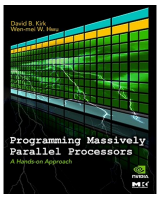
中国科学院大学计算机学院专业选修课

GPU架构与编程

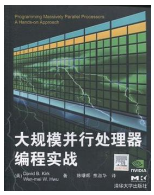
第三课：CUDA编程（一）

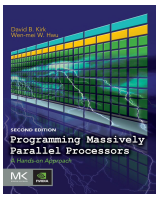
赵地
中科院计算所
2025年秋季学期

参考书：CUDA编程

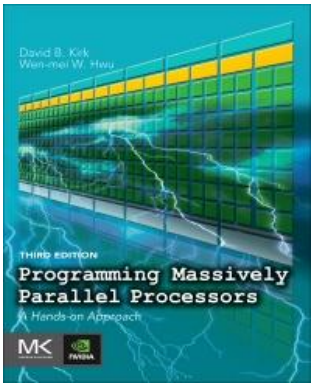


2010

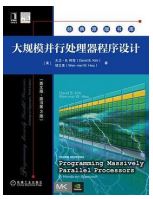


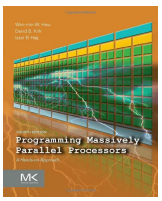


2012



2017





2022

✓**Third Edition:**

https://www.sciencedirect.com/book/9780128119860/programming-massively-parallel-processors

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach;

Contents

Preface xv
Acknowledgements xxi

CHAPTER 1 Introduction 1
1.1 Heterogeneous Parallel Computing 2
1.2 Architecture of a Modern GPU 6
1.3 Why More Speed or Parallelism? 8
1.4 Speeding Up Real Applications 10
1.5 Challenges in Parallel Programming 12
1.6 Parallel Programming Languages and Models 12
1.7 Overarching Goals 14
1.8 Organization of the Book 15
References 18

CHAPTER 2 Data Parallel Computing 19
2.1 Data Parallelism 20
2.2 CUDA C Program Structure 22
2.3 A Vector Addition Kernel 25
2.4 Device Global Memory and Data Transfer 27
2.5 Kernel Functions and Threading 32
2.6 Kernel Launch 37
2.7 Summary 38
Function Declarations 38
Kernel Launch 38
Built-in (Predefined) Variables 39
Run-time API 39
2.8 Exercises 39
References 41

CHAPTER 3 Scalable Parallel Execution 43
3.1 CUDA Thread Organization 43
3.2 Mapping Threads to Multidimensional Data 47
3.3 Image Blur: A More Complex Kernel 54
3.4 Synchronization and Transparent Scalability 58
3.5 Resource Assignment 60
3.6 Querying Device Properties 61
3.7 Thread Scheduling and Latency Tolerance 64
3.8 Summary 67
3.9 Exercises 67

CHAPTER 4 Memory and Data Locality 71
4.1 Importance of Memory Access Efficiency 72
4.2 Matrix Multiplication 73
4.3 CUDA Memory Types 77
4.4 Tiling for Reduced Memory Traffic 84
4.5 A Tiled Matrix Multiplication Kernel 90
4.6 Boundary Checks 94
4.7 Memory as a Limiting Factor to Parallelism 97
4.8 Summary 99
4.9 Exercises 100

CHAPTER 5 Performance Considerations 103
5.1 Global Memory Bandwidth 104
5.2 More on Memory Parallelism 112
5.3 Warps and SIMD Hardware 117
5.4 Dynamic Partitioning of Resources 125
5.5 Thread Granularity 127
5.6 Summary 128
5.7 Exercises 128
References 130

CHAPTER 6 Numerical Considerations 131
6.1 Floating-Point Data Representation 132
Normalized Representation of *M* 132
Excess Encoding of *E* 133
6.2 Representable Numbers 134
6.3 Special Bit Patterns and Precision in IEEE Format 138
6.4 Arithmetic Accuracy and Rounding 139
6.5 Algorithm Considerations 140
6.6 Linear Solvers and Numerical Stability 142
6.7 Summary 146
6.8 Exercises 147
References 147

CHAPTER 7 Parallel Patterns: Convolution 149
7.1 Background 150
7.2 1D Parallel Convolution—A Basic Algorithm 153
7.3 Constant Memory and Caching 156
7.4 Tiled 1D Convolution with Halo Cells 160
7.5 A Simpler Tiled 1D Convolution—General Caching 165
7.6 Tiled 2D Convolution with Halo Cells 166

7.7 Summary 172
7.8 Exercises 173

CHAPTER 8 Parallel Patterns: Prefix Sum 175
8.1 Background 176
8.2 A Simple Parallel Scan 177
8.3 Speed and Work Efficiency 181
8.4 A More Work-Efficient Parallel Scan 183
8.5 An Even More Work-Efficient Parallel Scan 187
8.6 Hierarchical Parallel Scan for Arbitrary-Length Inputs 189
8.7 Single-Pass Scan for Memory Access Efficiency 192
8.8 Summary 195
8.9 Exercises 195
References 196

CHAPTER 9 Parallel Patterns—Parallel Histogram Computation 199
9.1 Background 200
9.2 Use of Atomic Operations 202
9.3 Block versus Interleaved Partitioning 206
9.4 Latency versus Throughput of Atomic Operations 207
9.5 Atomic Operation in Cache Memory 210
9.6 Privatization 210
9.7 Aggregation 211
9.8 Summary 213
9.9 Exercises 213
Reference 214

CHAPTER 10 Parallel Patterns: Sparse Matrix Computation 215
10.1 Background 216
10.2 Parallel SpMV Using CSR 219
10.3 Padding and Transposition 221
10.4 Using a Hybrid Approach to Regulate Padding 224
10.5 Sorting and Partitioning for Regularization 227
10.6 Summary 229
10.7 Exercises 229
References 230

CHAPTER 11 Parallel Patterns: Merge Sort 231
11.1 Background 231
11.2 A Sequential Merge Algorithm 233
11.3 A Parallelization Approach 234
11.4 Co-Rank Function Implementation 236

11.5 A Basic Parallel Merge Kernel 241
11.6 A Tiled Merge Kernel 242
11.7 A Circular-Buffer Merge Kernel 249
11.8 Summary 256
11.9 Exercises 256
Reference 256

CHAPTER 12 Parallel Patterns: Graph Search 257
12.1 Background 258
12.2 Breadth-First Search 260
12.3 A Sequential BFS Function 262
12.4 A Parallel BFS Function 265
12.5 Optimizations 270
Memory Bandwidth 270
Hierarchical Queues 271
Kernel Launch Overhead 272
Load Balance 273
12.6 Summary 273
12.7 Exercises 273
References 274

CHAPTER 13 CUDA Dynamic Parallelism 275
13.1 Background 276
13.2 Dynamic Parallelism Overview 278
13.3 A Simple Example 279
13.4 Memory Data Visibility 281
Global Memory 281
Zero-Copy Memory 282
Constant Memory 282
Local Memory 282
Shared Memory 283
Texture Memory 283
13.5 Configurations and Memory Management 283
Launch Environment Configuration 283
Memory Allocation and Lifetime 283
Nesting Depth 284
Pending Launch Pool Configuration 284
Errors and Launch Failures 284
13.6 Synchronization, Streams, and Events 285
Synchronization 285
Synchronization Depth 285
Streams 286
Events 287

13.7 A More Complex Example 287
Linear Bezier Curves 288
Quadratic Bezier Curves 288
Bezier Curve Calculation (Without Dynamic Parallelism) 288
Bezier Curve Calculation (With Dynamic Parallelism) 290
Launch Pool Size 292
Streams 292
13.8 A Recursive Example 293
13.9 Summary 297
13.10 Exercises 299
References 301
A13.1 Code Appendix 301

CHAPTER 14 Application Case Study—non-Cartesian Magnetic Resonance Imaging 305
14.1 Background 306
14.2 Iterative Reconstruction 308
14.3 Computing F^HD 310
Step 1: Determine the Kernel Parallelism Structure 312
Step 2: Getting Around the Memory Bandwidth Limitation 317
Step 3: Using Hardware Trigonometry Functions 323
Step 4: Experimental Performance Tuning 326
14.4 Final Evaluation 327
14.5 Exercises 328
References 329

CHAPTER 15 Application Case Study—Molecular Visualization and Analysis 331
15.1 Background 332
15.2 A Simple Kernel Implementation 333
15.3 Thread Granularity Adjustment 337
15.4 Memory Coalescing 338
15.5 Summary 342
15.6 Exercises 343
References 344

CHAPTER 16 Application Case Study—Machine Learning 345
16.1 Background 346
16.2 Convolutional Neural Networks 347
ConvNets: Basic Layers 348
ConvNets: Backpropagation 351
16.3 Convolutional Layer: A Basic CUDA Implementation of Forward Propagation 355

16.4 Reduction of Convolutional Layer to Matrix Multiplication 359
16.5 cuDNN Library 364
16.6 Exercises 366
References 367

CHAPTER 17 Parallel Programming and Computational Thinking 369
17.1 Goals of Parallel Computing 370
17.2 Problem Decomposition 371
17.3 Algorithm Selection 374
17.4 Computational Thinking 379
17.5 Single Program, Multiple Data, Shared Memory and Locality 380
17.6 Strategies for Computational Thinking 382
17.7 A Hypothetical Example: Sodium Map of the Brain 383
17.8 Summary 386
17.9 Exercises 386
References 386

CHAPTER 18 Programming a Heterogeneous Computing Cluster 387
18.1 Background 388
18.2 A Running Example 388
18.3 Message Passing Interface Basics 391
18.4 Message Passing Interface Point-to-Point Communication 393
18.5 Overlapping Computation and Communication 400
18.6 Message Passing Interface Collective Communication 408
18.7 CUDA-Aware Message Passing Interface 409
18.8 Summary 410
18.9 Exercises 410
Reference 411

CHAPTER 19 Parallel Programming with OpenACC 413
19.1 The OpenACC Execution Model 414
19.2 OpenACC Directive Format 416
19.3 OpenACC by Example 418
The OpenACC Kernels Directive 419
The OpenACC Parallel Directive 422
Comparison of Kernels and Parallel Directives 424
OpenACC Data Directives 425
OpenACC Loop Optimizations 430
OpenACC Routine Directive 432
Asynchronous Computation and Data 434

19.4 Comparing OpenACC and CUDA 435
Portability 435
Performance 436
Simplicity 436
19.5 Interoperability with CUDA and Libraries 437
Calling CUDA or Libraries with OpenACC Arrays 437
Using CUDA Pointers in OpenACC 438
Calling CUDA Device Kernels from OpenACC 439
19.6 The Future of OpenACC 440
19.7 Exercises 441

