COMP 322: Fundamentals of Parallel Programming

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Lecture 33: GPGPU Programming with CUDA

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

Acknowledgments for Today's Lecture

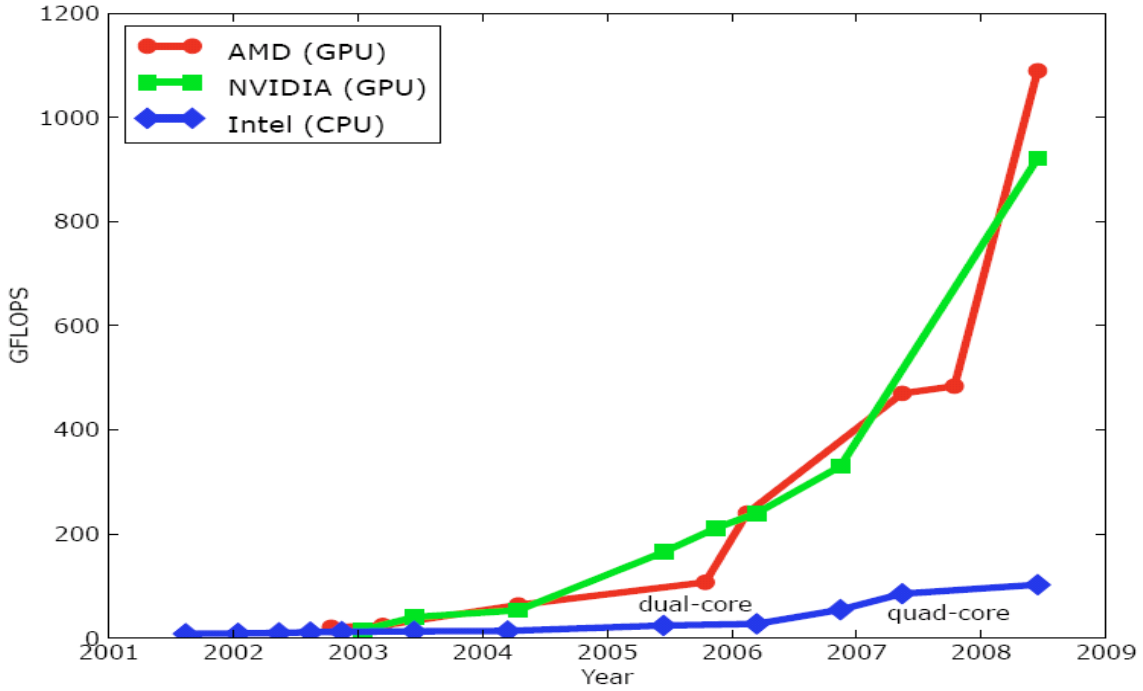
- Handout for Lecture 33
- David B. Kirk and Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

Announcements

- Homework 7 due by 5pm on Friday, April 22nd
  - Send email to comp322-staff if you're running into issues with accessing SUG@R nodes, or anything else

Why GPUs?

- Two major trends
  1. Increasing performance gap relative to mainstream CPUs
     - Calculation: 367 GFLOPS vs. 32 GFLOPS
     - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  2. Availability of more general (non-graphics) programming interfaces



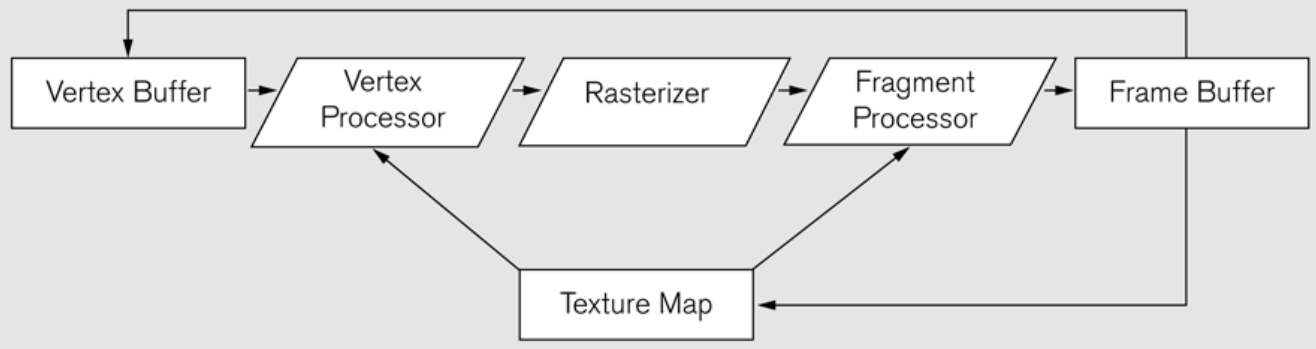
  — GPU in every PC and workstation – massive volume and potential impact

