GPU Teaching Kit

Accelerated Computing

Lecture 1.2 – Course Introduction

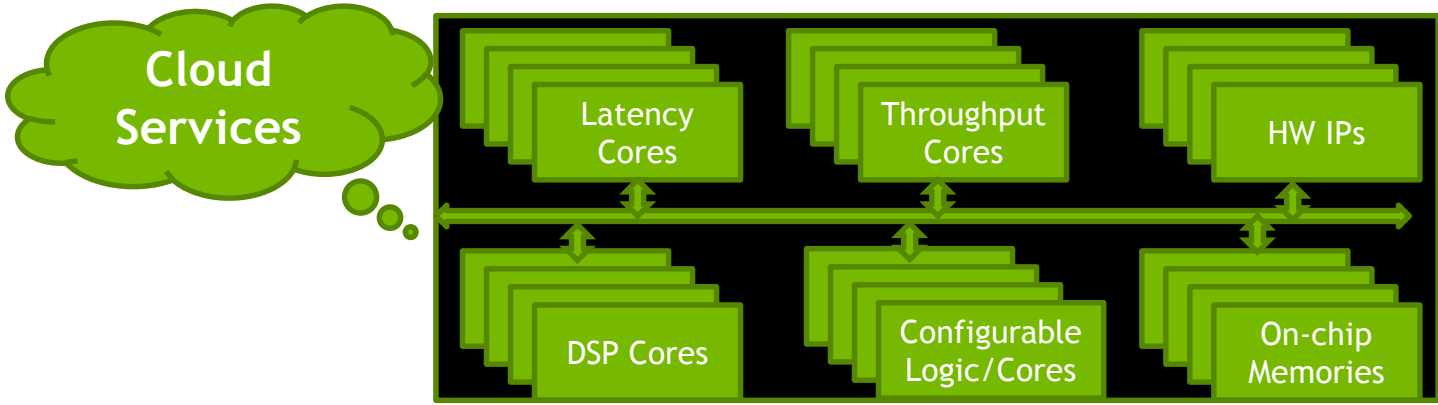
Introduction to Heterogeneous Parallel Computing

Objectives

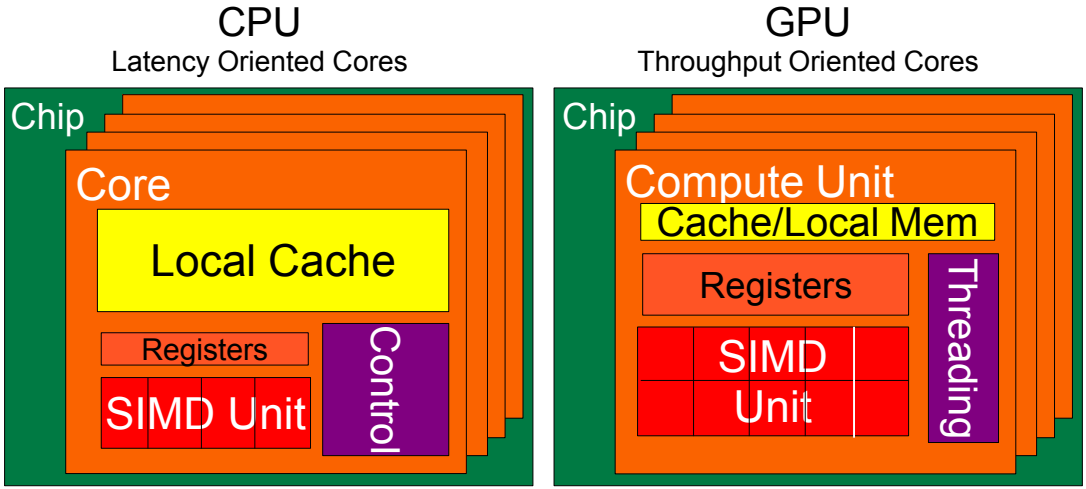
- To learn the major differences between latency devices (CPU cores) and throughput devices (GPU cores)
- To understand why winning applications increasingly use both types of devices

Heterogeneous Parallel Computing

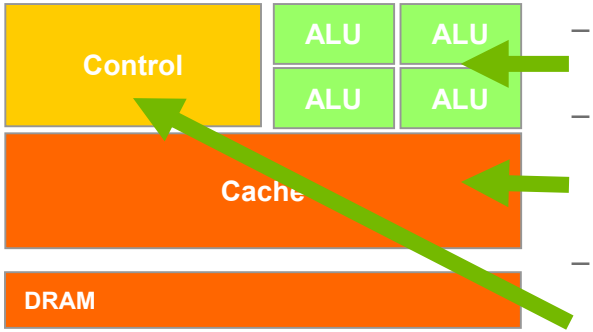
– Use the best match for the job (heterogeneity in mobile SOC)



CPU and GPU are designed very differently

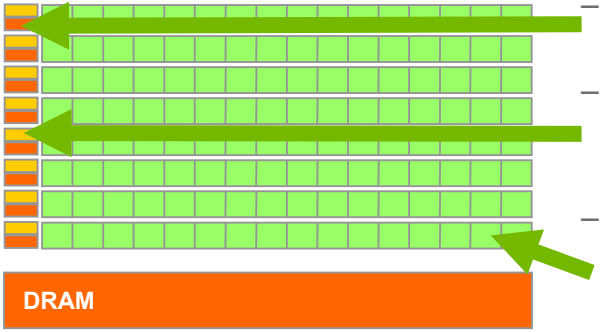


CPUs: Latency Oriented Design



- 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

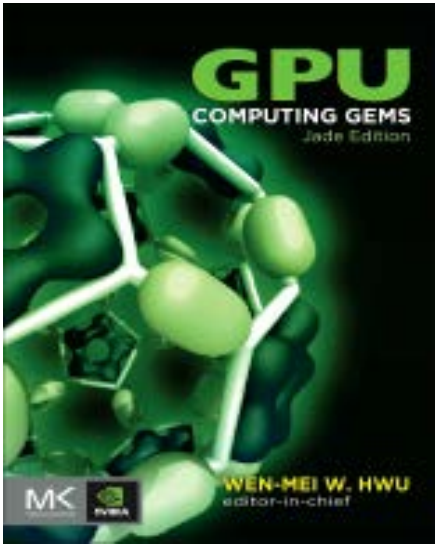


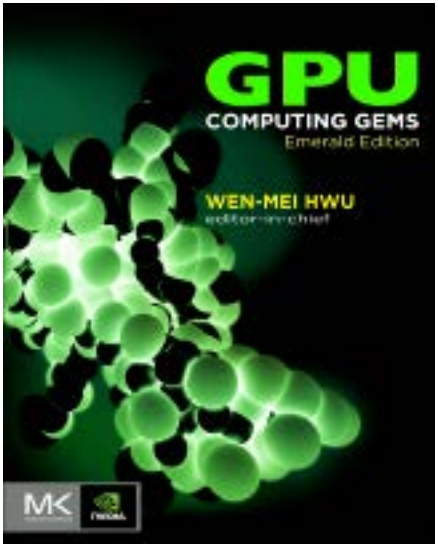
- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

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

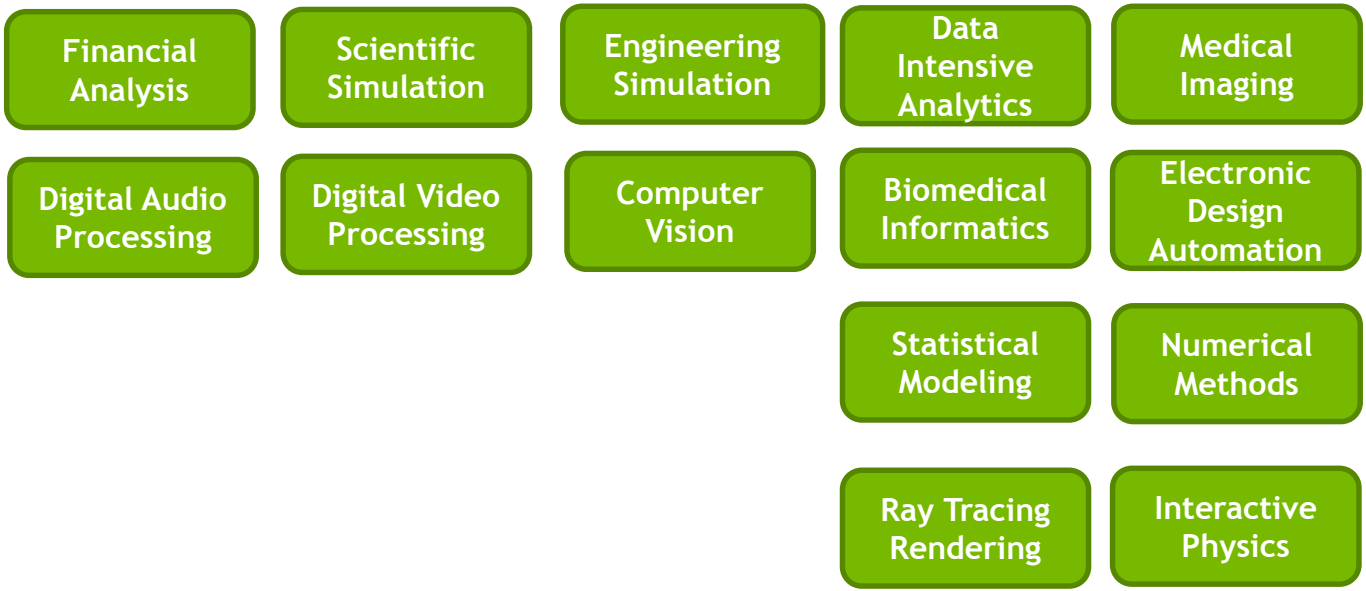
GPU computing reading resources





90 articles in two volumes

Heterogeneous Parallel Computing in Many Disciplines



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.

GPU Teaching Kit

Accelerated Computing

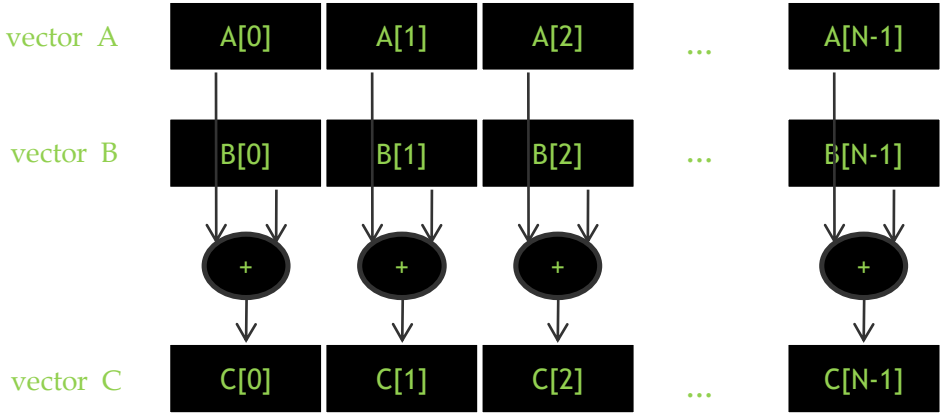
Lecture 2.2 - Introduction to CUDA C

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

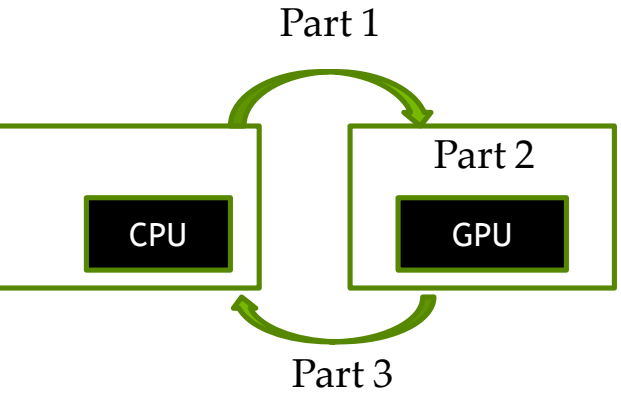
Data Parallelism - Vector Addition Example



Vector Addition – Traditional C Code

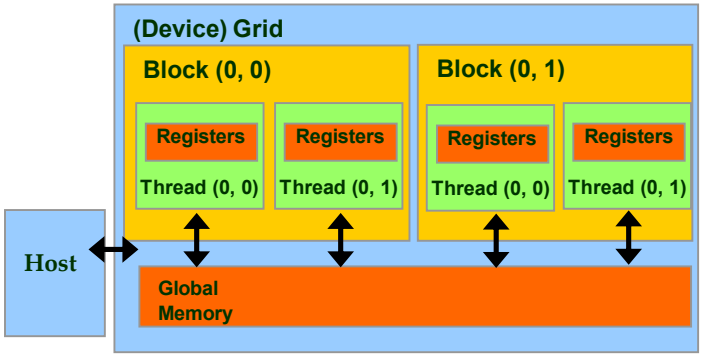
```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

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
}
```

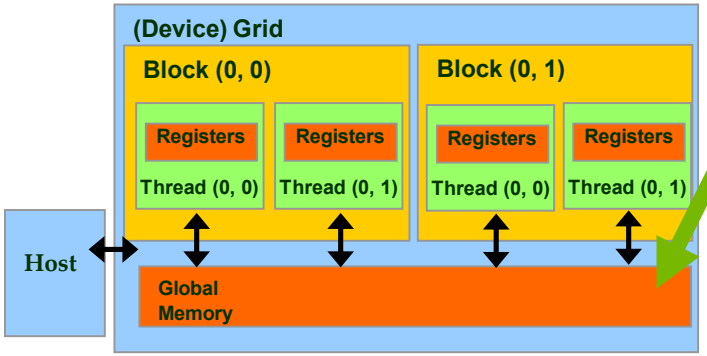
Partial Overview of CUDA Memories



- 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

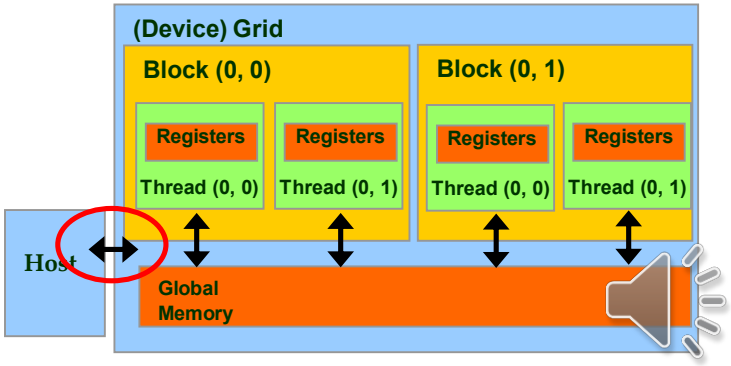
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointe**r 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



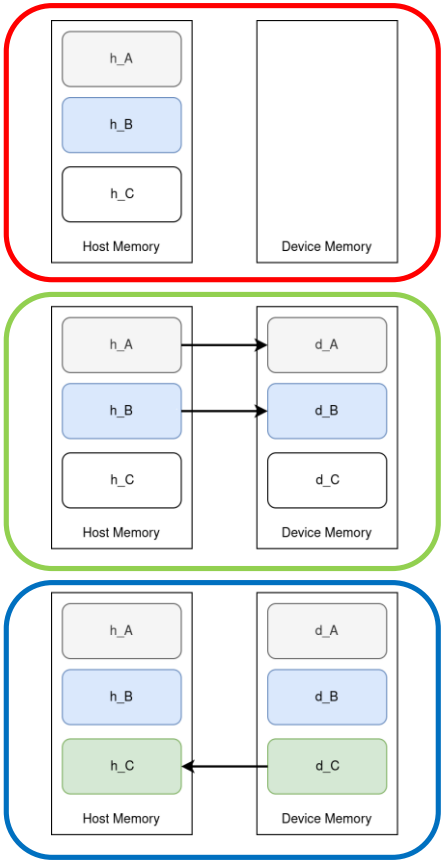
- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is synchronous with respect to the host

