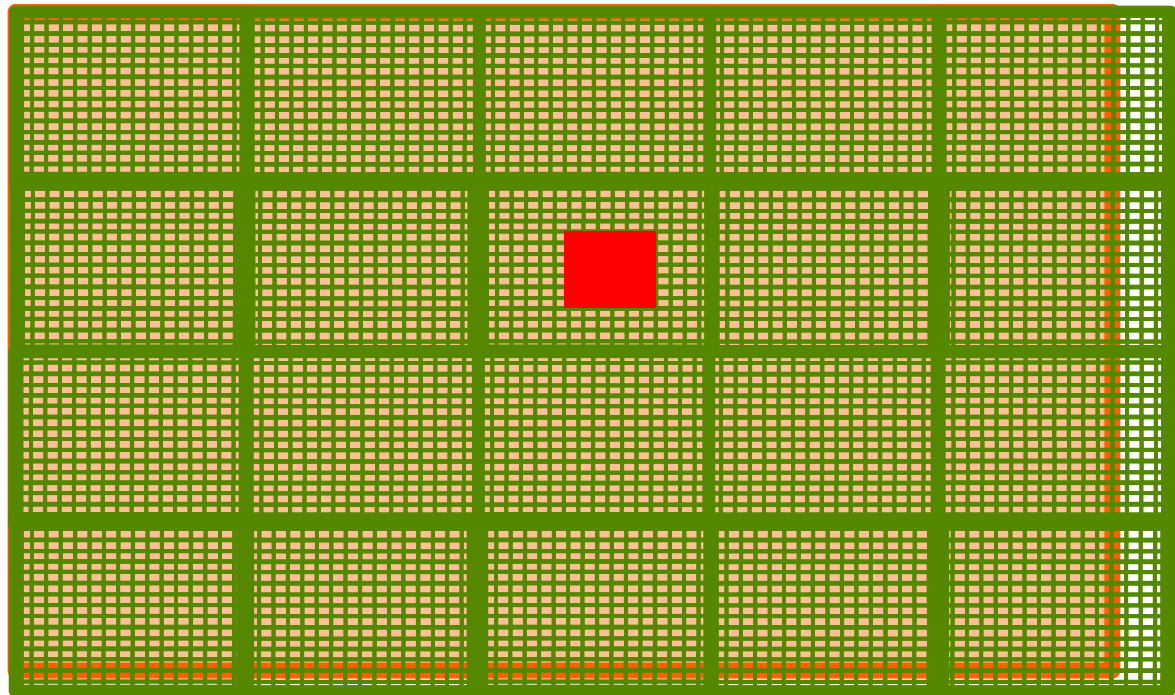


Image Blur as a 2D Kernel

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
{
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    int Row = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
        ... // Rest of our kernel
    }
}
```

```

__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    int Row = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
        int pixVal = 0;
        int pixels = 0;

        // Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
            for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

                int curRow = Row + blurRow;
                int curCol = Col + blurCol;
                // Verify we have a valid image pixel
                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                    pixVal += in[curRow * w + curCol];
                    pixels++; // Keep track of number of pixels in the accumulated total
                }
            }
        }

        // Write our new pixel value out
        out[Row * w + Col] = (unsigned char)(pixVal / pixels);
    }
}

```



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



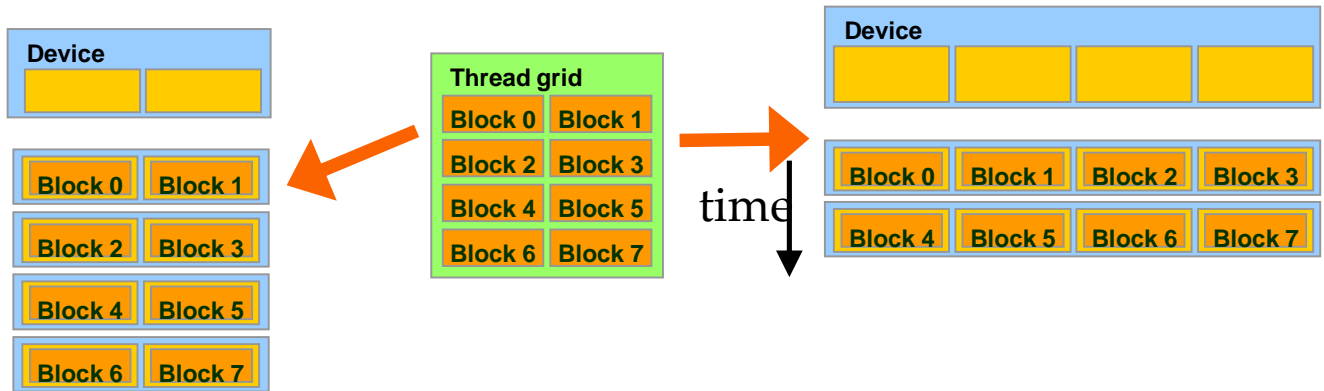
Lecture 3.5 – CUDA Parallelism Model

Thread Scheduling

Objective

- To learn how a CUDA kernel utilizes hardware execution resources
 - Assigning thread blocks to execution resources
 - Capacity constraints of execution resources
 - Zero-overhead thread scheduling

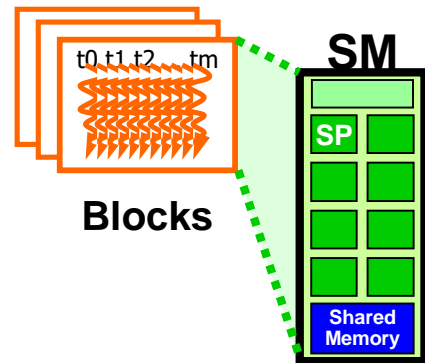
Transparent Scalability



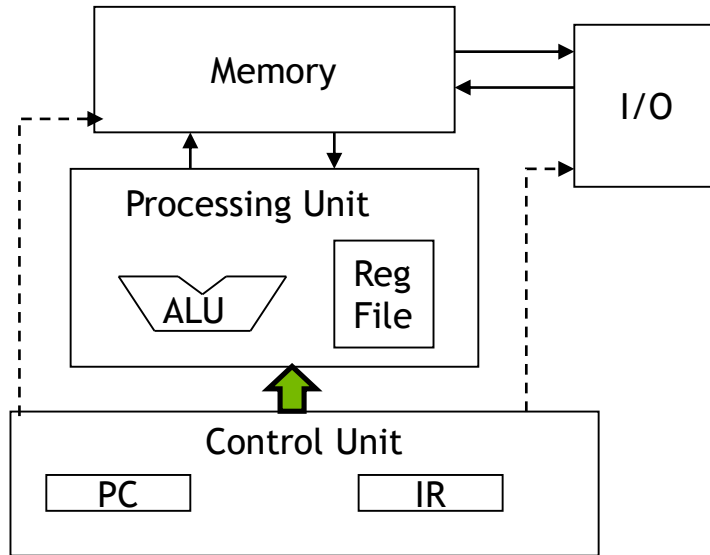
- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
 - A kernel scales to any number of parallel processors

Example: Executing Thread Blocks

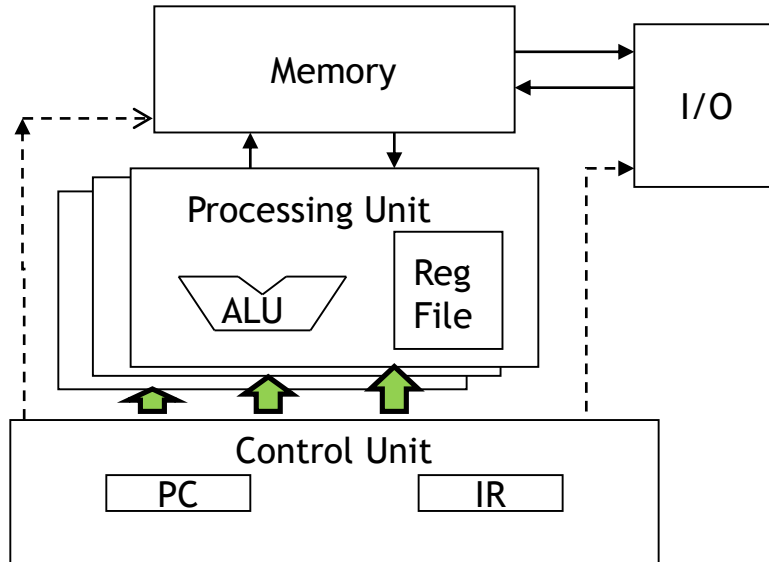
- Threads are assigned to **Streaming Multiprocessors (SM)** in block granularity
 - Up to 32 blocks to each SM as resource allows
 - Volta SM can take up to **2048** threads
 - Could be $256 \text{ (threads/block)} * 8 \text{ blocks}$
 - Or $512 \text{ (threads/block)} * 4 \text{ blocks, etc.}$
- SM maintains thread/block id_x #s
- SM manages/schedules thread execution



The Von-Neumann Model



The Von-Neumann Model with SIMD units



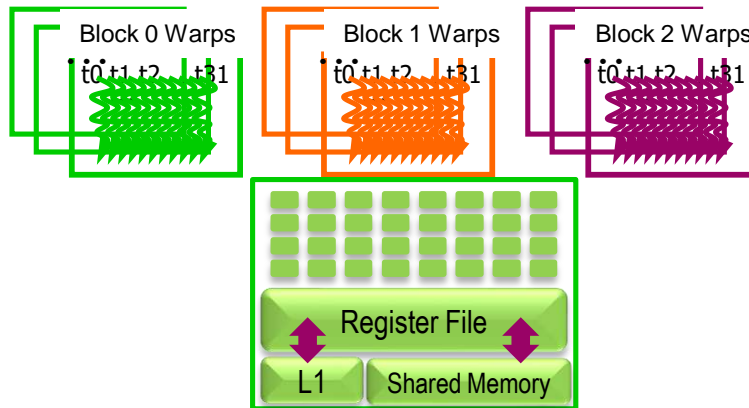
Single Instruction Multiple Data
(SIMD)

Warps as Scheduling Units

- Each Block is executed as 32-thread Warps
 - An implementation decision, not part of the CUDA programming model
 - Warps are scheduling units in SM
 - Threads in a warp execute in SIMD
 - Future GPUs may have different number of threads in each warp

Warp Example

- 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



Example: Thread Scheduling (Cont.)

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

Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should each block have 4X4, 8X8 or 30X30 threads for Volta?
- For 4X4, we have 16 threads per Block. Each SM can take up to 2048 threads, which translates to 128 Blocks. However, each SM can only take up to 32 Blocks, so only 512 threads will go into each SM!
- For 8X8, we have 64 threads per Block. Since each SM can take up to 2048 threads, it can take up to 32 Blocks and achieve full capacity unless other resource considerations overrule.
- For 30X30, we would have 900 threads per Block. Only two blocks could fit into an SM for Volta, so only 1800/2048 of the SM thread capacity would be utilized.



GPU Teaching Kit



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit
Accelerated Computing



Module 4.1 – Memory and Data Locality

CUDA Memories

Objective

- To learn to effectively use the CUDA memory types in a parallel program
 - Importance of memory access efficiency
 - Registers, shared memory, global memory
 - Scope and lifetime

Review: Image Blur Kernel.

```
// Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
    for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

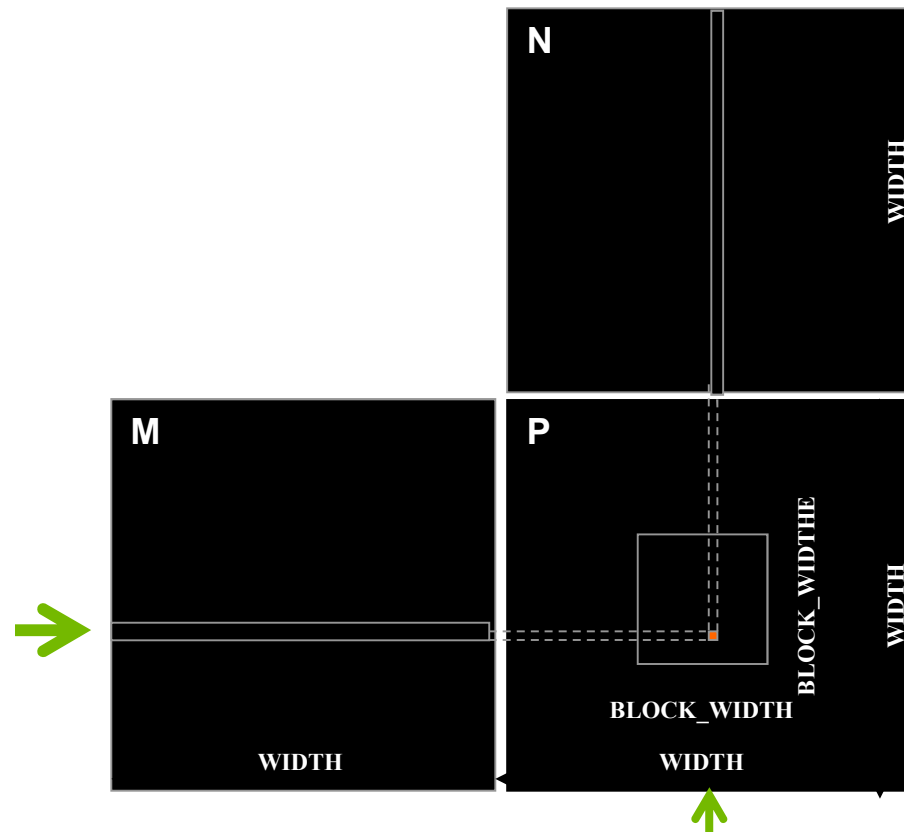
        int curRow = Row + blurRow;
        int curCol = Col + blurCol;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
            ➡ pixVal += in[curRow * w + curCol];
            pixels++; // Keep track of number of pixels in the accumulated total
        }
    }
}

// Write our new pixel value out
out[Row * w + Col] = (unsigned char)(pixVal / pixels);
```

How about performance on a GPU

- All threads access global memory for their input matrix elements
 - One memory accesses (4 bytes) per floating-point addition
 - 4B/s of memory bandwidth/FLOPS
- Assume a GPU with
 - Peak floating-point rate 1,600 GFLOPS with 600 GB/s DRAM bandwidth
 - $4 \times 1,600 = 6,400$ GB/s required to achieve peak FLOPS rating
 - The 600 GB/s memory bandwidth limits the execution at 150 GFLOPS
- This limits the execution rate to 9.3% (150/1600) of the peak floating-point execution rate of the device!
- Need to drastically cut down memory accesses to get close to the 1,600 GFLOPS

Example – Matrix Multiplication



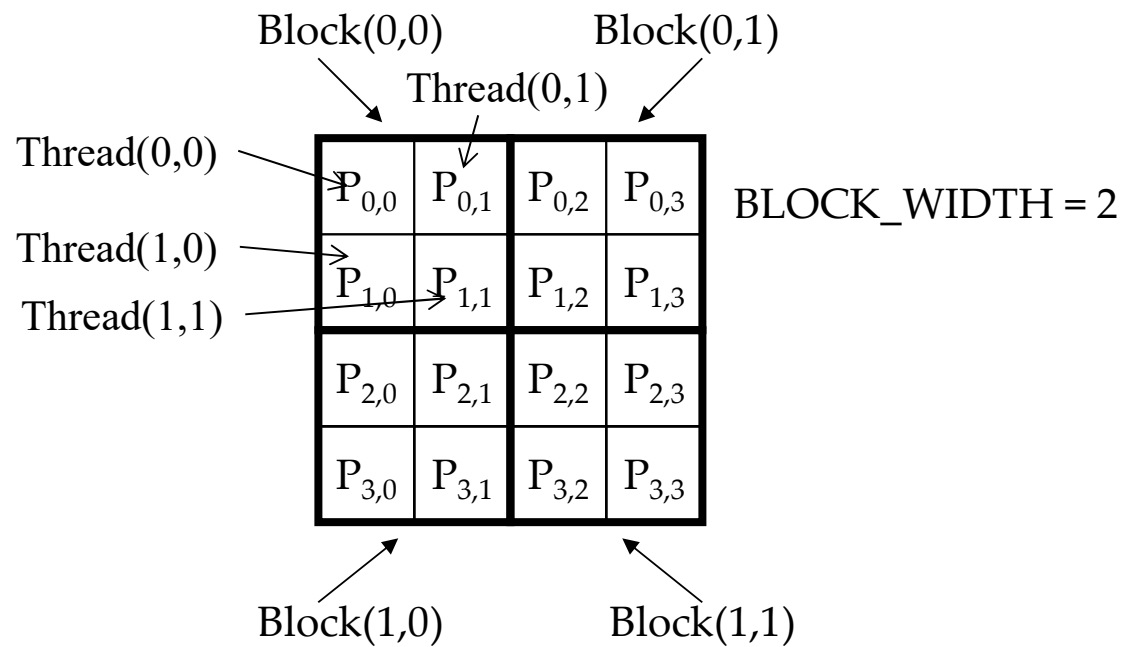
A Basic Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += M[Row*Width+k]*N[k*Width+Col];  
        }  
        P[Row*Width+Col] = Pvalue;  
    }  
}
```