What is GPGPU ?

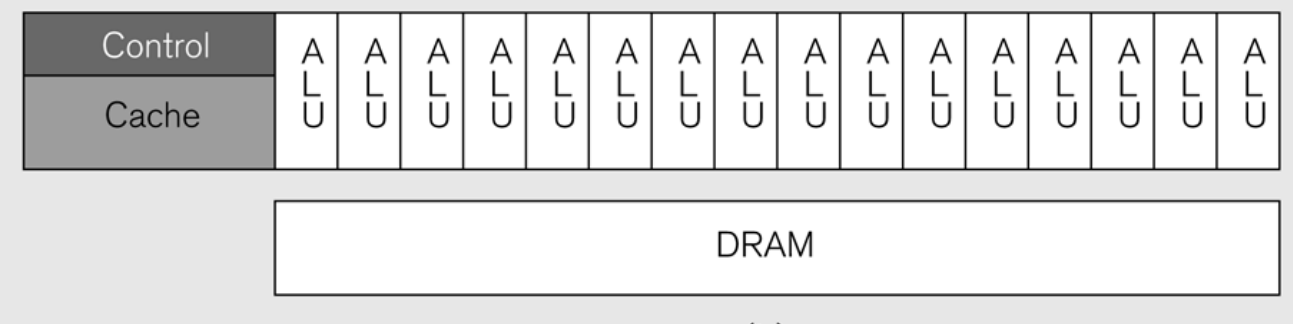
- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

Traditional vs. General Purpose GPUs

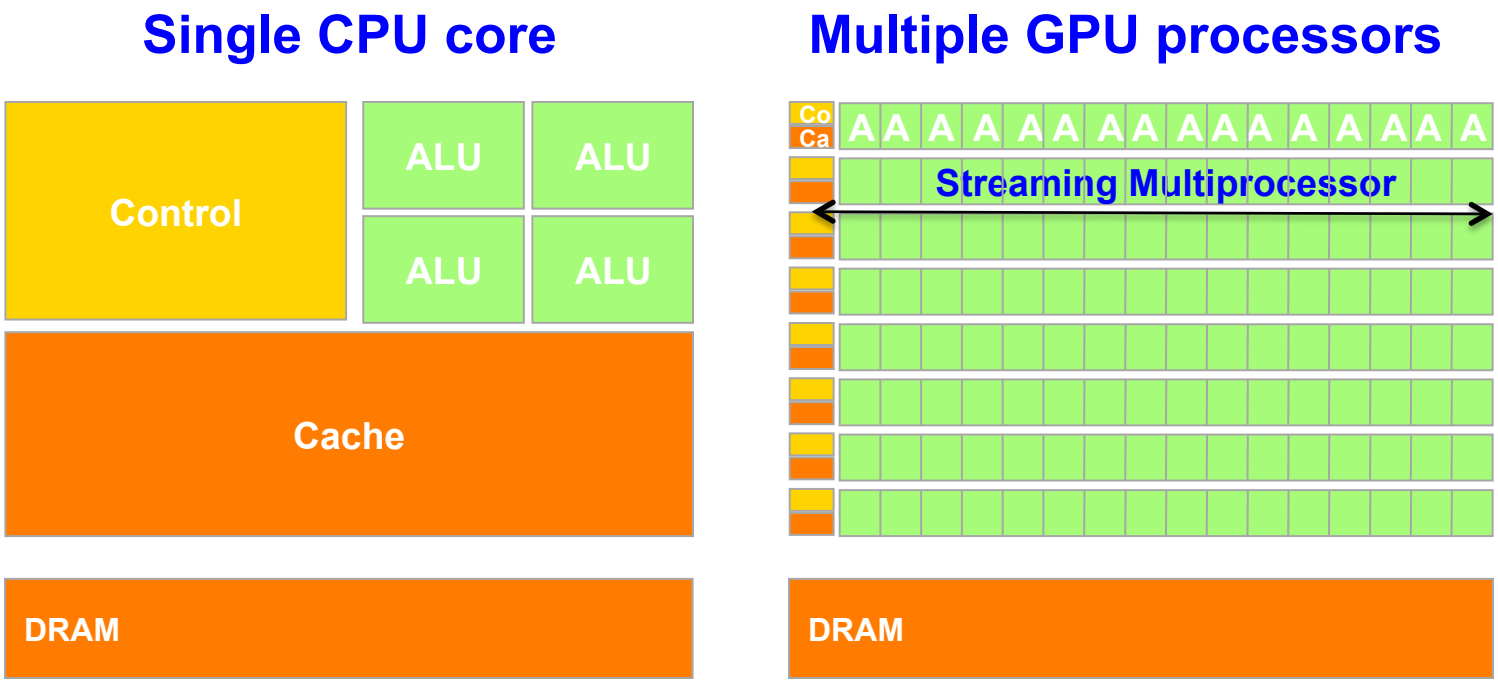
- Traditional graphics pipeline (Figure 10.3, Lin & Snyder)