Vector Addition, Explicit Memory Management

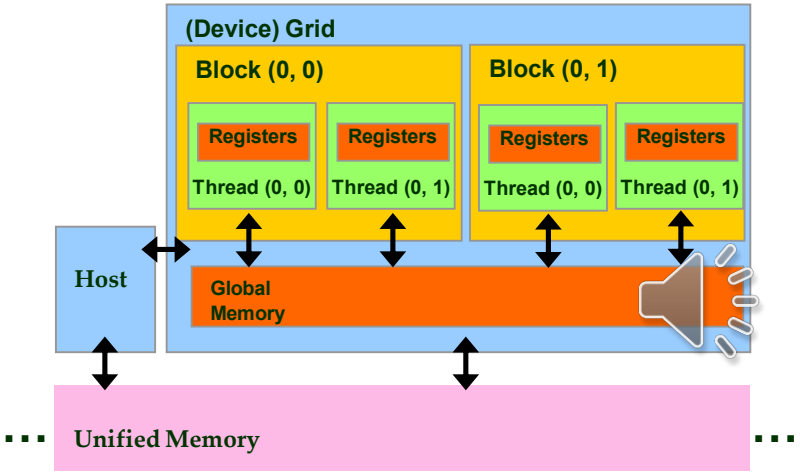
… Allocate h_A, h_B, h_C …

```
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 …



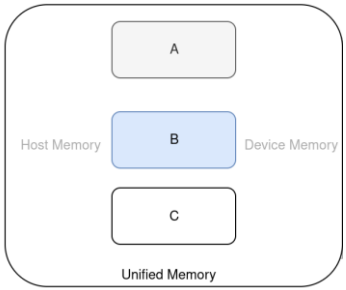
Unified Memory



- 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.

GPU Teaching Kit

Accelerated Computing

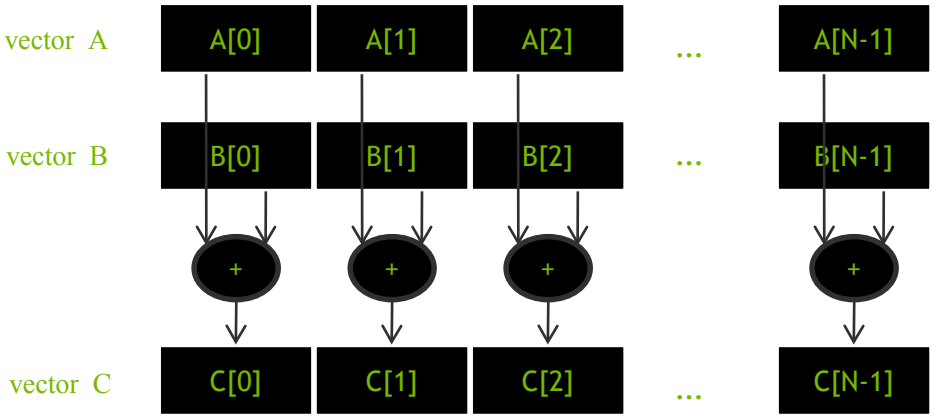
Lecture 2.3 – Introduction to CUDA C

Threads and Kernel Functions

Objective

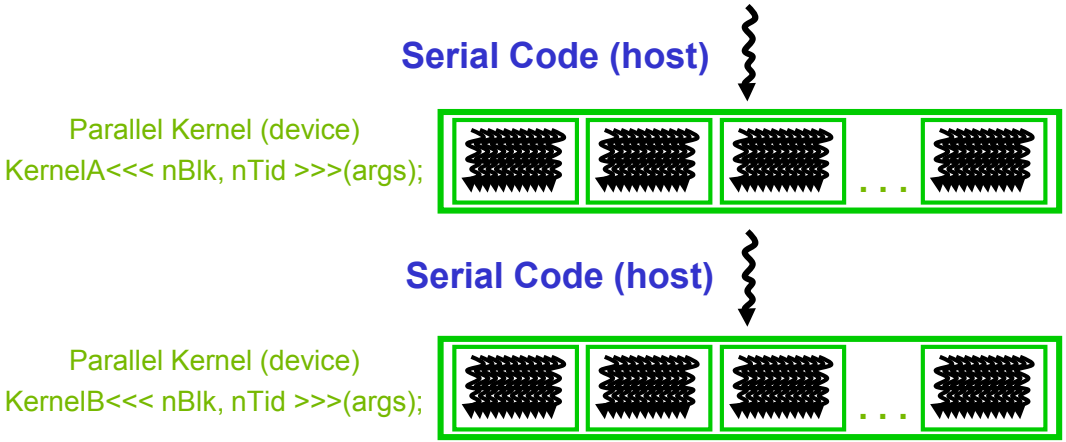
- To learn about CUDA threads, the main mechanism for exploiting of data parallelism
  - Hierarchical thread organization
  - Launching parallel execution
  - Thread index to data index mapping

Data Parallelism - Vector Addition Example

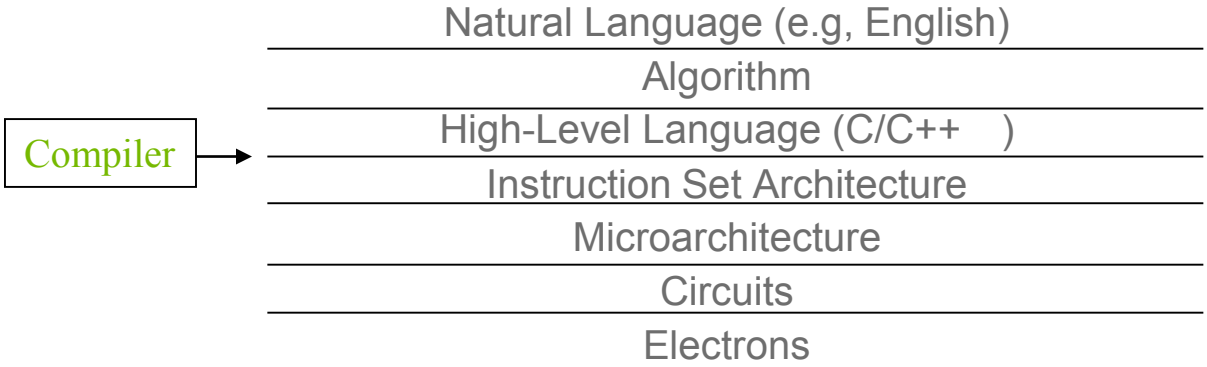


CUDA Execution Model

- 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*

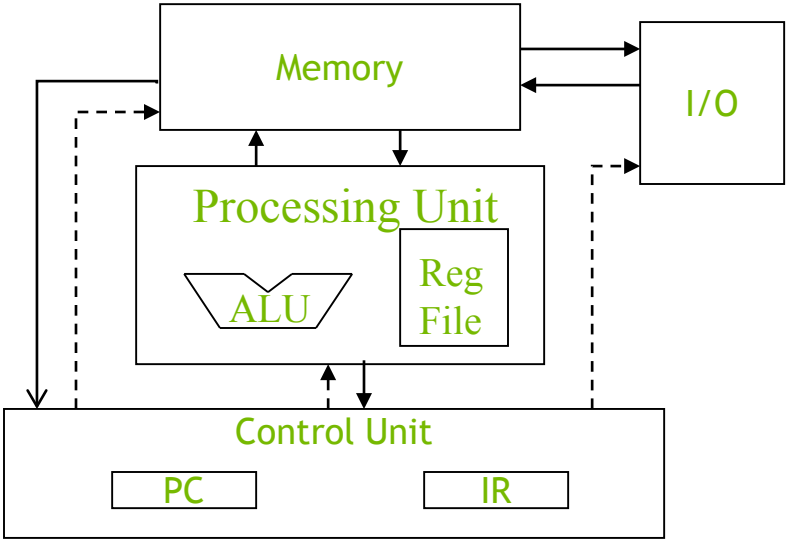
A program at the ISA level

- 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.

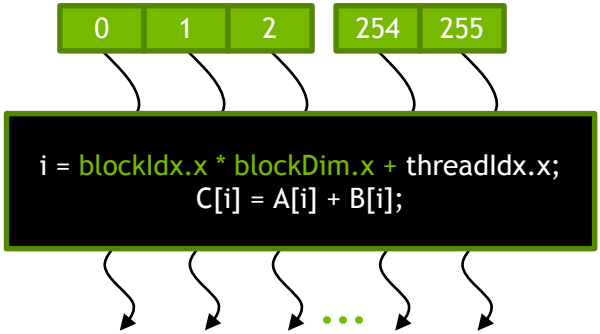
A Thread as a Von-Neumann Processor

A thread is a “virtualized” or
“abstracted”
Von-Neumann Processor

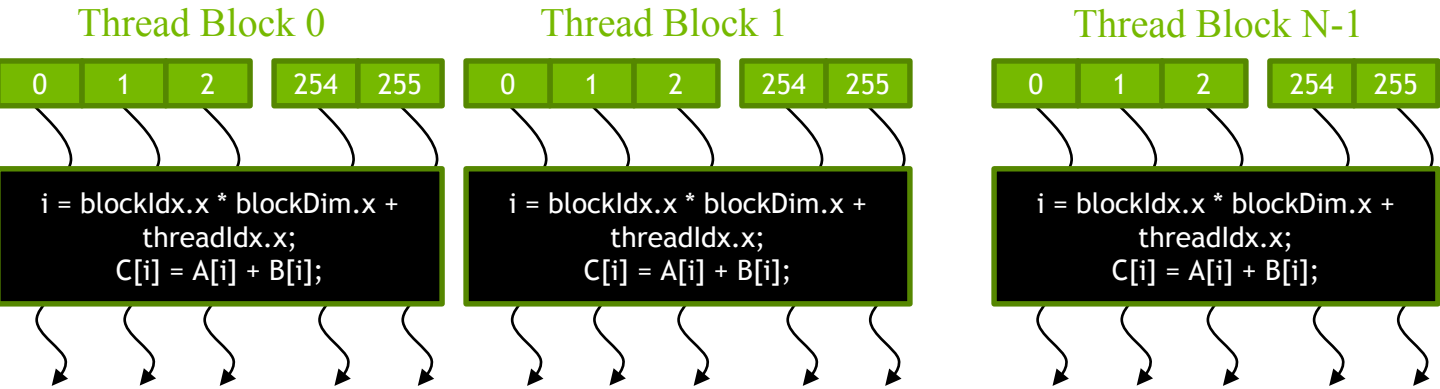


Arrays of Parallel Threads

- 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



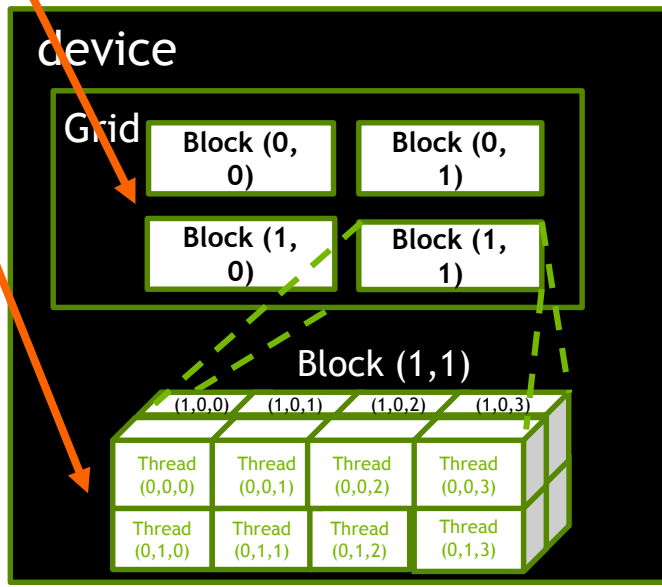
Thread Blocks: Scalable Cooperation



- 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.

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

NVIDIA ILLINOIS

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

Easy to use
Most Performance

Easy to use
Portable code

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 <cstdio>

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

NVIDIA ILLINOIS

CUDA Example 1: Hello World

```
#include <cstdio>

__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

NVIDIA ILLINOIS

Hello World! with Device Code

```
#include <cstdio>

__global__ void mykernel(void) {
}

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

Output:

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

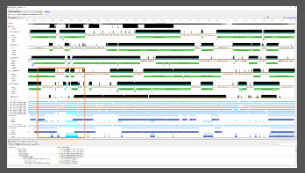
- `mykernel` (does nothing, somewhat anticlimactic!)

NVIDIA ILLINOIS

Developer Tools - Debuggers

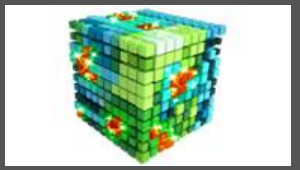
Nsight

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

NVIDIA ILLINOIS

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

NVIDIA ILLINOIS

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
```

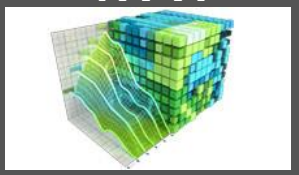
3. Fix Bug

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

Developer Tools - Profilers

NSIGHT

NVVP

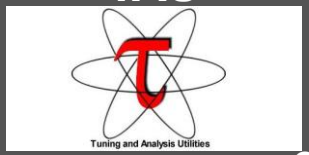


NVPROF

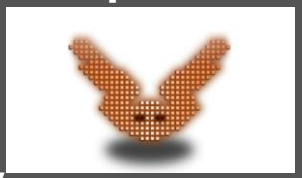
```
==20561== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.88%  866.69ms    504758  1.7170us  1.5040us  2.0160us  void th
int, thrust::detail::device_generate_functor<thrust::detail::fill_
 25.33%  440.05ms    252662  1.7410us  1.5360us  2.3680us  void th
t, thrust::detail::device_generate_functor<thrust::detail::fill_fu
 17.07%  296.60ms       200  1.4830ms  1.2040ms  1.7253ms  kerComp
  2.98%  51.819ms       200  259.09us  246.97us  264.83us  kerMake
  1.16%  20.173ms       501  40.265us     928ns  17.677ms  [CUDA m
  0.93%  16.198ms       200  80.991us  71.840us  90.751us  kerColl
  0.73%  12.636ms       400  31.589us  14.720us  50.432us  [CUDA m
  0.69%  12.075ms       200  60.376us  59.680us  62.304us  kerRema
  0.63%  10.993ms       200  54.963us  52.608us  58.208us  kerMake
  0.32%  5.5524ms       200  27.761us  22.559us  33.152us  [CUDA m
  0.12%  2.1342ms         1  2.1342ms  2.1342ms  2.1342ms  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

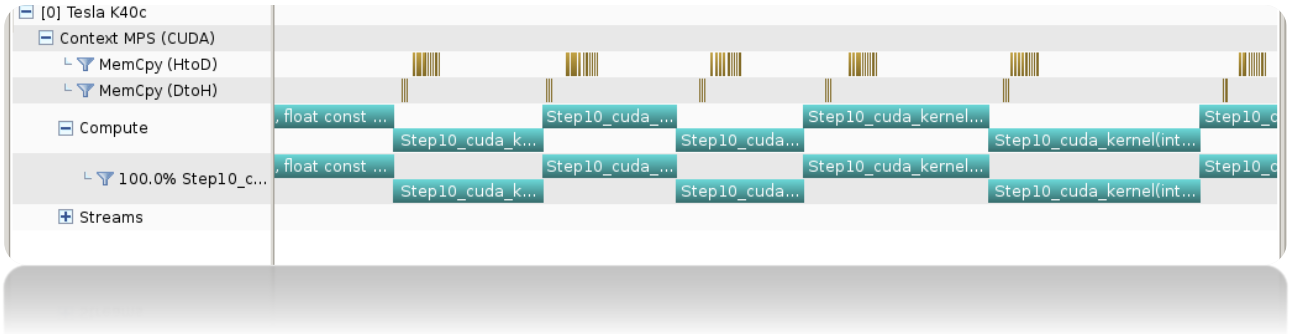
Instructions:

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

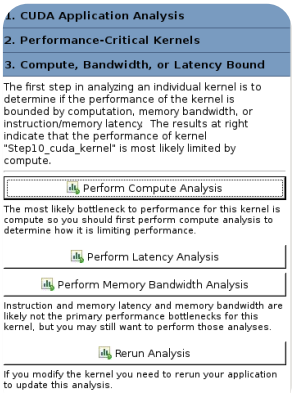
NVIDIA ILLINOIS

NVIDIA's Visual Profiler (NVVP)

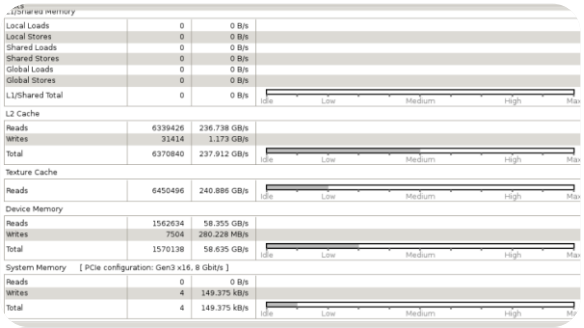
Timeline

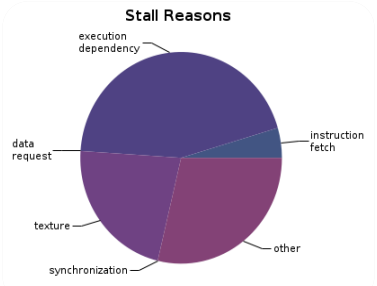


Guided System



Analysis





Example 4: NVVP

Instructions:

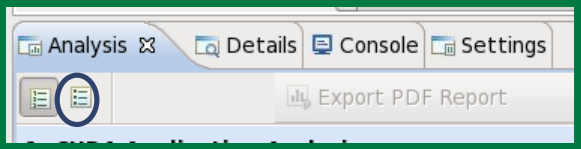
1. Import nvprof profile into NVVP
   - Launch nvvp
   - Click File/ Import/ Nvprof/ Next/ Single process/ Next / Browse
     - Select profile.timeline
   - Add Metrics to timeline
     - Click on 2nd Browse
     - Select profile.metrics
   - Click Finish
2. Explore Timeline
   - Control + mouse drag in timeline to zoom in
   - Control + mouse drag in measure bar (on top) to measure time

NVIDIA ILLINOIS

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:

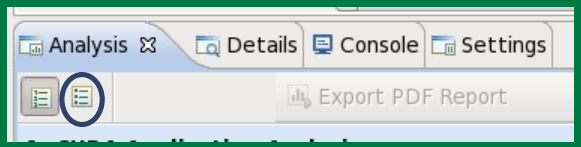
If kernel order is non-deterministic you can only load the timeline or the metrics but not both.

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



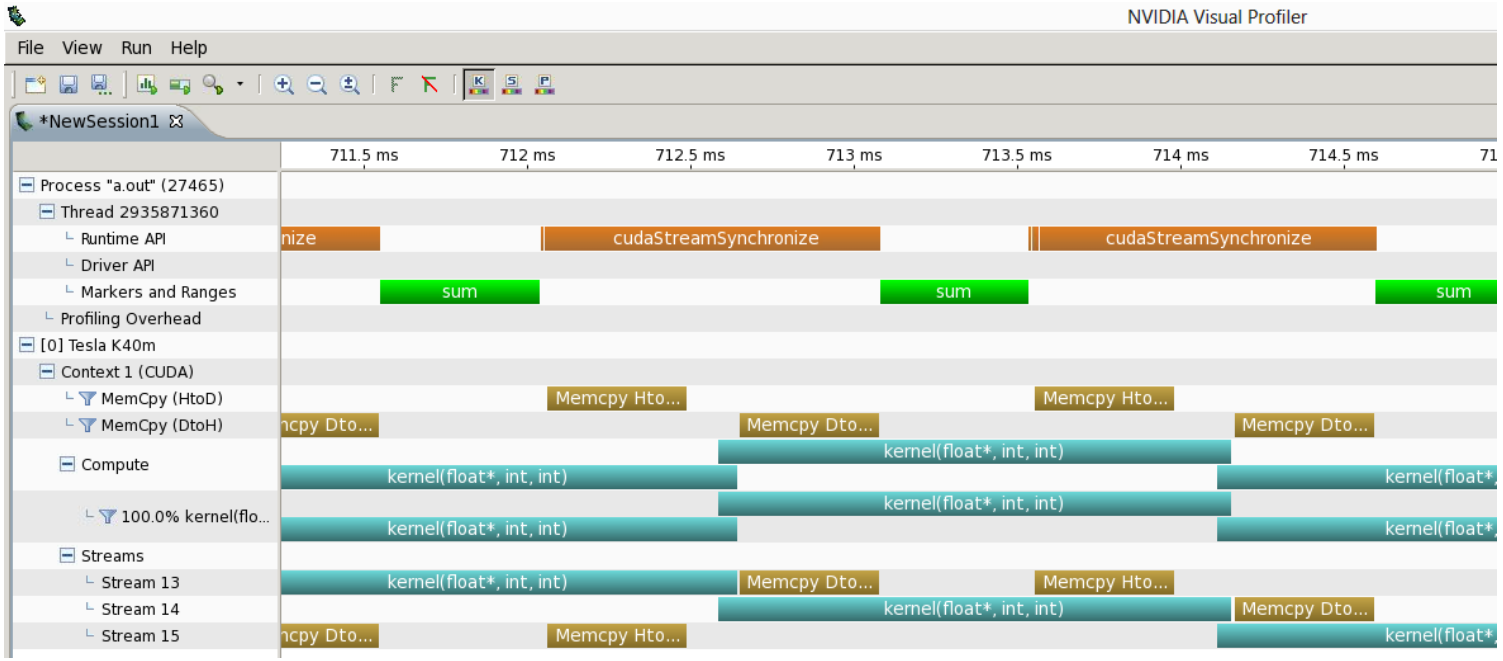
2. Click on a kernel
   Select Unguided Analysis
   Click Analyze All

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

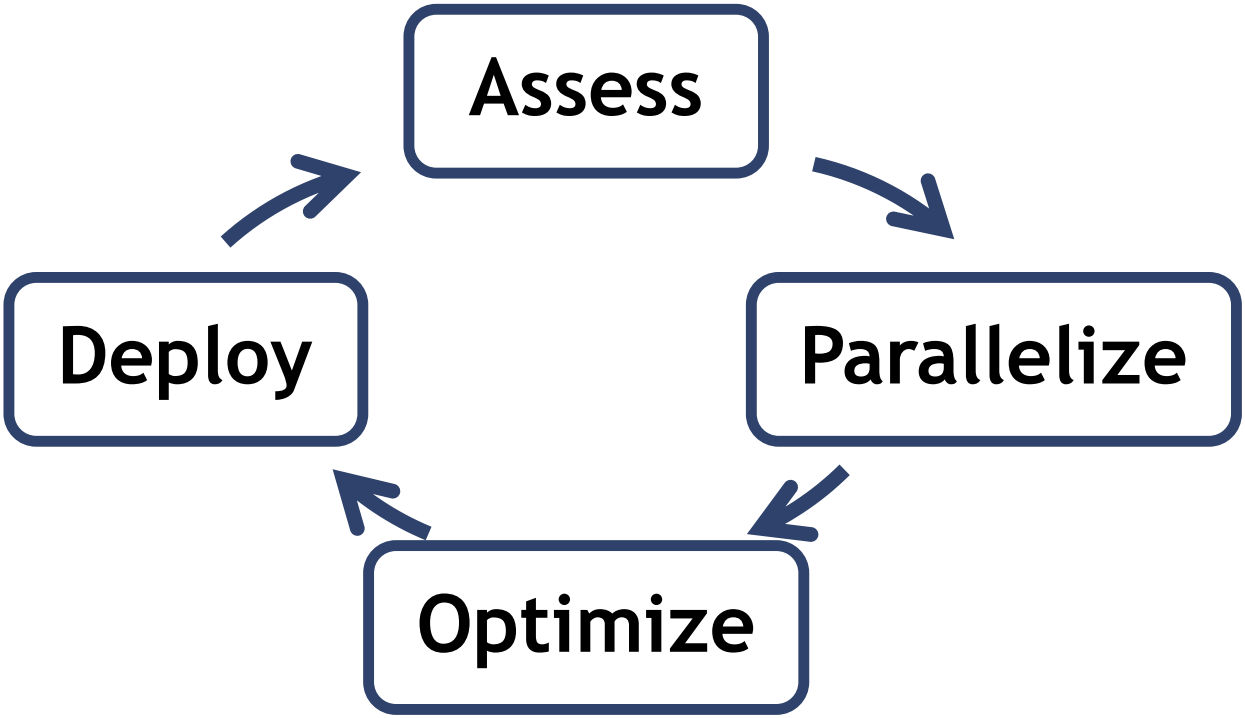
NVIDIA ILLINOIS

Profiler Summary

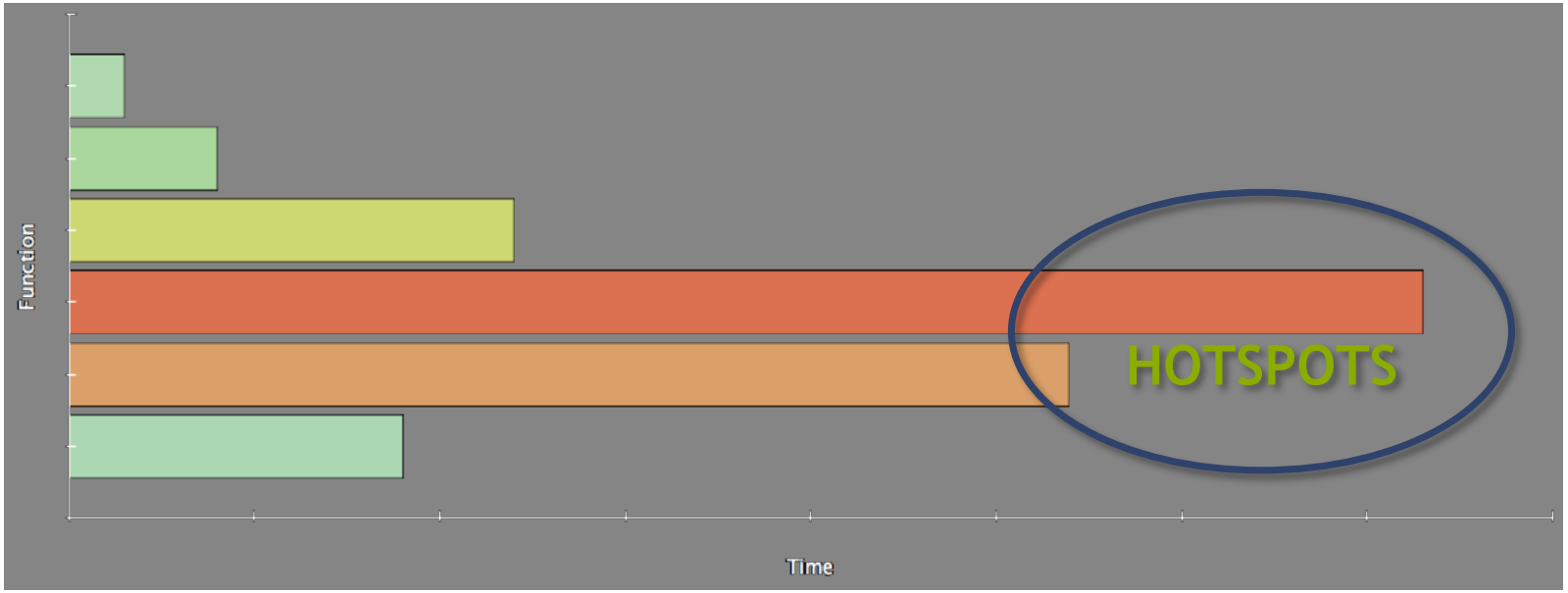
- Many profile tools are available
- NVIDIA Provided
  - NVPROF: Command Line
  - NVVP: Visual profiler
  - NSIGHT: IDE (Visual Studio and Eclipse)
- 3rd Party
  - TAU
  - VAMPIR

NVIDIA ILLINOIS

Optimization



Assess

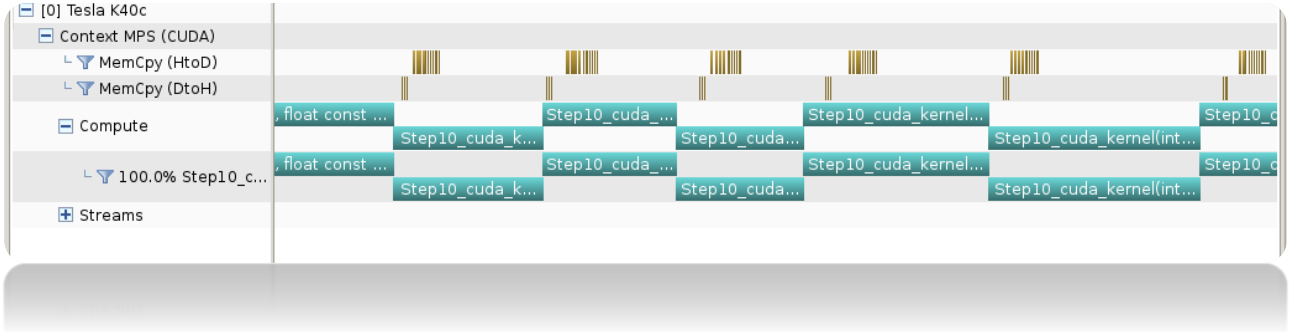


- Profile the code, find the hotspot(s)
- Focus your attention where it will give the most benefit

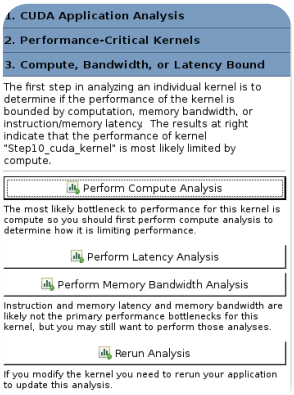
Parallelize

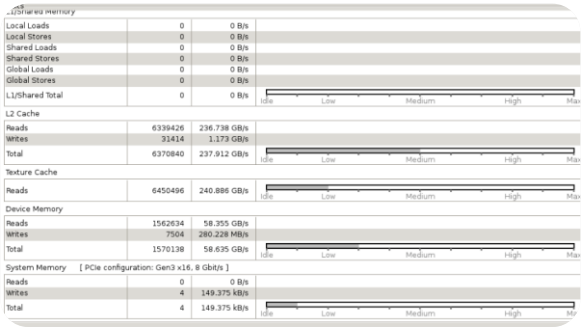
Optimize

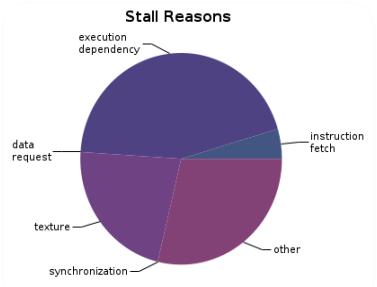
Timeline



Guided System Analysis







Bottleneck Analysis

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

129 GB/s → 84 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 Efficien	97.1%
▾ Occupancy	
Achieved	86.7%
Theoretical	100%
▾ Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB

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	Idle Low Medium

⚠ Shared Memory Alignment and Access Pattern

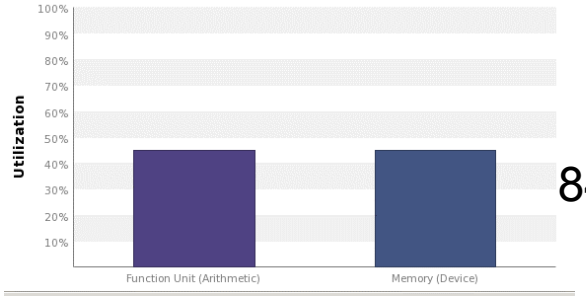
Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

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 Efficien	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	Idle Low Medium
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	Idle Low Medium
Texture Cache			
Reads	0	0 B/s	Idle Low Medium
Device Memory			
Reads	524968	69.306 GB/s	
Writes	524289	69.217 GB/s	
Total	1049257	138.523 GB/s	Idle Low Medium

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.

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

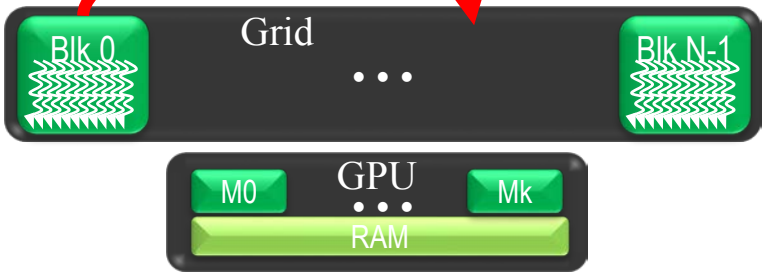
```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  dim3 DimGrid((n-1)/256 + 1, 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

