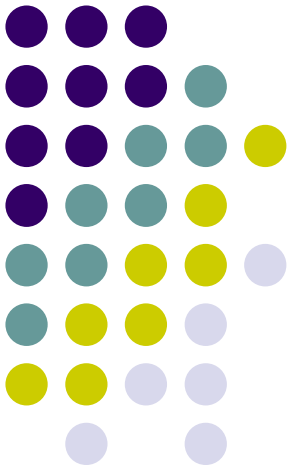
ME964
High Performance Computing for Engineering Applications

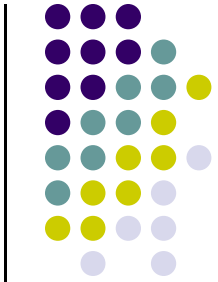
The CUDA API

February 08, 2011



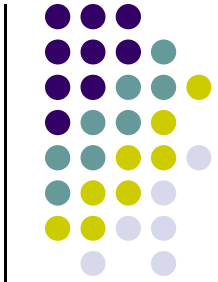
Most of the time I don't have much fun. The rest of the time I don't have any fun at all. – Woody Allen

Before We Get Started…

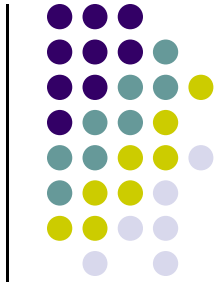


- Last time
  - Andrew: wrap up building CUDA apps in Visual Studio 2008
  - Andrew: running apps through the HPC scheduler on Newton
  - Very high-level overview of the CUDA programming model
  - Discussed Index issues in the context of the “execution configuration” and how the index of a thread translates into an ID of a thread
  - Brief discussion of the memory spaces in relation to GPU computing
- Today
  - Discussion of the CUDA API
  - One-on-one with Andrew if you have compile/build issues in CUDA
    - 3-5 PM in room 2042ME
- HW
  - HW2: due date was 02/08. Now 02/10
  - HW3 has been posted. Due date: 02/15
    - Small matrix-vector multiplication
    - Matrix addition – requires use of multiple blocks

Putting Things in Perspective…

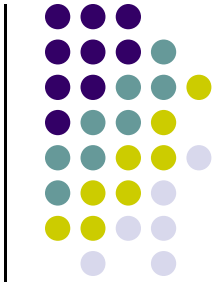


- CUDA programming model and execution configuration
  - Basic concepts and data types - just finished this…
- CUDA application programming interface
  - Working on it next
- Simple example to illustrate basic concepts and functionality
  - Coming up shortly
- Performance features will be covered later



The CUDA API

What is an API?

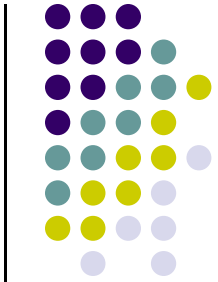


- Application Programming Interface (API)
  - A set of functions, procedures or classes that an operating system, library, or service provides to support requests made by computer programs (from Wikipedia)
  - Example: OpenGL, a graphics library, has its own API that allows one to draw a line, rotate it, resize it, etc.

- Cooked up analogy (for the mechanical engineer)
  - Think of a car, you can say it has a certain Device Operating Interface (DOI):
    - A series of pedals, gauges, steering wheel, etc. This would be its DOI

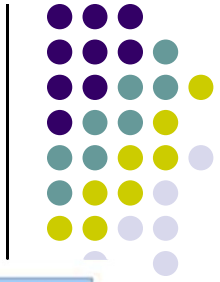
- In this context, CUDA provides an API that enables you to tap into the computational resources of the NVIDIA's GPUs
  - This is what replaced the old GPGPU way of programming the hardware