- General-purpose GPU (Figure 10.4(b), Lin & Snyder)



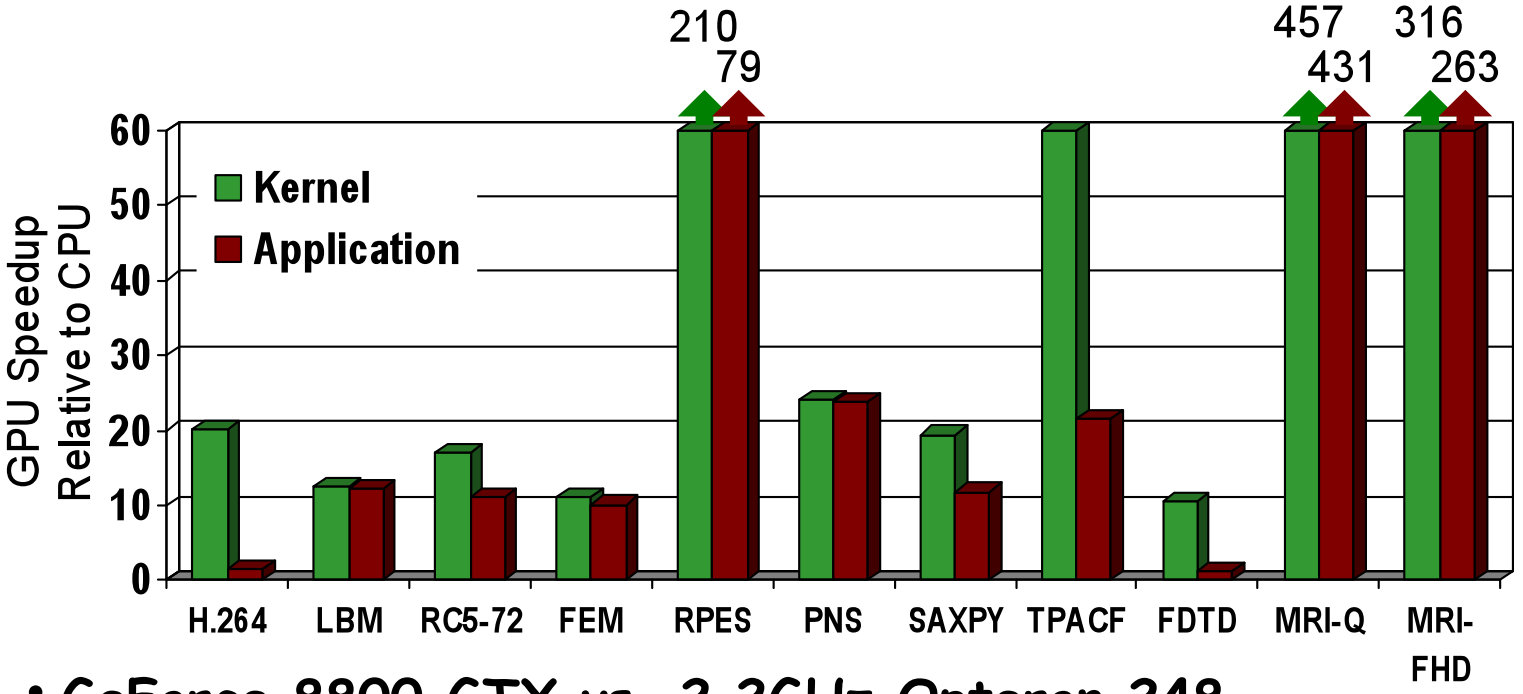
CPUs and GPUs have fundamentally different design philosophies (Figure 1)