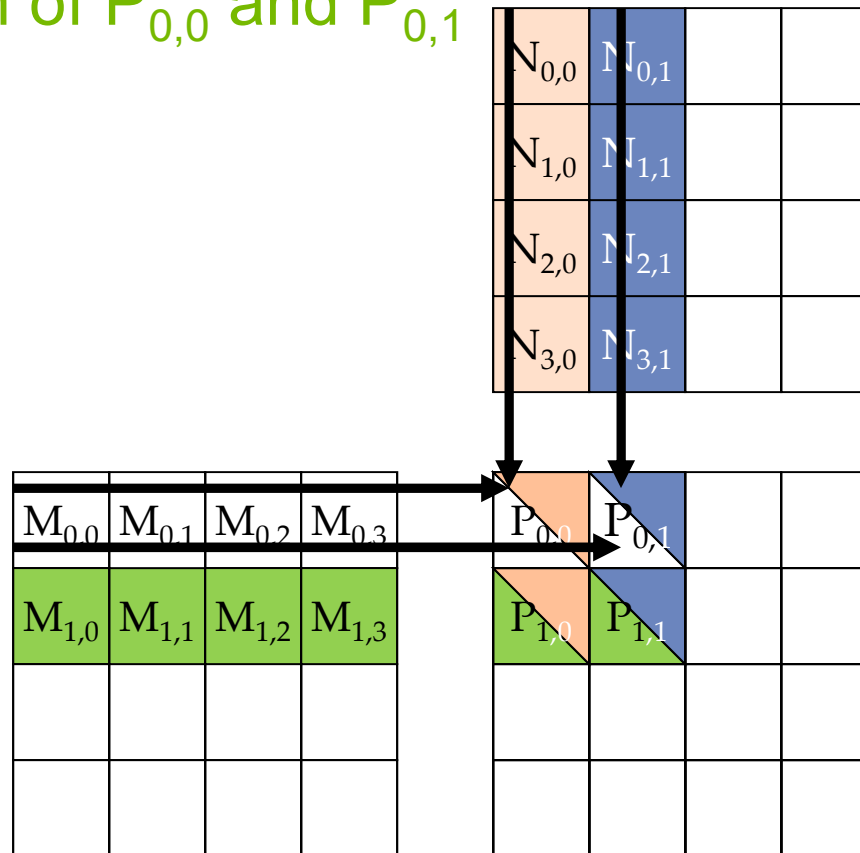
Example – Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {  
  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += M[Row*Width+k]*N[k*Width+Col];  
        }  
        P[Row*Width+Col] = Pvalue;  
    }  
}
```

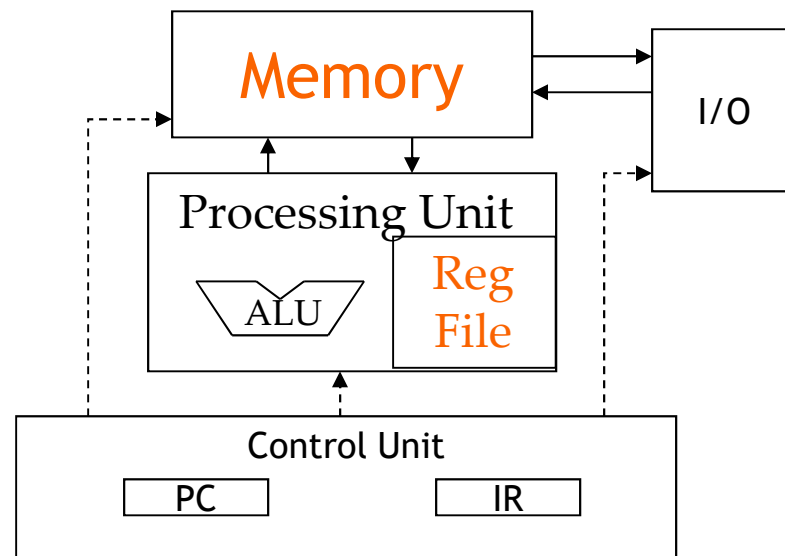
A Toy Example: Thread to P Data Mapping



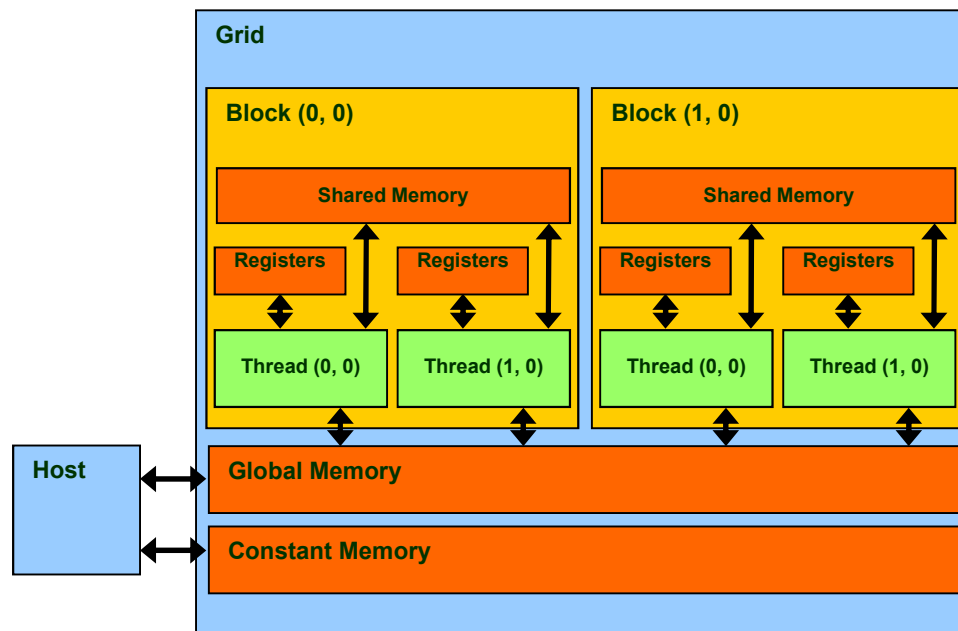
Calculation of $P_{0,0}$ and $P_{0,1}$



Memory and Registers in the Von-Neumann Model



Programmer View of CUDA Memories



Declaring CUDA Variables

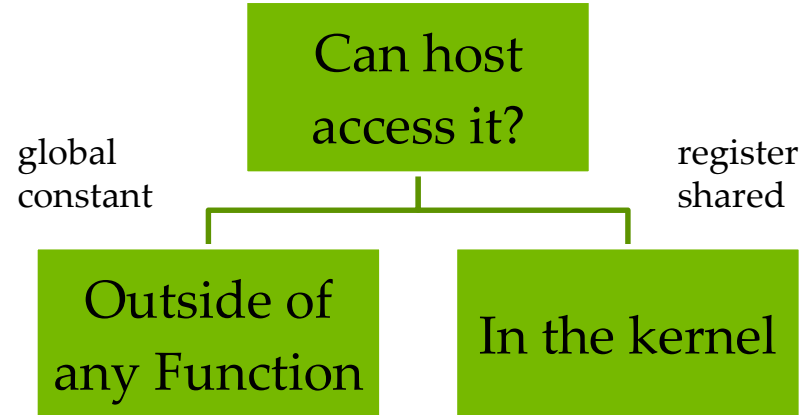
Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables reside in a register
 - Except per-thread arrays that reside in global memory

Example: Shared Memory Variable Declaration