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];
}
```

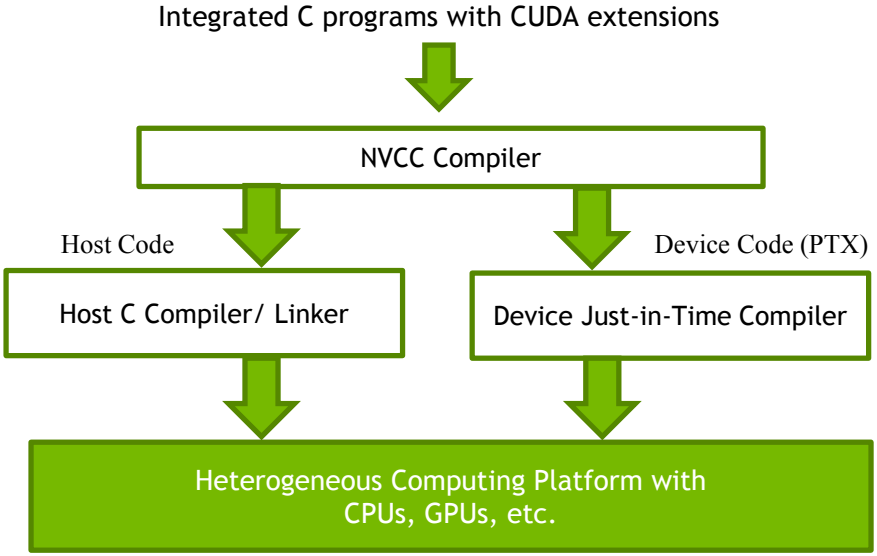


More on CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	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.

GPU Teaching Kit

Accelerated Computing

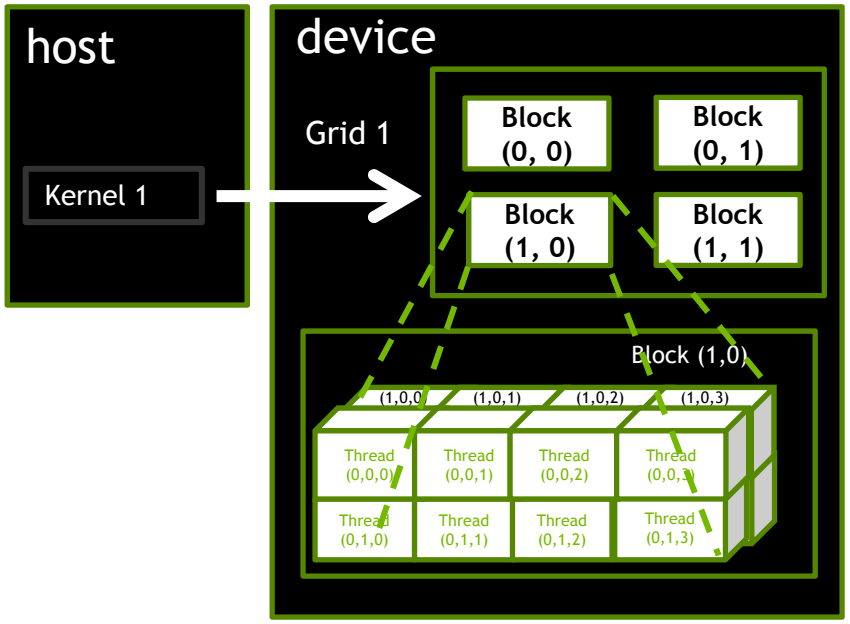
Lecture 3.2 – CUDA Parallelism Model

Multidimensional Kernel Configuration

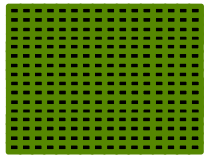
Objective

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

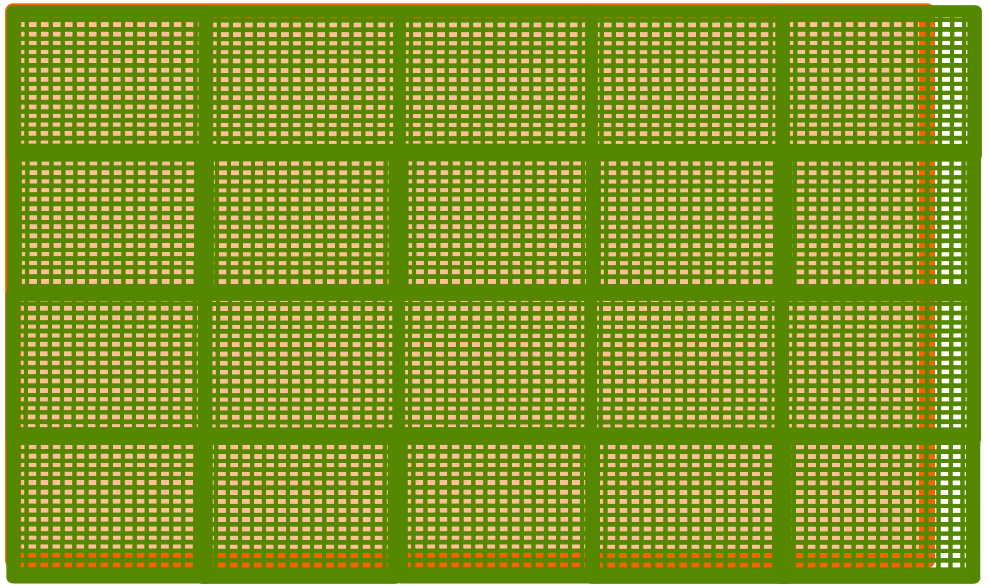
A Multi-Dimensional Grid Example



Processing a Picture with a 2D Grid

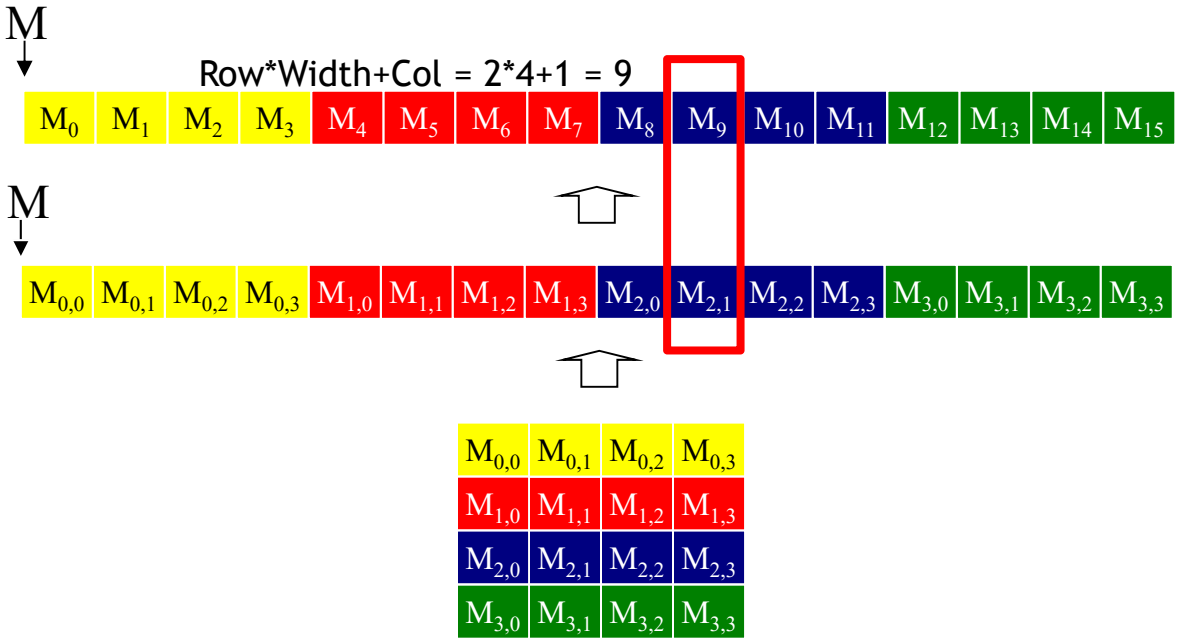


16×16 blocks



62×76 picture

Row-Major Layout in C/C++



Source Code of a PictureKernel

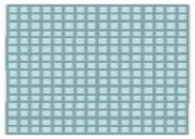
```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                              int height, int width)
{

  // 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];
  }
}
```

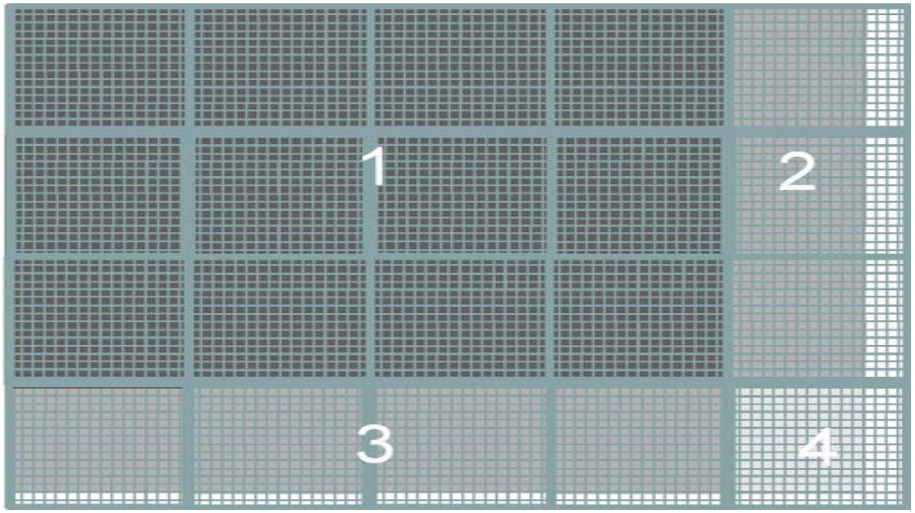
Scale every pixel value by 2.0

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.

GPU Teaching Kit

Accelerated Computing

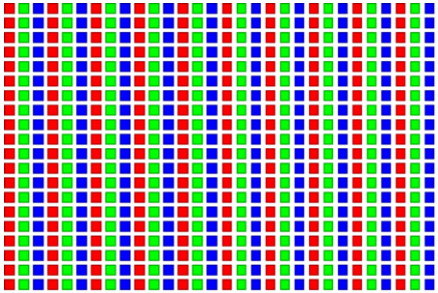
Lecture 3.3 – CUDA Parallelism Model

Color-to-Grayscale Image Processing Example

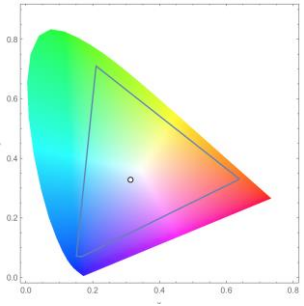
Objective

– 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

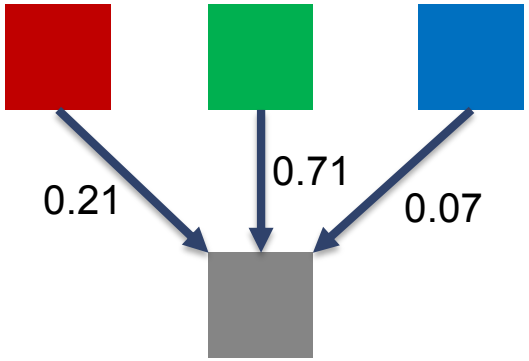


RGB to Grayscale Conversion

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:
  grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b
- This is just a dot product <[r,g,b],[0.21,0.71,0.07]> 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.

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

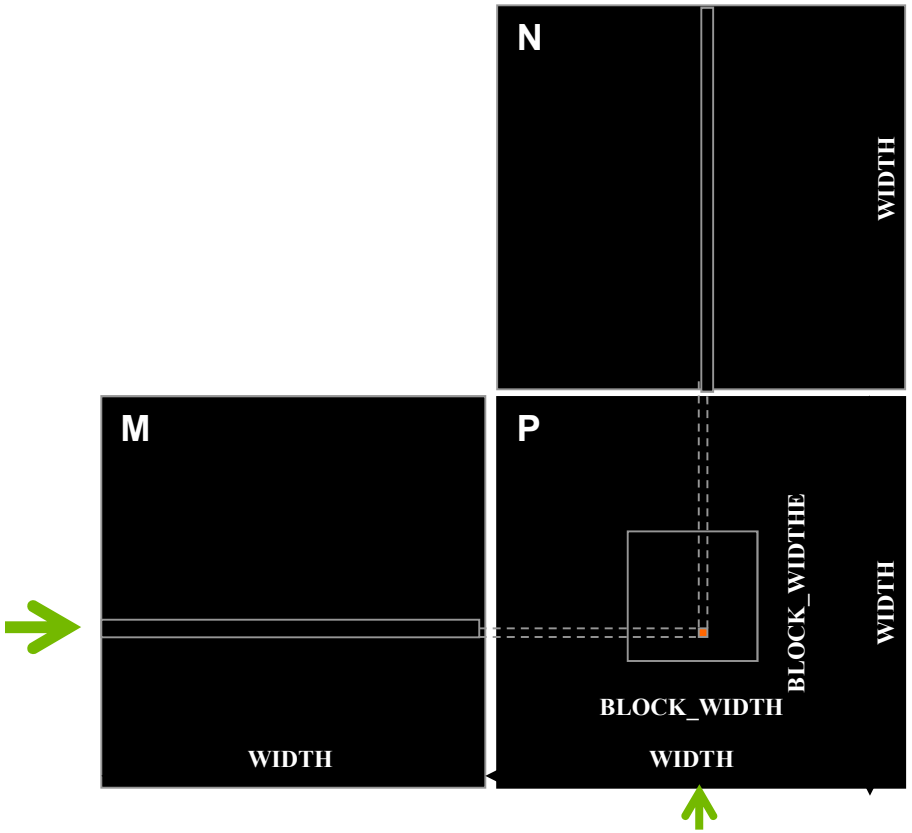
Review: Image Blur Kernel.