CHAPTER 20 More on CUDA and Graphics Processing Unit Computing 443
20.1 Model of Host/Device Interaction 444
20.2 Kernel Execution Control 449
20.3 Memory Bandwidth and Compute Throughput 451
20.4 Programming Environment 453
20.5 Future Outlook 455
References 456

CHAPTER 21 Conclusion and Outlook 457
21.1 Goals Revisited 457
21.2 Future Outlook 458

Appendix A: An Introduction to OpenCL 461
Appendix B: THRUST: a Productivity-oriented Library for CUDA 475
Appendix C: CUDA Fortran 493
Appendix D: An introduction to C++ AMP 515

Index 535

讲授内容

- Introduction to CUDA C
- CUDA Parallelism Model
- Memory and Data Locality

讲授内容：Introduction to CUDA C

① **CUDA C vs. Thrust vs. CUDA Libraries**

② **Memory Allocation and Data Movement API Functions**

③ **Threads and Kernel Functions**

④ **Introduction to the CUDA Toolkit**

⑤ **Nsight Compute and Nsight Systems**

⑥ **Unified Memory**

Libraries: Easy, High-Quality Acceleration

```
//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

– 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

Numerical analytics ▶	MATLAB,, Mathematica, LabVIEW
Python ▶	PyCUDA, Numba
Fortran ▶	CUDA Fortran, OpenACC
C ▶	CUDA C, OpenACC
C++ ▶	CUDA C++, Thrust
C# ▶	Hybridizer

讲授内容：Introduction to CUDA C

① CUDA C vs. Thrust vs. CUDA Libraries

② Memory Allocation and Data Movement API Functions

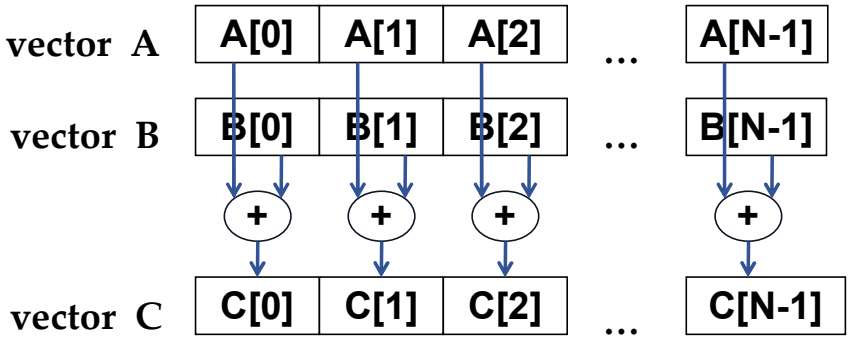
③ Threads and Kernel Functions

④ Introduction to the CUDA Toolkit

⑤ Nsight Compute and Nsight Systems

⑥ Unified Memory

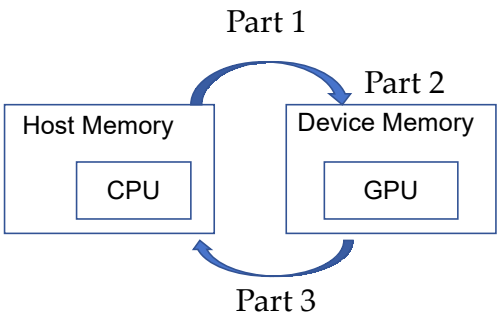
Data Parallelism - Vector Addition Example



Vector Addition – Traditional C Code

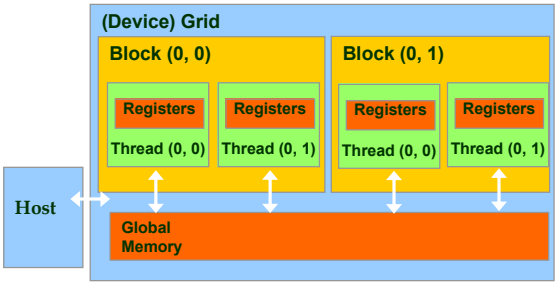
```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

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
}
```

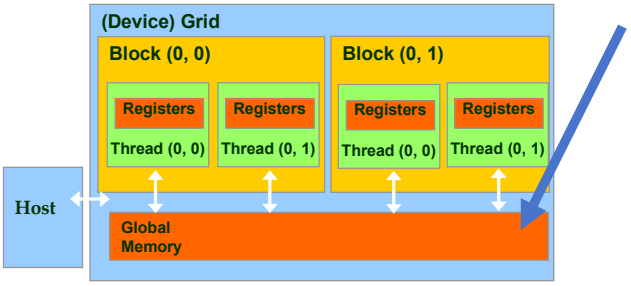

Partial Overview of CUDA Memories



- 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

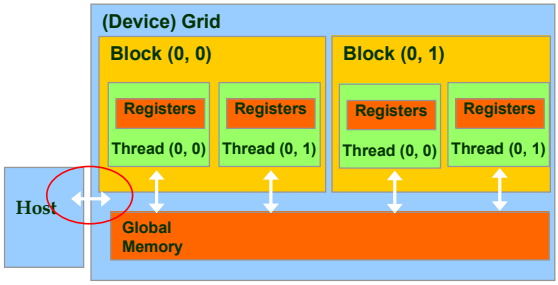
We will cover more memory types and more sophisticated memory models later.

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

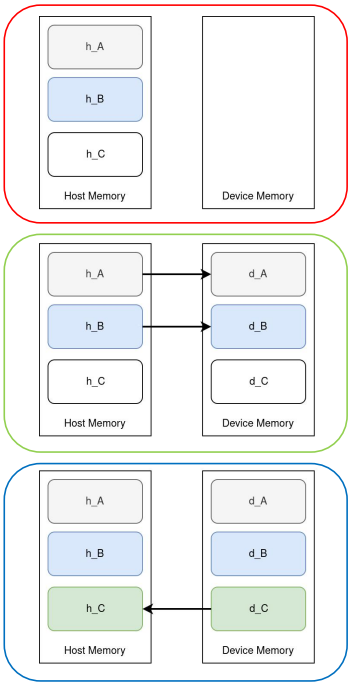
Host-Device Data Transfer API functions



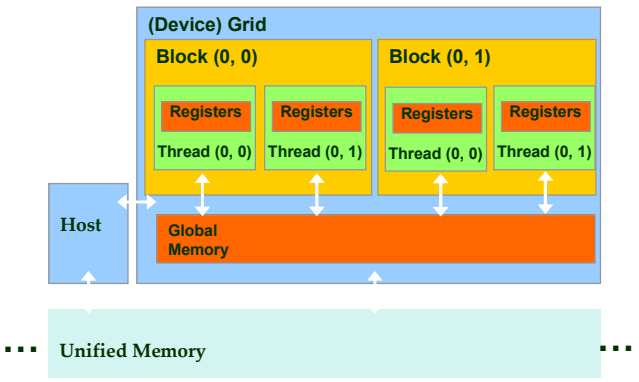
- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is synchronous with respect to the host

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 …
```



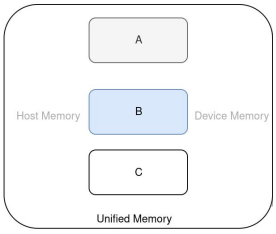
Unified Memory



- 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);