Some applications that are well-suited for GPU execution

Application	Description	Source	Kernel	% time
H.264	SPEC '06 version, change in guess vector	34,811	194	35%
LBM	SPEC '06 version, change to single precision and print fewer reports	1,481	285	>99%
RC5-72	Distributed.net RC5-72 challenge client code	1,979	218	>99%
FEM	Finite element modeling, simulation of 3D graded materials	1,874	146	99%
RPES	Rye Polynomial Equation Solver, quantum chem, 2-electron repulsion	1,104	281	99%
PNS	Petri Net simulation of a distributed system	322	160	>99%
SAXPY	Single-precision implementation of saxpy, used in Linpack's Gaussian elim. routine	952	31	>99%
TRACF	Two Point Angular Correlation Function	536	98	96%
FDTD	Finite-Difference Time Domain analysis of 2D electromagnetic wave propagation	1,365	93	16%
MRI-Q	Computing a matrix Q, a scanner's configuration in MRI reconstruction	490	33	>99%

Speedup of these applications relative to a single CPU core



- GeForce 8800 GTX vs. 2.2GHz Opteron 248
- 10× speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads
- 25× to 400× speedup if the function's data requirements and control flow suit the GPU and the application is optimized

Process Flow of a CUDA Kernel Call (Figure 2)

- Data parallel programming architecture from NVIDIA
  - Execute programmer-defined kernels on extremely parallel GPUs
  - CUDA program flow:
    1. Push data on device
    2. Launch kernel
    3. Execute kernel and memory accesses in parallel
    4. Pull data off device
- Device threads are launched in batches
  - Blocks of Threads, Grid of Blocks
- Explicit device memory management
  - cudaMalloc, cudaMemcpy, cudaFree, etc.