```
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
{
    __shared__ float ds_in[TILE_WIDTH][TILE_WIDTH];
    ...
}
```

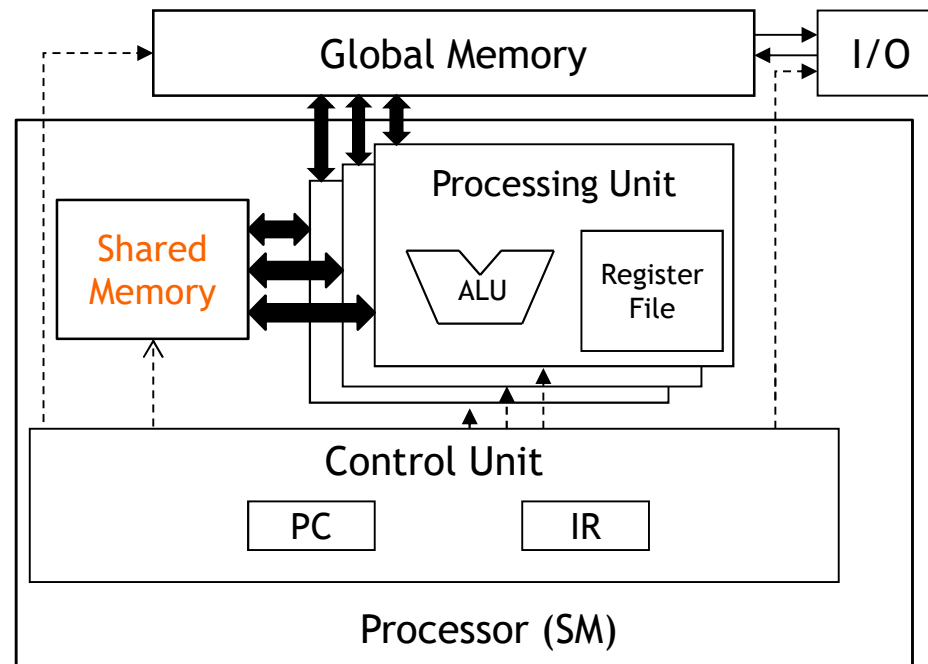
Where to Declare Variables?



Shared Memory in CUDA

- A special type of memory whose contents are explicitly defined and used in the kernel source code
 - One in each SM
 - Accessed at much higher speed (in both latency and throughput) than global memory
 - Scope of access and sharing - thread blocks
 - Lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution
 - Accessed by memory load/store instructions
 - A form of scratchpad memory in computer architecture

Hardware View of CUDA Memories





GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit
Accelerated Computing



Module 4.2 – Memory and Data Locality

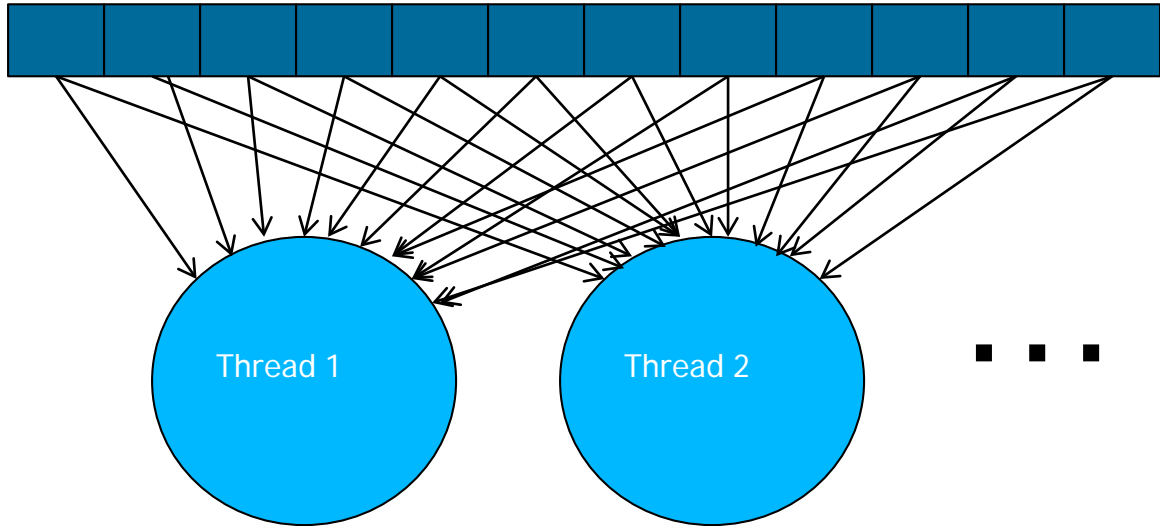
Tiled Parallel Algorithms

Objective

- To understand the motivation and ideas for tiled parallel algorithms
 - Reducing the limiting effect of memory bandwidth on parallel kernel performance
 - Tiled algorithms and barrier synchronization

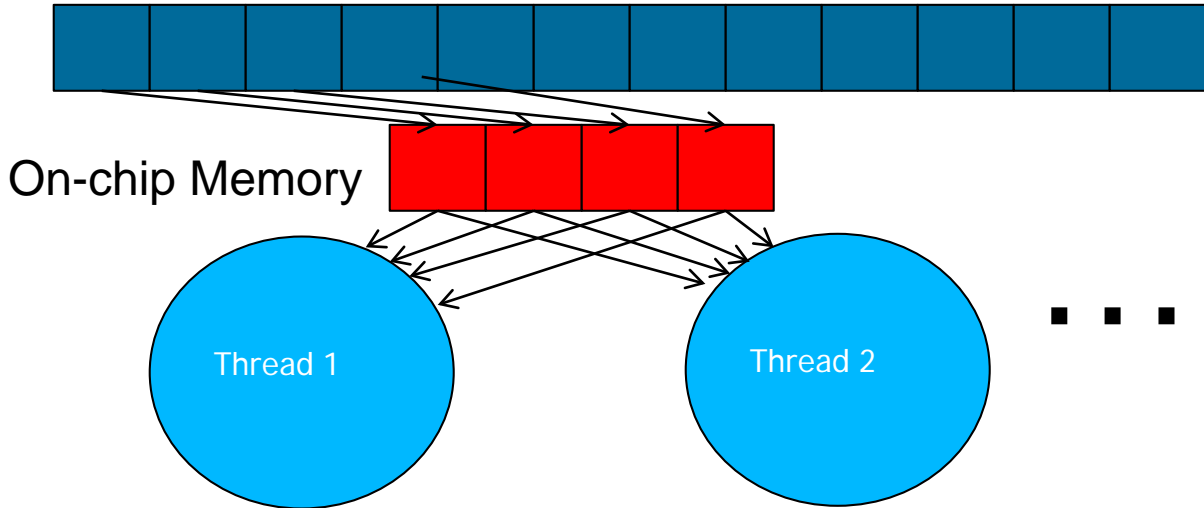
Global Memory Access Pattern of the Basic Matrix Multiplication Kernel

Global Memory



Tiling/Blocking - Basic Idea

Global Memory



Divide the global memory content into tiles

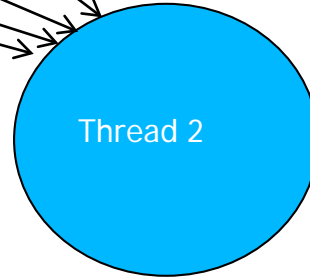
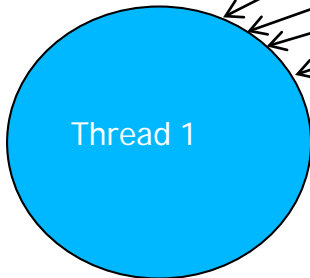
Focus the computation of threads on one or a small number of tiles at each point in time

Tiling/Blocking - Basic Idea

Global Memory



On-chip Memory



...

Basic Concept of Tiling

- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
 - Carpooling for commuters
 - Tiling for global memory accesses
 - drivers = threads accessing their memory data operands
 - cars = memory access requests



Some Computations are More Challenging to Tile

- Some carpools may be easier than others
 - Car pool participants need to have similar work schedule
 - Some vehicles may be more suitable for carpooling
- Similar challenges exist in tiling



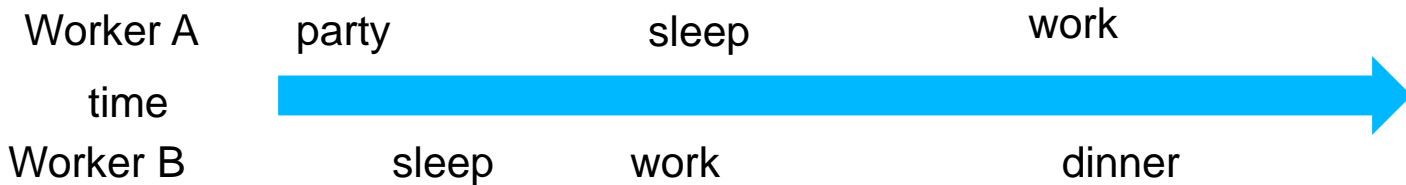
Carpools need synchronization.

- Good: when people have similar schedule



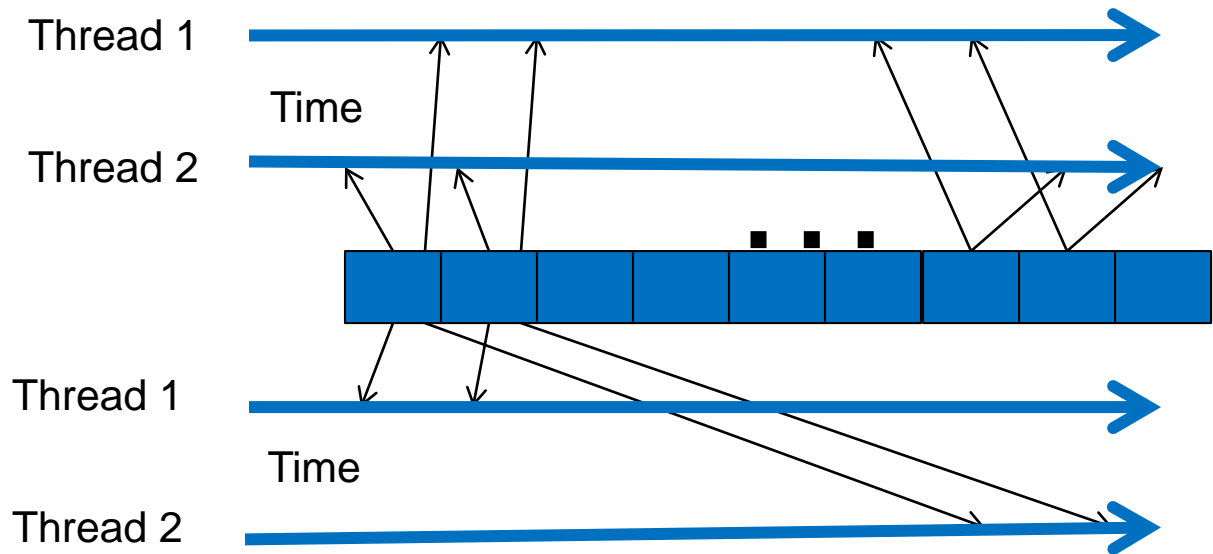
Carpools need synchronization.

- Bad: when people have very different schedule



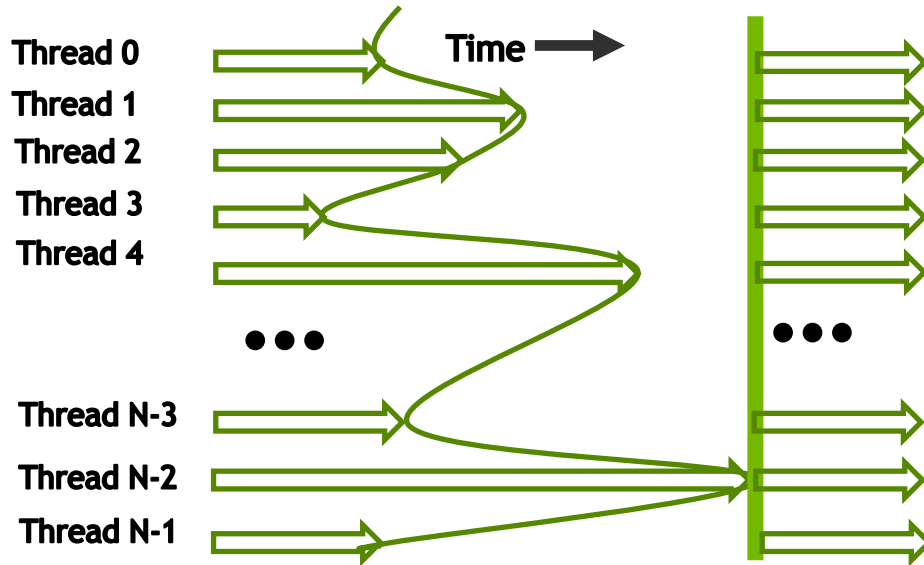
Same with Tiling

- Good: when threads have similar access timing



- Bad: when threads have very different timing

Barrier Synchronization for Tiling



Outline of Tiling Technique

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Use barrier synchronization to make sure that all threads are ready to start the phase
- Have the multiple threads to access their data from the on-chip memory
- Use barrier synchronization to make sure that all threads have completed the current phase
- Move on to the next tile



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit
Accelerated Computing



Module 4.3 - Memory Model and Locality

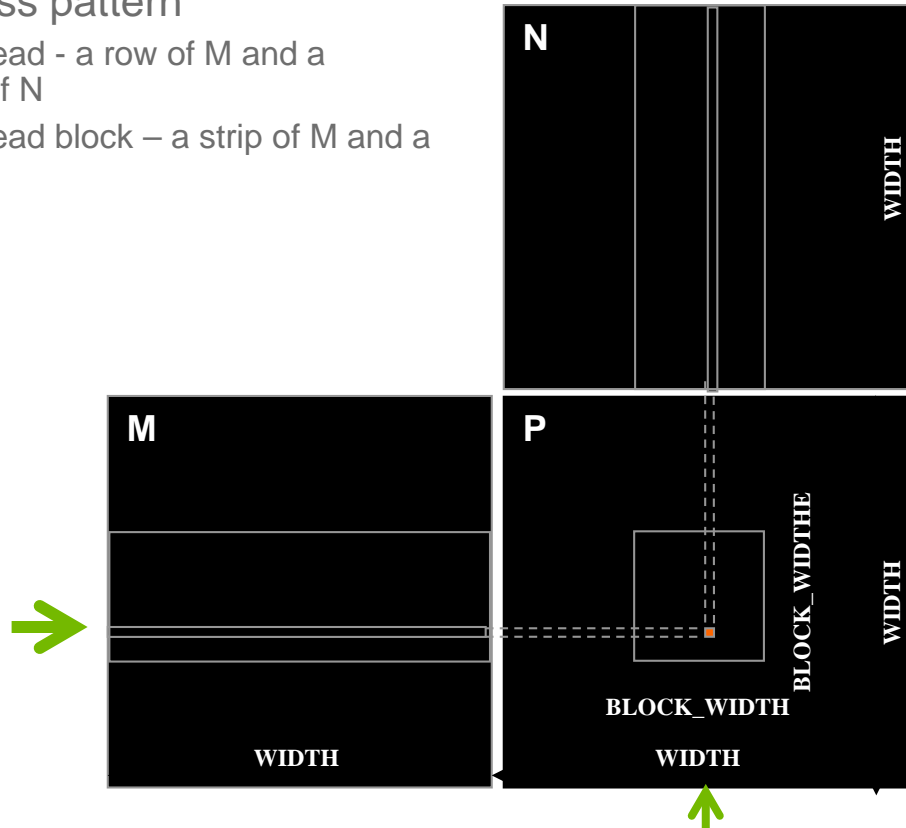
Tiled Matrix Multiplication

Objective

- To understand the design of a tiled parallel algorithm for matrix multiplication
 - Loading a tile
 - Phased execution
 - Barrier Synchronization

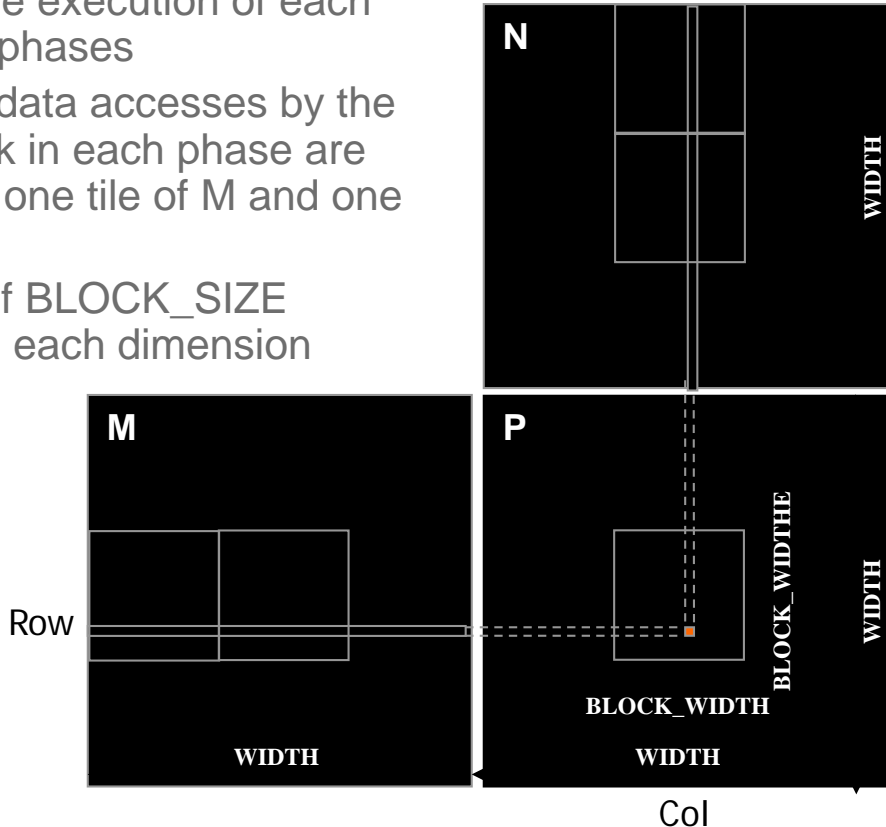
Matrix Multiplication

- Data access pattern
 - Each thread - a row of M and a column of N
 - Each thread block – a strip of M and a strip of N



Tiled Matrix Multiplication

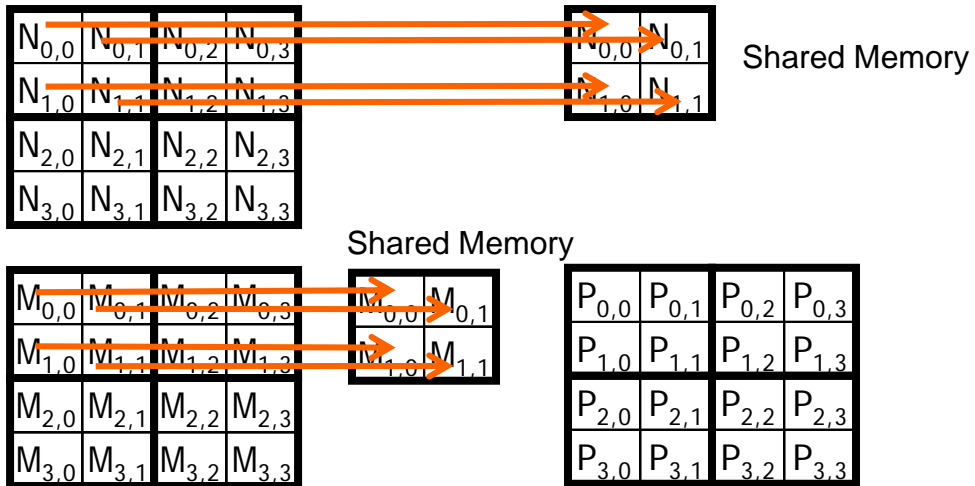
- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of BLOCK_SIZE elements in each dimension



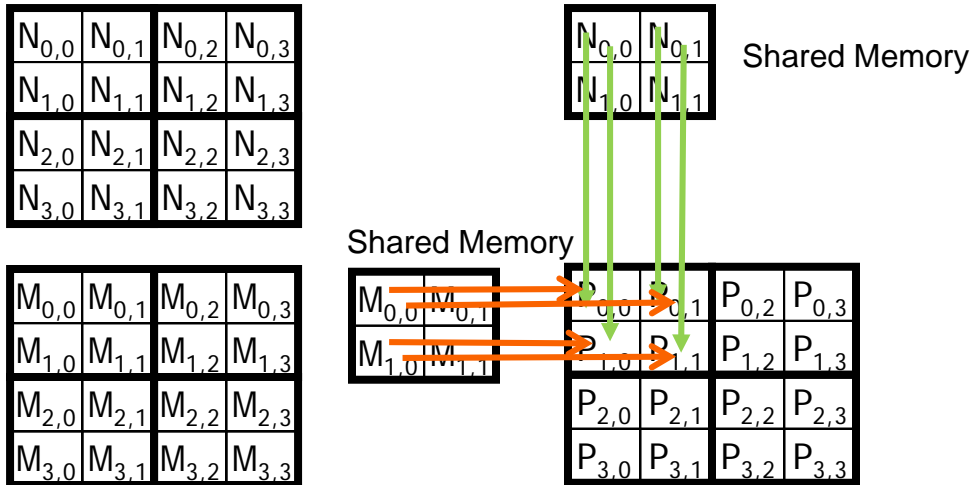
Loading a Tile

- All threads in a block participate
 - Each thread loads one M element and one N element in tiled code

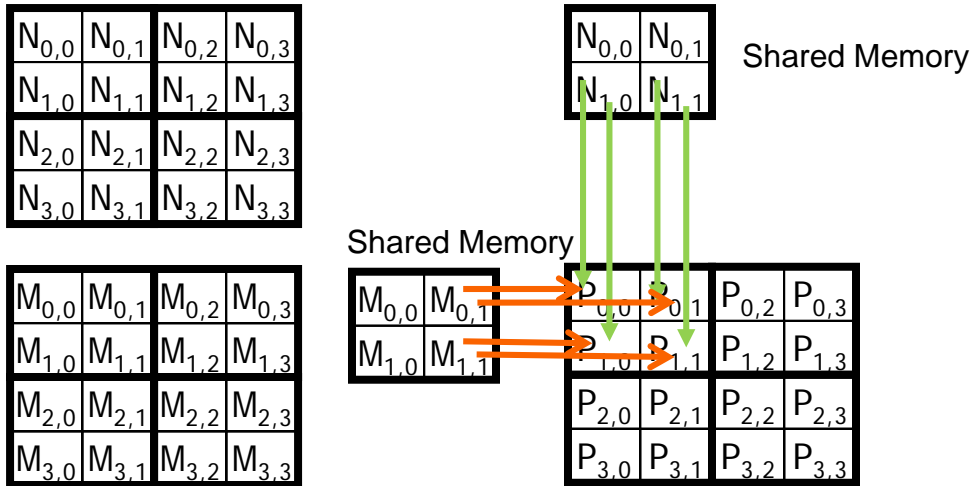
Phase 0 Load for Block (0,0)



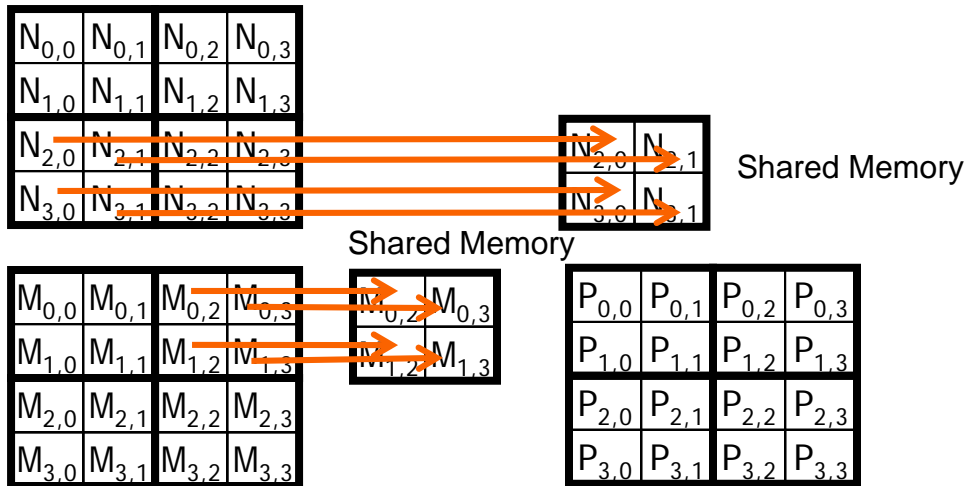
Phase 0 Use for Block (0,0) (iteration 0)



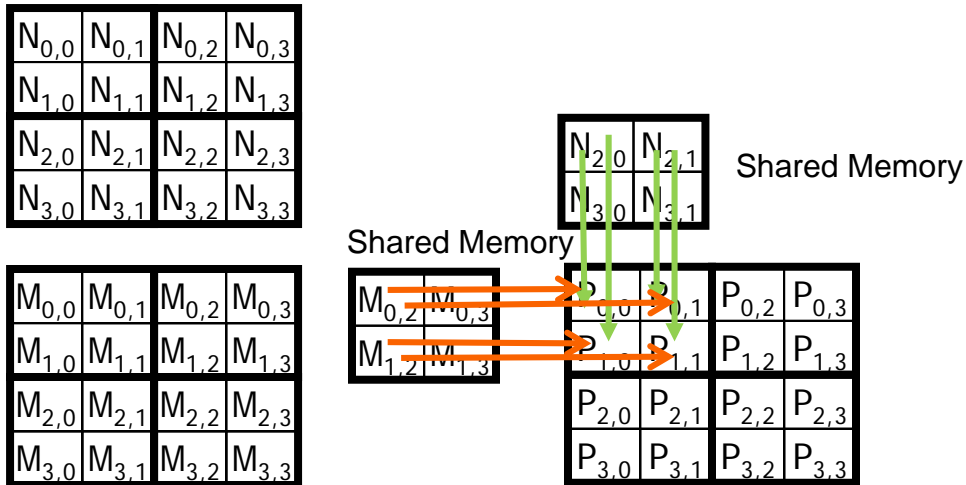
Phase 0 Use for Block (0,0) (iteration 1)



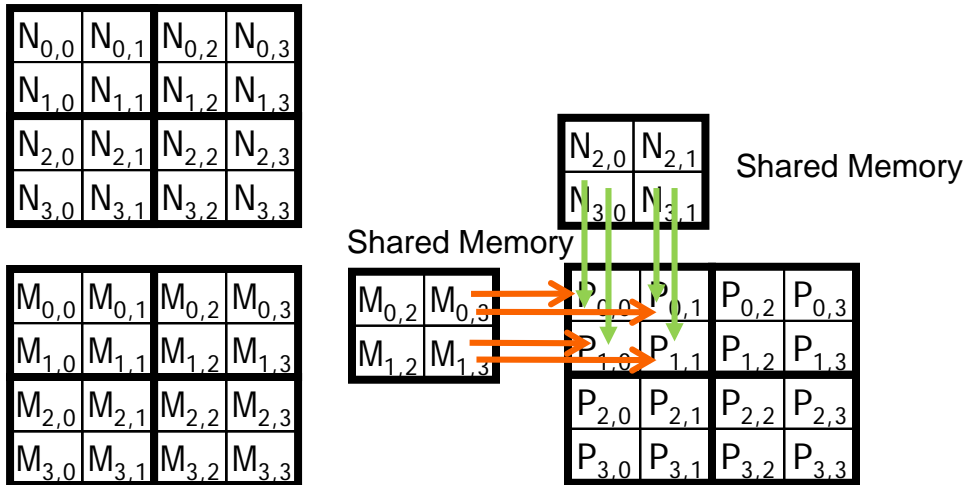
Phase 1 Load for Block (0,0)



Phase 1 Use for Block (0,0) (iteration 0)



Phase 1 Use for Block (0,0) (iteration 1)



Execution Phases of Toy Example

	Phase 0			Phase 1		
thread _{0,0}	M_{0,0} ↓ Mds _{0,0}	N_{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	M_{0,2} ↓ Mds _{0,0}	N_{2,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	M_{0,1} ↓ Mds _{0,1}	N_{0,1} ↓ Nds _{1,0}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	M_{0,3} ↓ Mds _{0,1}	N_{2,1} ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	M_{1,0} ↓ Mds _{1,0}	N_{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	M_{1,2} ↓ Mds _{1,0}	N_{3,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	M_{1,1} ↓ Mds _{1,1}	N_{1,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	M_{1,3} ↓ Mds _{1,1}	N_{3,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}

time



Execution Phases of Toy Example (cont.)

	Phase 0			Phase 1		
thread _{0,0}	M_{0,0} ↓ Mds _{0,0}	N_{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds_{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	M_{0,2} ↓ Mds _{0,0}	N_{2,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	M_{0,1} ↓ Mds _{0,1}	N_{0,1} ↓ Nds _{1,0}	PValue _{0,1} += Mds_{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	M_{0,3} ↓ Mds _{0,1}	N_{2,1} ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	M_{1,0} ↓ Mds _{1,0}	N_{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	M_{1,2} ↓ Mds _{1,0}	N_{3,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	M_{1,1} ↓ Mds _{1,1}	N_{1,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	M_{1,3} ↓ Mds _{1,1}	N_{3,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}

time →

Shared memory allows each value to be accessed by multiple threads

Barrier Synchronization

- Synchronize all threads in a block
 - `__syncthreads()`
- All threads in the same block must reach the `__syncthreads()` before any of them can move on
- Best used to coordinate the phased execution tiled algorithms
 - To ensure that all elements of a tile are loaded at the beginning of a phase
 - To ensure that all elements of a tile are consumed at the end of a phase



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit
Accelerated Computing



Module 4.4 - Memory and Data Locality

Tiled Matrix Multiplication Kernel

Objective

- To learn to write a tiled matrix-multiplication kernel
 - Loading and using tiles for matrix multiplication
 - Barrier synchronization, shared memory
 - Resource Considerations
 - Assume that Width is a multiple of tile size for simplicity

Loading Input Tile 0 of M (Phase 0)

- Have each thread load an M element and an N element at the same relative position as its P element.

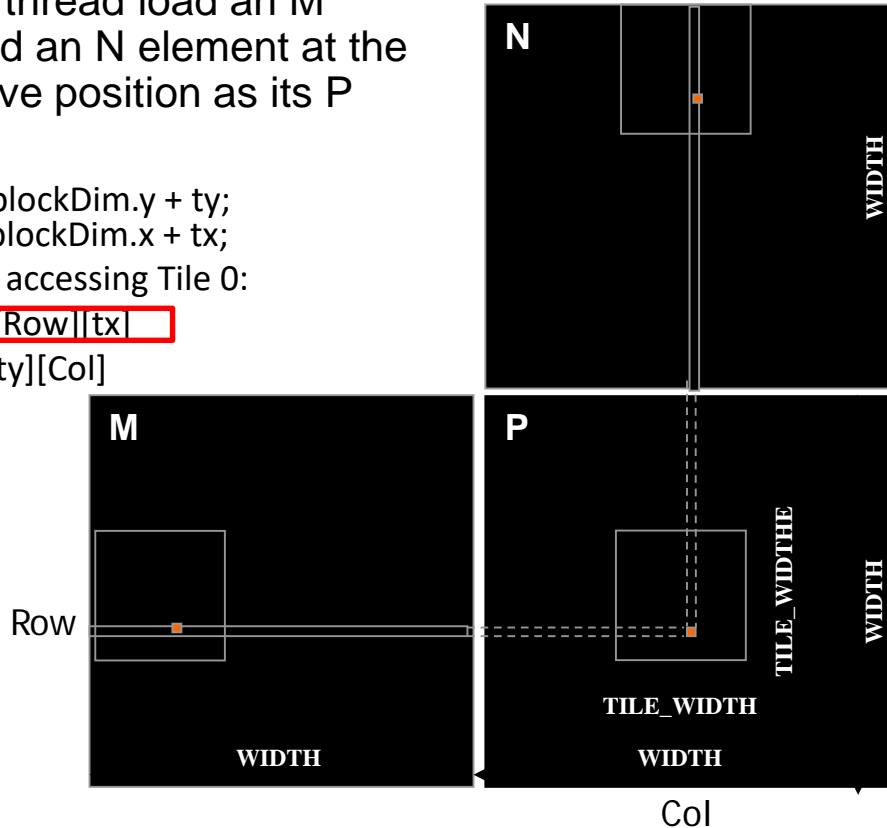
```
int Row = by * blockDim.y + ty;
```