}
```

讲授内容：Introduction to CUDA C

① CUDA C vs. Thrust vs. CUDA Libraries

② Memory Allocation and Data Movement API Functions

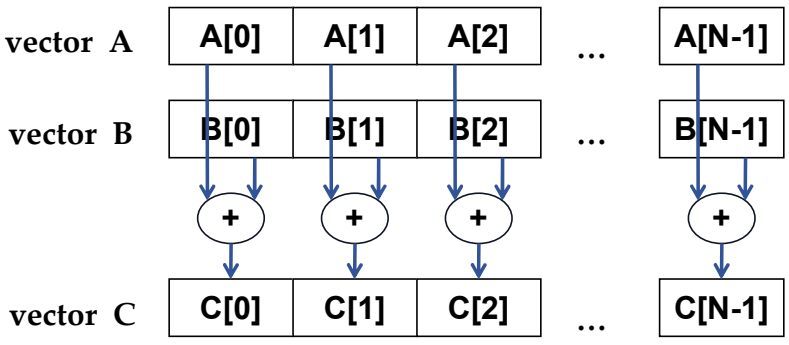
③ Threads and Kernel Functions

④ Introduction to the CUDA Toolkit

⑤ Nsight Compute and Nsight Systems

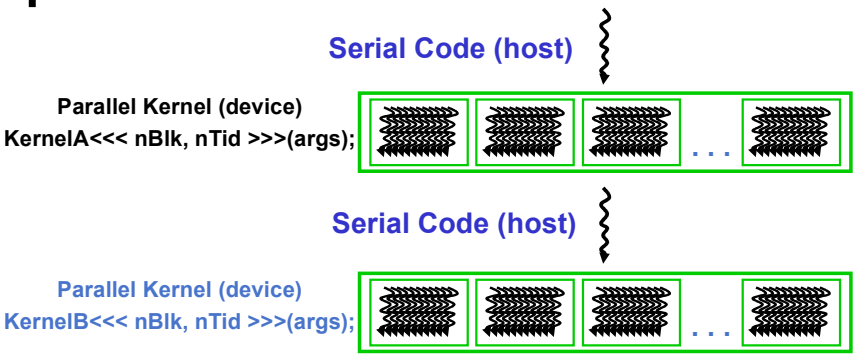
⑥ Unified Memory

Data Parallelism - Vector Addition Example



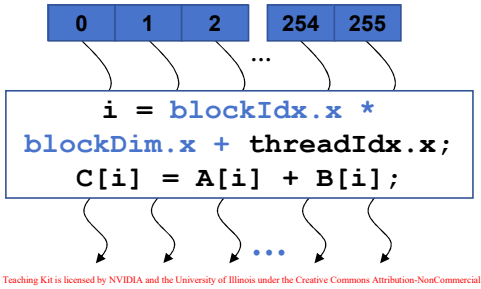
CUDA Execution Model

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

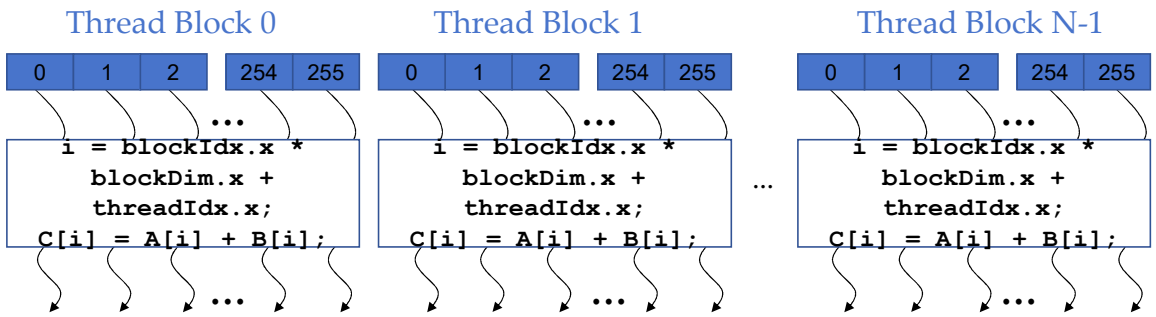


Arrays of Parallel Threads

- 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



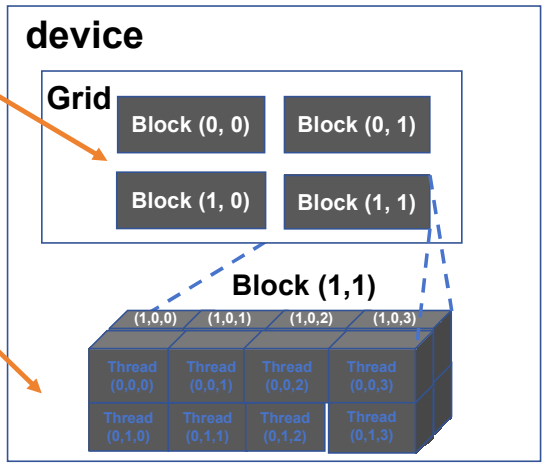
Thread Blocks: Scalable Cooperation



- 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
  - …



讲授内容：Introduction to CUDA C

① CUDA C vs. Thrust vs. CUDA Libraries

② Memory Allocation and Data Movement API Functions

③ Threads and Kernel Functions

④ Introduction to the CUDA Toolkit

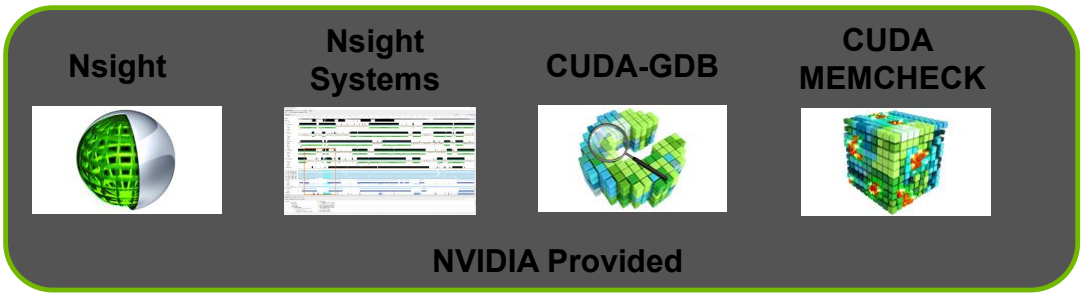
⑤ Nsight Compute and Nsight Systems

⑥ Unified Memory

NVCC Compiler

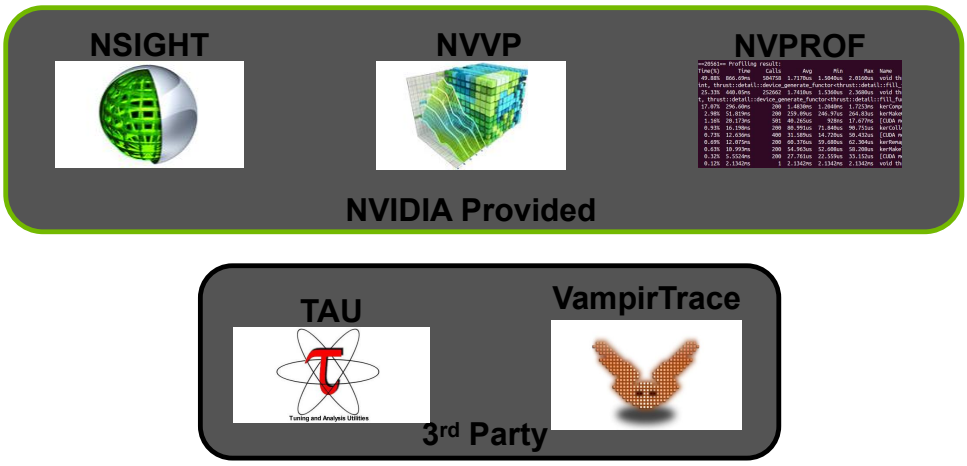
- NVIDIA provides a CUDA-C compiler
  - nvcc
- NVCC compiles device code then forwards code on to the host compiler (e.g. g++)
- Can be used to compile & link host only applications

Developer Tools - Debuggers



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

Developer Tools - Profilers



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

讲授内容：Introduction to CUDA C

① CUDA C vs. Thrust vs. CUDA Libraries

② Memory Allocation and Data Movement API Functions

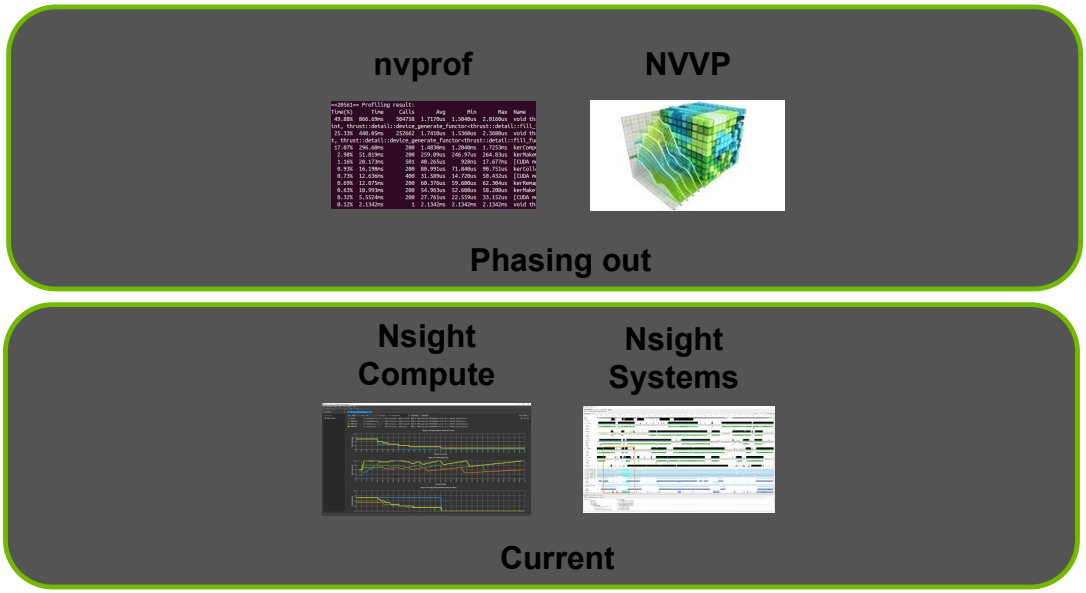
③ Threads and Kernel Functions

④ Introduction to the CUDA Toolkit

⑤ Nsight Compute and Nsight Systems

⑥ Unified Memory

Profiling Tools



讲授内容：Introduction to CUDA C

① CUDA C vs. Thrust vs. CUDA Libraries

② Memory Allocation and Data Movement API Functions

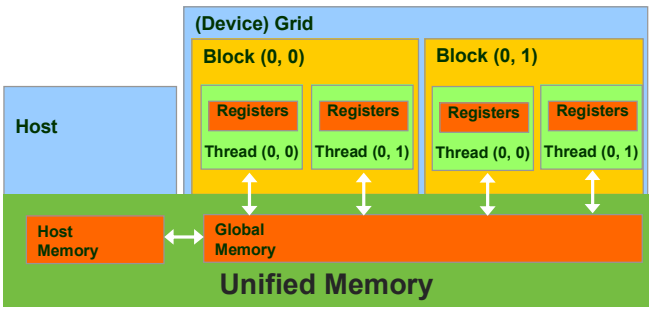
③ Threads and Kernel Functions

④ Introduction to the CUDA Toolkit

⑤ Nsight Compute and Nsight Systems

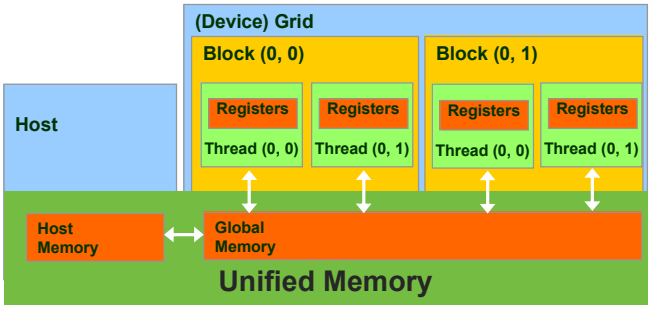
⑥ Unified Memory

Partial Overview of CUDA Memories



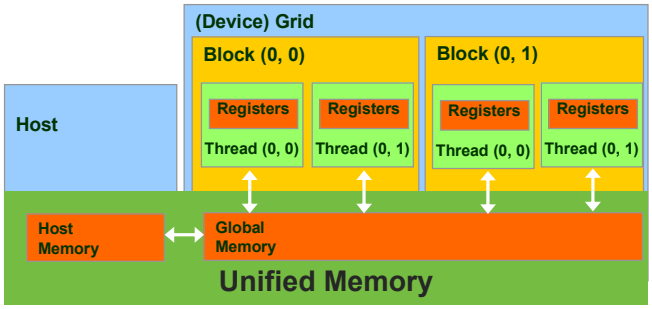
- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
  - R/W managed memory (Unified Memory)
- Host code can
  - Transfer data to/from per grid global memory
  - R/W managed memory

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);

    // Memory initialization on the Host

    // Kernel invocation code - to be shown
later

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

Allocation of Managed Memory

m_A, m_B gets initialized on the host

The device performs the actual vector addition

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.

讲授内容

- Introduction to CUDA C
- CUDA Parallelism Model
- Memory and Data Locality

讲授内容：CUDA Parallelism Model

① **Kernel-Based SPMD Parallel Programming**

② **Multidimensional Kernel Configuration**

③ **Color-to-Grayscale Image Processing Example**

④ **Image Blur Example**

⑤ **Thread Scheduling**

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.

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

讲授内容：CUDA Parallelism Model

① Kernel-Based SPMD Parallel Programming

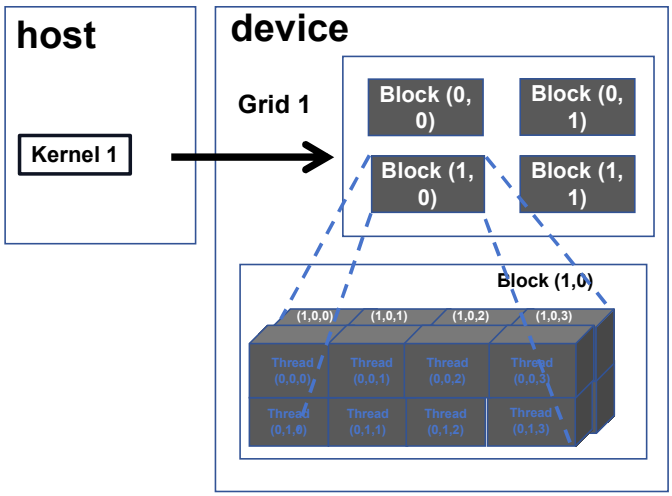
② Multidimensional Kernel Configuration

③ Color-to-Grayscale Image Processing Example

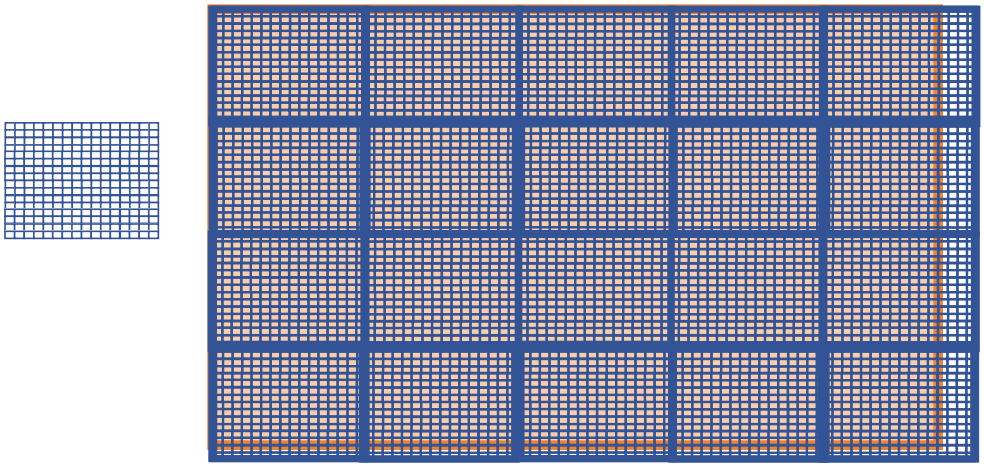
④ Image Blur Example

⑤ Thread Scheduling

A Multi-Dimensional Grid Example

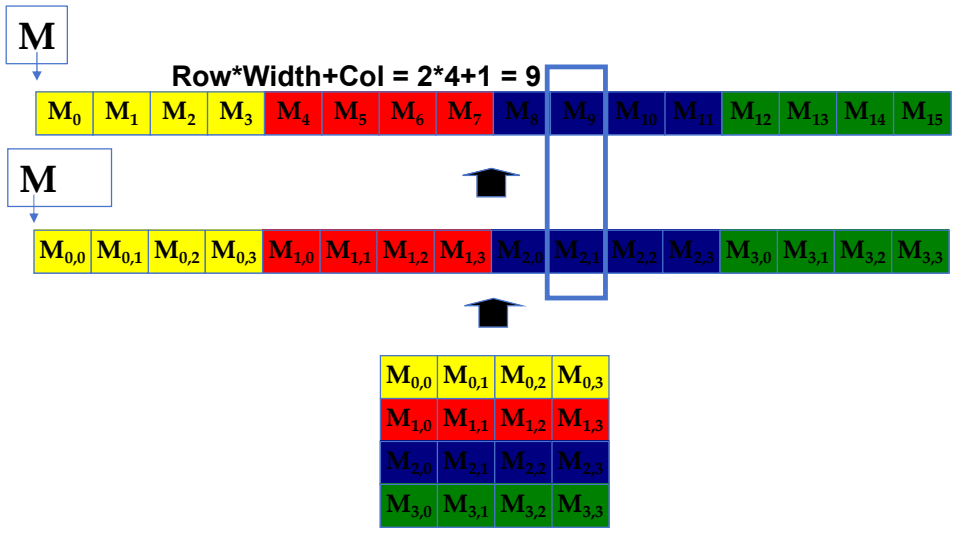


Processing a Picture with a 2D Grid



62×76 picture

Row-Major Layout in C/C++



Source Code of a PictureKernel

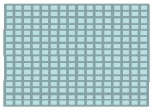
```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                              int height, int width)
{