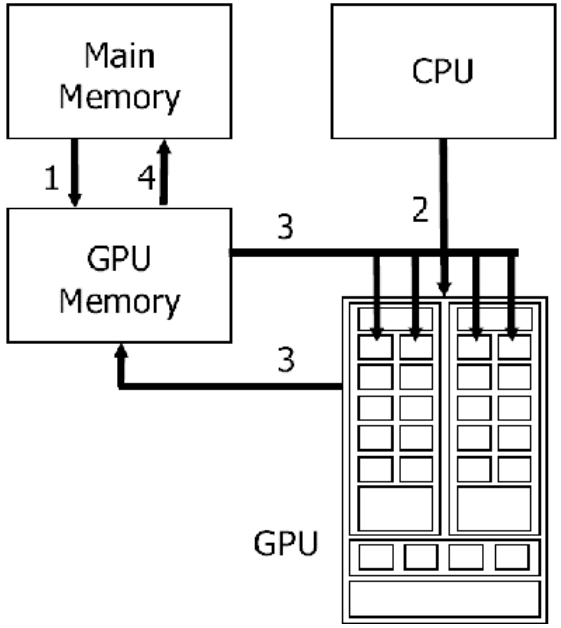
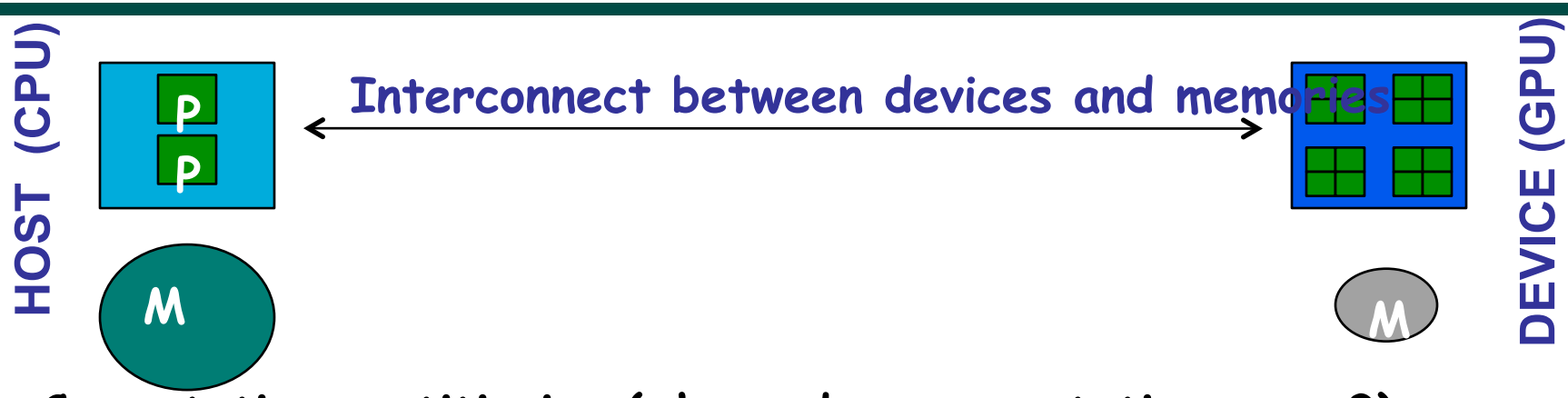


Figure source: Y. Yan et. al "JCUDA: a Programmer Friendly Interface for Accelerating Java Programs with CUDA." Euro-Par 2009.