```
int Col = bx * blockDim.x + tx;
```

2D indexing for accessing Tile 0:

M|Row||tx|

N[ty][Col]



Loading Input Tile 0 of N (Phase 0)

- Have each thread load an M element and an N element at the same relative position as its P element.

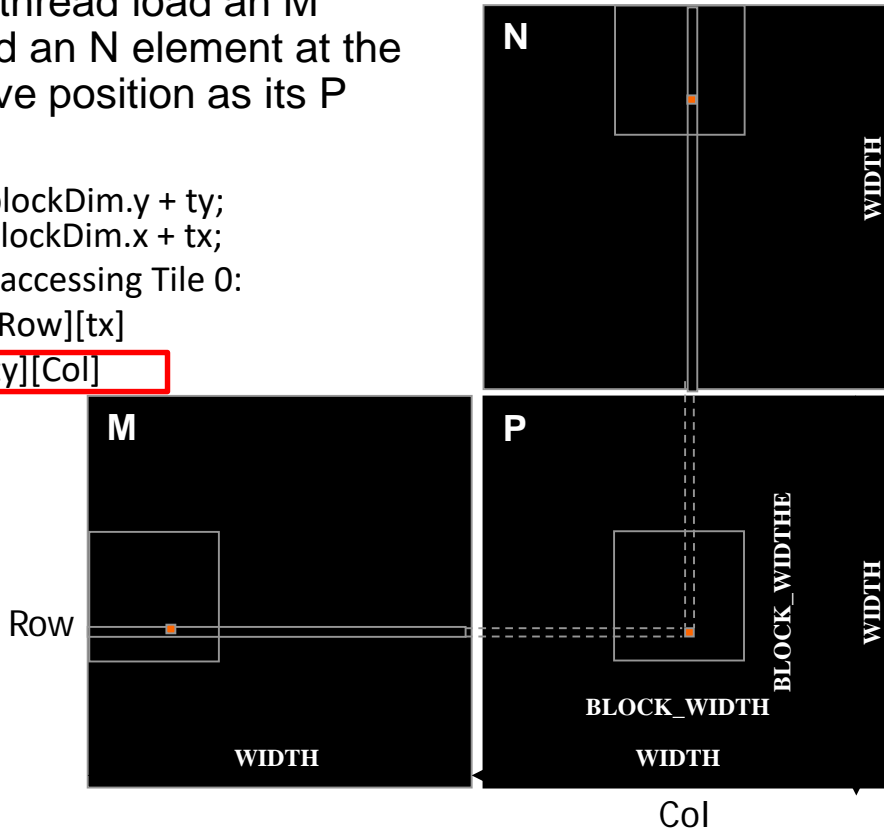
```
int Row = by * blockDim.y + ty;
```

```
int Col = bx * blockDim.x + tx;
```

2D indexing for accessing Tile 0:

```
M[Row][tx]
```

```
N[ty][Col]
```

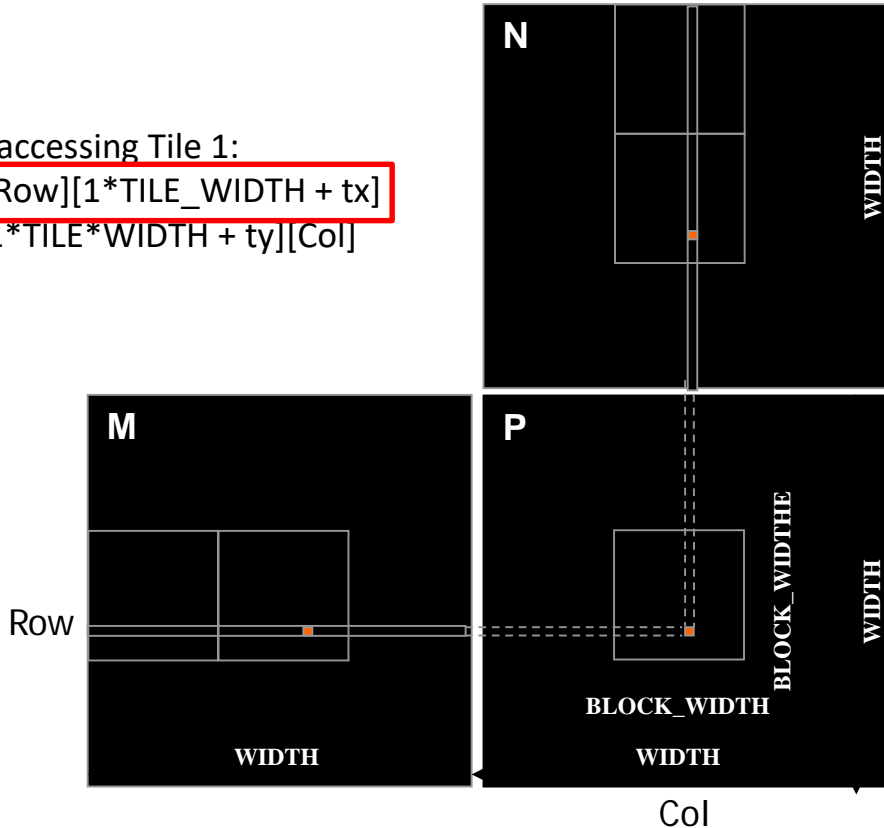


Loading Input Tile 1 of M (Phase 1)

2D indexing for accessing Tile 1:

$M[\text{Row}][1 * \text{TILE_WIDTH} + \text{tx}]$

$N[1 * \text{TILE} * \text{WIDTH} + \text{ty}][\text{Col}]$

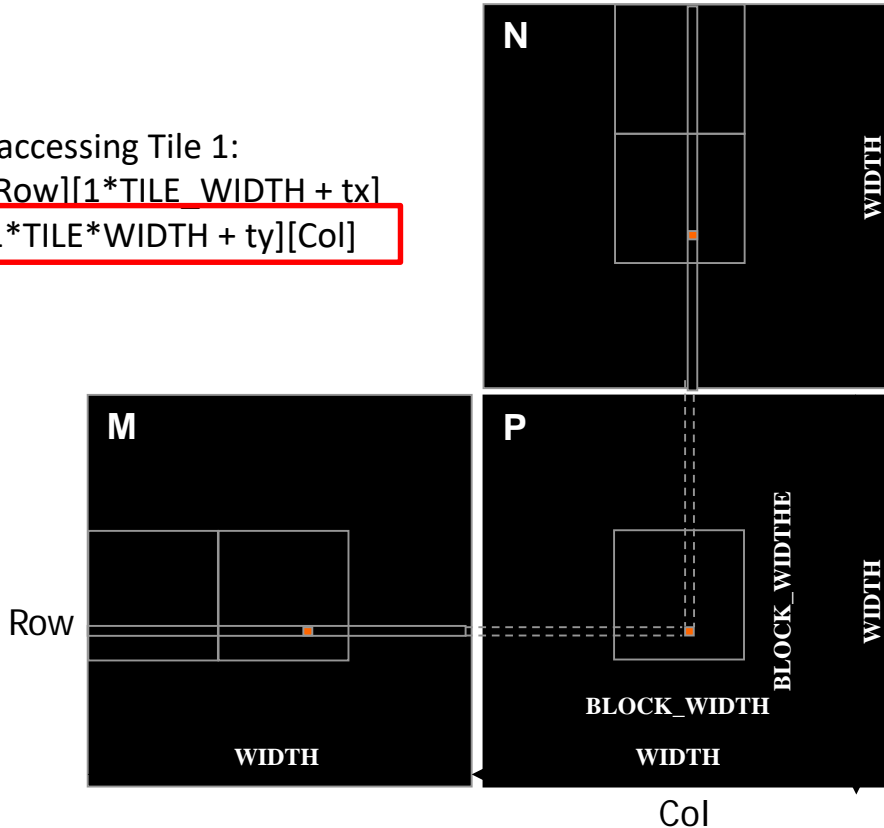


Loading Input Tile 1 of N (Phase 1)

2D indexing for accessing Tile 1:

$M[\text{Row}][1 * \text{TILE_WIDTH} + \text{tx}]$

$N[1 * \text{TILE} * \text{WIDTH} + \text{ty}][\text{Col}]$



M and N are dynamically allocated - use 1D indexing

➡ $M[\text{Row}][p * \text{TILE_WIDTH} + tx]$
➡ $M[\text{Row} * \text{Width} + p * \text{TILE_WIDTH} + tx]$

➡ $N[p * \text{TILE_WIDTH} + ty][\text{Col}]$
➡ $N[(p * \text{TILE_WIDTH} + ty) * \text{Width} + \text{Col}]$

where p is the sequence number of the current phase

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
```

```
{
```

```
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
```

```
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
```

```
    int bx = blockIdx.x;  int by = blockIdx.y;
```

```
    int tx = threadIdx.x; int ty = threadIdx.y;
```

```
    int Row = by * blockDim.y + ty;
```

```
    int Col = bx * blockDim.x + tx;
```

```
    float Pvalue = 0;
```

```
    // Loop over the M and N tiles required to compute the P element
```

```
    for (int p = 0; p < n/TILE_WIDTH; ++p) {
```

```
        // Collaborative loading of M and N tiles into shared memory
```

```
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
```

```
        ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
```

```
        __syncthreads();
```

```
        for (int i = 0; i < TILE_WIDTH; ++i) Pvalue += ds_M[ty][i] * ds_N[i][tx];
```

```
        __syncthreads();
```

```
    }
```

```
    P[Row*Width+Col] = Pvalue;
```

```
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < n/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i) Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < n/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i) Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

Tile (Thread Block) Size Considerations

- Each **thread block** should have many threads
 - TILE_WIDTH of 16 gives $16*16 = 256$ threads
 - TILE_WIDTH of 32 gives $32*32 = 1024$ threads
- For 16, in each phase, each block performs $2*256 = 512$ float loads from global memory for $256 * (2*16) = 8,192$ mul/add operations. (16 floating-point operations for each memory load)
- For 32, in each phase, each block performs $2*1024 = 2048$ float loads from global memory for $1024 * (2*32) = 65,536$ mul/add operations. (32 floating-point operation for each memory load)

Shared Memory and Threading

- For an SM with 16KB shared memory
 - Shared memory size is implementation dependent!
 - For `TILE_WIDTH = 16`, each thread block uses $2 \times 256 \times 4\text{B} = 2\text{KB}$ of shared memory.
 - For 16KB shared memory, one can potentially have up to 8 thread blocks executing
 - This allows up to $8 \times 512 = 4,096$ pending loads. (2 per thread, 256 threads per block)
 - The next `TILE_WIDTH 32` would lead to $2 \times 32 \times 32 \times 4\text{Byte} = 8\text{K Byte}$ shared memory usage per thread block, allowing 2 thread blocks active at the same time
 - However, in a GPU where the thread count is limited to 1536 threads per SM, the number of blocks per SM is reduced to one!
- Each `__syncthread()` can reduce the number of active threads for a block
 - More thread blocks can be advantageous



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



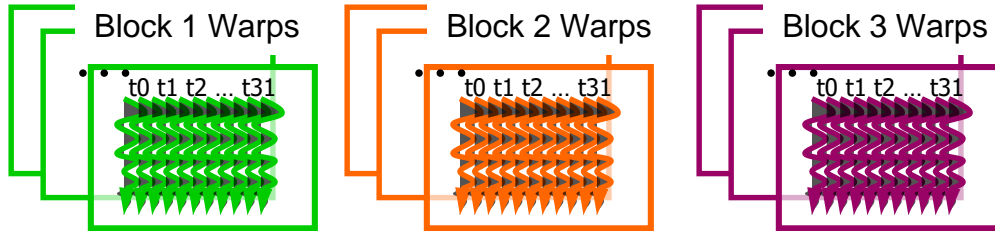
Module 5.1 – Thread Execution Efficiency

Warps and SIMD Hardware

Objective

- To understand how CUDA threads execute on SIMD Hardware
 - Warp partitioning
 - SIMD Hardware
 - Control divergence

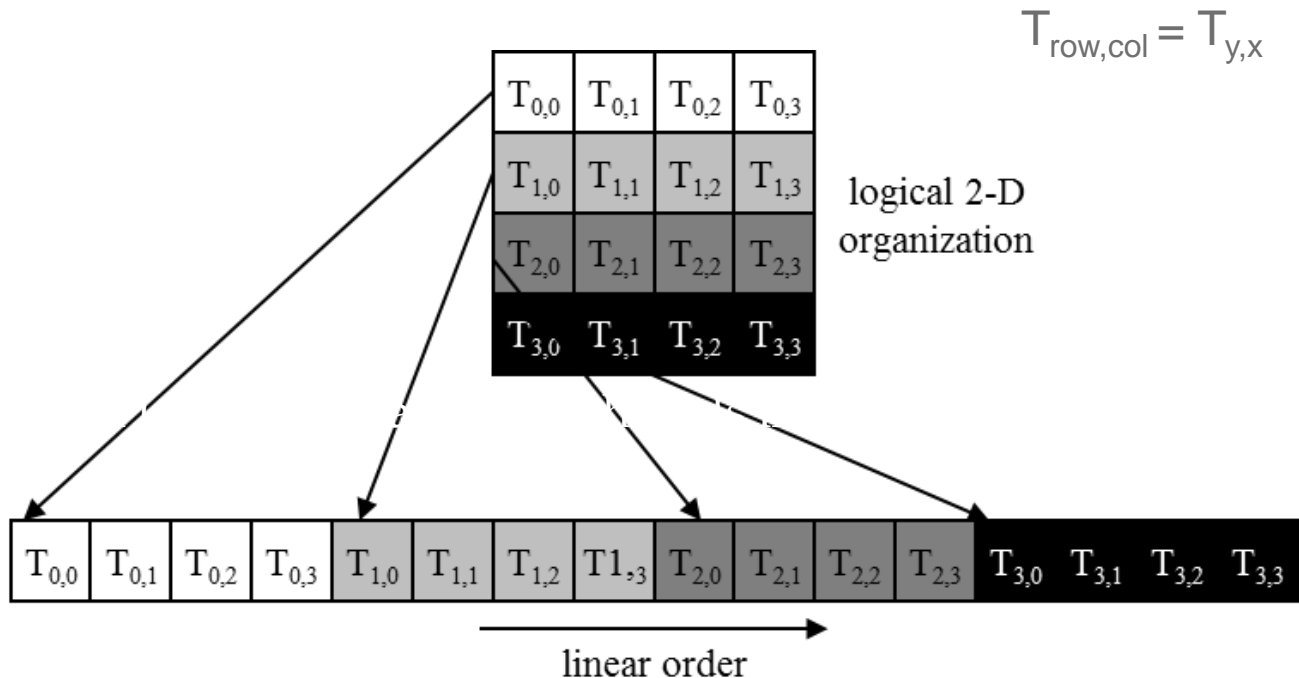
Warps as Scheduling Units



- Each block is divided into 32-thread warps
 - An implementation technique, not part of the CUDA programming model
 - Warps are scheduling units in SM
 - Threads in a warp execute in Single Instruction Multiple Data (SIMD) manner
 - The number of threads in a warp may vary in future generations

Warps in Multi-dimensional Thread Blocks

- The thread blocks are first linearized into 1D in row major order
 - In x-dimension first, y-dimension next, and z-dimension last

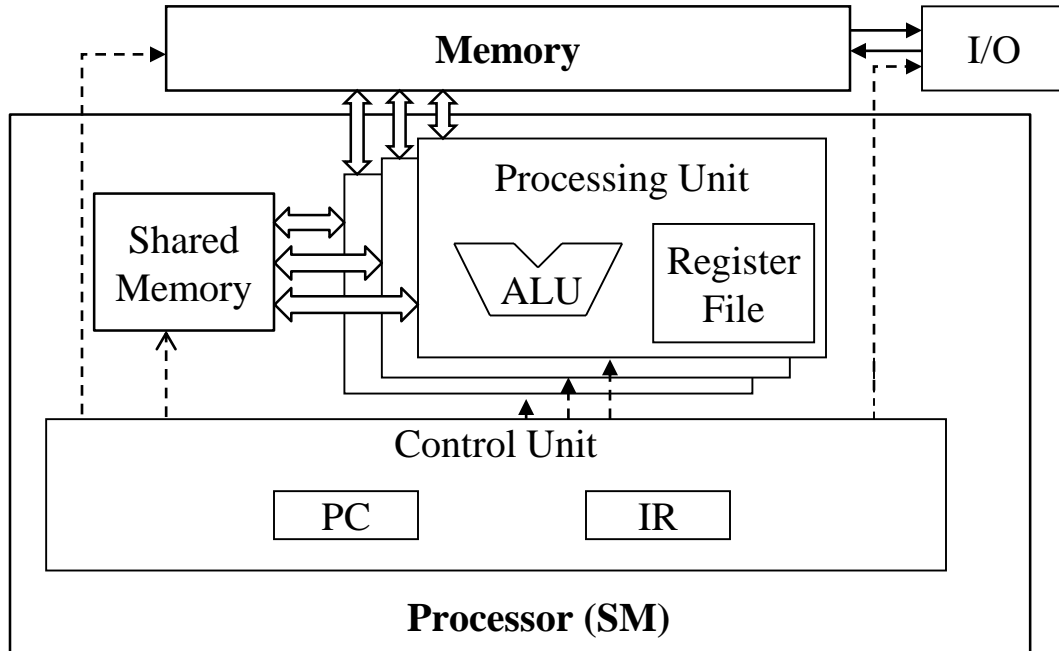


Blocks are partitioned after linearization

- Linearized thread blocks are partitioned
 - Thread indices within a warp are consecutive and increasing
 - Warp 0 starts with Thread 0
- Partitioning scheme is consistent across devices
 - Thus you can use this knowledge in control flow
 - However, the exact size of warps may change from generation to generation
- DO NOT rely on any ordering within or between warps
 - If there are any dependencies between threads, you must `__syncthreads()` to get correct results (more later).

SMs are SIMD Processors

- Control unit for instruction fetch, decode, and control is shared among multiple processing units
 - Control overhead is minimized (Module 1)



SIMD Execution Among Threads in a Warp

- All threads in a warp must execute the same instruction at any point in time
- This works efficiently if all threads follow the same control flow path
 - All if-then-else statements make the same decision
 - All loops iterate the same number of times

Control Divergence

- Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
 - Some take the then-path and others take the else-path of an if-statement
 - Some threads take different number of loop iterations than others
- The execution of threads taking different paths are serialized in current GPUs
 - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
 - During the execution of each path, all threads taking that path will be executed in parallel
 - The number of different paths can be large when considering nested control flow statements

Control Divergence Examples

- Divergence can arise when branch or loop condition is a function of thread indices
- Example kernel statement with divergence:
 - `if (threadIdx.x > 2) { }`
 - This creates two different control paths for threads in a block
 - Decision granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
- Example without divergence:
 - `If (blockIdx.x > 2) { }`
 - Decision granularity is a multiple of blocks size; all threads in any given warp follow the same path

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum  $C = A + B$   
// Each thread performs one pair-wise addition  
  
__global__  
void vecAddKernel(float* A, float* B, float* C,  
    int n)  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
    if(i < n) C[i] = A[i] + B[i];  
}
```