  // 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];
  }
}
```

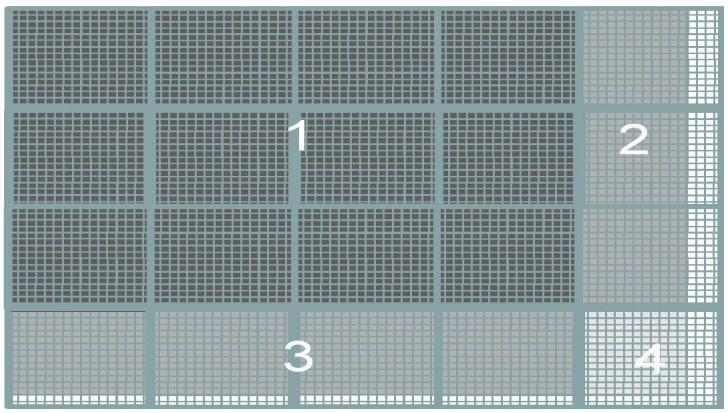
Scale every pixel value by 2.0

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.

讲授内容：CUDA Parallelism Model

① Kernel-Based SPMD Parallel Programming

② Multidimensional Kernel Configuration

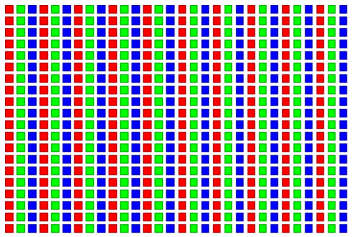
③ Color-to-Grayscale Image Processing Example

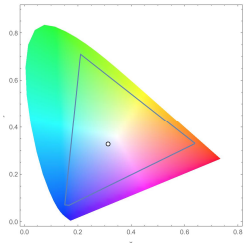
④ Image Blur Example

⑤ Thread Scheduling

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

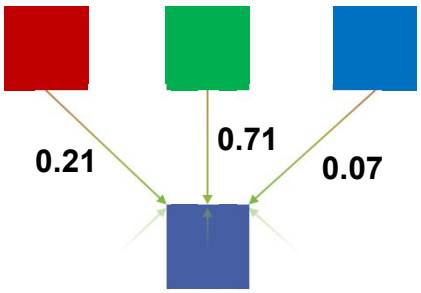
A grayscale digital image is an image in which the value of each pixel carries only intensity information.

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) {
...
 }
}
```

```
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
    // perform the rescaling and store it
    // We multiply by floating point constants
    grayImage[grayOffset] = 0.21f*r + 0.71f*g +
    0.07f*b;
}
```

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) {

...

 }

}
```

```
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

    // perform the rescaling and store it

    // We multiply by floating point constants

    grayImage[grayOffset] = 0.21f*r + 0.71f*g +
    0.07f*b;

}
```


RGB to Grayscale Conversion Code

```
// 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) {

...

 }

}
```

```
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

    // perform the rescaling and store it

    // We multiply by floating point constants

    grayImage[grayOffset] = 0.21f*r + 0.71f*g +
    0.07f*b;

}
```

讲授内容：CUDA Parallelism Model

① Kernel-Based SPMD Parallel Programming

② Multidimensional Kernel Configuration

③ Color-to-Grayscale Image Processing Example

④ Image Blur Example

⑤ Thread Scheduling

Image Blurring

Blurring Box

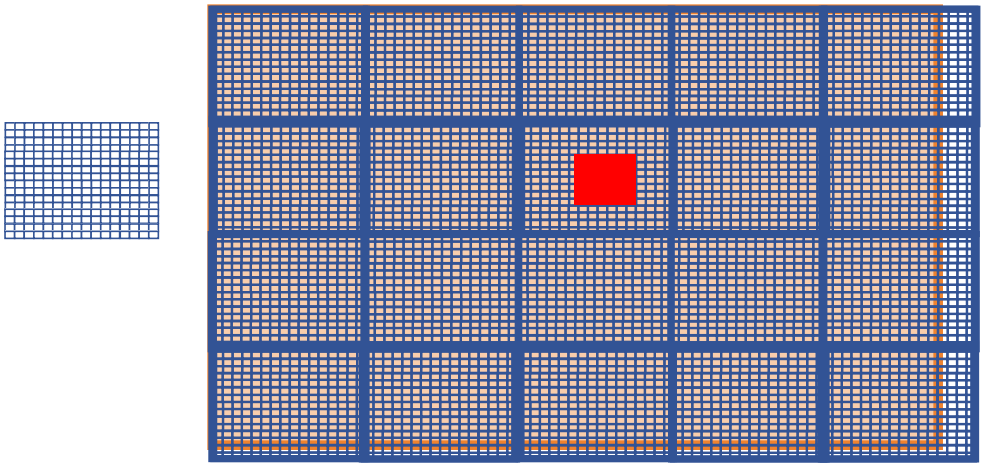


Image Blur as a 2D Kernel

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