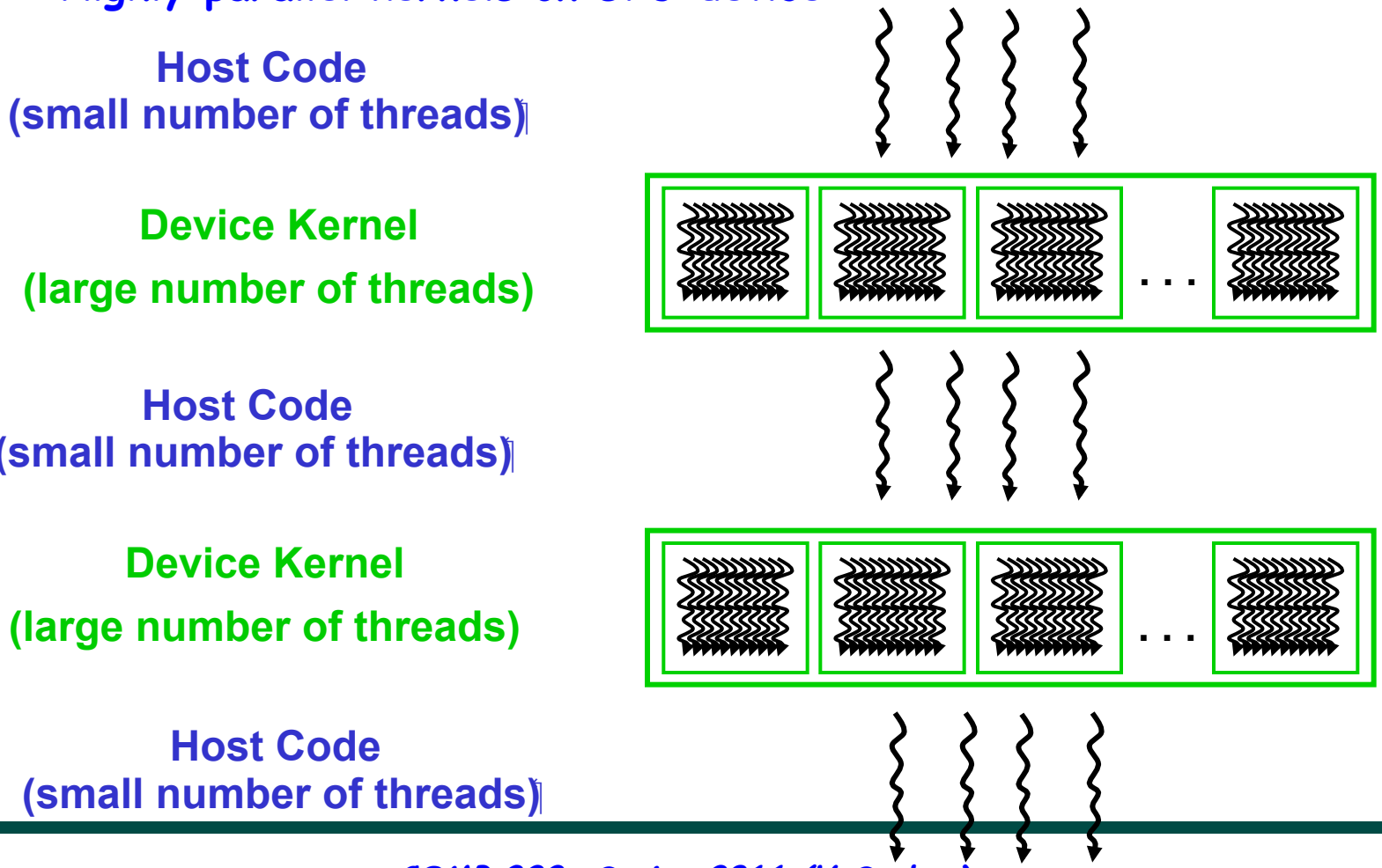
What Programmer Expresses in CUDA



- Computation partitioning (where does computation occur?)
  - Declarations on functions __host__, __global__, __device__
  - Mapping of thread programs to device: compute <<<gs, bs>>>(<args>)
- Data partitioning (where does data reside, who may access it and how?)
  - Declarations on data __shared__, __device__, __constant__, …
- Data management and orchestration
  - Copying to/from host: *e.g.*, cudaMemcpy(h_obj,d_obj, cudaMemcpyDevicetoHost)
- Concurrency management
  - *E.g.* __synchthreads()

Execution of a CUDA program (Figure 3)

- Integrated host+device application
  - Serial or modestly parallel parts on CPU host
  - Highly parallel kernels on GPU device