Analysis for vector size of 1,000 elements

- Assume that block size is 256 threads
 - 8 warps in each block
- All threads in Blocks 0, 1, and 2 are within valid range
 - i values from 0 to 767
 - There are 24 warps in these three blocks, none will have control divergence
- Most warps in Block 3 will not control divergence
 - Threads in the warps 0-6 are all within valid range, thus no control divergence
- One warp in Block 3 will have control divergence
 - Threads with i values 992-999 will all be within valid range
 - Threads with i values of 1000-1023 will be outside valid range
- Effect of serialization on control divergence will be small
 - 1 out of 32 warps has control divergence
 - The impact on performance will likely be less than 3%



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



Module 5.2 – Thread Execution Efficiency

Performance Impact of Control Divergence

Objective

- To learn to analyze the performance impact of control divergence
 - Boundary condition checking
 - Control divergence is data-dependent

Performance Impact of Control Divergence

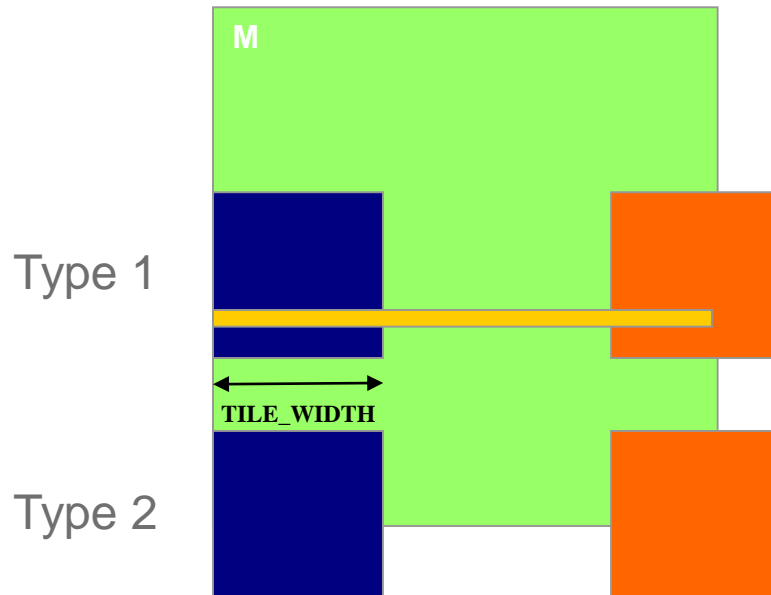
- Boundary condition checks are vital for complete functionality and robustness of parallel code
 - The tiled matrix multiplication kernel has many boundary condition checks
 - The concern is that these checks may cause significant performance degradation
 - For example, see the tile loading code below:

```
if(Row < Width && t * TILE_WIDTH+tx < Width) {  
    ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];  
} else {  
    ds_M[ty][tx] = 0.0;  
}
```

```
if (p*TILE_WIDTH+ty < Width && Col < Width) {  
    ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];  
} else {  
    ds_N[ty][tx] = 0.0;  
}
```

Two types of blocks in loading M Tiles

- 1. Blocks whose tiles are all within valid range until the last phase.
- 2. Blocks whose tiles are partially outside the valid range all the way

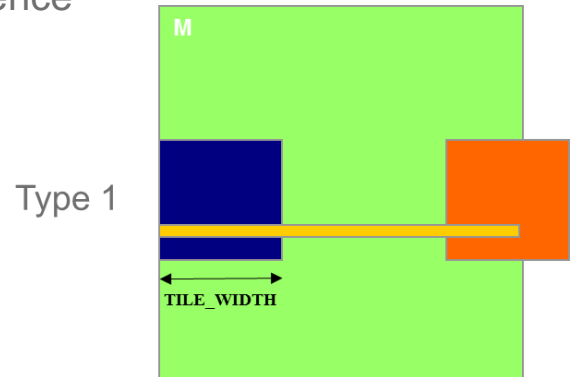


Analysis of Control Divergence Impact

- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps ($256/32$)
- Assume square matrices of 100x100
- Each thread will go through 7 phases (ceiling of $100/16$)
- There are 49 thread blocks (7 in each dimension)

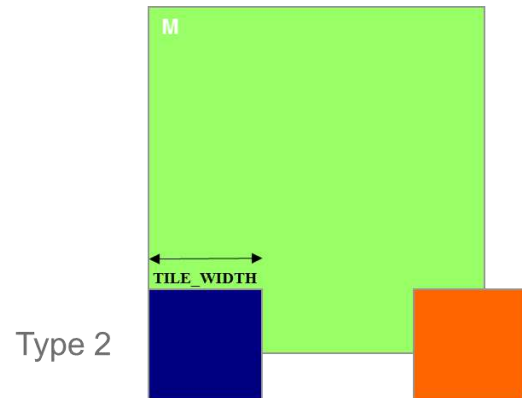
Control Divergence in Loading M Tiles

- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each warp will go through 7 phases (ceiling of $100/16$)
- There are 42 (6*7) Type 1 blocks, with a total of 336 (8*42) warps
- They all have 7 phases, so there are 2,352 (336*7) warp-phases
- The warps have control divergence only in their last phase
- 336 warp-phases have control divergence



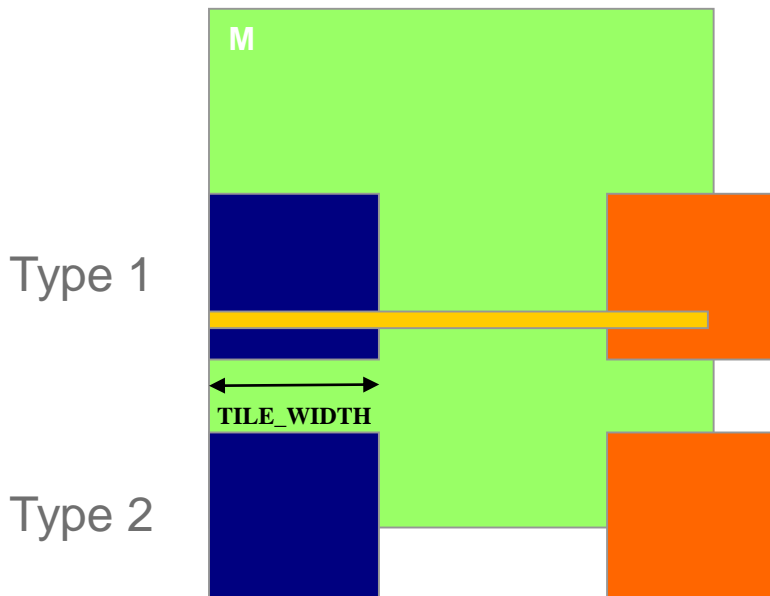
Control Divergence in Loading M Tiles (Type 2)

- Type 2: the 7 block assigned to load the bottom tiles, with a total of 56 (8×7) warps
- They all have 7 phases, so there are 392 (56×7) warp-phases
- The first 2 warps in each Type 2 block will stay within the valid range until the last phase
- The 6 remaining warps stay outside the valid range
- So, only 14 (2×7) warp-phases have control divergence



Overall Impact of Control Divergence

- Type 1 Blocks: 336 out of 2,352 warp-phases have control divergence
- Type 2 Blocks: 14 out of 392 warp-phases have control divergence
- The performance impact is expected to be less than 12% ($350/2,944$ or $(336+14)/(2352+14)$)



Additional Comments

- The calculation of impact of control divergence in loading N tiles is somewhat different and is left as an exercise
- The estimated performance impact is data dependent.
 - For larger matrices, the impact will be significantly smaller
- In general, the impact of control divergence for boundary condition checking for large input data sets should be insignificant
 - One should not hesitate to use boundary checks to ensure full functionality
- The fact that a kernel is full of control flow constructs does not mean that there will be heavy occurrence of control divergence
- We will cover some algorithm patterns that naturally incur control divergence (such as parallel reduction) in the Parallel Algorithm Patterns modules



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



Module 6.1 – Memory Access Performance

DRAM Bandwidth

Objective

- To learn that memory bandwidth is a first-order performance factor in a massively parallel processor
 - DRAM bursts, banks, and channels
 - All concepts are also applicable to modern multicore processors

Global Memory (DRAM) Bandwidth

– Ideal

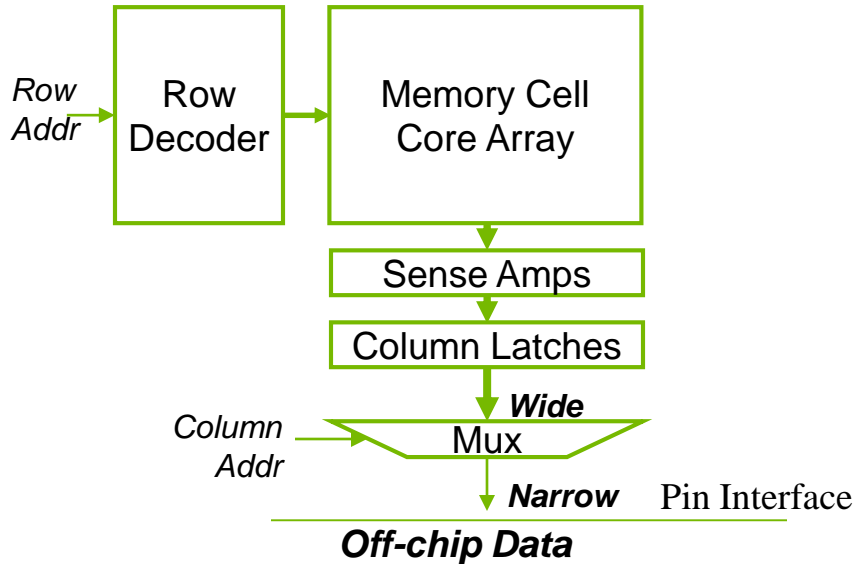


– Reality

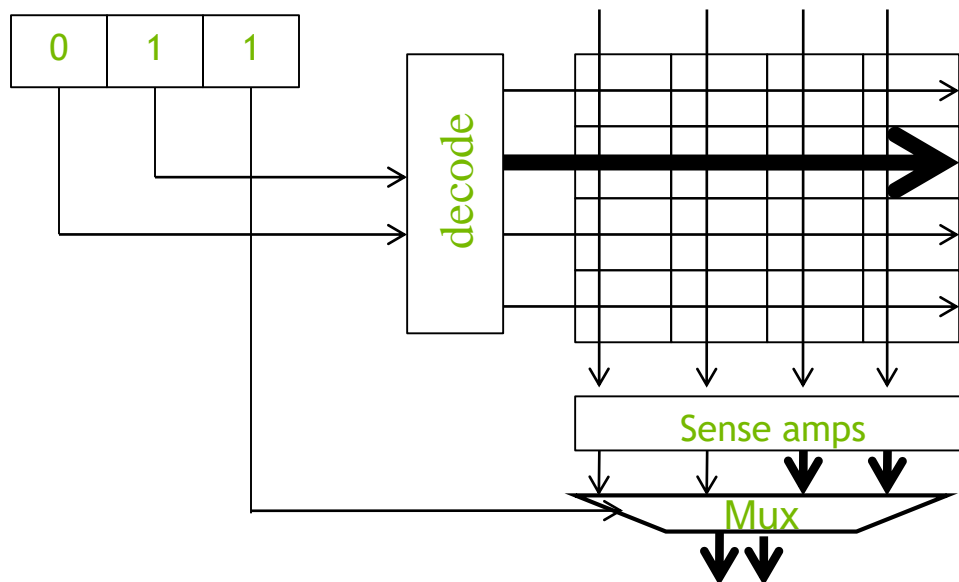


DRAM Core Array Organization

- Each DRAM core array has about 16M bits
- Each bit is stored in a tiny capacitor made of one transistor

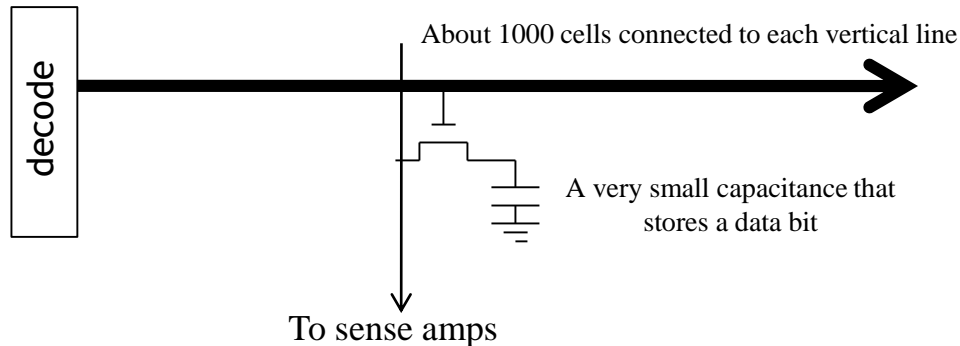


A very small (8x2-bit) DRAM Core Array



DRAM Core Arrays are Slow

- Reading from a cell in the core array is a very slow process
 - DDR: Core speed = $\frac{1}{2}$ interface speed
 - DDR2/GDDR3: Core speed = $\frac{1}{4}$ interface speed
 - DDR3/GDDR4: Core speed = $\frac{1}{8}$ interface speed
 - ... likely to be worse in the future