```
// 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*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 the1,600 GFLOPS

Example – Matrix Multiplication



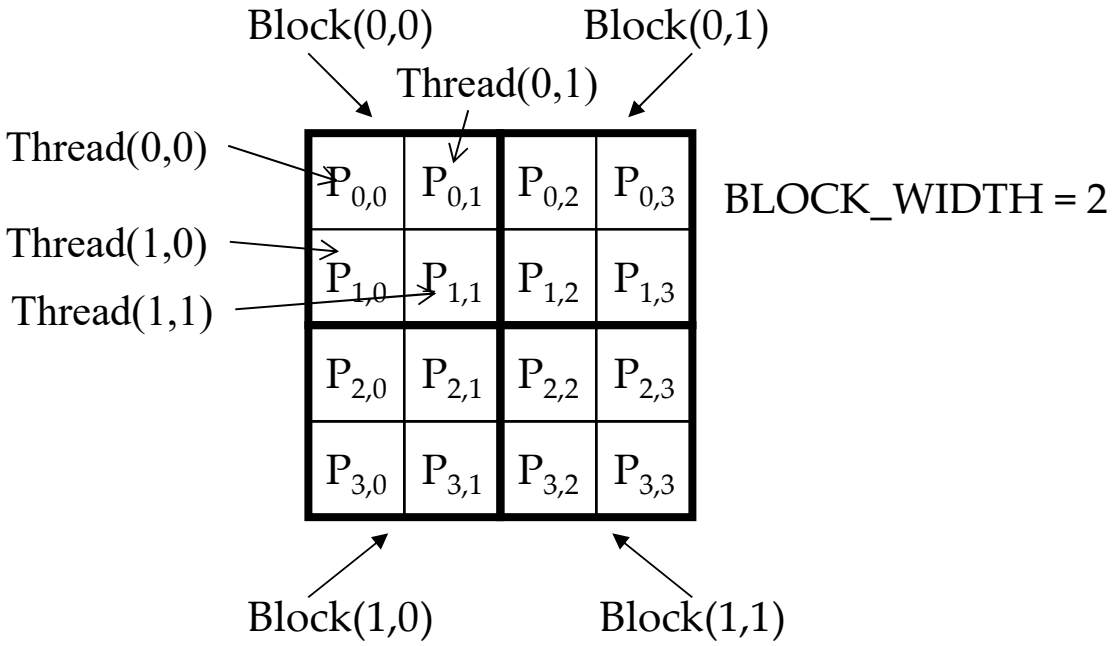
A Basic Matrix Multiplication

```
__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;
  }

}
```

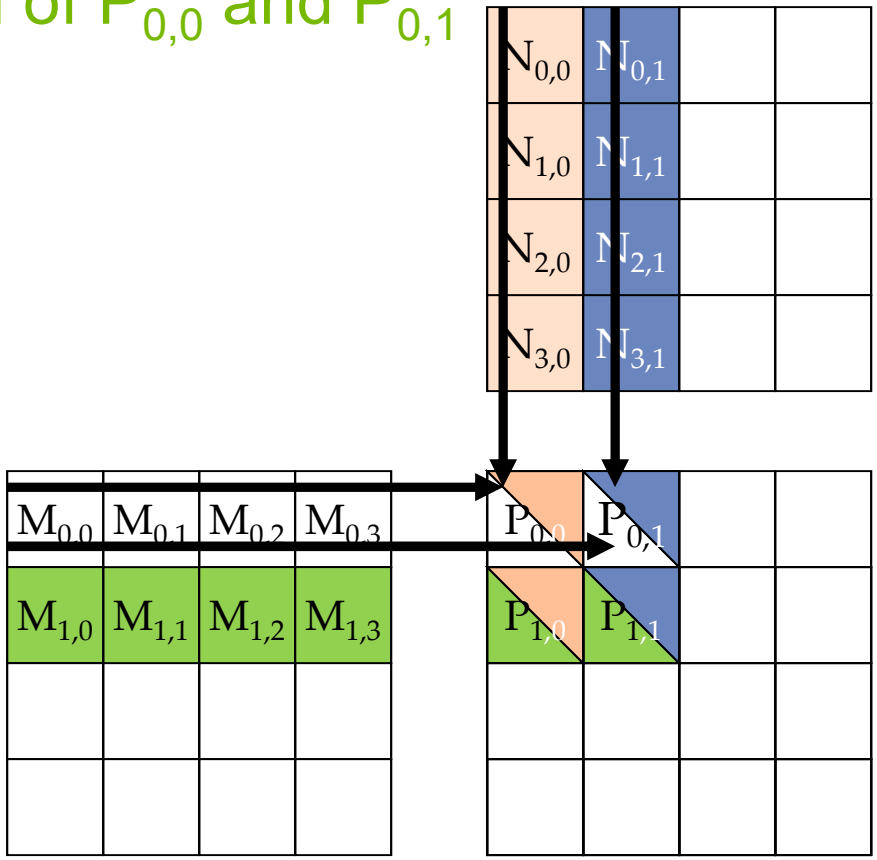
Example – Matrix Multiplication

```
__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;
  }

}
```

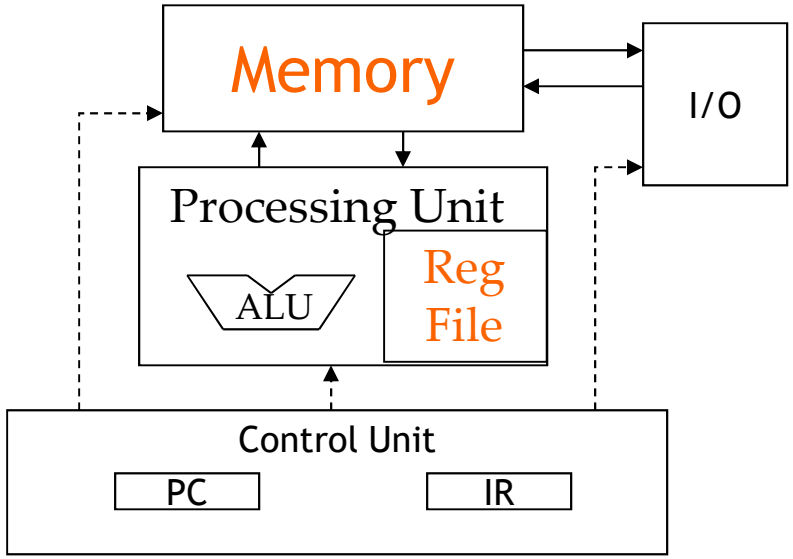
A Toy Example: Thread to P Data Mapping



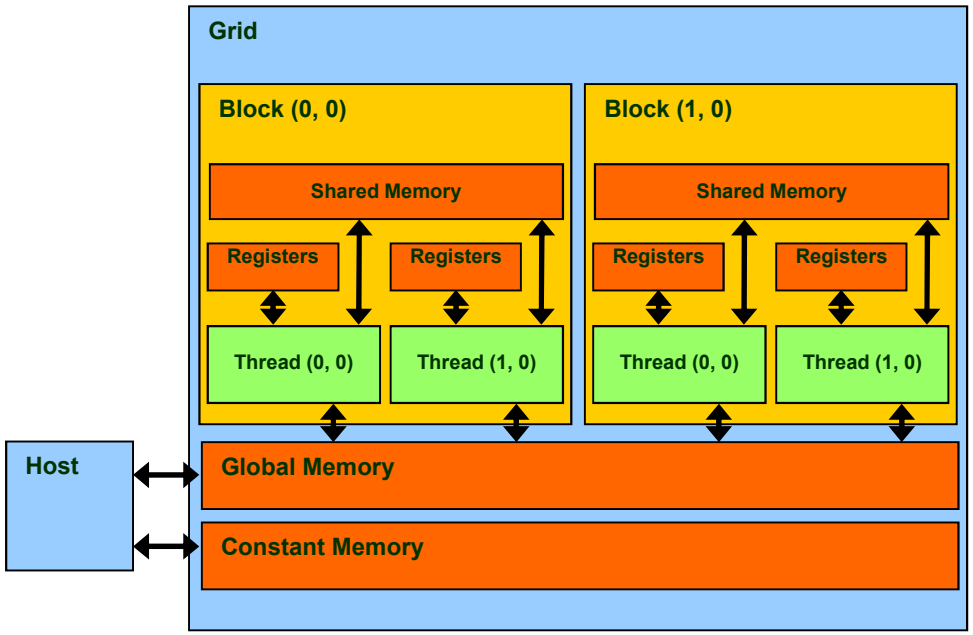
Calculation of $P_{0,0}$ and $P_{0,1}$



Memory and Registers in the Von-Neumann Model



Programmer View of CUDA Memories



Declaring CUDA Variables

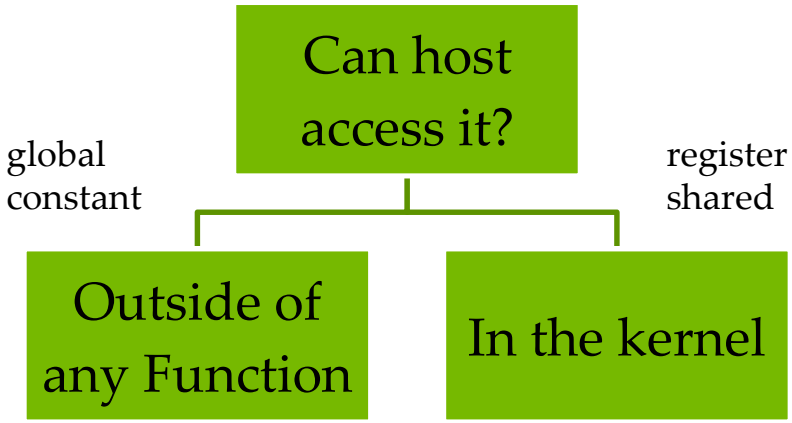
Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- **`__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];