```
      // 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
              }
          }
      }
```


讲授内容：CUDA Parallelism Model

① Kernel-Based SPMD Parallel Programming

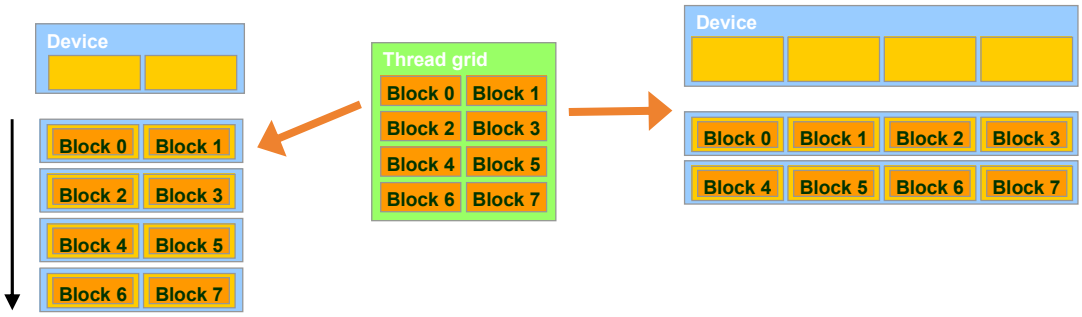
② Multidimensional Kernel Configuration

③ Color-to-Grayscale Image Processing Example

④ Image Blur Example

⑤ Thread Scheduling

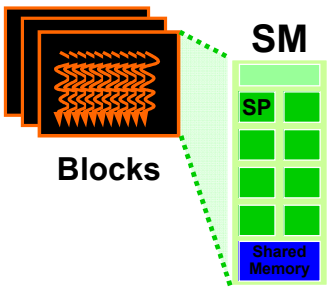
Transparent Scalability



- 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 (threads/block) * 8 blocks
    - Or 512 (threads/block) * 4 blocks, etc.
- SM maintains thread/block idx #s
- SM manages/schedules thread execution

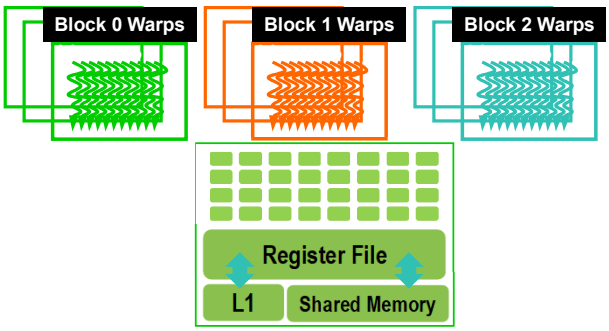


Warps as Scheduling Units

- 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.

讲授内容

- Introduction to CUDA C
- CUDA Parallelism Model
- Memory and Data Locality

讲授内容：Memory and Data Locality

① CUDA Memories

② Tiled Parallel Algorithms

③ Tiled Matrix Multiplication

④ Tiled Matrix Multiplication Kernel

⑤ Handling Arbitrary Matrix Sizes in Tiled Algorithms

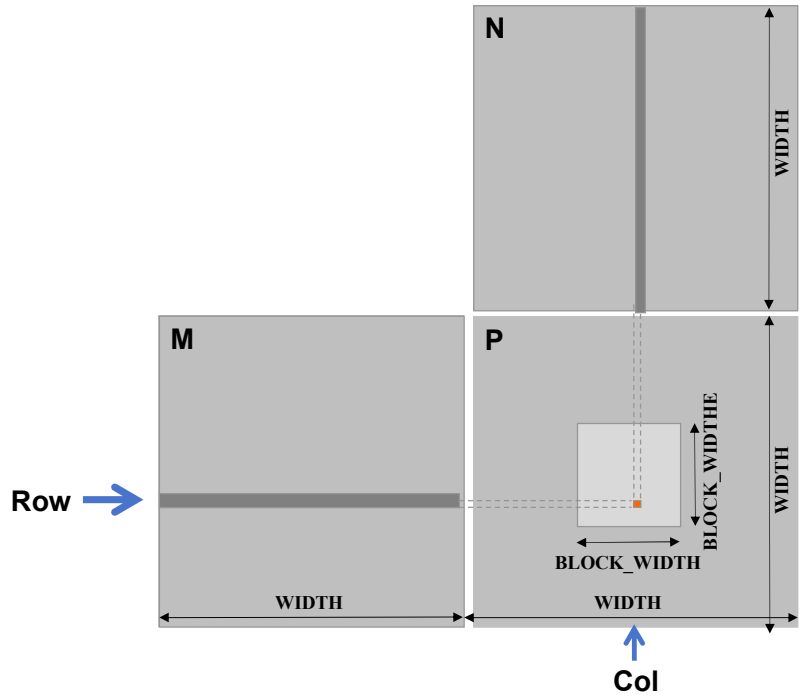
Review: Image Blur Kernel.

```
// 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



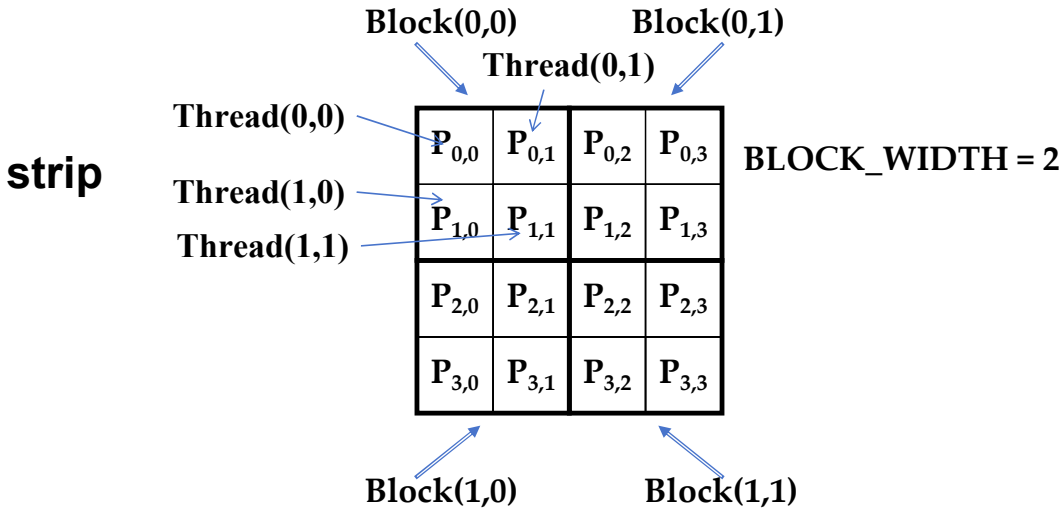
A Basic Matrix Multiplication

```
__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;
  }
}
```

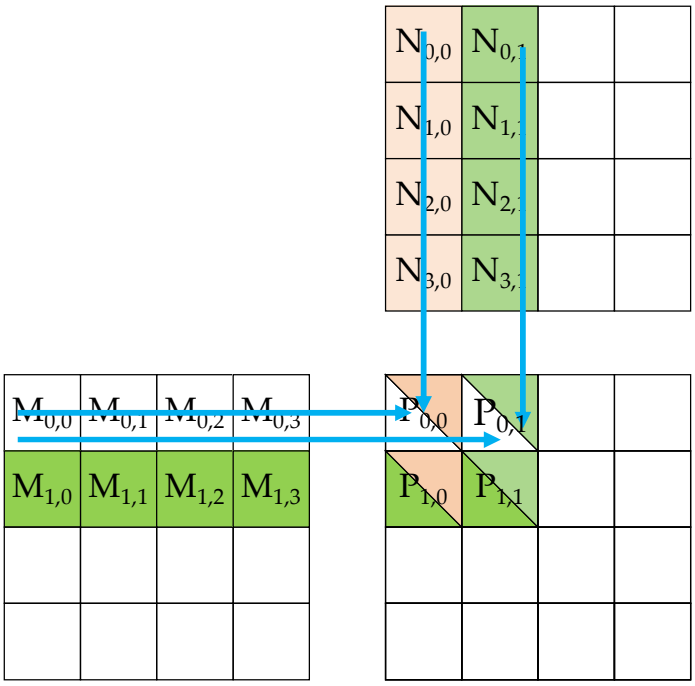
Example – Matrix Multiplication

```
__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;
  }
}
```

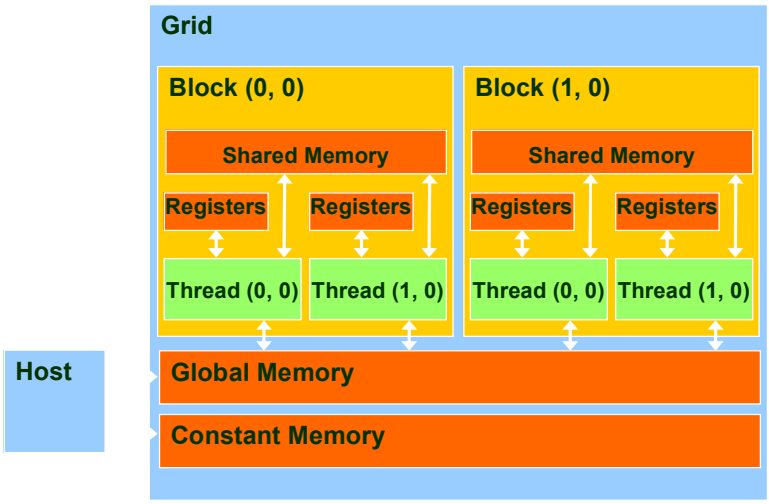

A Toy Example: Thread to P Data Mapping



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



Programmer View of CUDA Memories



Declaring CUDA Variables

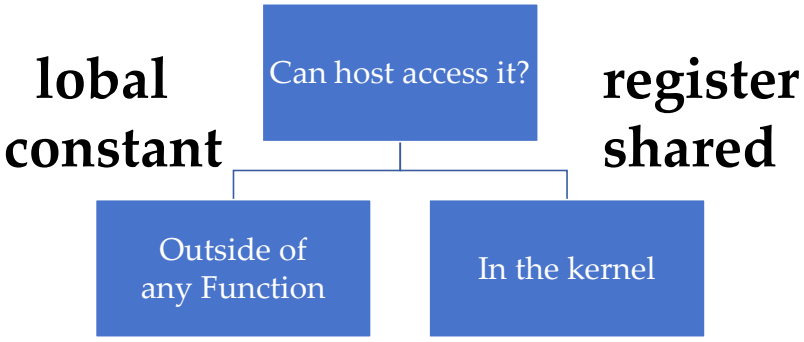
Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- __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