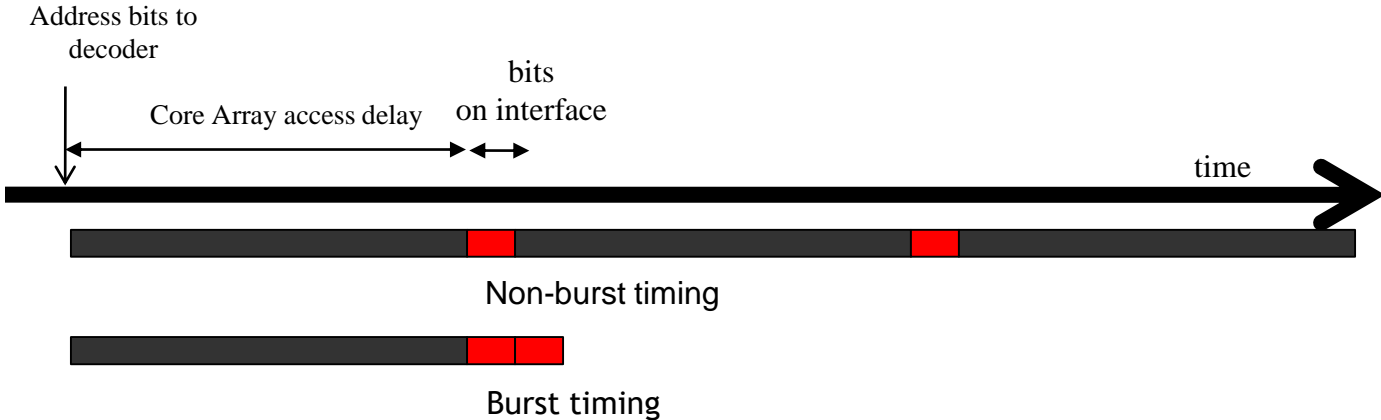


DRAM Bursting

- For DDR{2,3} SDRAM cores clocked at $1/N$ speed of the interface:
 - Load ($N \times$ interface width) of DRAM bits from the same row at once to an internal buffer, then transfer in N steps at interface speed
 - DDR3/GDDR4: buffer width = $8X$ interface width

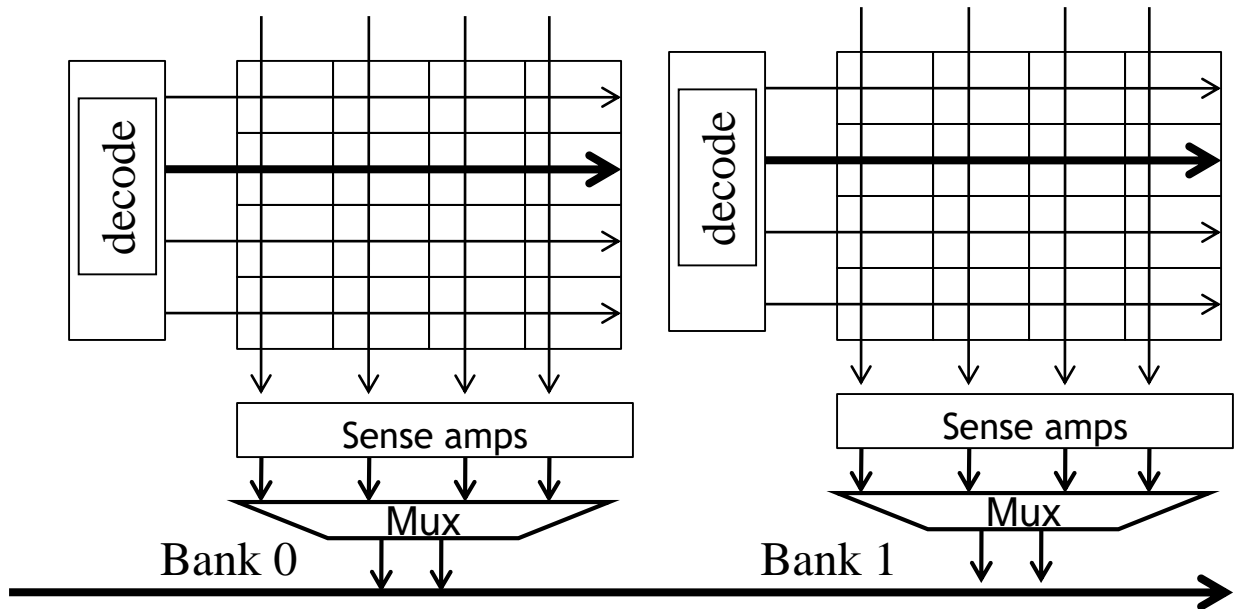


DRAM Bursting Timing Example



Modern DRAM systems are designed to always be accessed in burst mode. Burst bytes are transferred to the processor but discarded when accesses are not to sequential locations.

Multiple DRAM Banks



DRAM Bursting with Banking



Single-Bank burst timing, dead time on interface



Multi-Bank burst timing, reduced dead time

GPU off-chip memory subsystem

- NVIDIA RTX6000 GPU:
 - Peak global memory bandwidth = 672GB/s
- Global memory (GDDR6) interface @ 7GHz
 - 14 Gbps pin speed
 - For GDDR6 32-bit interface, we can sustain only about 56 GB/s
 - We need a lot more bandwidth (672 GB/s) – thus 12 memory channels



GPU Teaching Kit



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



Lecture 6.2 – Performance Considerations

Memory Coalescing in CUDA

Objective

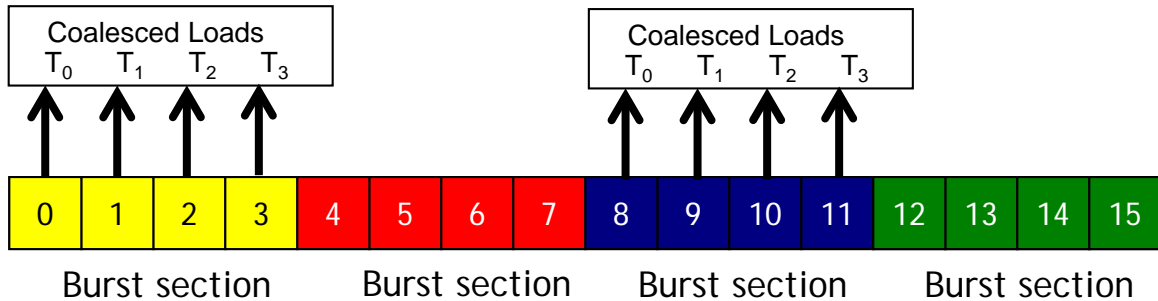
- To learn that memory coalescing is important for effectively utilizing memory bandwidth in CUDA
 - Its origin in DRAM burst
 - Checking if a CUDA memory access is coalesced
 - Techniques for improving memory coalescing in CUDA code

DRAM Burst – A System View



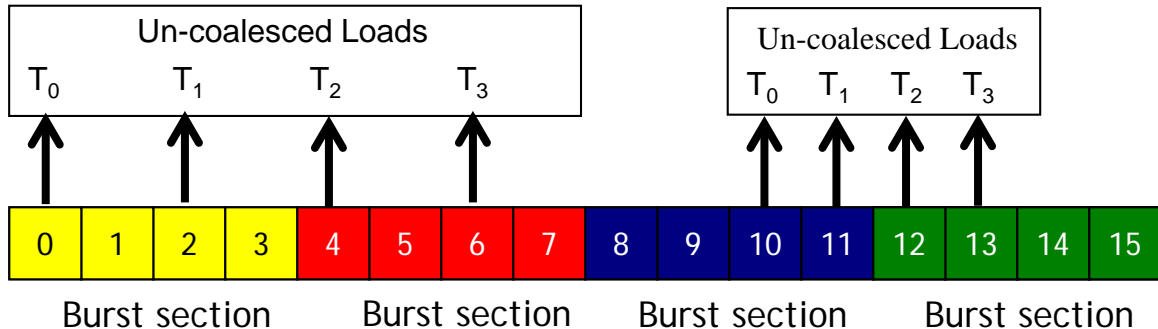
- Each address space is partitioned into burst sections
 - Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
 - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

Memory Coalescing



- When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.

Un-coalesced Accesses

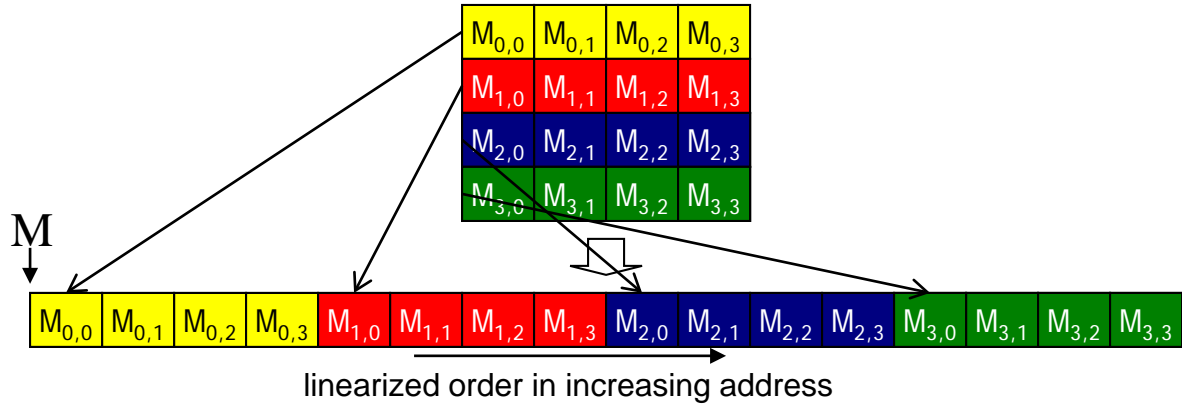


- When the accessed locations spread across burst section boundaries:
 - Coalescing fails
 - Multiple DRAM requests are made
 - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

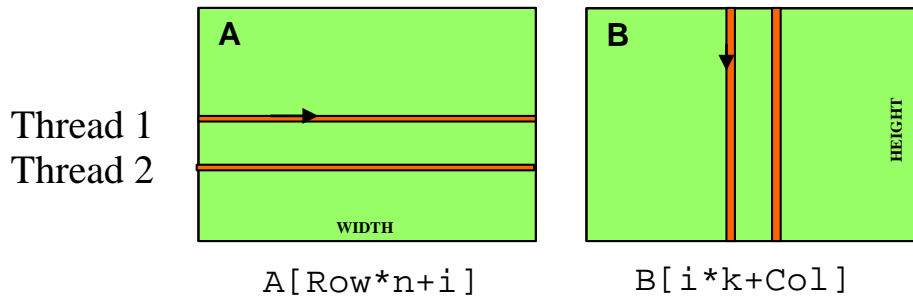
How to judge if an access is coalesced?

- Accesses in a warp are to consecutive locations if the index in an array access is in the form of
 - $A[(\text{expression with terms independent of threadIdx.x}) + \text{threadIdx.x}]$;

A 2D C Array in Linear Memory Space



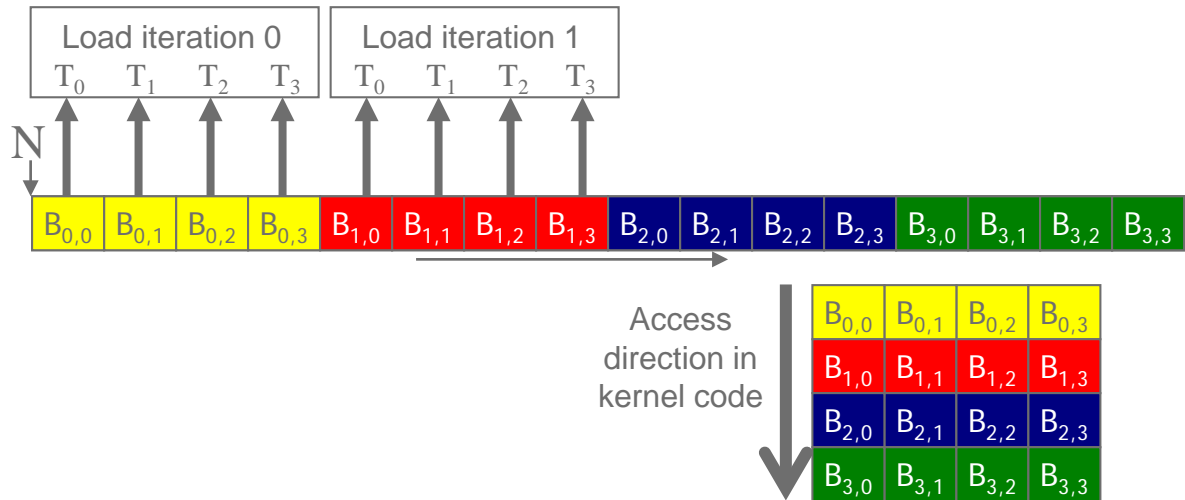
Two Access Patterns of Basic Matrix Multiplication



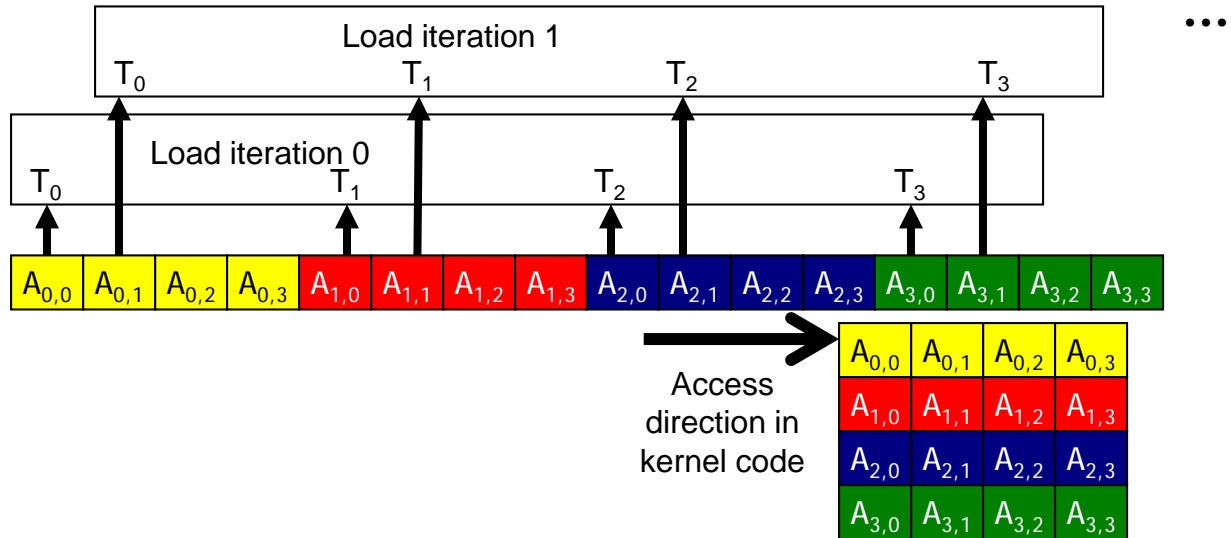
i is the loop counter in the inner product loop of the kernel code

A is $m \times n$, B is $n \times k$
 $\text{Col} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$

B accesses are coalesced



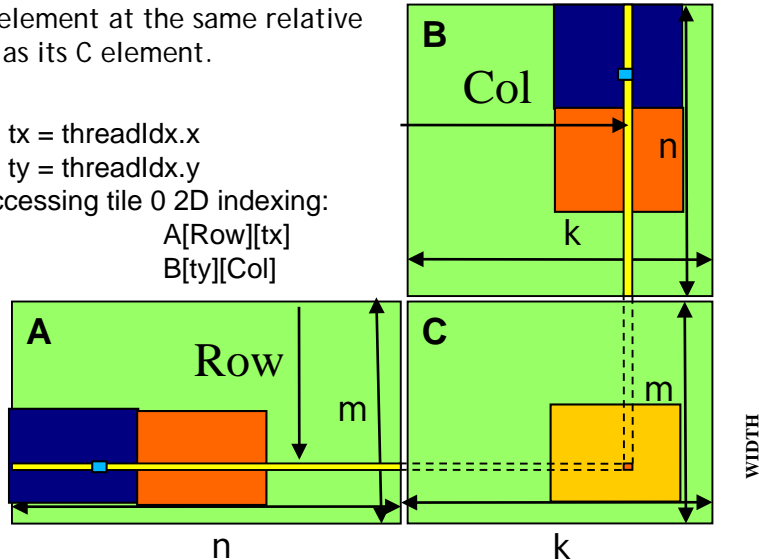
A Accesses are Not Coalesced



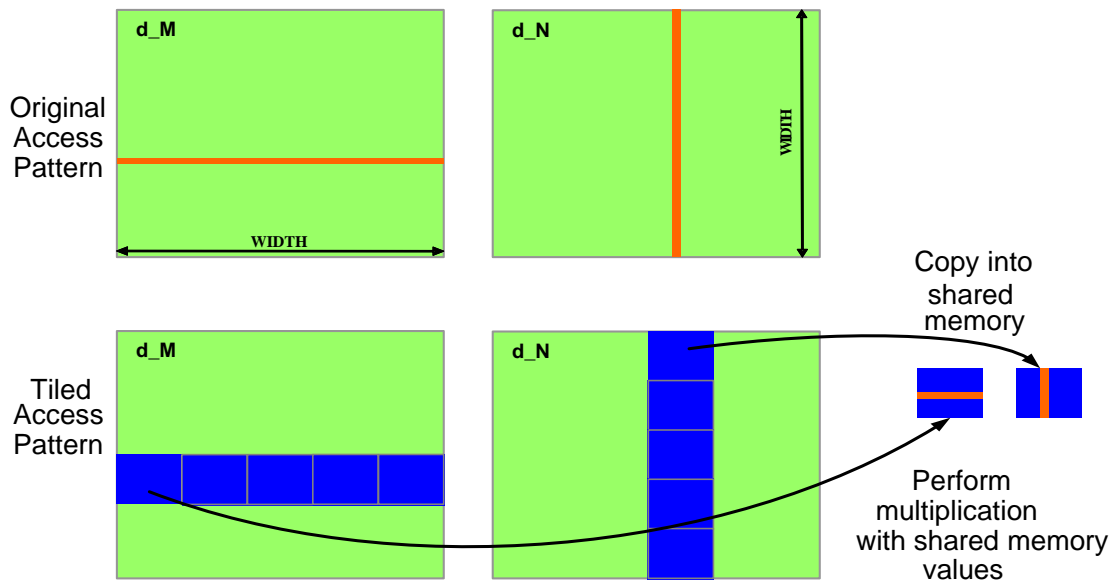
Loading an Input Tile

Have each thread load an A element and a B element at the same relative position as its C element.

```
int tx = threadIdx.x  
int ty = threadIdx.y  
Accessing tile 0 2D indexing:  
A[Row][tx]  
B[ty][Col]
```