  …
}
```

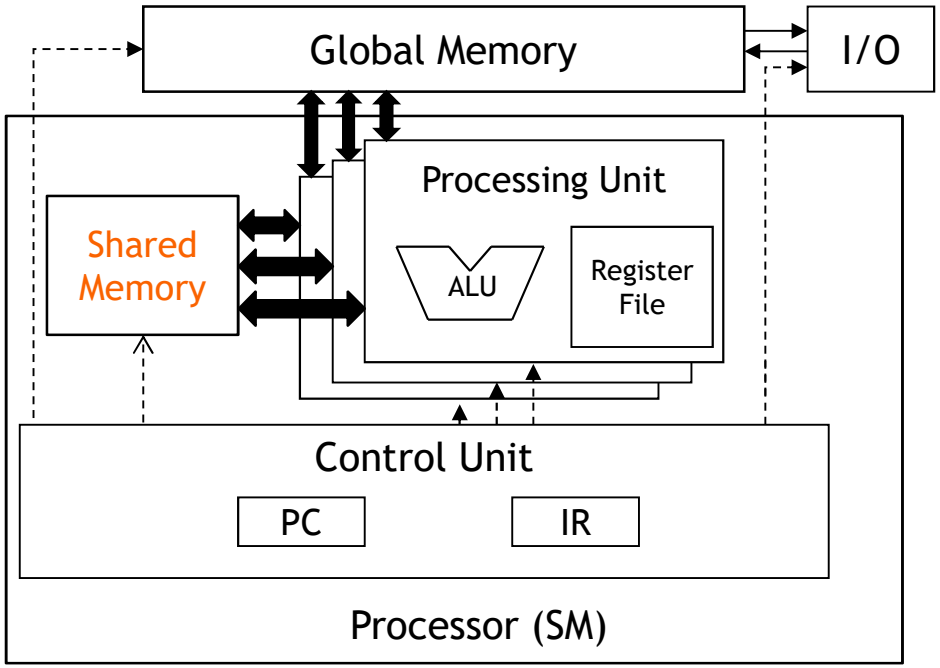
Where to Declare Variables?



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.

GPU Teaching Kit

Accelerated Computing

Module 4.2 – Memory and Data Locality

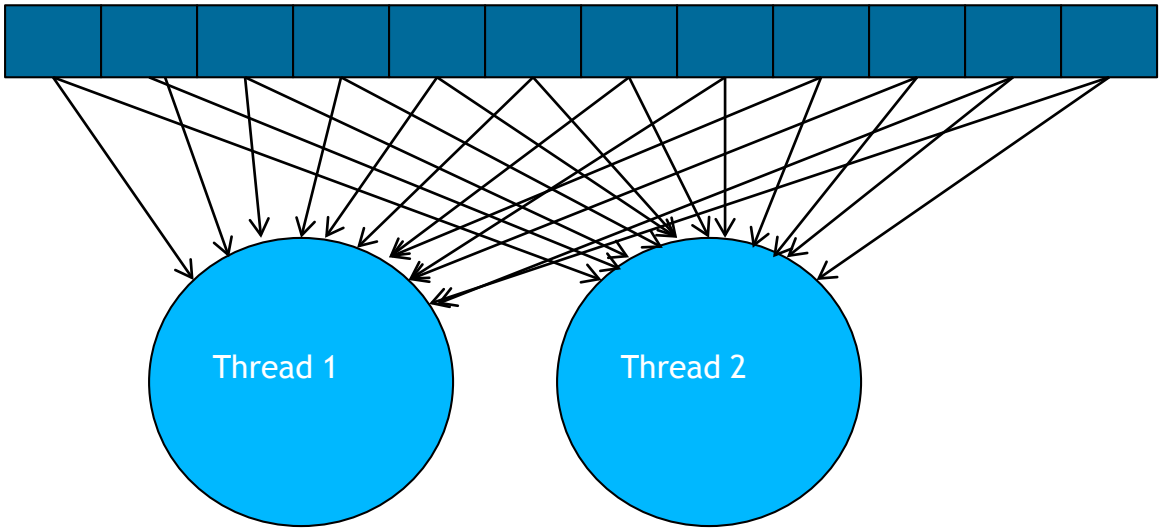
Tiled Parallel Algorithms

Objective

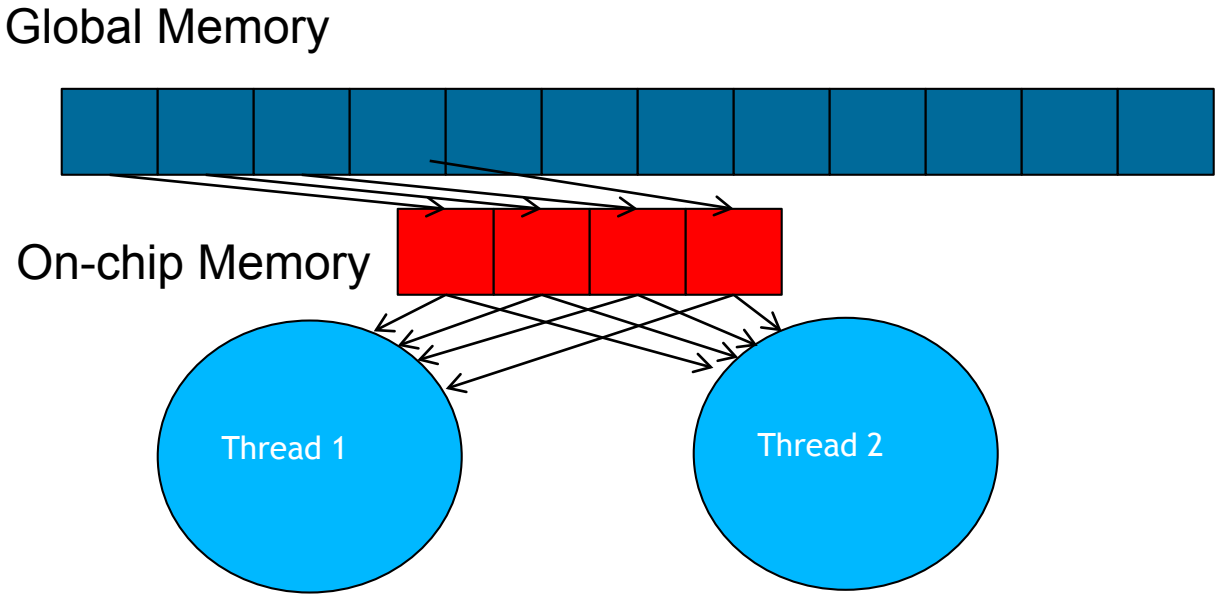
- To understand the motivation and ideas for tiled parallel algorithms
  - Reducing the limiting effect of memory bandwidth on parallel kernel performance
  - Tiled algorithms and barrier synchronization

Global Memory Access Pattern of the Basic Matrix Multiplication Kernel

Global Memory



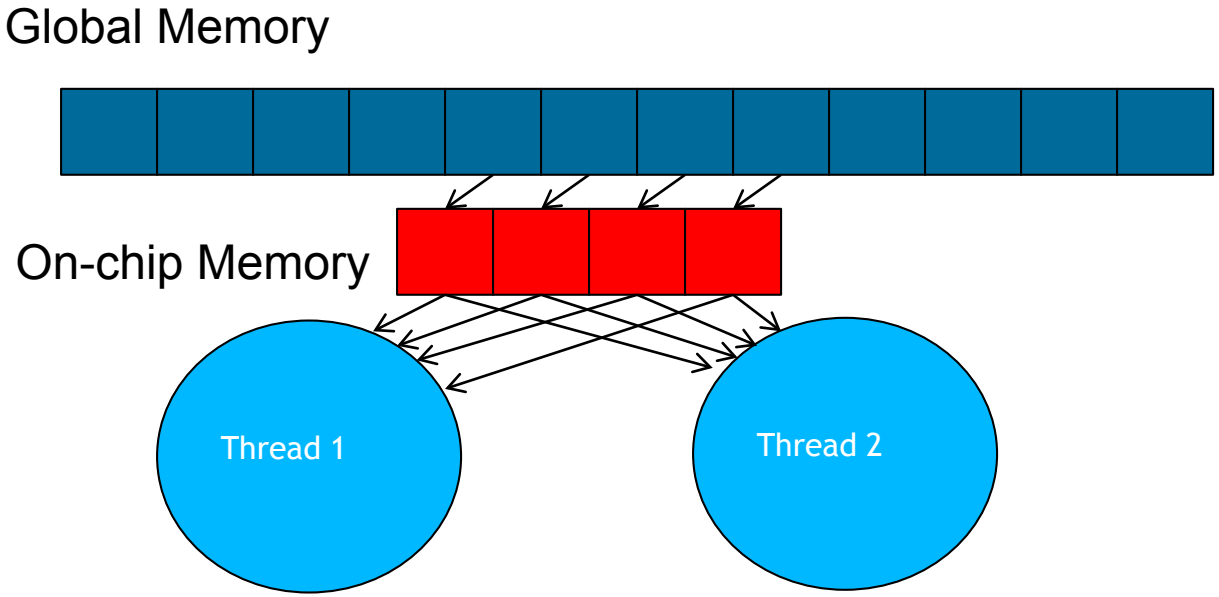
Tiling/Blocking - Basic Idea



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



Basic Concept of Tiling

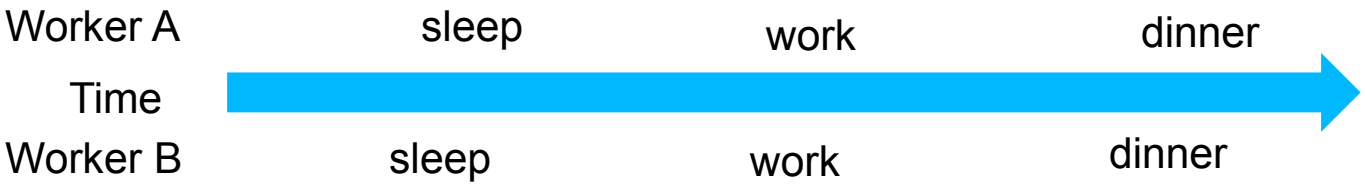
- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
  - Carpooling for commuters
  - Tiling for global memory accesses
    - drivers = threads accessing their memory data operands
    - cars = memory access requests

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

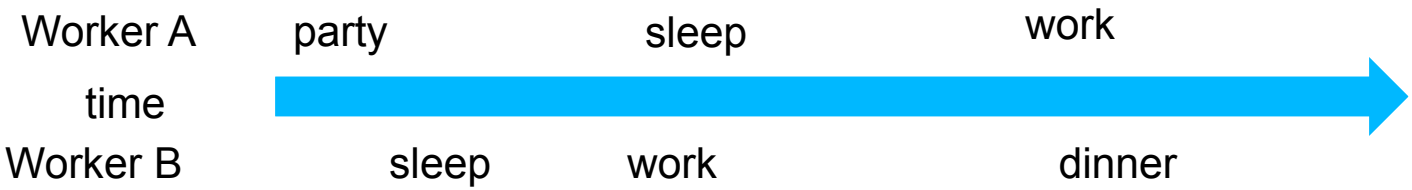
Carpools need synchronization.

– Good: when people have similar schedule



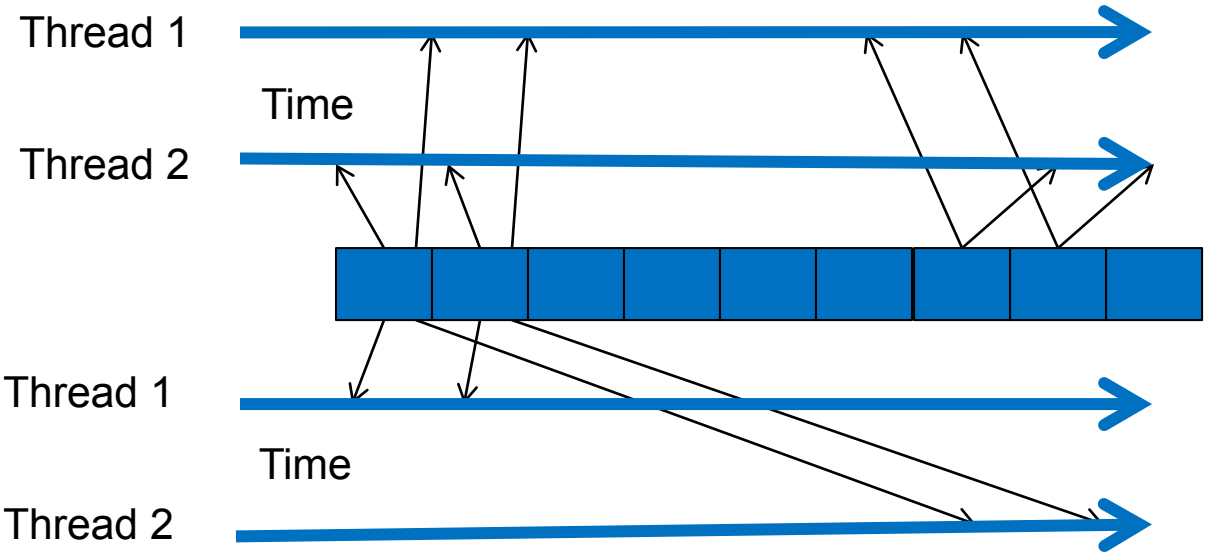
Carpools need synchronization.

– Bad: when people have very different schedule



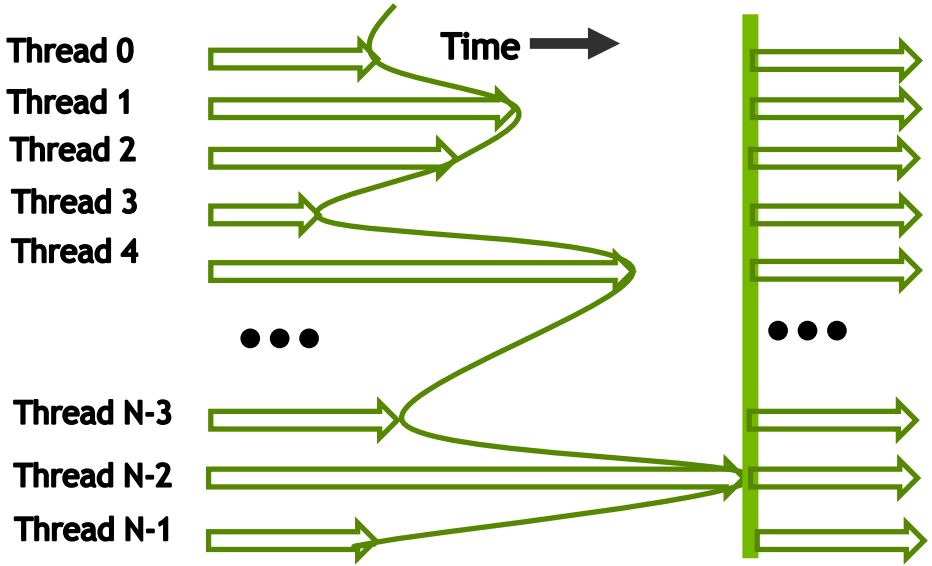
Same with Tiling

– 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.

GPU Teaching Kit

Accelerated Computing

Module 4.3 - Memory Model and Locality

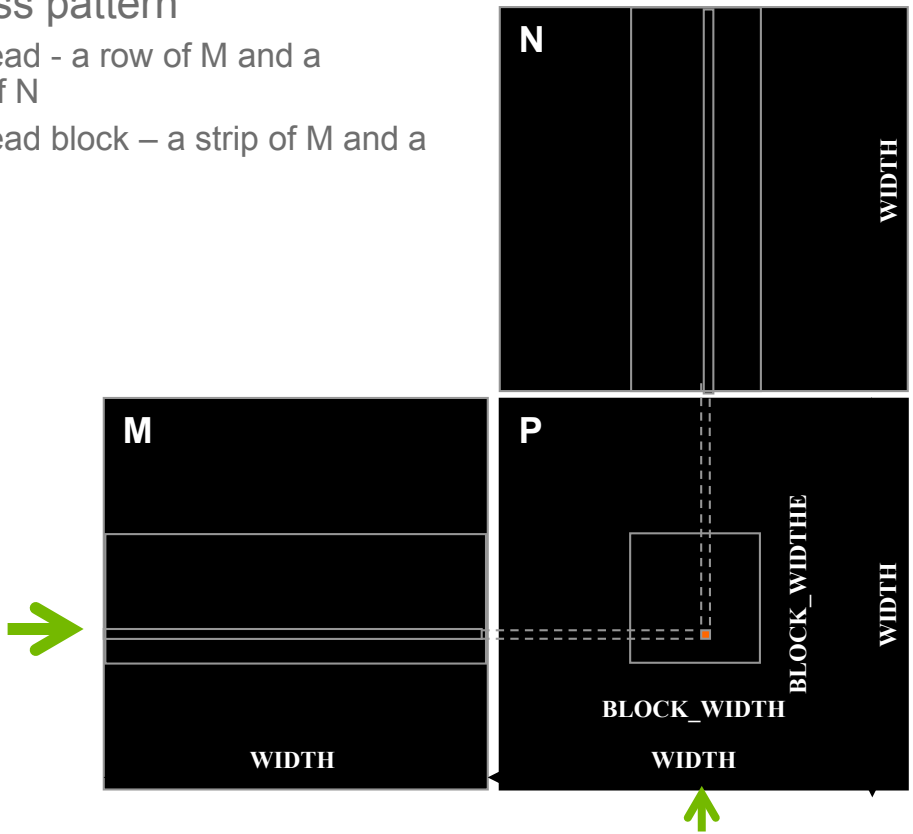
Tiled Matrix Multiplication

Objective

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

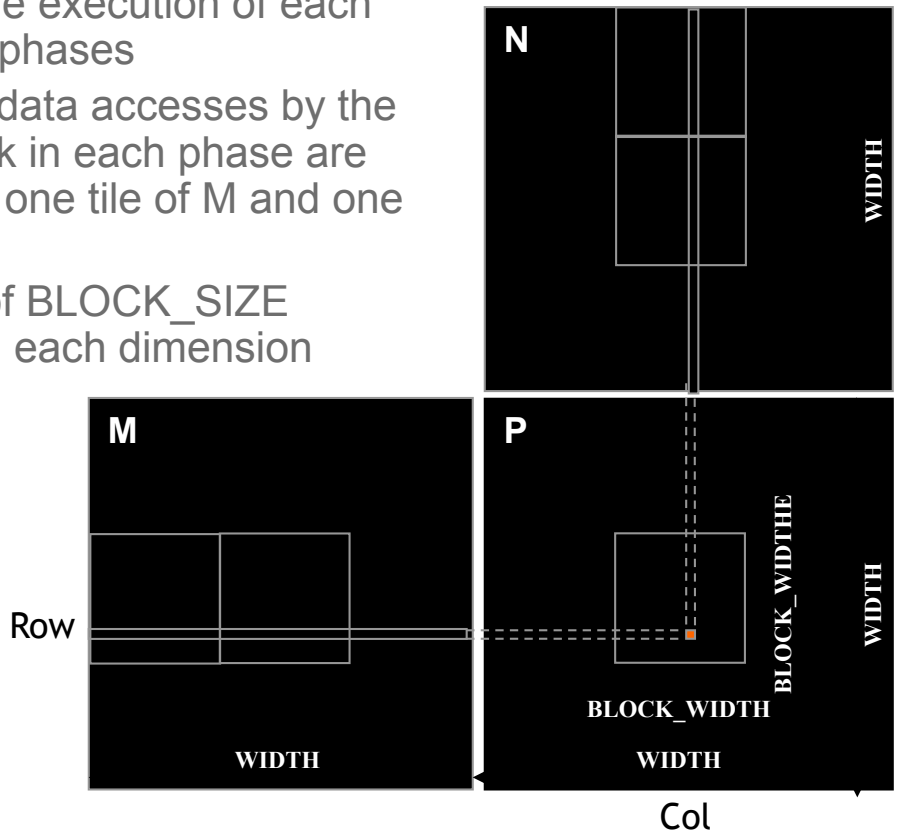
Matrix Multiplication

- 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

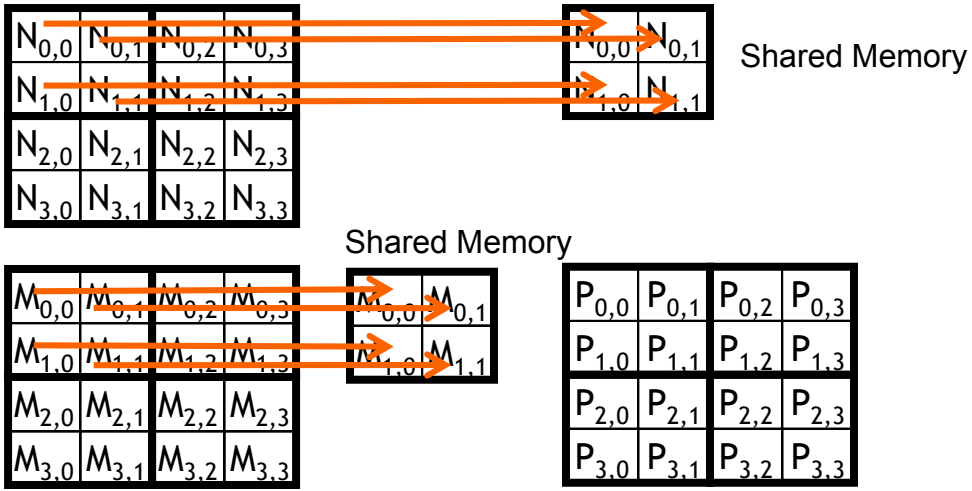
- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of BLOCK_SIZE elements in each dimension



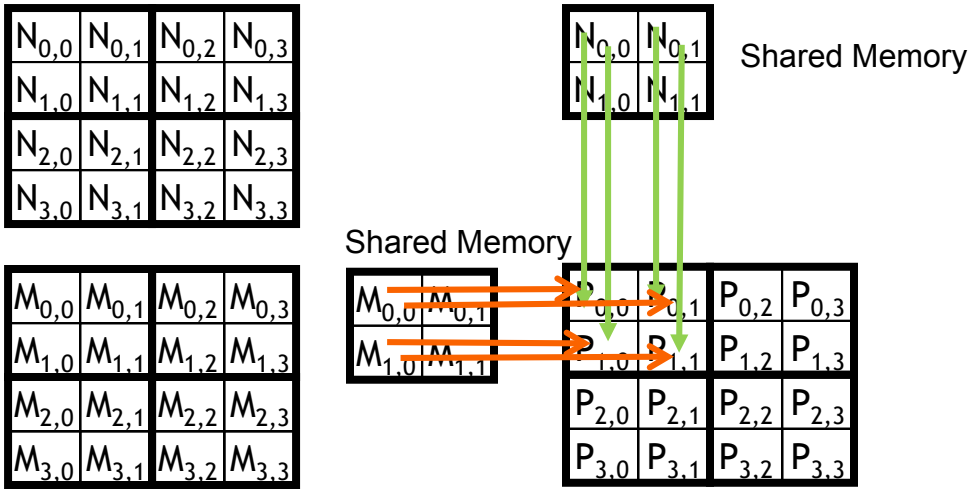
Loading a Tile

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

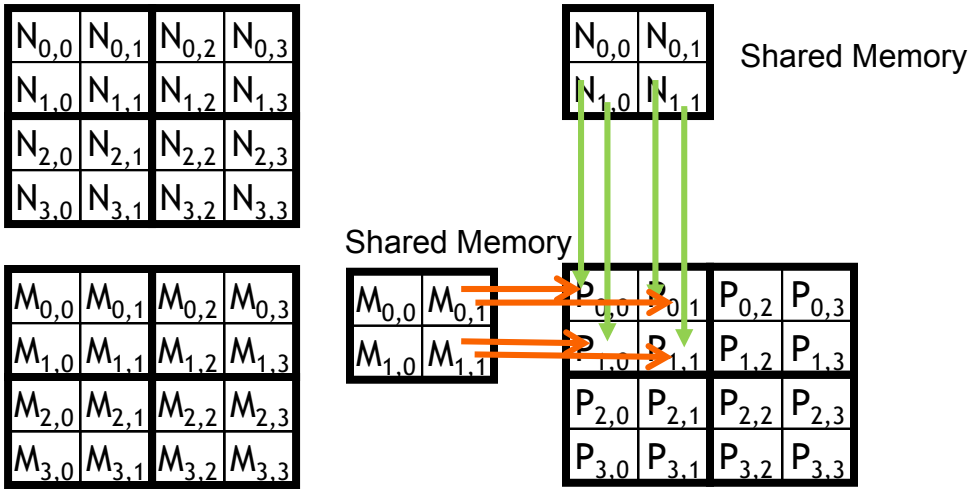
Phase 0 Load for Block (0,0)



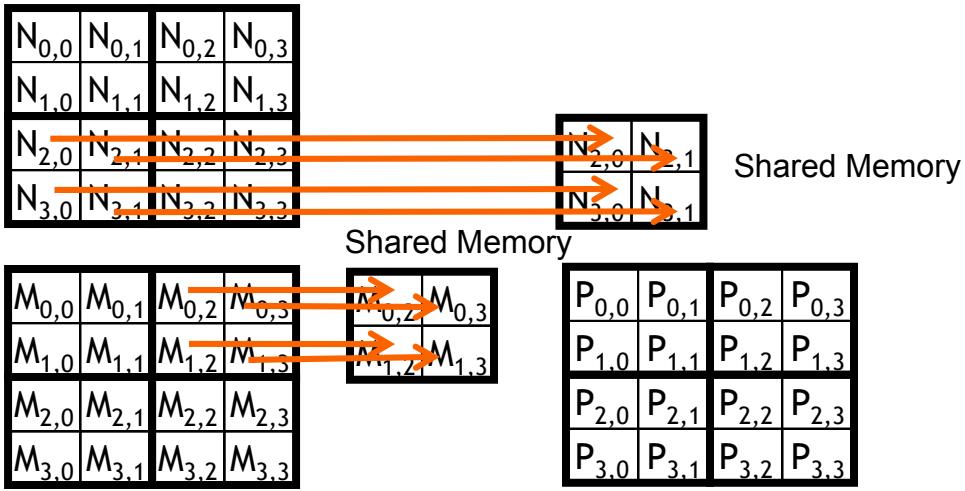
Phase 0 Use for Block (0,0) (iteration 0)



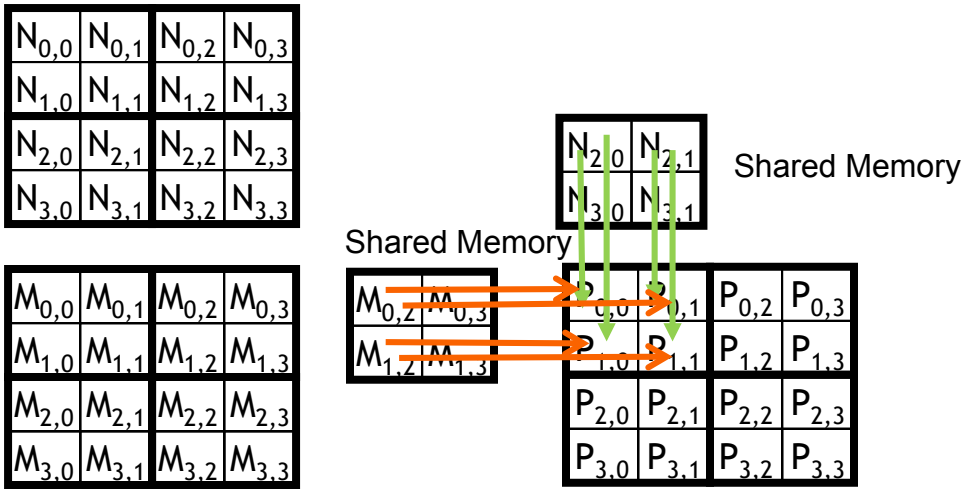
Phase 0 Use for Block (0,0) (iteration 1)



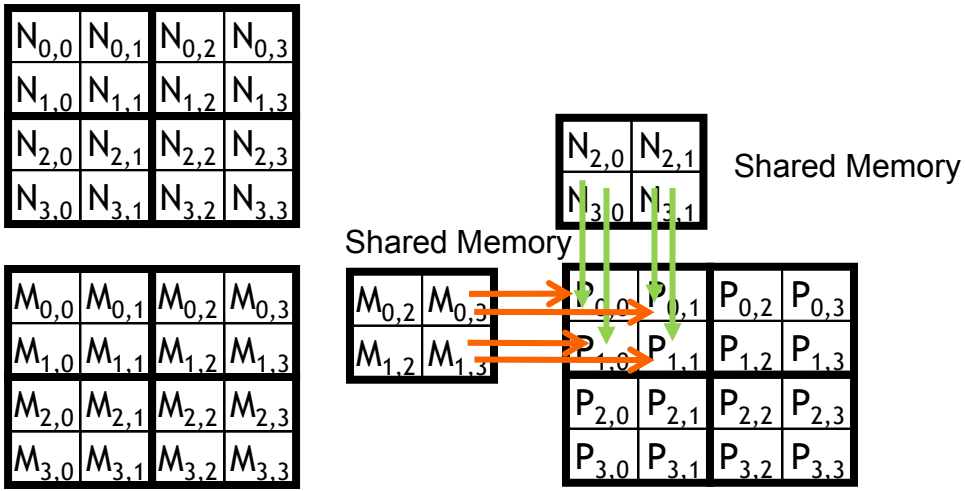
Phase 1 Load for Block (0,0)



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		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$

time →

Execution Phases of Toy Example (cont.)

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

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 the 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.

GPU Teaching Kit

Accelerated Computing

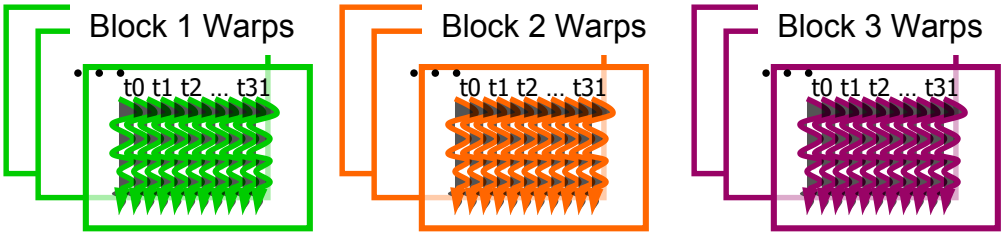
Module 5.1 – Thread Execution Efficiency

Warps and SIMD Hardware

Objective

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

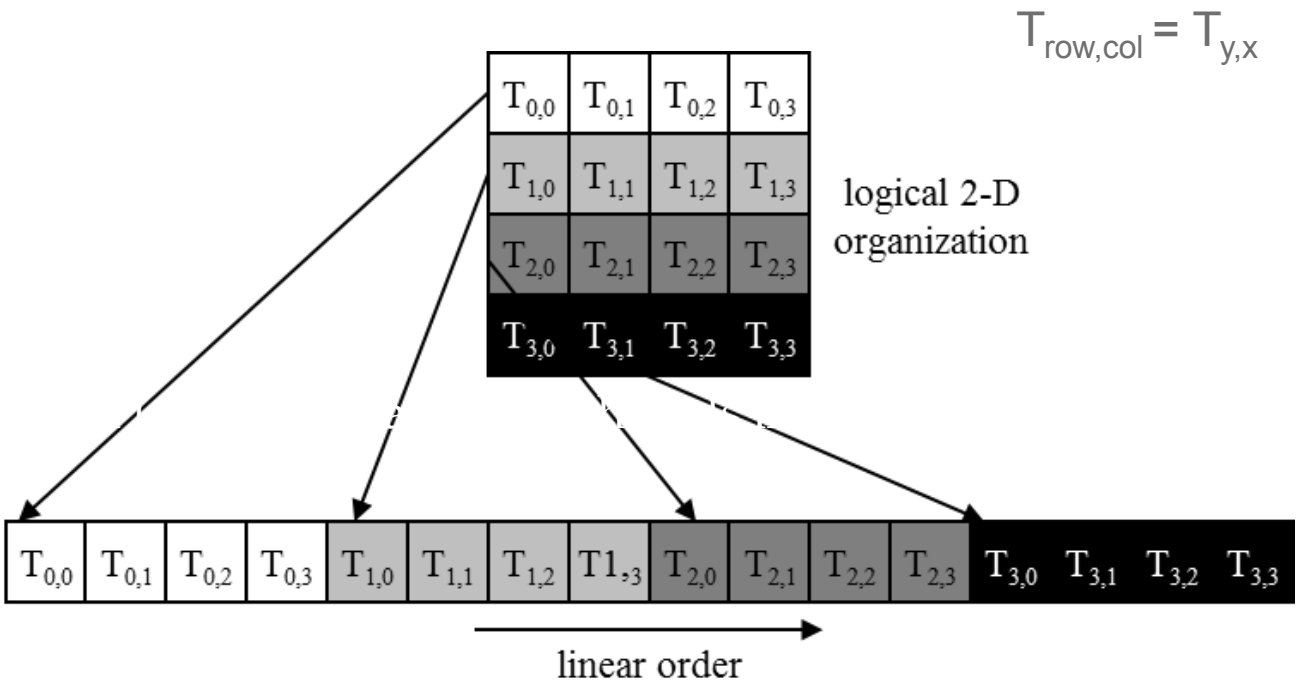
Warps as Scheduling Units



- 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



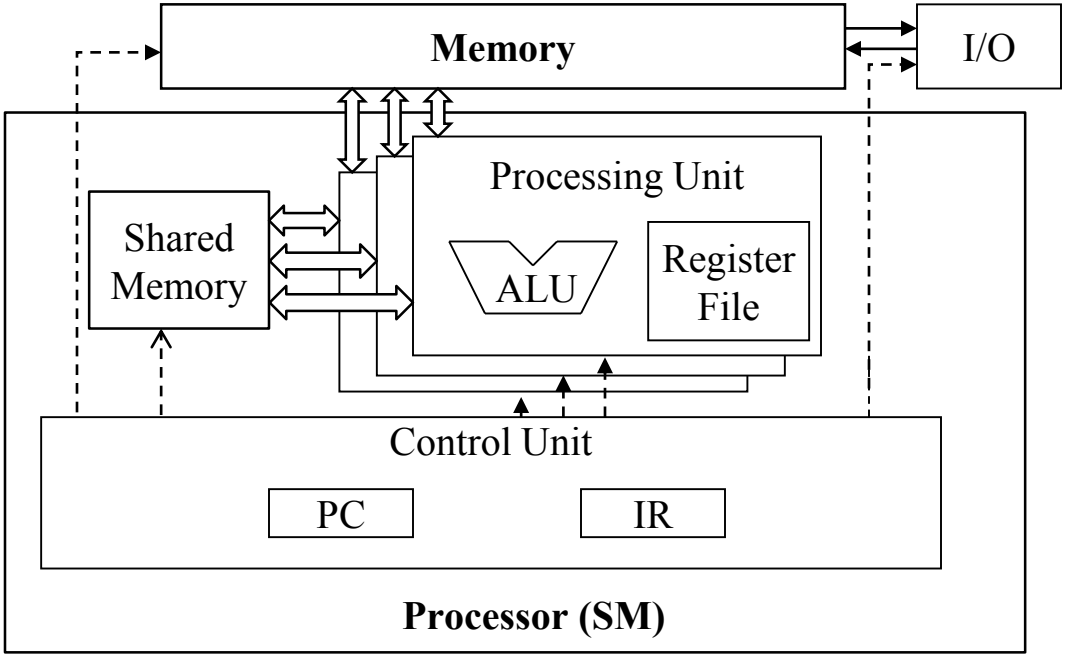
Blocks are partitioned after linearization

- 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).

NVIDIA ILLINOIS

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

NVIDIA ILLINOIS

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%

NVIDIA ILLINOIS

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.

GPU Teaching Kit

Accelerated Computing

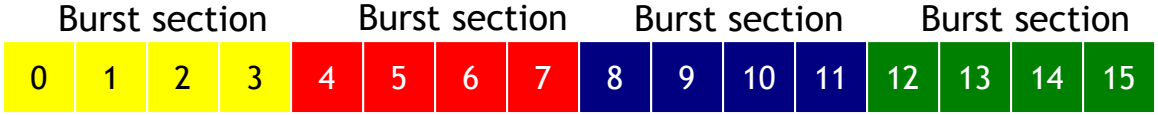
Lecture 6.2 – Performance Considerations

Memory Coalescing in CUDA

Objective

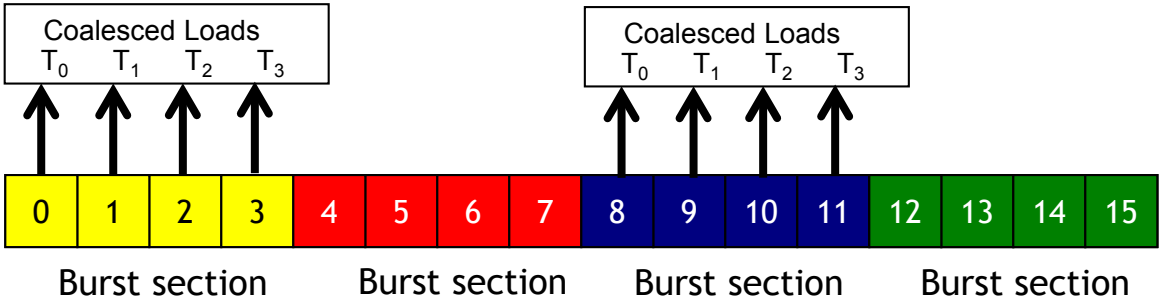
- To learn that memory coalescing is important for effectively utilizing memory bandwidth in CUDA
  - Its origin in DRAM burst
  - Checking if a CUDA memory access is coalesced
  - Techniques for improving memory coalescing in CUDA code

DRAM Burst – A System View



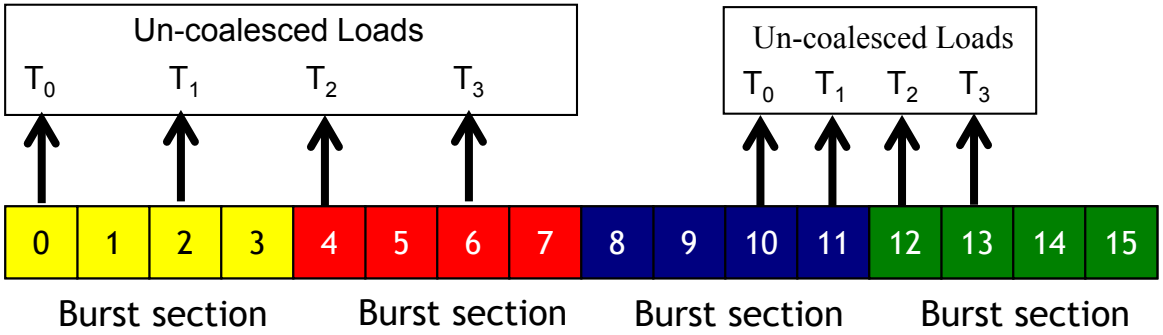
- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes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

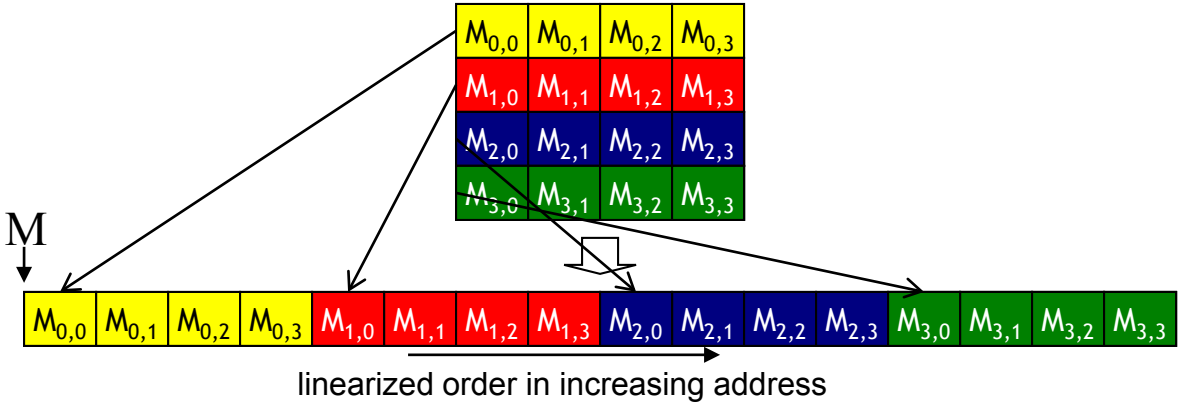


- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

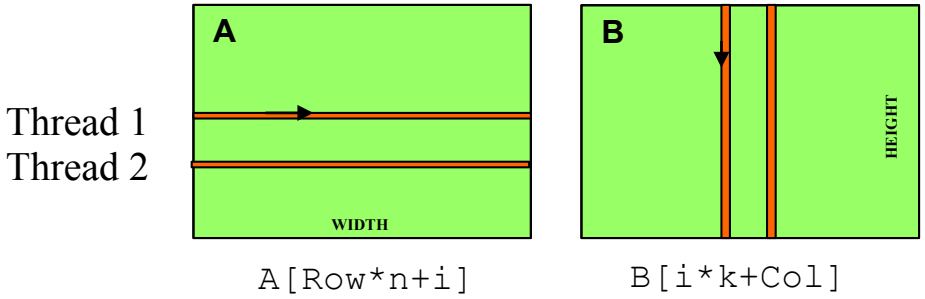
How to judge if an access is coalesced?

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

A 2D C Array in Linear Memory Space



Two Access Patterns of Basic Matrix Multiplication

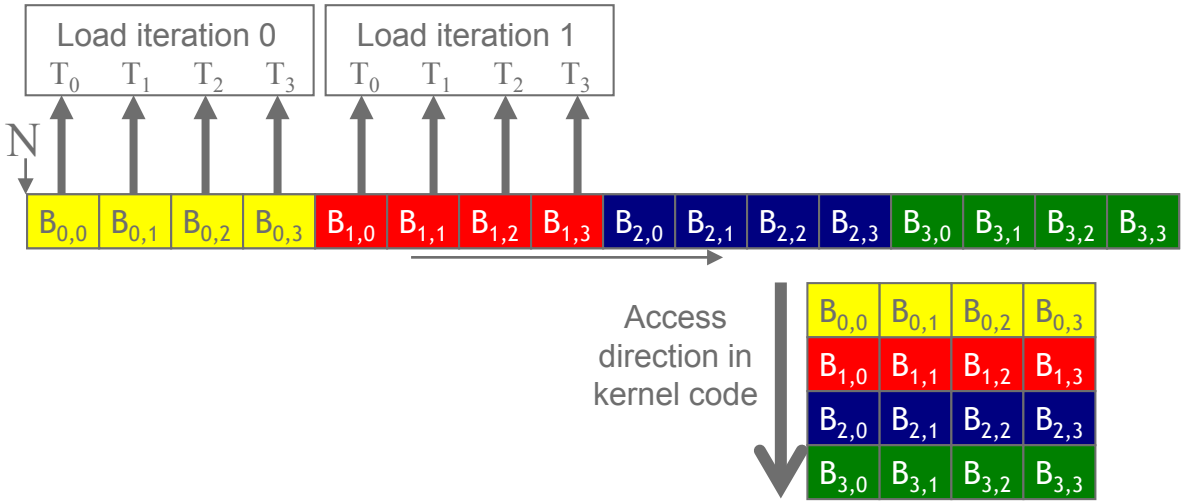


i is the loop counter in the inner product loop of the kernel code

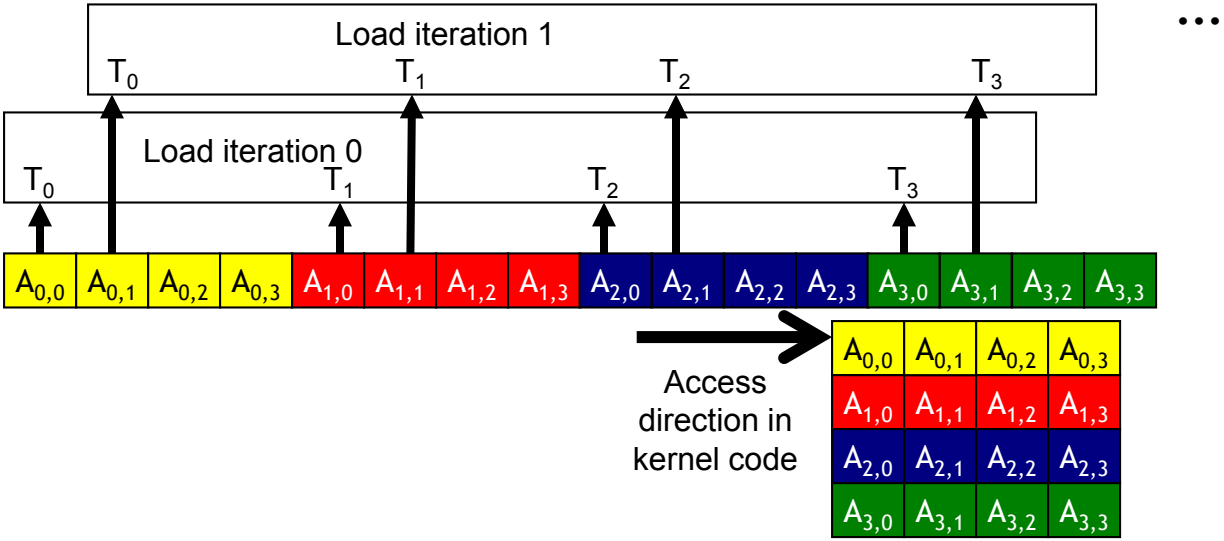
A is m × n, B is n × k

Col = blockIdx.x*blockDim.x + threadIdx.x

B accesses are coalesced



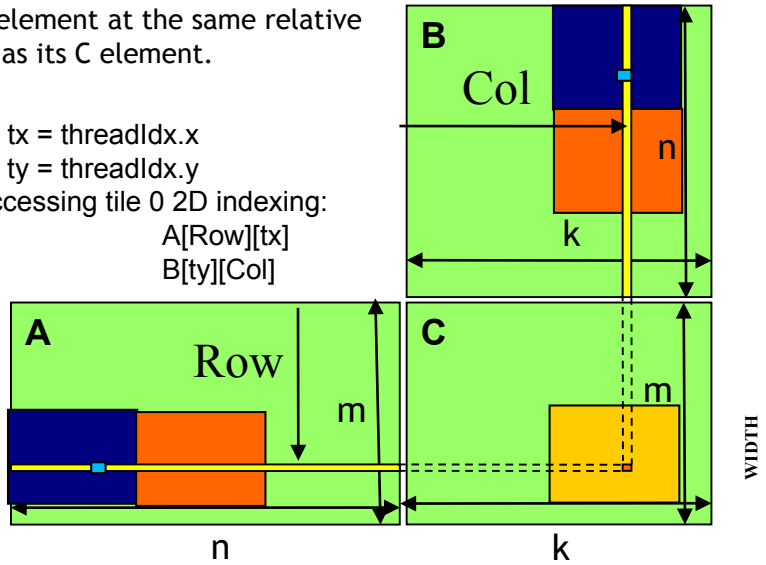
A Accesses are Not Coalesced



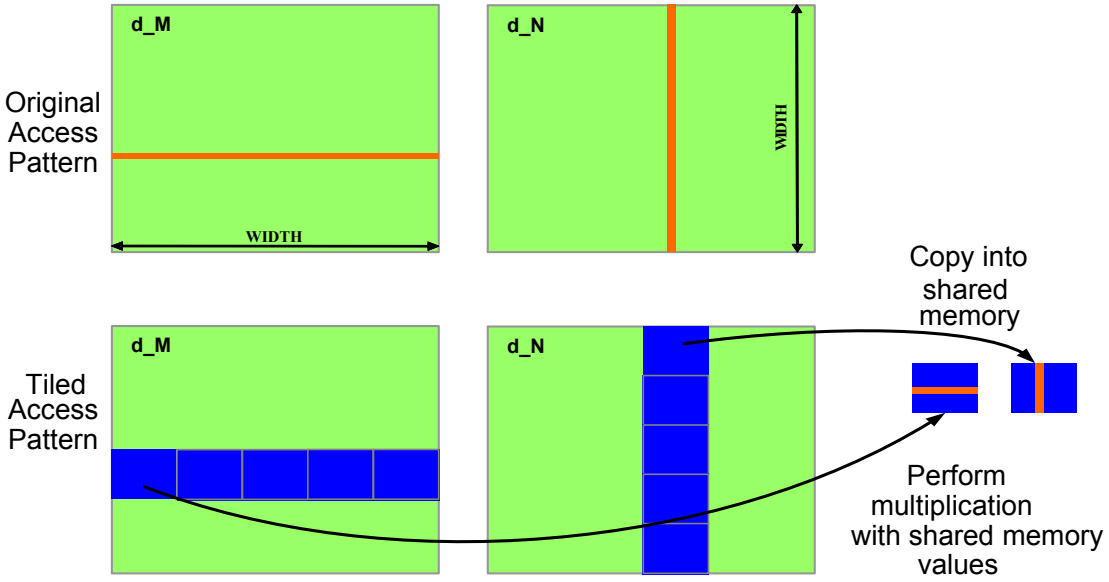
Loading an Input Tile

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.

GPU Teaching Kit

Accelerated Computing

Module 7.1 – Parallel Computation Patterns (Histogram)

Histogramming

Objective

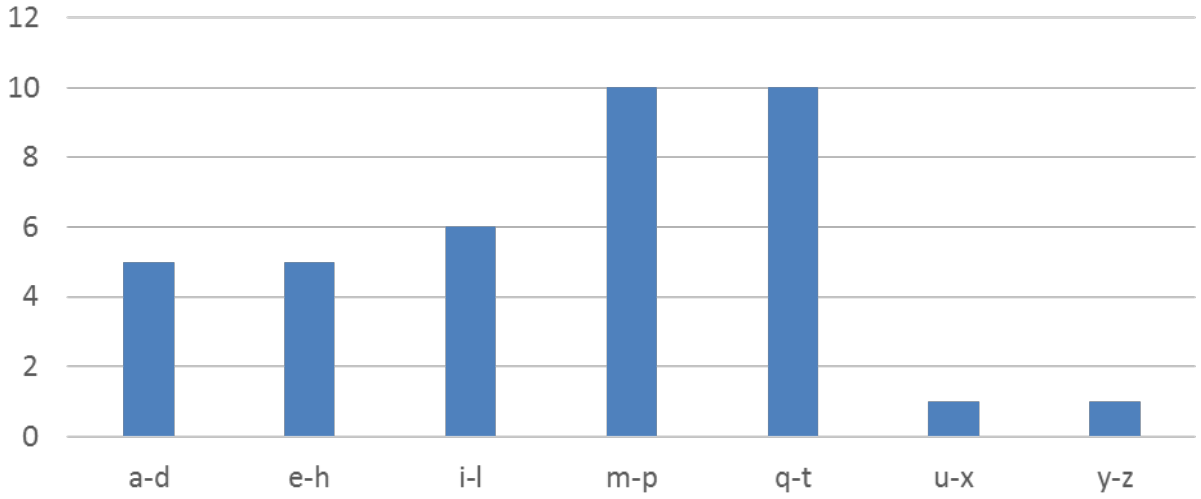
- To learn the parallel histogram computation pattern
  - An important, useful computation
  - Very different from all the patterns we have covered so far in terms of output behavior of each thread
  - A good starting point for understanding output interference in parallel computation

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

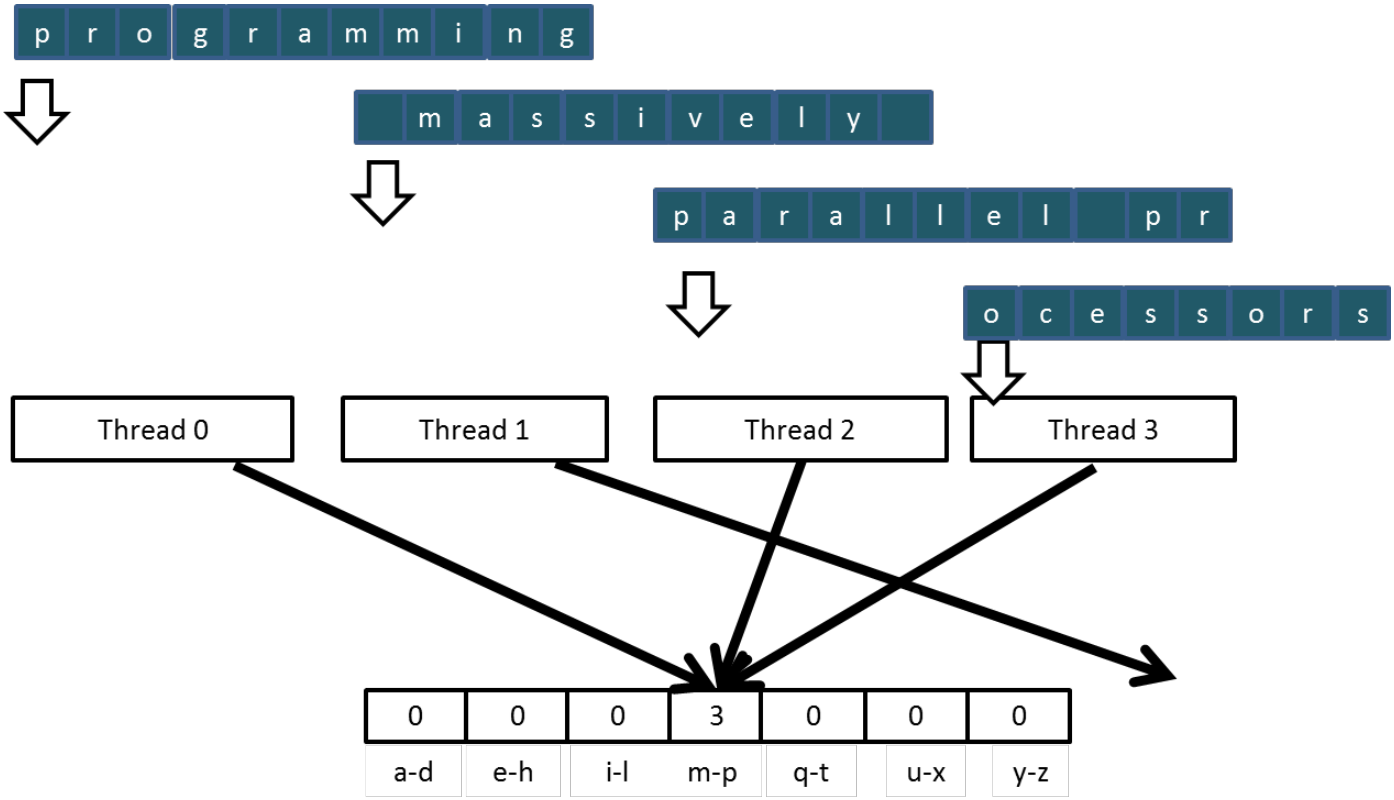
- Define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, n-p, …
- For each character in an input string, increment the appropriate bin counter.
- In the phrase “Programming Massively Parallel Processors” the output histogram is shown below:



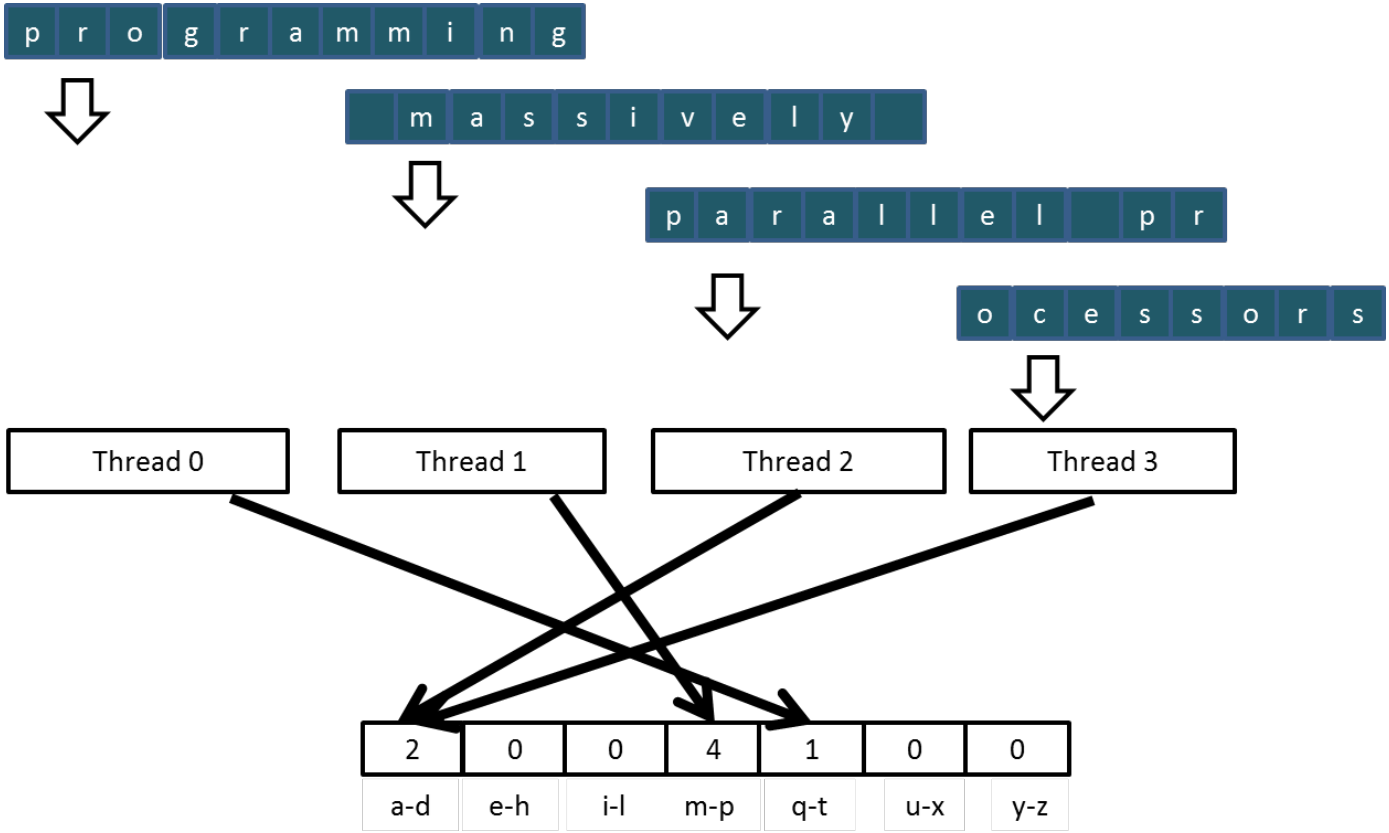
A simple parallel histogram algorithm

- 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)

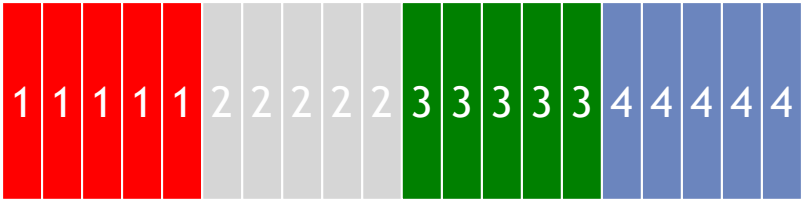


Input Partitioning Affects Memory Access Efficiency

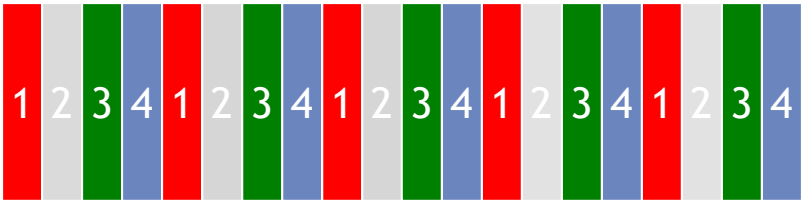
- Sectioned partitioning results in poor memory access efficiency
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized

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

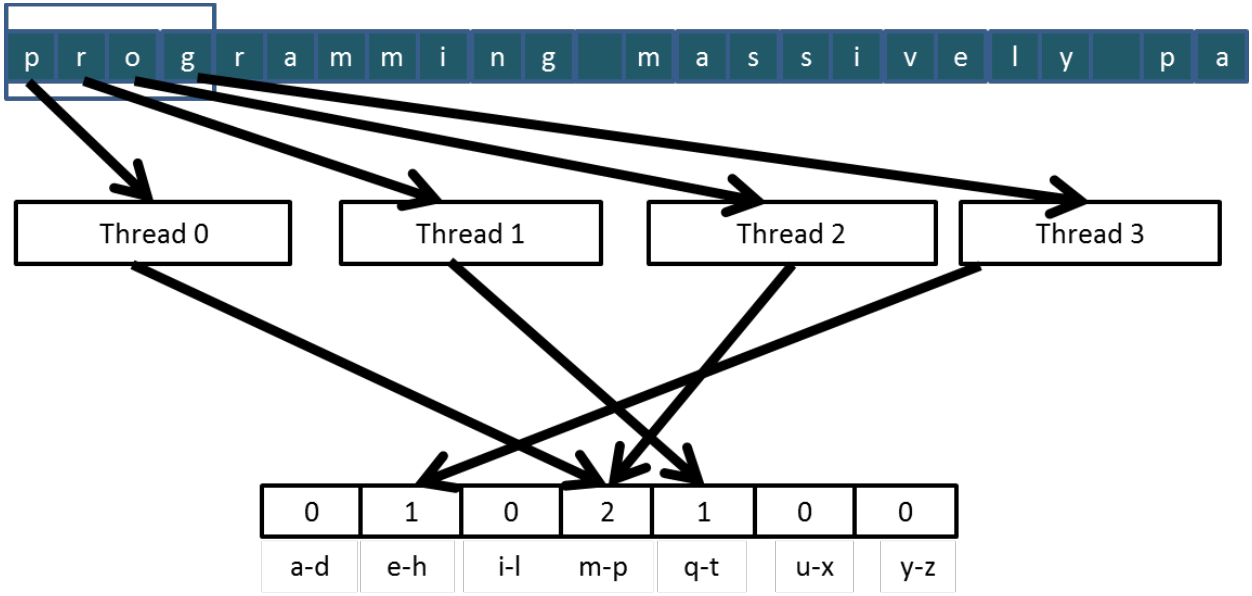


- Change to interleaved partitioning
  - All threads process a contiguous section of elements
  - They all move to the next section and repeat
  - The memory accesses are coalesced

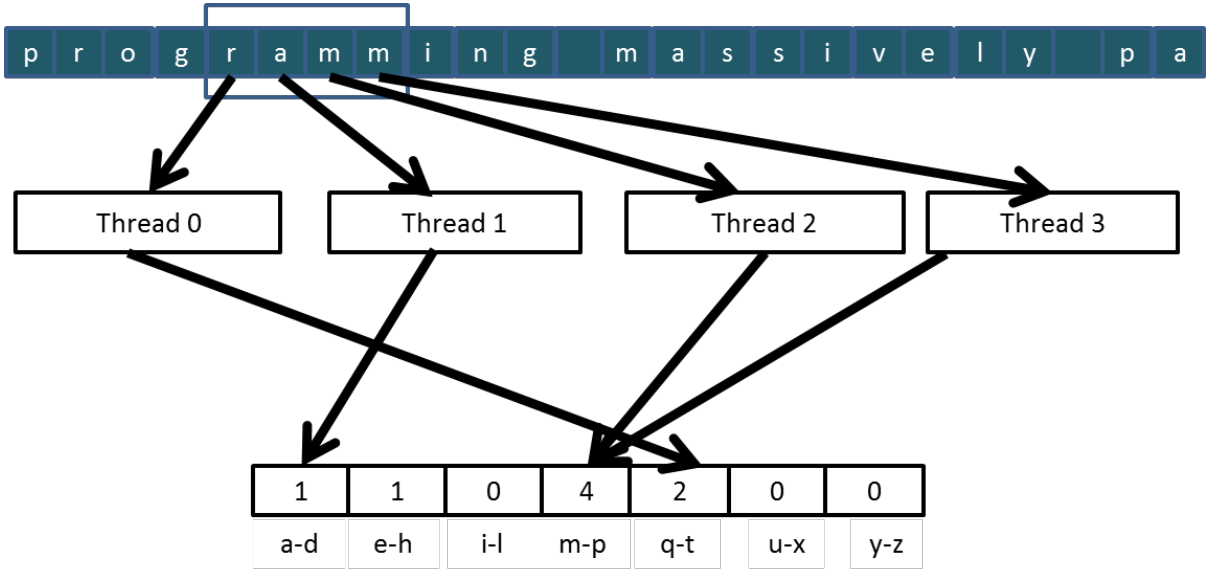


Interleaved Partitioning of Input

– 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.

GPU Teaching Kit

Accelerated Computing

Module 7.2 – Parallel Computation Patterns (Histogram)

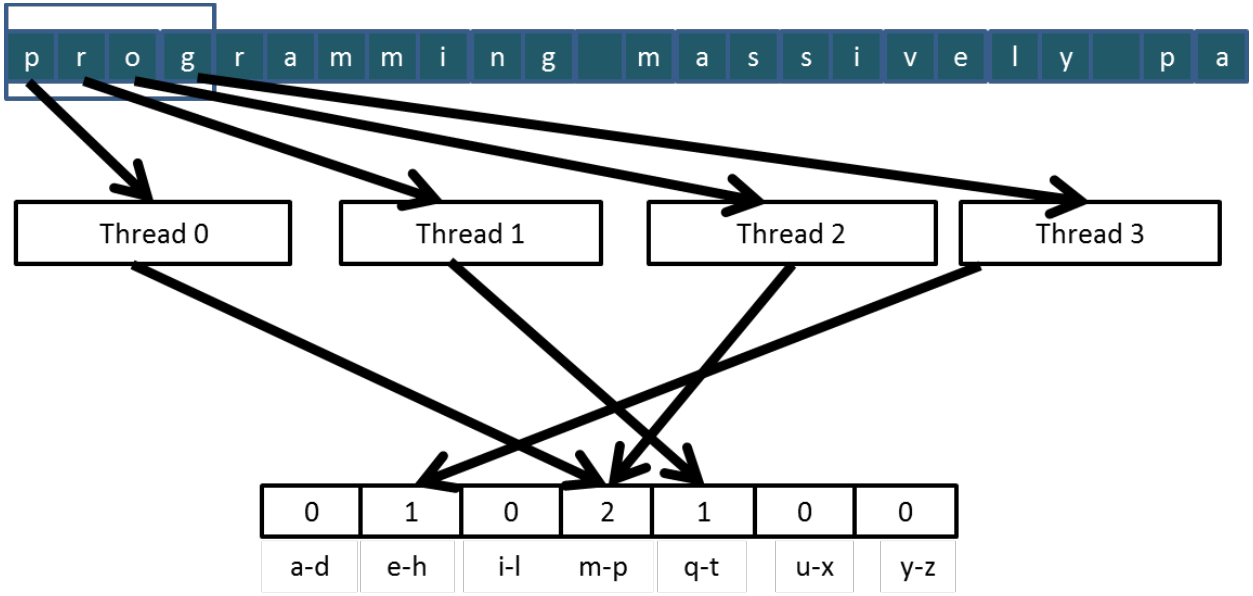
Introduction to Data Races

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

NVIDIA ILLINOIS

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: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Old and New are per-thread register variables.

Question 1: If Mem[x] was initially 0, what would the value of 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 ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← 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 ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← 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 ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

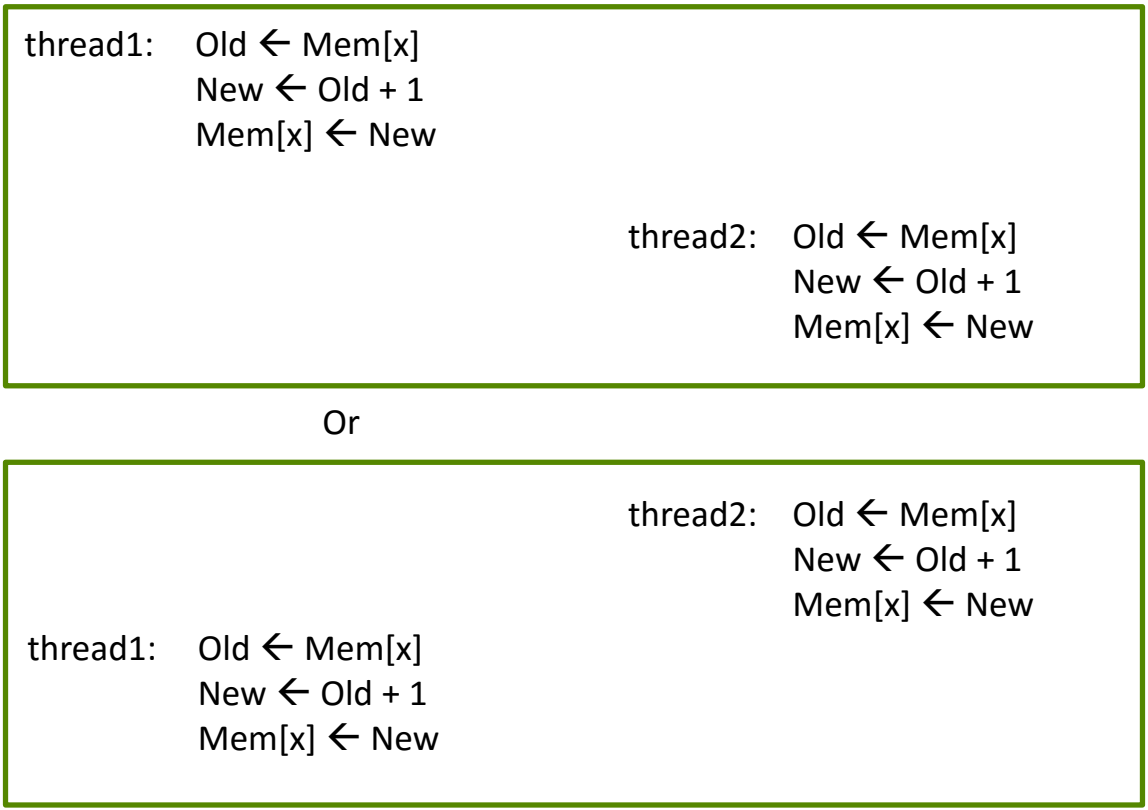
- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Timing Scenario #4

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

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

Purpose of Atomic Operations – To Ensure Good Outcomes



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.

GPU Teaching Kit

Accelerated Computing

Module 7.3 – Parallel Computation Patterns (Histogram)

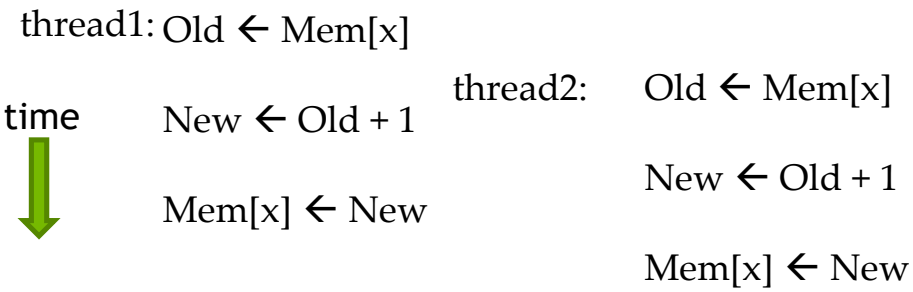
Atomic Operations in CUDA

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 && alpha_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.

GPU Teaching Kit

Accelerated Computing

Module 7.4 – Parallel Computation Patterns (Histogram)