讲授内容：Memory and Data Locality

① CUDA Memories

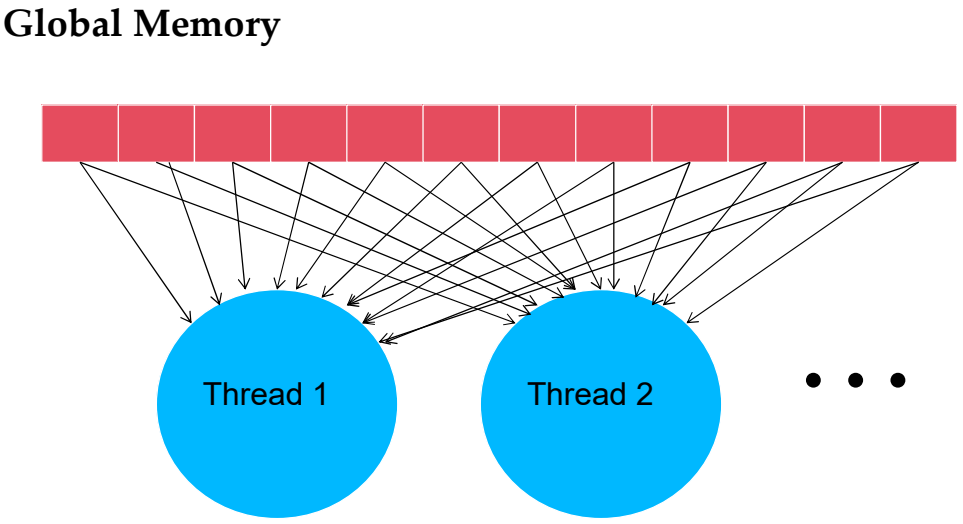
② Tiled Parallel Algorithms

③ Tiled Matrix Multiplication

④ Tiled Matrix Multiplication Kernel

⑤ Handling Arbitrary Matrix Sizes in Tiled Algorithms

Global Memory Access Pattern of the Basic Matrix Multiplication Kernel



Basic Concept of Tiling

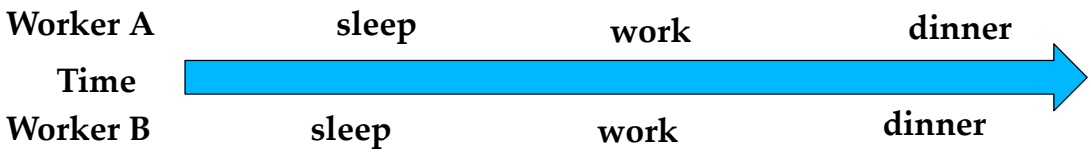
- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
  - Carpooling for commuters
  - Tiling for global memory accesses
    - drivers = threads accessing their memory data operands
    - cars = memory access requests

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

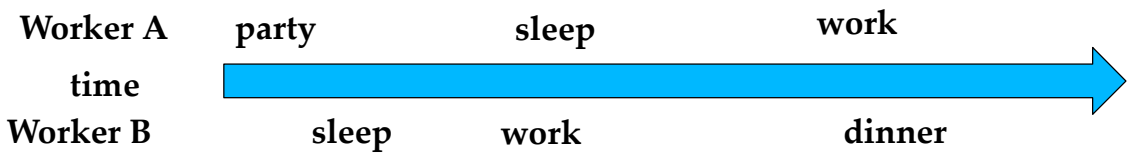
Carpools need synchronization.

– Good: when people have similar schedule



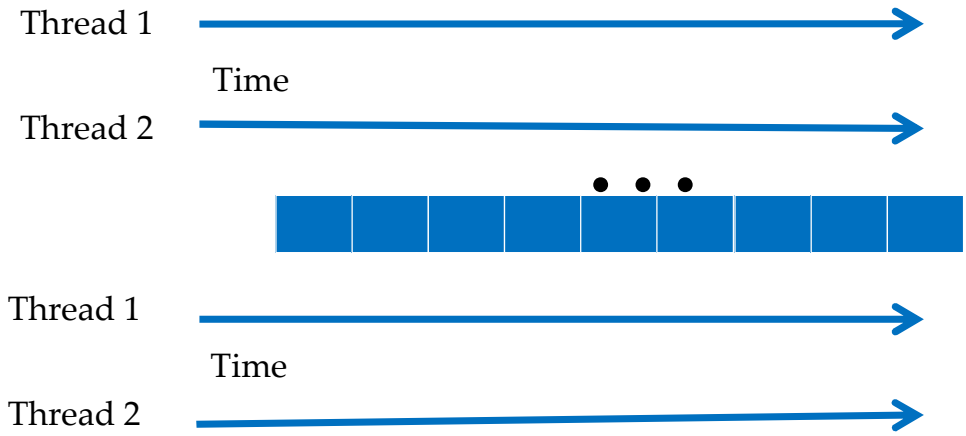
Carpools need synchronization.

– Bad: when people have very different schedule



Same with Tiling

– Good: when threads have similar access timing



– Bad: when threads have very different timing

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

讲授内容：Memory and Data Locality

① CUDA Memories

② Tiled Parallel Algorithms

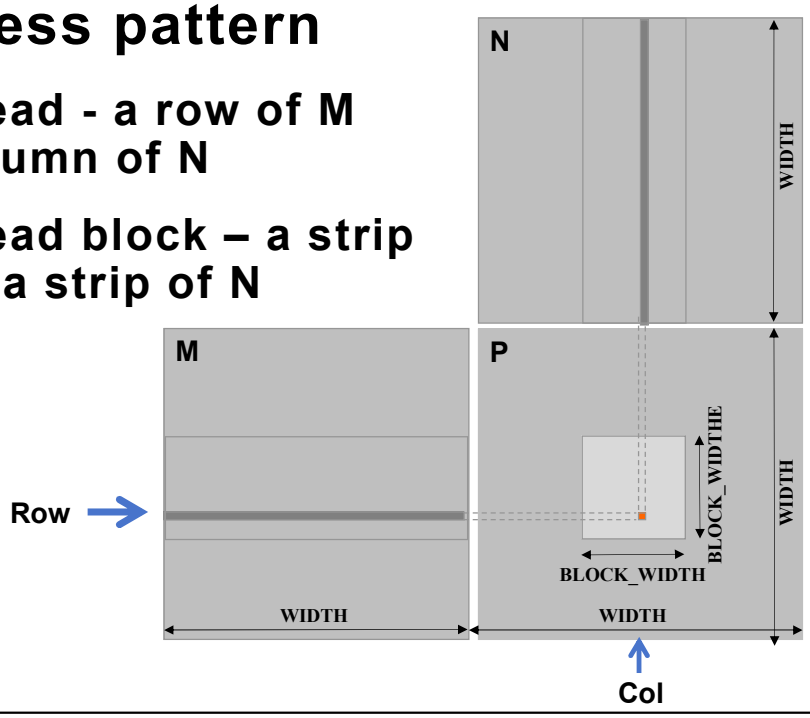
③ Tiled Matrix Multiplication

④ Tiled Matrix Multiplication Kernel

⑤ Handling Arbitrary Matrix Sizes in Tiled Algorithms

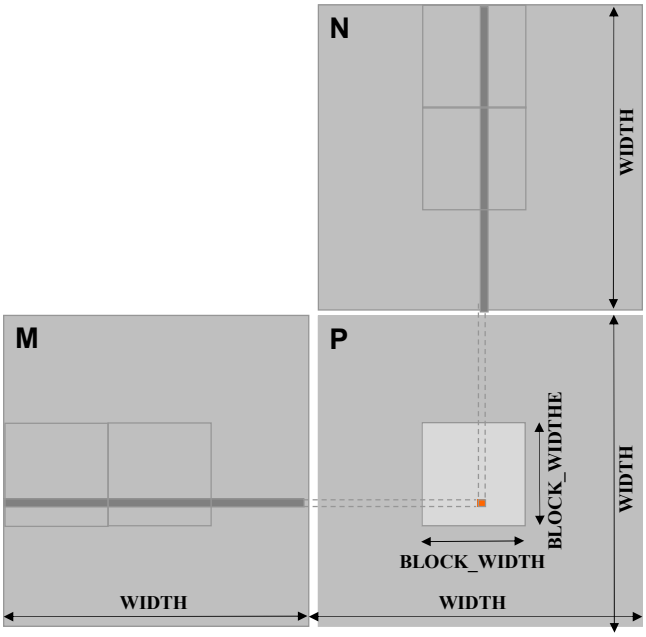
Matrix Multiplication

- 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

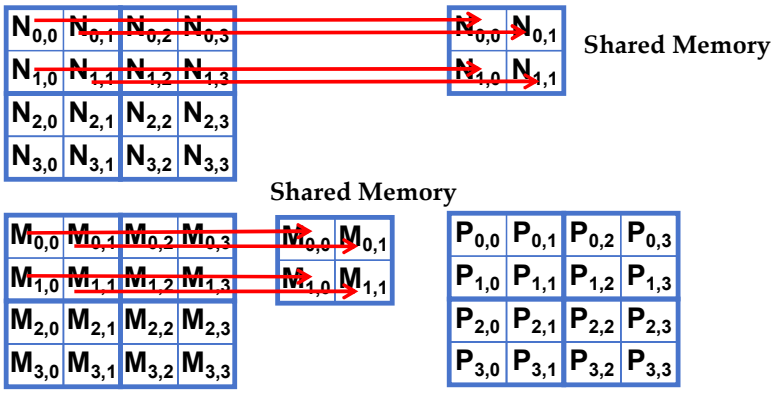
- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of BLOCK_SIZE elements in each dimension



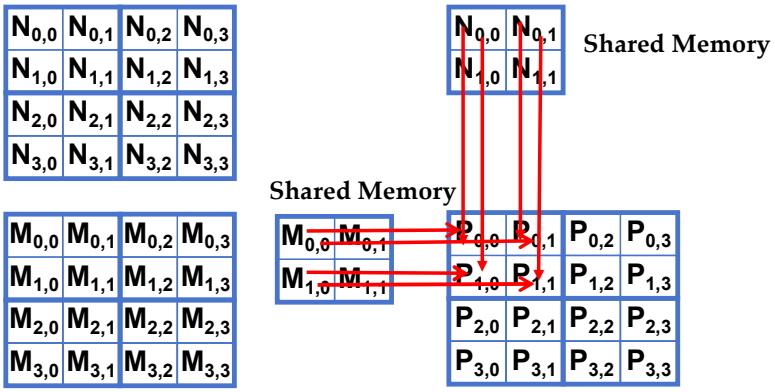
Loading a Tile

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

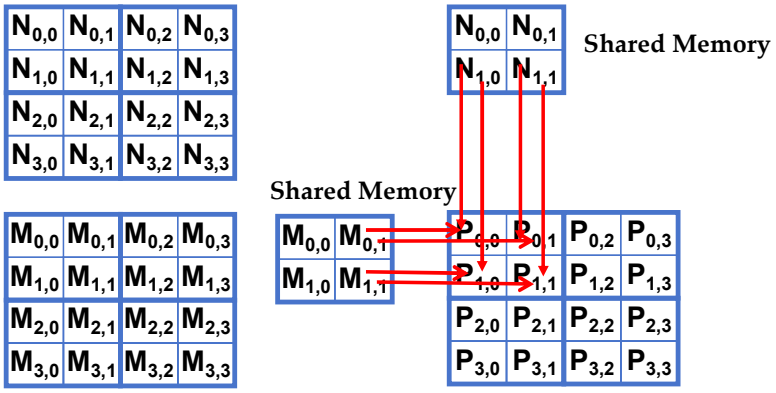
Phase 0 Load for Block (0,0)