Corner Turning





GPU Teaching Kit



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



Module 7.1 – Parallel Computation Patterns (Histogram)

Histogramming

Objective

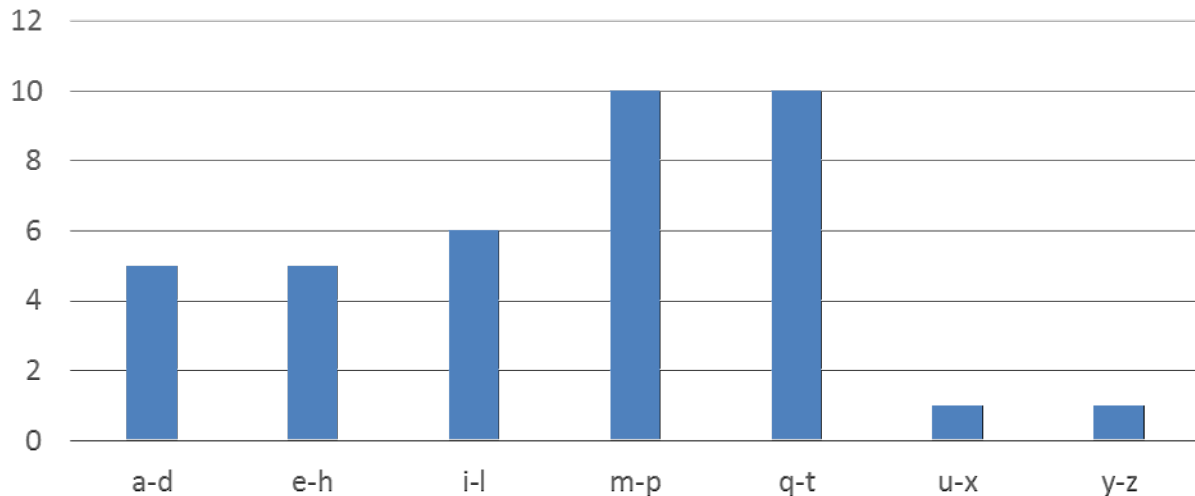
- To learn the parallel histogram computation pattern
 - An important, useful computation
 - Very different from all the patterns we have covered so far in terms of output behavior of each thread
 - A good starting point for understanding output interference in parallel computation

Histogram

- A method for extracting notable features and patterns from large data sets
 - Feature extraction for object recognition in images
 - Fraud detection in credit card transactions
 - Correlating heavenly object movements in astrophysics
 - ...
- Basic histograms - for each element in the data set, use the value to identify a “bin counter” to increment

A Text Histogram Example

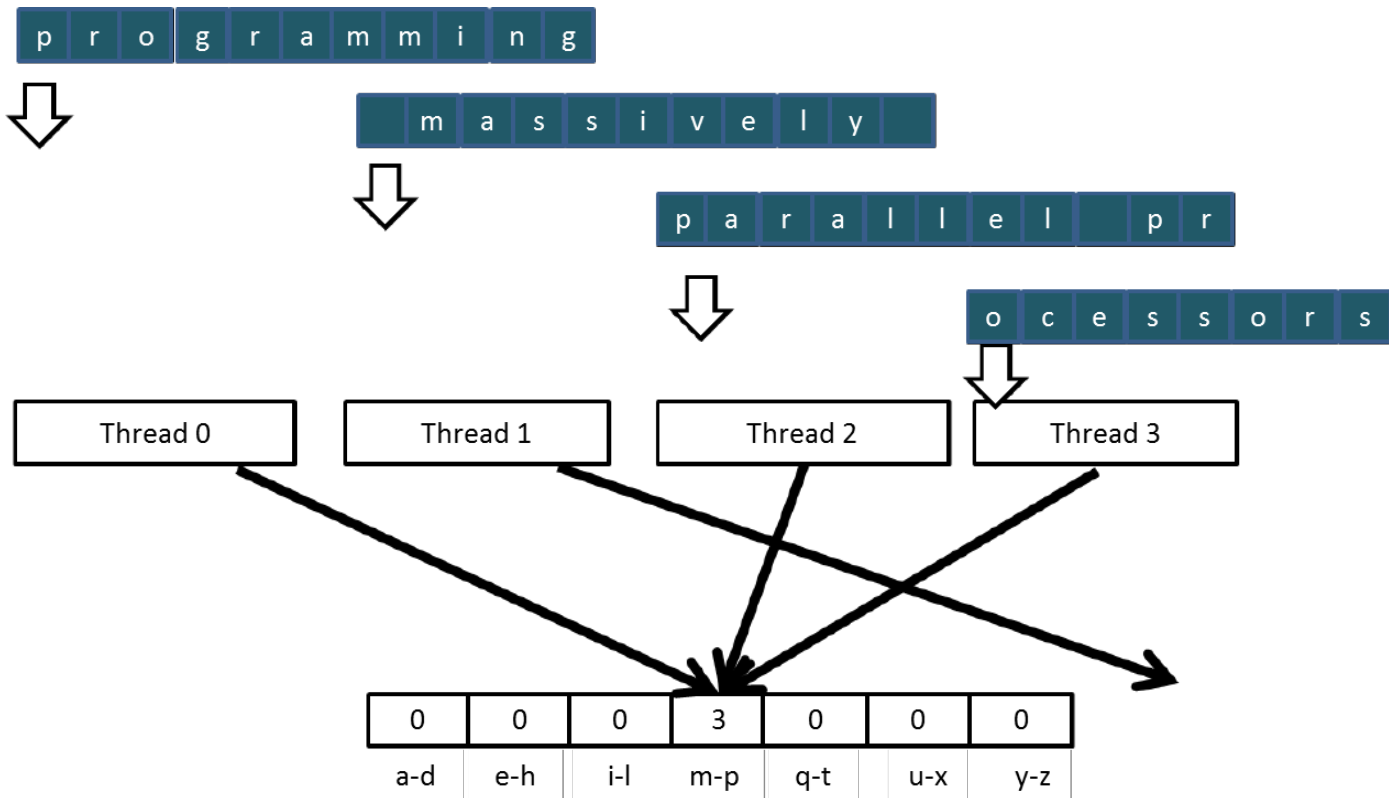
- Define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, n-p, ...
- For each character in an input string, increment the appropriate bin counter.
- In the phrase “Programming Massively Parallel Processors” the output histogram is shown below:



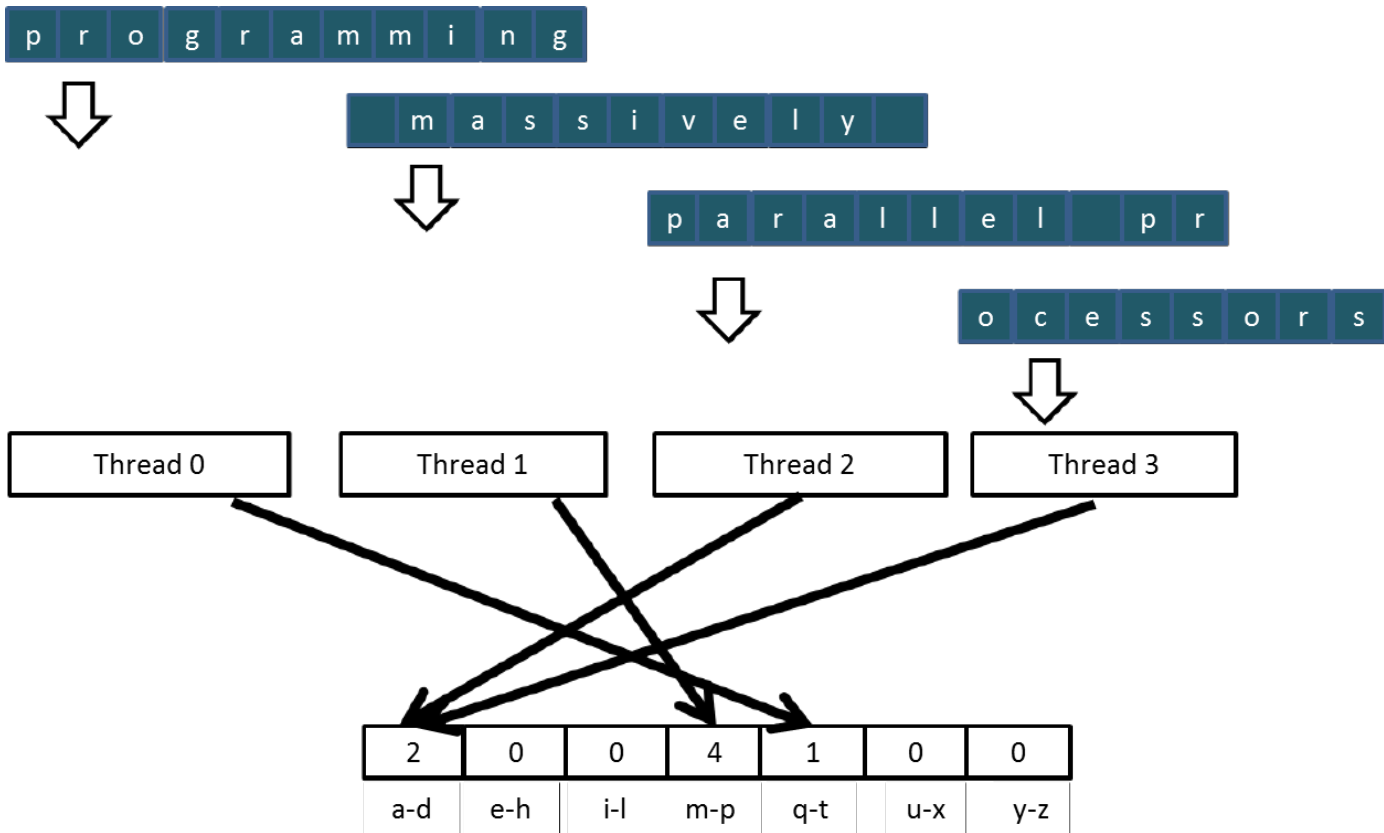
A simple parallel histogram algorithm

- Partition the input into sections
- Have each thread to take a section of the input
- Each thread iterates through its section.
- For each letter, increment the appropriate bin counter

Sectioned Partitioning (Iteration #1)



Sectioned Partitioning (Iteration #2)



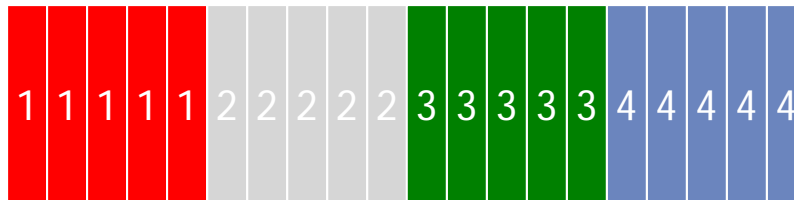
Input Partitioning Affects Memory Access Efficiency

- Sectioned partitioning results in poor memory access efficiency
 - Adjacent threads do not access adjacent memory locations
 - Accesses are not coalesced
 - DRAM bandwidth is poorly utilized

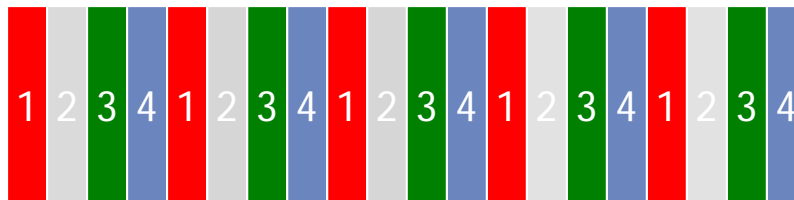


Input Partitioning Affects Memory Access Efficiency

- Sectioned partitioning results in poor memory access efficiency
 - Adjacent threads do not access adjacent memory locations
 - Accesses are not coalesced
 - DRAM bandwidth is poorly utilized

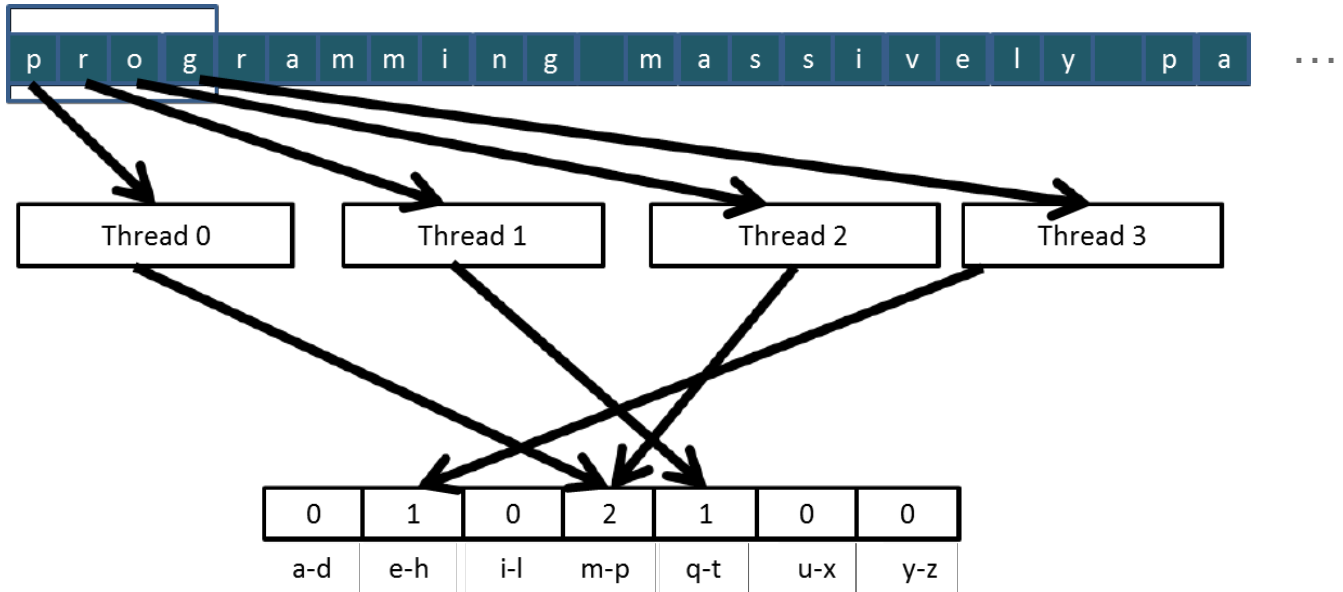


- Change to interleaved partitioning
 - All threads process a contiguous section of elements
 - They all move to the next section and repeat
 - The memory accesses are coalesced

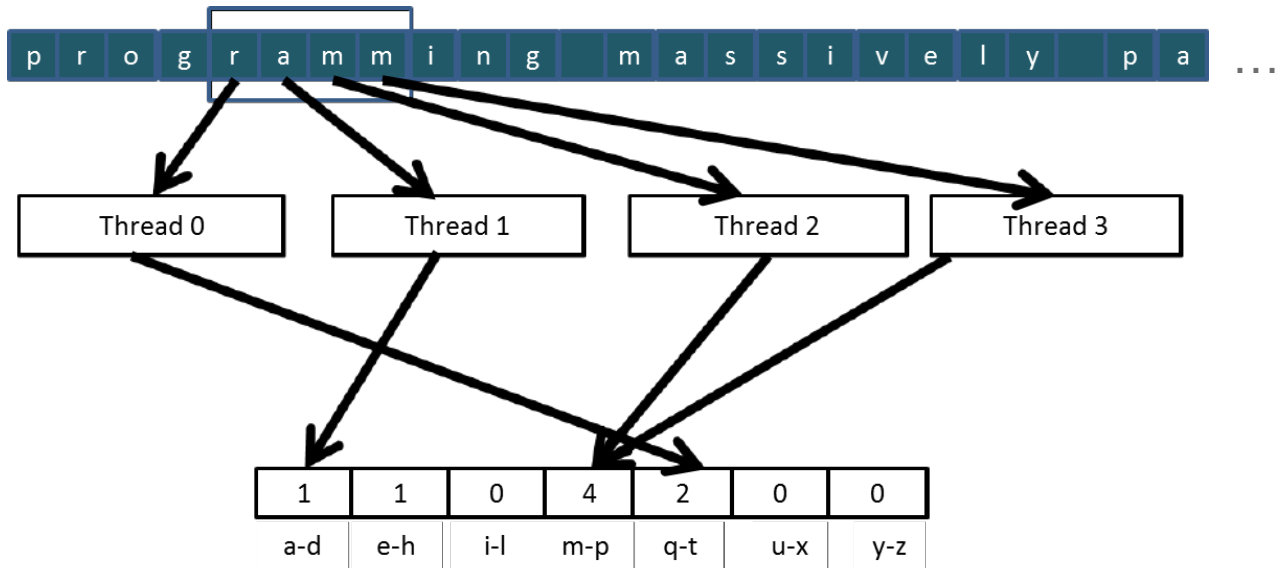


Interleaved Partitioning of Input

- For coalescing and better memory access performance



Interleaved Partitioning (Iteration 2)





GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



Module 7.2 – Parallel Computation Patterns (Histogram)