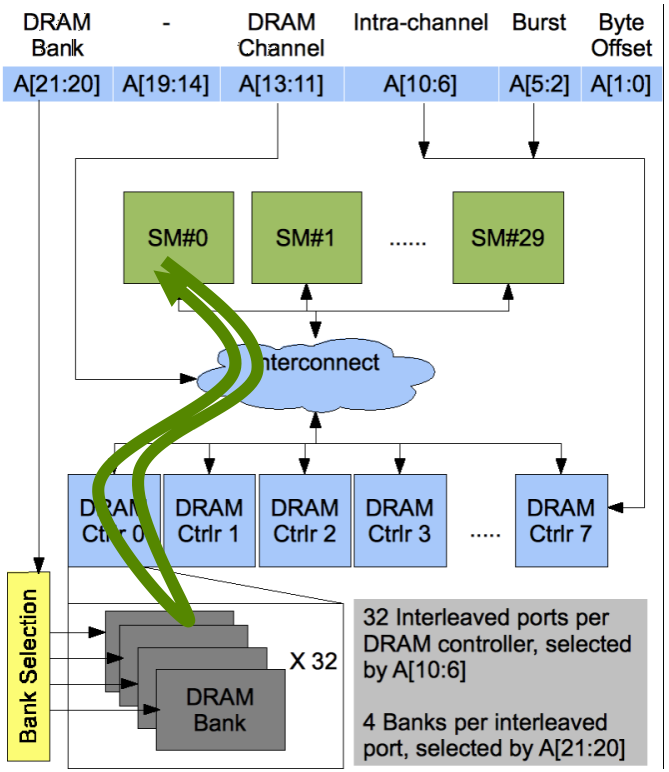
Atomic Operation Performance

Objective

- To learn about the main performance considerations of atomic operations
  - Latency and throughput of atomic operations
  - Atomic operations on global memory
  - Atomic operations on shared L2 cache
  - Atomic operations on shared memory

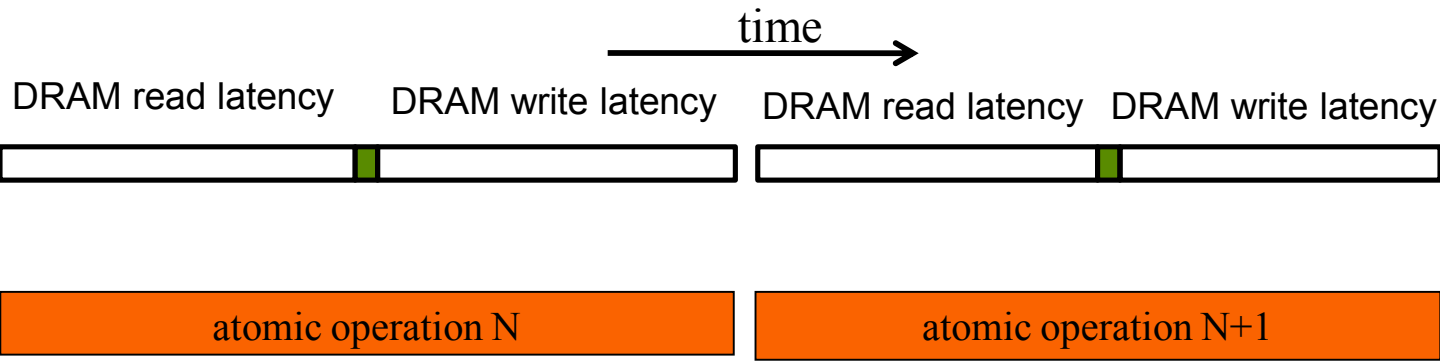
Atomic Operations on Global Memory (DRAM)

- An atomic operation on a DRAM location starts with a read, which has a latency of a few hundred cycles
- The atomic operation ends with a write to the same location, with a latency of a few hundred cycles
- During this whole time, no one else can access the location



Atomic Operations on DRAM

- Each Read-Modify-Write has two full memory access delays
  - All atomic operations on the same variable (DRAM location) are serialized



Latency determines throughput

- Throughput of atomic operations on the same DRAM location is the rate at which the application can execute an atomic operation.
- The rate for atomic operation on a particular location is limited by the total latency of the read-modify-write sequence, typically more than 1000 cycles for global memory (DRAM) locations.
- This means that if many threads attempt to do atomic operation on the same location (contention), the memory throughput is reduced to < 1/1000 of the peak bandwidth of one memory channel!

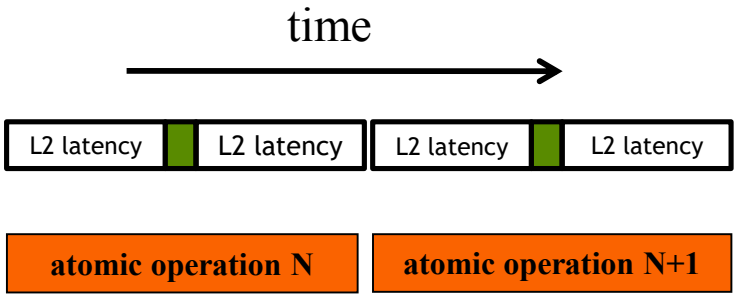
NVIDIA ILLINOIS

You may have a similar experience in supermarket checkout

- Some customers realize that they missed an item after they started to check out
- They run to the isle and get the item while the line waits
  - The rate of checkout is drastically reduced due to the long latency of running to the isle and back.
- Imagine a store where every customer starts the check out before they even fetch any of the items
  - The rate of the checkout will be 1 / (entire shopping time of each customer)

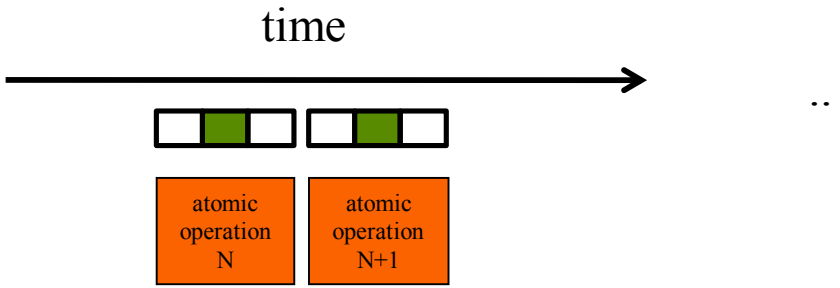
Hardware Improvements

- Atomic operations on Fermi L2 cache
  - Medium latency, about 1/10 of the DRAM latency
  - Shared among all blocks
  - “Free improvement” on Global Memory atomics



Hardware Improvements

- Atomic operations on Shared Memory
  - Very short latency
  - Private to each thread block
  - Need algorithm work by programmers (more later)



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.

GPU Teaching Kit

Accelerated Computing

Module 7.5 – Parallel Computation Patterns (Histogram)

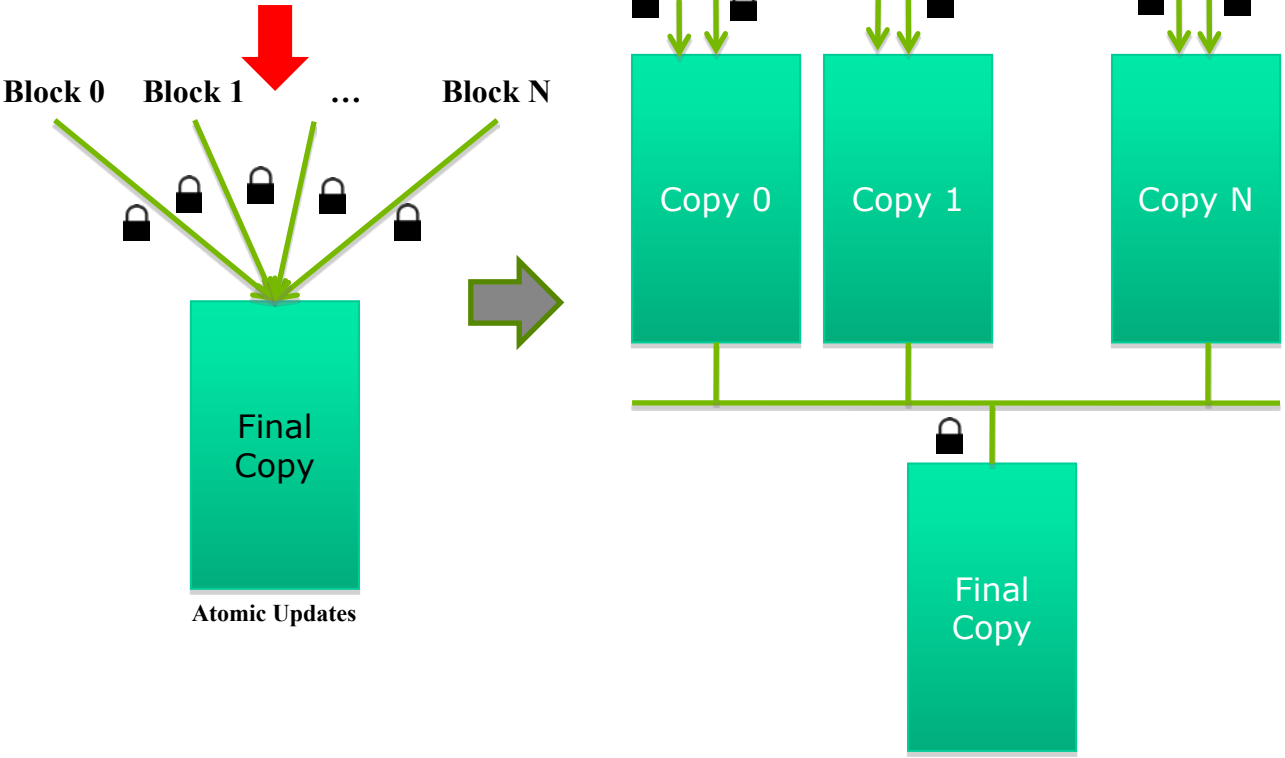
Privatization Technique for Improved Throughput

Objective

- Learn to write a high performance kernel by privatizing outputs
  - Privatization as a technique for reducing latency, increasing throughput, and reducing serialization
  - A high performance privatized histogram kernel
  - Practical example of using shared memory and L2 cache atomic operations

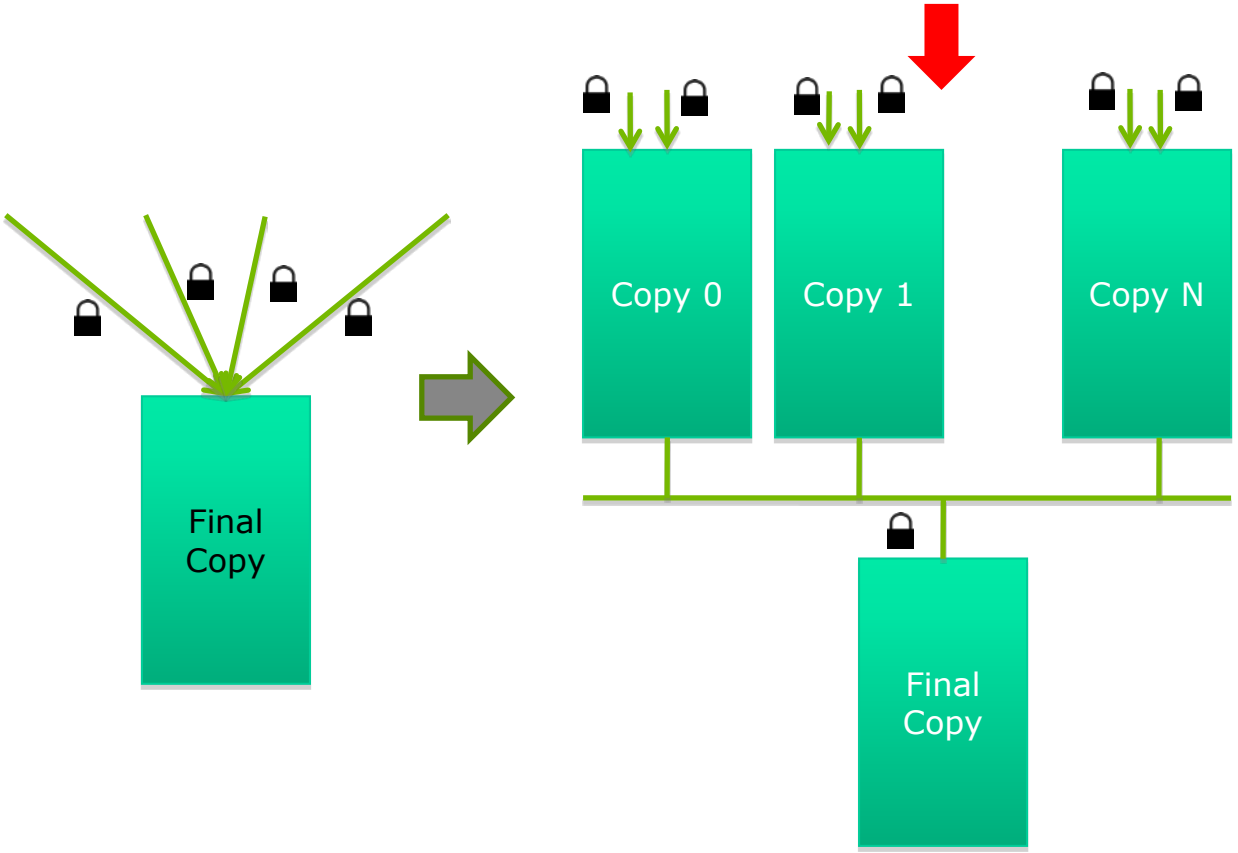
Privatization

Heavy contention and serialization

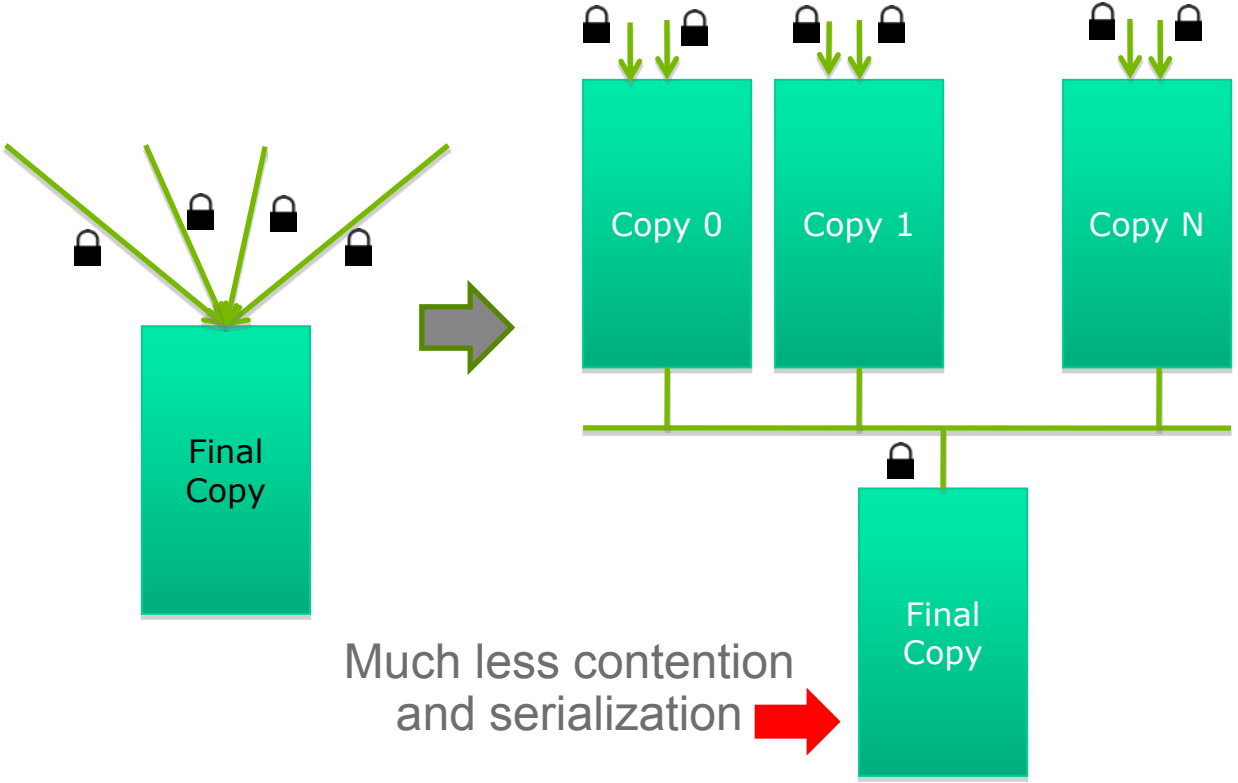


Privatization (cont.)

Much less contention and serialization



Privatization (cont.)



Cost and Benefit of Privatization

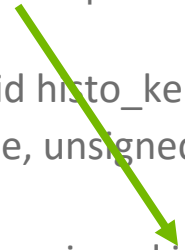
- Cost
  - Overhead for creating and initializing private copies
  - Overhead for accumulating the contents of private copies into the final copy
- Benefit
  - Much less contention and serialization in accessing both the private copies and the final copy
  - The overall performance can often be improved more than 10x

Shared Memory Atomics for Histogram

- Each subset of threads are in the same block
- Much higher throughput than DRAM (100x) or L2 (10x) atomics
- Less contention – only threads in the same block can access a shared memory variable
- This is a very important use case for shared memory!

Shared Memory Atomics Requires Privatization

– Create private copies of the histo[] array for each thread block