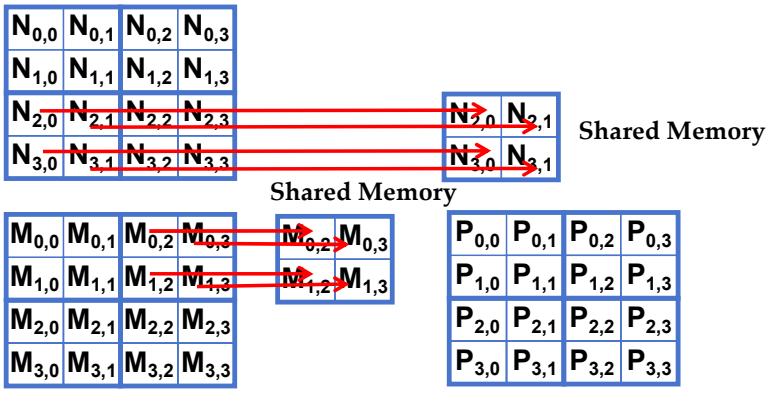
Phase 0 Use for Block (0,0) (iteration 0)



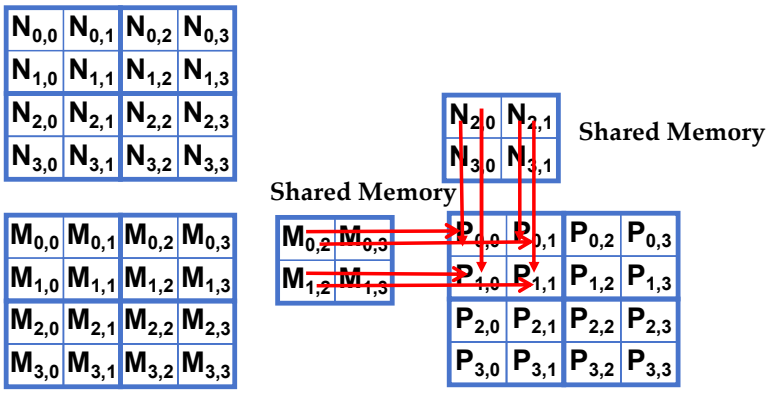
Phase 0 Use for Block (0,0) (iteration 1)



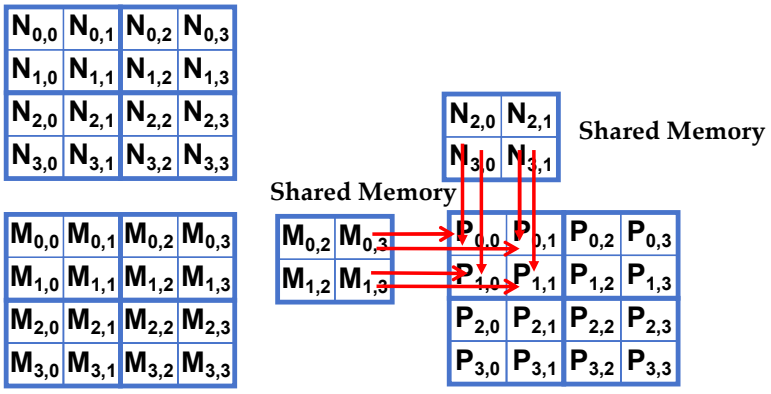
Phase 1 Load for Block (0,0)



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 →

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

讲授内容：Memory and Data Locality

① CUDA Memories

② Tiled Parallel Algorithms

③ Tiled Matrix Multiplication

④ Tiled Matrix Multiplication Kernel

⑤ Handling Arbitrary Matrix Sizes in Tiled Algorithms

Loading Input Tile 0 of M (Phase 0)

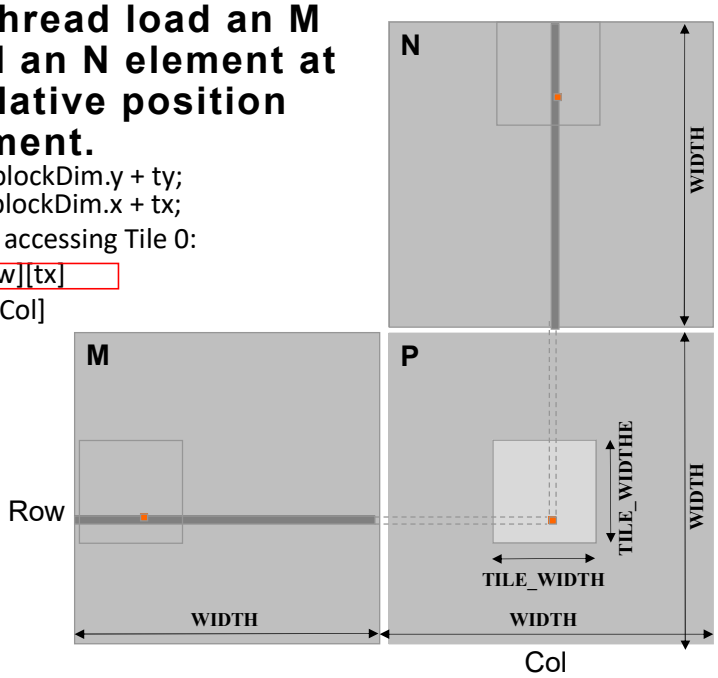
- **Have each thread load an M element and an N element at the same relative position as its P element.**

  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)

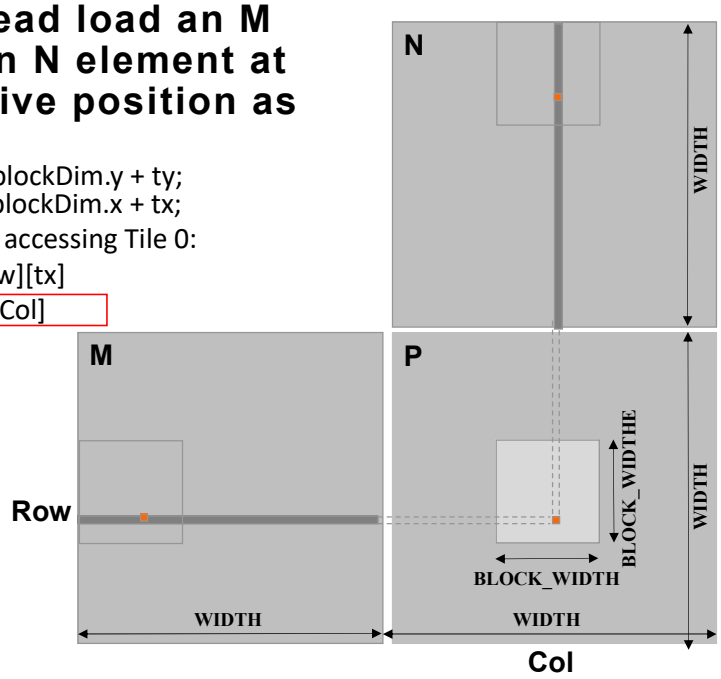
- **Have each thread load an M element and an N element at the same relative position as its P element.**

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

  2D indexing for accessing Tile 0:

  M[Row][tx]

  N[ty][Col]

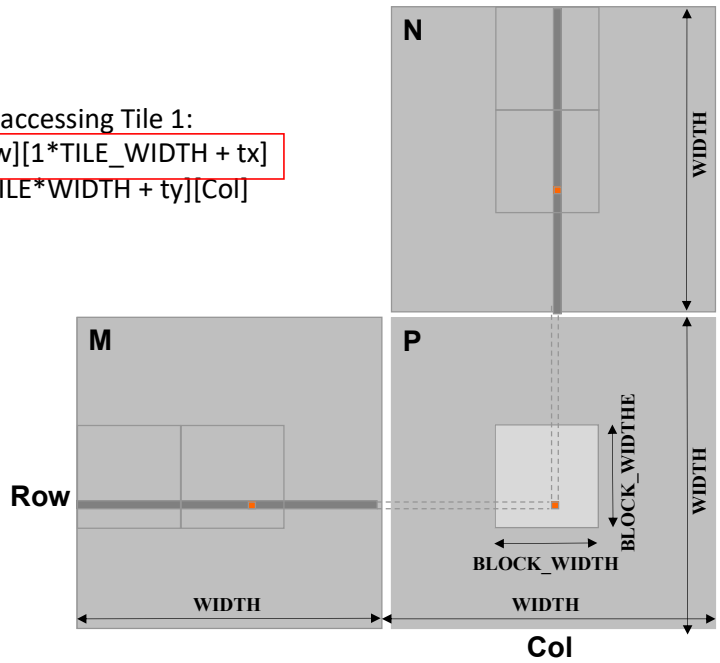


Loading Input Tile 1 of M (Phase 1)

2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]

N[1*TILE*WIDTH + ty][Col]

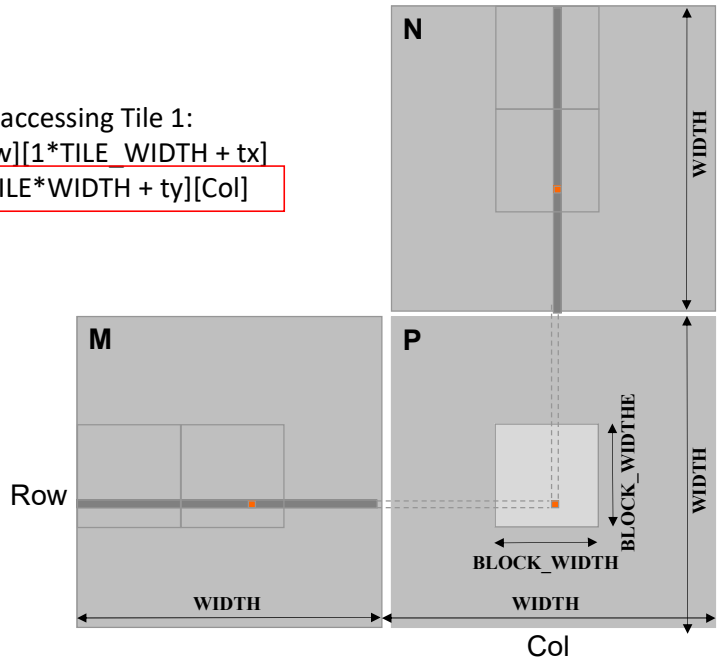


Loading Input Tile 1 of N (Phase 1)

2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]

N[1*TILE*WIDTH + ty][Col]



M and N are dynamically allocated - use 1D indexing

```
M[Row][p*TILE_WIDTH+tx]
M[Row*Width + p*TILE_WIDTH + tx]

N[p*TILE_WIDTH+ty][Col]
N[(p*TILE_WIDTH+ty)*Width + 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) {
   ...
  }
  P[Row*Width+Col] = Pvalue;
}
```