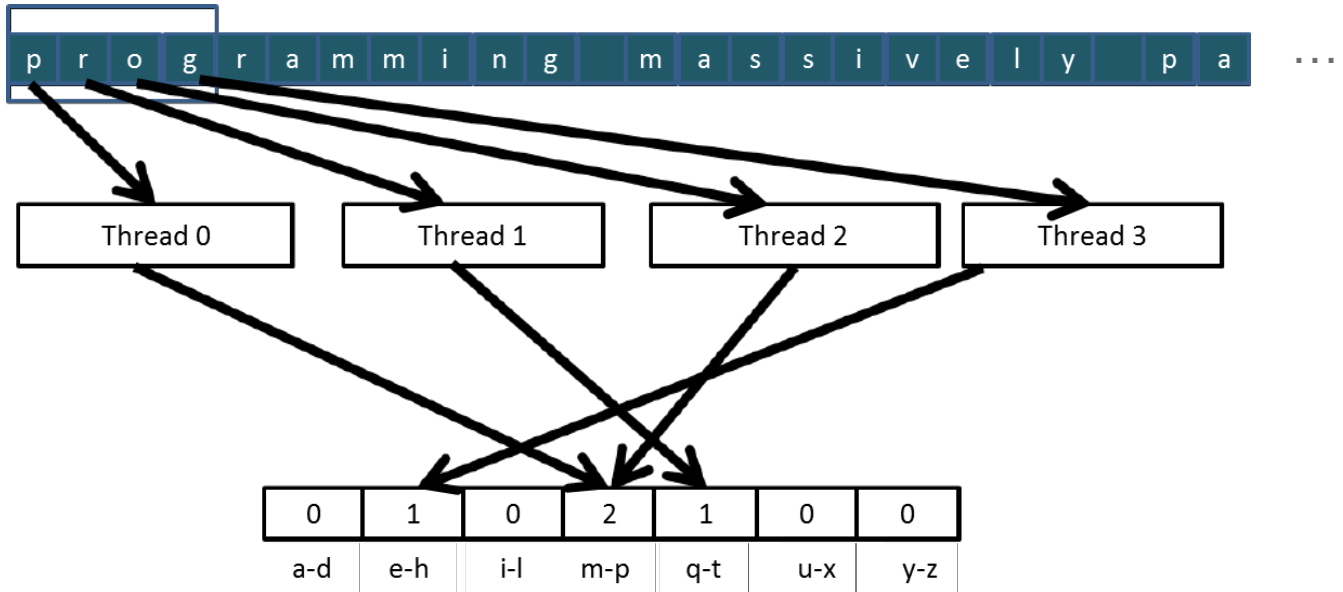
Introduction to Data Races

Objective

- To understand data races in parallel computing
 - Data races can occur when performing read-modify-write operations
 - Data races can cause errors that are hard to reproduce
 - Atomic operations are designed to eliminate such data races

Read-modify-write in the Text Histogram Example

- For coalescing and better memory access performance



Read-Modify-Write Used in Collaboration Patterns

- For example, multiple bank tellers count the total amount of cash in the safe
- Each grab a pile and count
- Have a central display of the running total
- Whenever someone finishes counting a pile, read the current running total (read) and add the subtotal of the pile to the running total (modify-write)
- A bad outcome
 - Some of the piles were not accounted for in the final total

A Common Parallel Service Pattern

- For example, multiple customer service agents serving waiting customers
- The system maintains two numbers,
 - the number to be given to the next incoming customer (I)
 - the number for the customer to be served next (S)
- The system gives each incoming customer a number (read I) and increments the number to be given to the next customer by 1 (modify-write I)
- A central display shows the number for the customer to be served next
- When an agent becomes available, he/she calls the number (read S) and increments the display number by 1 (modify-write S)
- Bad outcomes
 - Multiple customers receive the same number, only one of them receives service
 - Multiple agents serve the same number

A Common Arbitration Pattern

- For example, multiple customers booking airline tickets in parallel
- Each
 - Brings up a flight seat map (read)
 - Decides on a seat
 - Updates the seat map and marks the selected seat as taken (modify-write)
- A bad outcome
 - Multiple passengers ended up booking the same seat

Data Race in Parallel Thread Execution

thread1: $\text{Old} \leftarrow \text{Mem}[x]$
 $\text{New} \leftarrow \text{Old} + 1$
 $\text{Mem}[x] \leftarrow \text{New}$

thread2: $\text{Old} \leftarrow \text{Mem}[x]$
 $\text{New} \leftarrow \text{Old} + 1$
 $\text{Mem}[x] \leftarrow \text{New}$

Old and New are per-thread register variables.

Question 1: If $\text{Mem}[x]$ was initially 0, what would the value of $\text{Mem}[x]$ be after threads 1 and 2 have completed?

Question 2: What does each thread get in their Old variable?

Unfortunately, the answers may vary according to the relative execution timing between the two threads, which is referred to as a **data race**.

Timing Scenario #1

Time	Thread 1	Thread 2
1	(0) Old \leftarrow Mem[x]	
2	(1) New \leftarrow Old + 1	
3	(1) Mem[x] \leftarrow New	
4		(1) Old \leftarrow Mem[x]
5		(2) New \leftarrow Old + 1
6		(2) Mem[x] \leftarrow New

- Thread 1 Old = 0
- Thread 2 Old = 1
- Mem[x] = 2 after the sequence

Timing Scenario #2

Time	Thread 1	Thread 2
1		(0) Old \leftarrow Mem[x]
2		(1) New \leftarrow Old + 1
3		(1) Mem[x] \leftarrow New
4	(1) Old \leftarrow Mem[x]	
5	(2) New \leftarrow Old + 1	
6	(2) Mem[x] \leftarrow New	

- Thread 1 Old = 1
- Thread 2 Old = 0
- Mem[x] = 2 after the sequence

Timing Scenario #3

Time	Thread 1	Thread 2
1	(0) Old \leftarrow Mem[x]	
2	(1) New \leftarrow Old + 1	
3		(0) Old \leftarrow Mem[x]
4	(1) Mem[x] \leftarrow New	
5		(1) New \leftarrow Old + 1
6		(1) Mem[x] \leftarrow New

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Timing Scenario #4

Time	Thread 1	Thread 2
1		(0) Old \leftarrow Mem[x]
2		(1) New \leftarrow Old + 1
3	(0) Old \leftarrow Mem[x]	
4		(1) Mem[x] \leftarrow New
5	(1) New \leftarrow Old + 1	
6	(1) Mem[x] \leftarrow New	

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Purpose of Atomic Operations – To Ensure Good Outcomes

thread1: $\text{Old} \leftarrow \text{Mem}[x]$
 $\text{New} \leftarrow \text{Old} + 1$
 $\text{Mem}[x] \leftarrow \text{New}$

thread2: $\text{Old} \leftarrow \text{Mem}[x]$
 $\text{New} \leftarrow \text{Old} + 1$
 $\text{Mem}[x] \leftarrow \text{New}$

Or

thread1: $\text{Old} \leftarrow \text{Mem}[x]$
 $\text{New} \leftarrow \text{Old} + 1$
 $\text{Mem}[x] \leftarrow \text{New}$

thread2: $\text{Old} \leftarrow \text{Mem}[x]$
 $\text{New} \leftarrow \text{Old} + 1$
 $\text{Mem}[x] \leftarrow \text{New}$



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



Module 7.3 – Parallel Computation Patterns (Histogram)

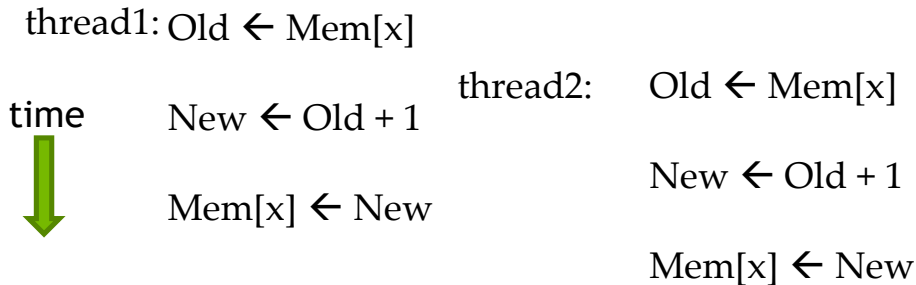
Atomic Operations in CUDA

Objective

- To learn to use atomic operations in parallel programming
 - Atomic operation concepts
 - Types of atomic operations in CUDA
 - Intrinsic functions
 - A basic histogram kernel

Data Race Without Atomic Operations

Mem[x] initialized to 0



- Both threads receive 0 in Old
- Mem[x] becomes 1

Key Concepts of Atomic Operations

- A read-modify-write operation performed by a single hardware instruction on a memory location *address*
 - Read the old value, calculate a new value, and write the new value to the location
- The hardware ensures that no other threads can perform another read-modify-write operation on the same location until the current atomic operation is complete
 - Any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue
 - All threads perform their atomic operations **serially** on the same location

Atomic Arithmetic Operations in CUDA

- Performed by calling functions that are translated into single instructions (a.k.a. *intrinsic functions* or *intrinsics*)
 - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
 - Read CUDA C programming Guide for details

- Atomic Add

```
int atomicAdd(int* address, int val);
```

- reads the 32-bit word **old** from the location pointed to by **address** in global or shared memory, computes (**old + val**), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

More Atomic Adds in CUDA

- Unsigned 32-bit integer atomic add

```
unsigned int atomicAdd(unsigned int* address,  
    unsigned int val);
```

- Unsigned 64-bit integer atomic add

```
unsigned long long int atomicAdd(unsigned long long  
    int* address, unsigned long long int val);
```

- Single-precision floating-point atomic add (Compute capability 2.x+)

```
float atomicAdd(float* address, float val);
```

- Double-precision floating-point atomic add (Compute capability 6.x+)

```
double atomicAdd(double* address, double val);
```

- 16-bit floating-point atomic add (Compute capability 7.x+)

```
__half atomicAdd(__half* address, __half val);
```

A Basic Text Histogram Kernel

- The kernel receives a pointer to the input buffer of byte values
- Each thread process the input in a strided pattern

```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;

    // All threads handle blockDim.x * gridDim.x
    // consecutive elements
    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1);
        i += stride;
    }
}
```

A Basic Histogram Kernel (cont.)

- The kernel receives a pointer to the input buffer of byte values
- Each thread process the input in a strided pattern

```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;

    // All threads handle blockDim.x * gridDim.x
    // consecutive elements
    while (i < size) {
        int alphabet_position = buffer[i] - "a";
        if (alphabet_position >= 0 && alphabet_position < 26)
            atomicAdd(&(histo[alphabet_position/4]), 1);
        i += stride;
    }
}
```



GPU Teaching Kit

Accelerated Computing



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

1. **How many warps can an SM hold (hardware/occupancy limit)?**
2. **How many active warps do you need to hide memory/latency (performance/throughput requirement)?**

- **Warp size** (almost all NVIDIA GPUs): 32 threads/warp.
- **Active warps per SM (hardware max)** depends on the GPU architecture. You don't guess this — the GPU datasheet gives it. Typical modern values: SM can support dozens of warps (e.g., 64 warps = 2048 threads if warp size = 32).
- **Occupancy** = active threads / max threads per SM. You can compute active warps from resource limits (threads, registers, shared memory).

To compute active warps per SM:

1. Find the max blocks per SM limited by each resource:
 - by threads: $\text{blocks_threads_limit} = \text{floor}(\text{max_threads_per_SM} / \text{threads_per_block})$
 - by registers: $\text{blocks_regs_limit} = \text{floor}(\text{total_regs_per_SM} / (\text{regs_per_thread} * \text{threads_per_block}))$
 - by shared mem: $\text{blocks_shmem_limit} = \text{floor}(\text{shared_mem_per_SM} / \text{shared_mem_per_block})$
2. $\text{blocks_per_SM} = \min(\text{blocks_threads_limit}, \text{blocks_regs_limit}, \text{blocks_shmem_limit}, \text{hardware_block_limit})$
3. **Active warps per SM** = $\text{blocks_per_SM} * (\text{threads_per_block} / \text{warp_size})$
4. **Occupancy** = $(\text{active warps per SM} * \text{warp_size}) / \text{max_threads_per_SM}$.

Example:

GPU limits: $\text{max_threads_per_SM} = 2048$,
 $\text{total_regs_per_SM} = 65536$, $\text{shared_mem_per_SM} = 64 \text{ KB}$.

Kernel: $\text{threads_per_block} = 256$, $\text{regs_per_thread} = 32$, $\text{shared_mem_per_block} = 16 \text{ KB}$.

- threads limit: $2048 / 256 = 8 \text{ blocks}$
- regs per block $= 256 * 32 = 8192 \rightarrow$ regs limit: $65536 / 8192 = 8 \text{ blocks}$
- shmem limit: $64\text{KB} / 16\text{KB} = 4 \text{ blocks}$

So $\text{blocks_per_SM} = \min(8, 8, 4) = 4 \text{ blocks}$.

Active warps per SM $= 4 * (256 / 32) = 4 * 8 = 32$
warps

Active threads $= 32 * 32 = 1024 \rightarrow$ occupancy $= 1024 / 2048 = 50\%$.

2) Warps needed to *hide latency* (performance / throughput)

This is a modeling estimate: how many *independent* warps you need so the SM can keep doing useful work while some warps are waiting (e.g., for memory).

A simple useful formula:

- Let L = memory latency in cycles (e.g., 300 cycles).
- Let M = average number of **useful** cycles a single warp executes between its memory requests (i.e., instructions between memory ops, in cycles). If a warp issues a memory access every M cycles, that warp will be stalled for L cycles on that memory access.
- Then **additional** warps required to cover that stall $\approx \text{ceil}(L / M)$. Total warps needed $\approx 1 + \text{ceil}(L / M)$ (the +1 is the original warp that issued the mem op).

So:

$\text{required_warps_to_hide} \approx 1 + \text{ceil}(L / M)$

Worked numeric example:

- Memory latency $L = 300$ cycles.
- A warp issues a memory access every $M = 50$ cycles (on average).

$\text{ceil}(300 / 50) = 6 \rightarrow \text{required_warps} \approx 1 + 6 = 7$
warps active to hide that memory stall.

Note: That's *modeling*. Real systems need more warps because:

- Different warps may hit different latencies (L varies).
- Warps may contend for other resources (execution units).
- Instruction throughput per warp may be less than one useful instruction per cycle.
- Branch divergence reduces effective parallelism.

So in practice, GPU architects target dozens of warps per SM

Tricks :

- Identify hardware limits: `max_threads_per_SM`, `total_regs_per_SM`, `shared_mem_per_SM`, `warp_size` (usually 32), `max_blocks_per_SM`.
- Compute blocks-per-SM limited by each resource and take minimum.
- Convert blocks \rightarrow warps: $\text{warps} = \text{blocks} * (\text{threads_per_block} / \text{warp_size})$.
- For latency-hiding questions, compute $\text{required_warps} \approx 1 + \text{ceil}(L / M)$ and compare to the active warps

Q1 . For a vector addition, assume that the vector length is 8000, each thread calculates one output element, and the thread block size is 1024 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?

Blocks needed = $\text{ceil}(8000 / 1024)$.

Compute:

- $1024 \times 7 = 7168$
- $1024 \times 8 = 8192 \rightarrow \text{so } \text{ceil}(8000/1024) = 8$ blocks.

Threads in the grid = $8 \text{ blocks} \times 1024 \text{ threads/block} = 8192 \text{ threads}$.

(So 8192 threads are launched; $8192 - 8000 = 192$ threads will be idle / do no useful work.)

Q2. If we want to copy 3000 bytes of data from host array h_A (h_A is a pointer to element 0 of the source array) to device array d_A (d_A is a pointer to element 0 of the destination array), what would be an appropriate API call for this data copy in CUDA?

`cudaMemcpy(d_A, h_A, 3000, cudaMemcpyHostToDevice);`

1. **Destination pointer** \rightarrow d_A (device memory).
2. **Source pointer** \rightarrow h_A (host memory).
3. **Size in bytes** \rightarrow 3000.
4. **Transfer direction** \rightarrow cudaMemcpyHostToDevice.

If `cudaMemcpy` returns `cudaSuccess`, the copy succeeded.

Q3. If the SM of a CUDA device can take up to 1536 threads and up to 4 thread blocks. Which of the following block configuration would result in the largest number of threads in the SM?

Rule:

Active threads per SM = $\min(\text{threads_per_block} \times \text{blocks_per_SM}, \text{max_threads_per_SM})$

where $\text{blocks_per_SM} \leq \text{max_blocks_per_SM}$.

Threads per block	Max blocks possible (limited by 4 blocks/SM & 1536 threads)	Total threads
256	$\min(4, \text{floor}(1536 / 256) = 6) \rightarrow 4$ blocks	$256 \times 4 =$ 1024
384	$\min(4, \text{floor}(1536 / 384) = 4) \rightarrow 4$ blocks	$384 \times 4 =$ 1536 ✓
512	$\min(4, \text{floor}(1536 / 512) = 3) \rightarrow 3$ blocks	$512 \times 3 =$ 1536 ✓
768	$\min(4, \text{floor}(1536 / 768) = 2) \rightarrow 2$ blocks	$768 \times 2 =$ 1536 ✓
1024	$\min(4, \text{floor}(1536 / 1024) = 1) \rightarrow 1$ block	1024

Possible

- 384 threads/block (4 blocks/SM),
- **or** 512 threads/block (3 blocks/SM),
- **or** 768 threads/block (2 blocks/SM).

Q4. For a vector addition, assume that the vector length is 2000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid?

Find number of blocks needed

$$\text{blocks} = \lceil \frac{\text{vector length}}{\text{threads per block}} \rceil$$

$$\text{blocks} = \lceil \frac{2000}{512} \rceil$$

- $512 \times 3 = 1536$

- $512 \times 4 = 2048$

So ceil =4 block

threads in grid=blocks×threads per block

so $4 \times 512 = 2048$ threads

Q5. each block executed 32T warps if we assign 3B to SM and each block has 256T. how many warps are there in sm ? also solved utilized and unutilized

1) Warps per block

Warp size = 32 threads.

$$\text{warps per block} = \frac{256 \text{ threads}}{32 \text{ threads/warp}} = 8 \text{ warps}$$

2) Total warps in the SM

With 3 blocks per SM:

$$\text{total warps} = 3 \text{ blocks} \times 8 \text{ warps/block} = 24 \text{ warps}$$

$$\text{Total threads} = 3 \times 256 = 768 \text{ threads.}$$

3) Utilized vs. unutilized (assuming SM max = 1536 threads)

- Max warps per SM:

$$\frac{1536 \text{ threads}}{32 \text{ threads/warp}} = 48 \text{ warps}$$

- Utilized warps: 24
- Unutilized warps: $48 - 24 = 24$

Or in threads:

- Utilized threads = 768
- Unutilized threads = $1536 - 768 = 768$

- **Active warps in SM = 24**
- **Active threads = 768**
- **Utilized** = $24/48=50\%$
- **Unutilized = 50%** (24 warps / 768 threads idle)