```
__global__ void histo_kernel(unsigned char *buffer,
        long size, unsigned int *histo)
{
   __shared__ unsigned int histo_private[7];
```

Shared Memory Atomics Requires Privatization

– Create private copies of the histo[] array for each thread block

```
__global__ void histo_kernel(unsigned char *buffer,
        long size, unsigned int *histo)
{
  __shared__ unsigned int histo_private[7];

  if (threadIdx.x < 7) histo_private[threadidx.x] = 0;
  __syncthreads();
```

Initialize the bin counters in the private copies of histo[]

Build Private Histogram

```
    int i = threadIdx.x + blockIdx.x * blockDim.x;
// stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &(private_histo[buffer[i]/4), 1);
        i += stride;
    }
```

Build Final Histogram

```
 // wait for all other threads in the block to finish
__syncthreads();

if (threadIdx.x < 7) {
      atomicAdd(&(histo[threadIdx.x]), private_histo[threadIdx.x] );
}

}
```

More on Privatization

- Privatization is a powerful and frequently used technique for parallelizing applications
- The operation needs to be associative and commutative
  - Histogram add operation is associative and commutative
  - No privatization if the operation does not fit the requirement
- The private histogram size needs to be small
  - Fits into shared memory
- What if the histogram is too large to privatize?
  - Sometimes one can partially privatize an output histogram and use range testing to go to either global memory or shared 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.