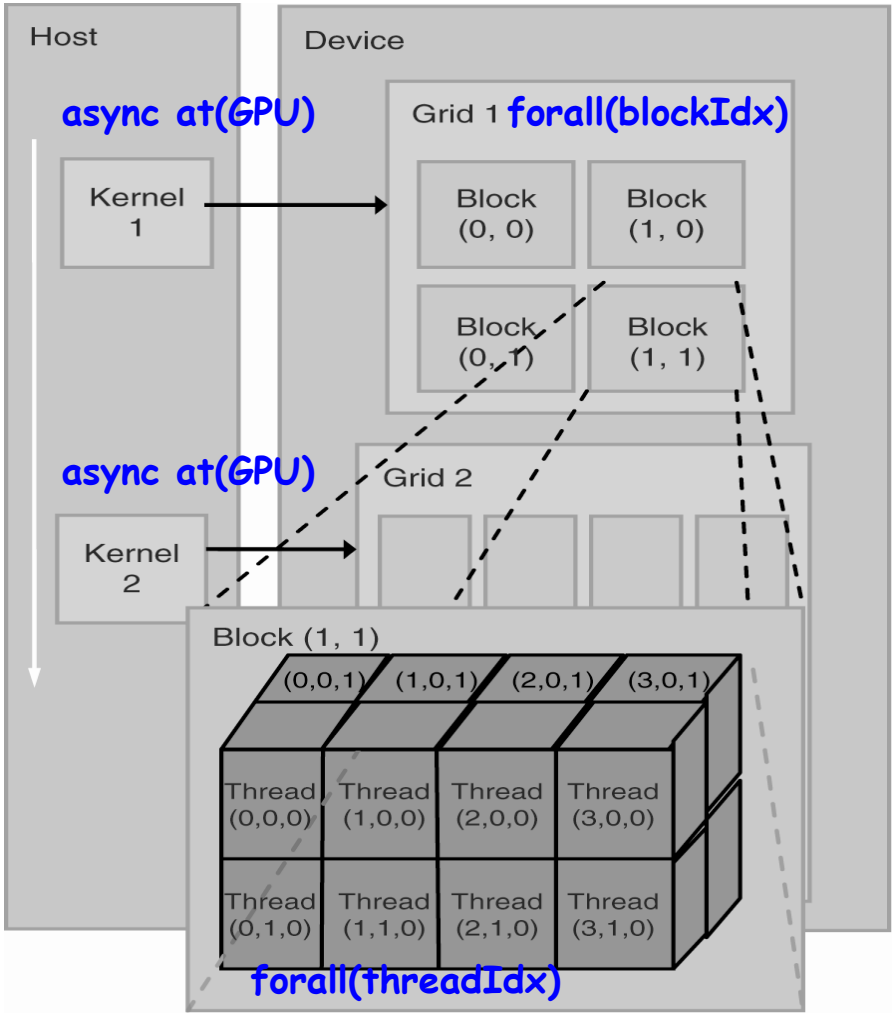


Logical Structure of a CUDA kernel invocation (Listing 1)

```
1  finish async at(GPU) {
2    // Parallel execution of blocks in grid
3    forall (point[blockIdx.x,blockIdx.y] : [0:gridDim.x-1,0:gridDim.y-1]) {
4      // Parallel execution of threads in block (blockIdx.x,blockIdx.y)
5      forall (point[threadIdx.x,threadIdx.y,threadIdx.z]
6                    : [0:blockDim.x-1,0:blockDim.y-1,0:blockDim.z-1]) {
7        // Perform kernel computation as function of blockIdx.x,blockIdx.y
8        // and threadIdx.x,threadIdx.y,threadIdx.z
9        . . .
10       next; // barrier synchronizes inner forall only (__syncthreads)
11       . . .
12     } // forall threadIdx.x,threadIdx.y,threadIdx.z
13   } // forall blockIdx.x, blockIdx.y
14 } // finish async (GPU)
```