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

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) {
   ...  }
  P[Row*Width+Col] = Pvalue;
}
```

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

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) {
...
  }
  P[Row*Width+Col] = Pvalue;
}
```

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

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*256*4B = 2KB of shared memory.
  - For 16KB shared memory, one can potentially have up to 8 thread blocks executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to 2*32*32*4 Byte= 8K 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

讲授内容：Memory and Data Locality

① CUDA Memories

② Tiled Parallel Algorithms

③ Tiled Matrix Multiplication

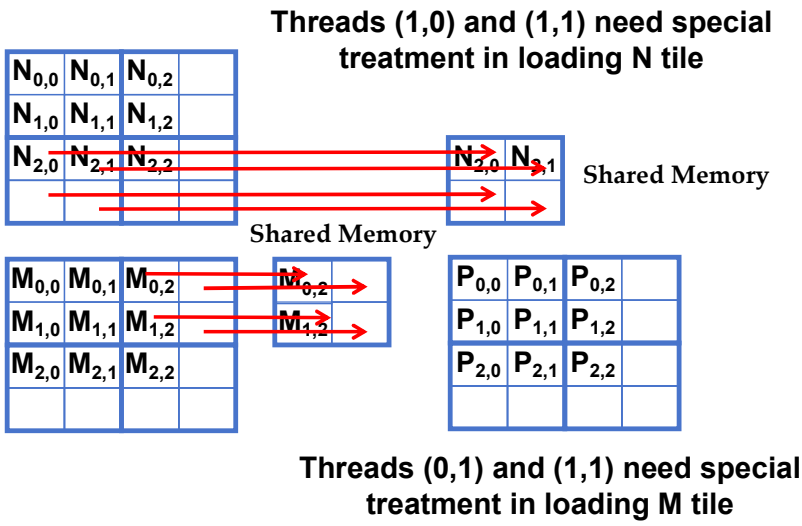
④ Tiled Matrix Multiplication Kernel

⑤ Handling Arbitrary Matrix Sizes in Tiled Algorithms

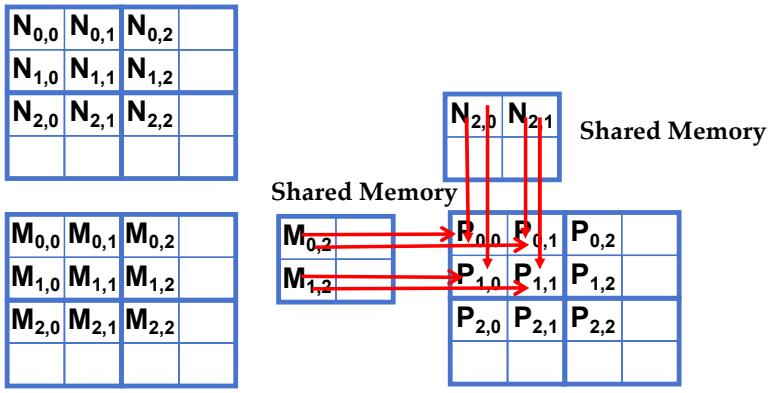
Handling Matrix of Arbitrary Size

- The tiled matrix multiplication kernel we presented so far can handle only square matrices whose dimensions (Width) are multiples of the tile width (TILE_WIDTH)
  - However, real applications need to handle arbitrary sized matrices.
  - One could pad (add elements to) the rows and columns into multiples of the tile size, but would have significant space and data transfer time overhead.
- We will take a different approach.

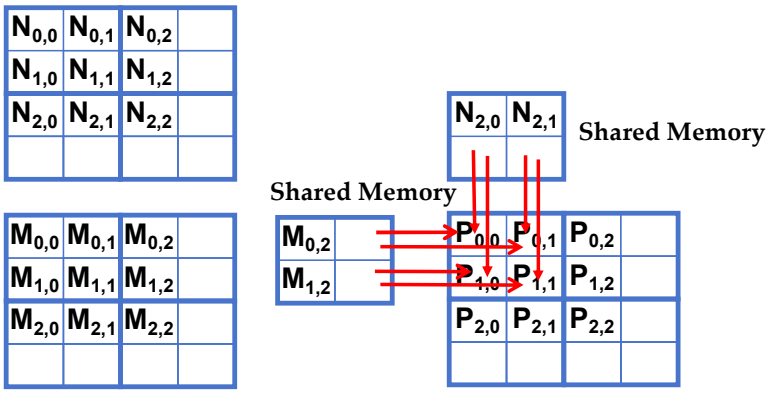
Phase 1 Loads for Block (0,0) for a 3x3 Example



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

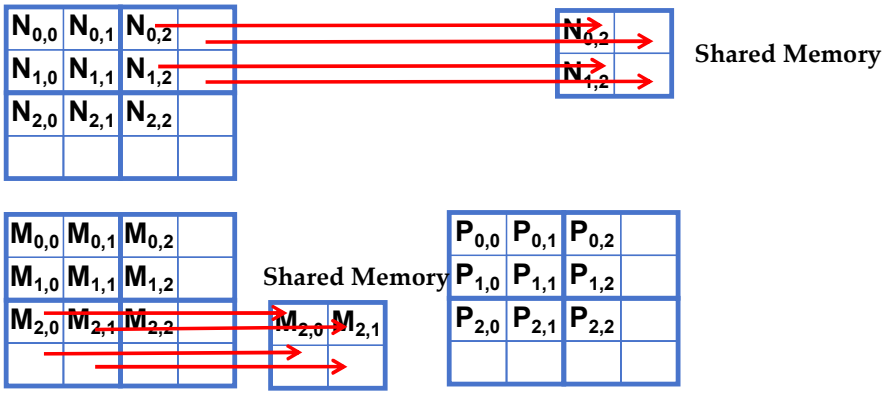


Phase 1 Use for Block (0,0) (iteration 1)



All Threads need special treatment. None of them should introduce invalidate contributions to their P elements.

Phase 0 Loads for Block (1,1) for a 3x3 Example



Threads (1,0) and (1,1) need special treatment in loading M tile

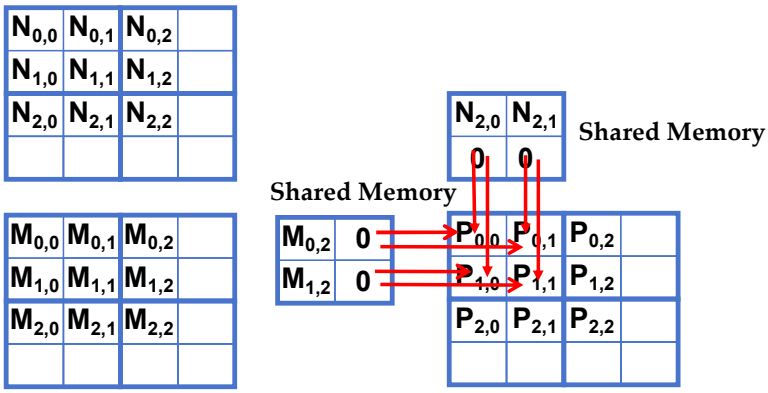
Major Cases in Toy Example

- Threads that do not calculate valid P elements but still need to participate in loading the input tiles
  - Phase 0 of Block(1,1), Thread(1,0), assigned to calculate non-existent P[3,2] but need to participate in loading tile element N[1,2]
- Threads that calculate valid P elements may attempt to load non-existing input elements when loading input tiles
  - Phase 0 of Block(0,0), Thread(1,0), assigned to calculate valid P[1,0] but attempts to load non-existing N[3,0]

A "Simple" Solution

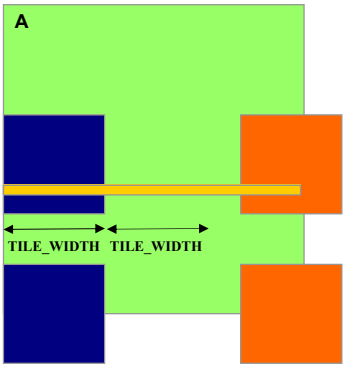
- When a thread is to load any input element, test if it is in the valid index range
  - If valid, proceed to load
  - Else, do not load, just write a 0
- Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element
- The condition tested for loading input elements is different from the test for calculating output P element
  - A thread that does not calculate valid P element can still participate in loading input tile elements

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



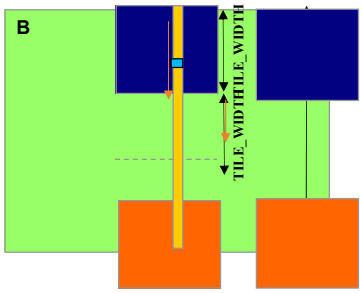
Boundary Condition for Input M Tile

- Each thread loads
  - M[Row][p*TILE_WIDTH+tx]
  - M[Row*Width + p*TILE_WIDTH+tx]
- Need to test
  - (Row < Width) && (p*TILE_WIDTH+tx < Width)
  - If true, load M element
  - Else , load 0



Boundary Condition for Input N Tile

- Each thread loads
  - N[p*TILE_WIDTH+ty][Col]
  - N[(p*TILE_WIDTH+ty)*Width+ Col]
- Need to test
  - (p*TILE_WIDTH+ty < Width) && (Col< Width)
  - If true, load N element
  - Else , load 0



Loading Elements – with boundary check

```
8     for (int p = 0; p < (Width-1) / TILE_WIDTH + 1; ++p) {
++        if(Row < Width && t * TILE_WIDTH+tx < Width) {
9               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) {
10              ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];
++        } else {
++              ds_N[ty][tx] = 0.0;
++        }
11       __syncthreads();
```

Inner Product – Before and After

```
++    if(Row < Width && Col < Width) {
12      for (int i = 0; i < TILE_WIDTH; ++i) {
13                    Pvalue += ds_M[ty][i] *
ds_N[i][tx];
        }
14      __syncthreads();
15    } /* end of outer for loop */
++    if (Row < Width && Col < Width)
16          P[Row*Width + Col] = Pvalue;
    } /* end of kernel */
```

Some Important Points

- For each thread the conditions are different for
  - Loading M element
  - Loading N element
  - Calculating and storing output elements
- The effect of control divergence should be small for large matrices

Handling General Rectangular Matrices

- In general, the matrix multiplication is defined in terms of rectangular matrices
  - A j x k M matrix multiplied with a k x l N matrix results in a j x l P matrix
- We have presented square matrix multiplication, a special case
- The kernel function needs to be generalized to handle general rectangular matrices
  - The Width argument is replaced by three arguments: j, k, l
  - When Width is used to refer to the height of M or height of P, replace it with j
  - When Width is used to refer to the width of M or height of N, replace it with k
  - When Width is used to refer to the width of N or width of P, replace it with l