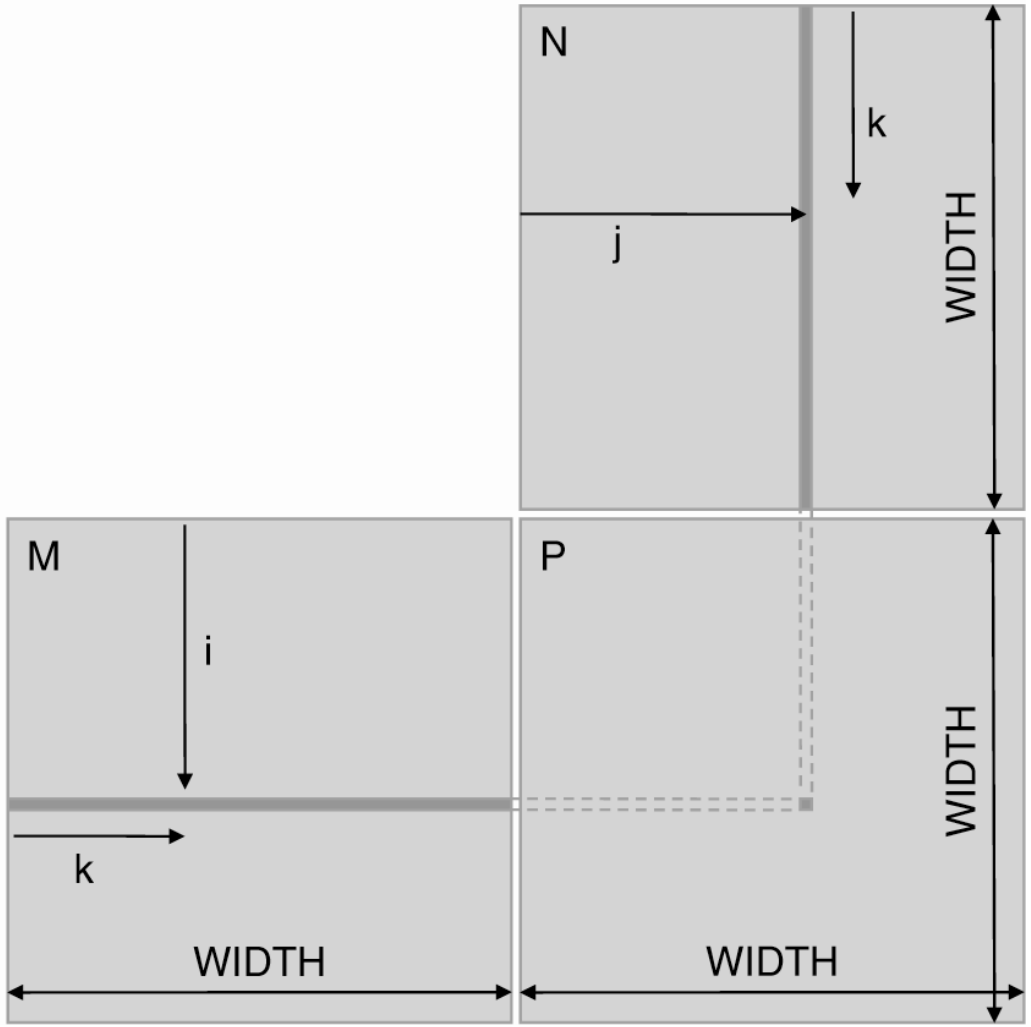
Listing 1: Logical structure of a CUDA kernel invocation

Organization of a CUDA grid (Figure 4)



Sequential CPU version of matrix multiply written in C (Figure 5)

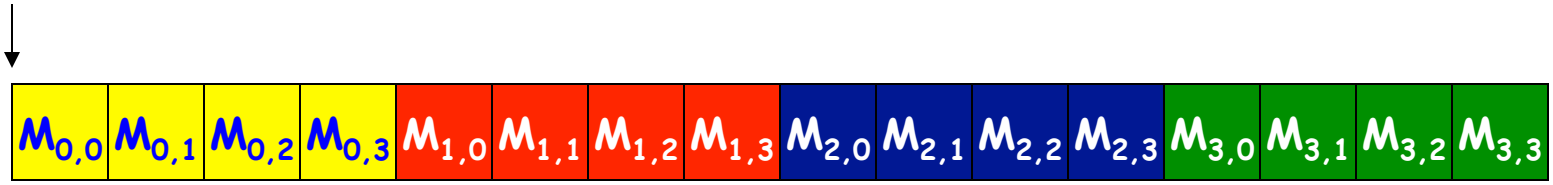
```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k  = 0; k < Width; ++k) {
                float a = M[i * width + k];
                float b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Using a 1-D array to store a 2-D matrix (Row major layout)

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

M



Assume square matrix for simplicity

Matrix multiplication kernel code in CUDA (Figure 6)

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   // 2D Thread ID
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // Pvalue stores the Pd element that is computed by the thread
   float Pvalue = 0;

   for (int k = 0; k < Width; ++k)
   {
      float Mdelement = Md[ty * Width + k];
      float Ndelement = Nd[k * Width + tx];
      Pvalue += Mdelement * Ndelement;
   }

   // Write the matrix to device memory each thread writes one element
   Pd[ty * Width + tx] = Pvalue;
}
```


Launching the Kernel

- Four steps in CUDA execution
  1. Push data on device
     - Use cudaMalloc() and cudaMemCpy()
       - Will be discussed in next lecture
  2. Launch kernel (Figure 7)
     - Two-level forall loops implied by <<<…>>> parameters

```
// Setup the execution configuration
dim3 dimBlock(Width, Width);
dim3 dimGrid(1, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

  3. Execute kernel (Figure 6)
  4. Pull data off device
     - Use cudaMemCpy()