On the CUDA API



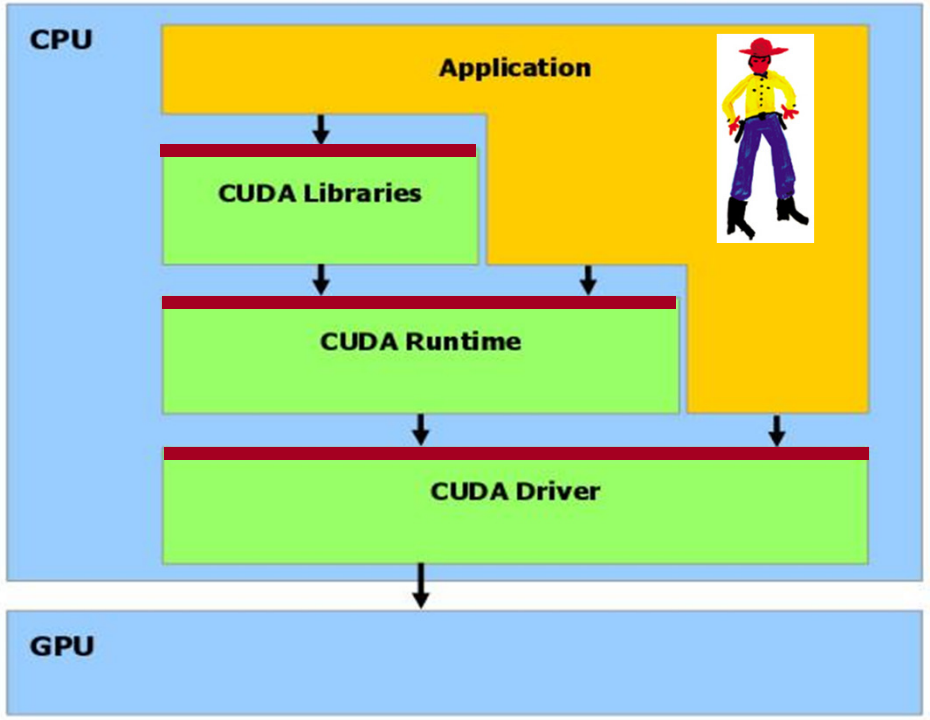
- Reading the CUDA Programming Guide you'll run numerous references to the CUDA Runtime API and CUDA Driver API
  - Many time they talk about "CUDA runtime" and "CUDA driver". What they mean is CUDA Runtime API and CUDA Driver API
- CUDA Runtime API – is the friendly face that you can choose to see when interacting with the GPU. This is what gets identified with "C CUDA"
  - Needs nvcc compiler to generate an executable
- CUDA Driver API – this is more like how it was back in the day: low level way of interacting with the GPU
  - You have significantly more control over the host-device interaction
  - Significantly clunkier way to dialogue with the GPU, typically only needs a C compiler
- I don't anticipate any reason to use the CUDA Driver API

Talking about the API: The C CUDA Software Stack



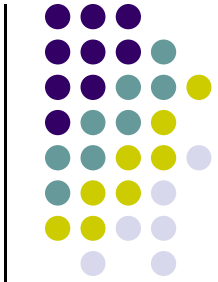
- Image at right indicates where the API fits in the picture

An API layer is indicated by a thick red line: —



- NOTE: any CUDA runtime function has a name that starts with “cuda”
  - Examples: cudaMalloc, cudaFree, cudaMemcpy, etc.
- Examples of CUDA Libraries: CUFFT, CUBLAS, CUSP, thrust, etc.

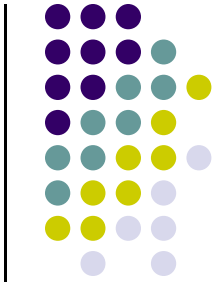
CUDA Function Declarations



	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
  - Must return `void`
- `__device__` and `__host__` can be used together

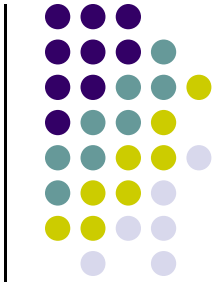
CUDA Function Declarations (cont.)



- `__device__` functions can't have their address taken

- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments
    - Something like *printf* would not work…

Compiling CUDA

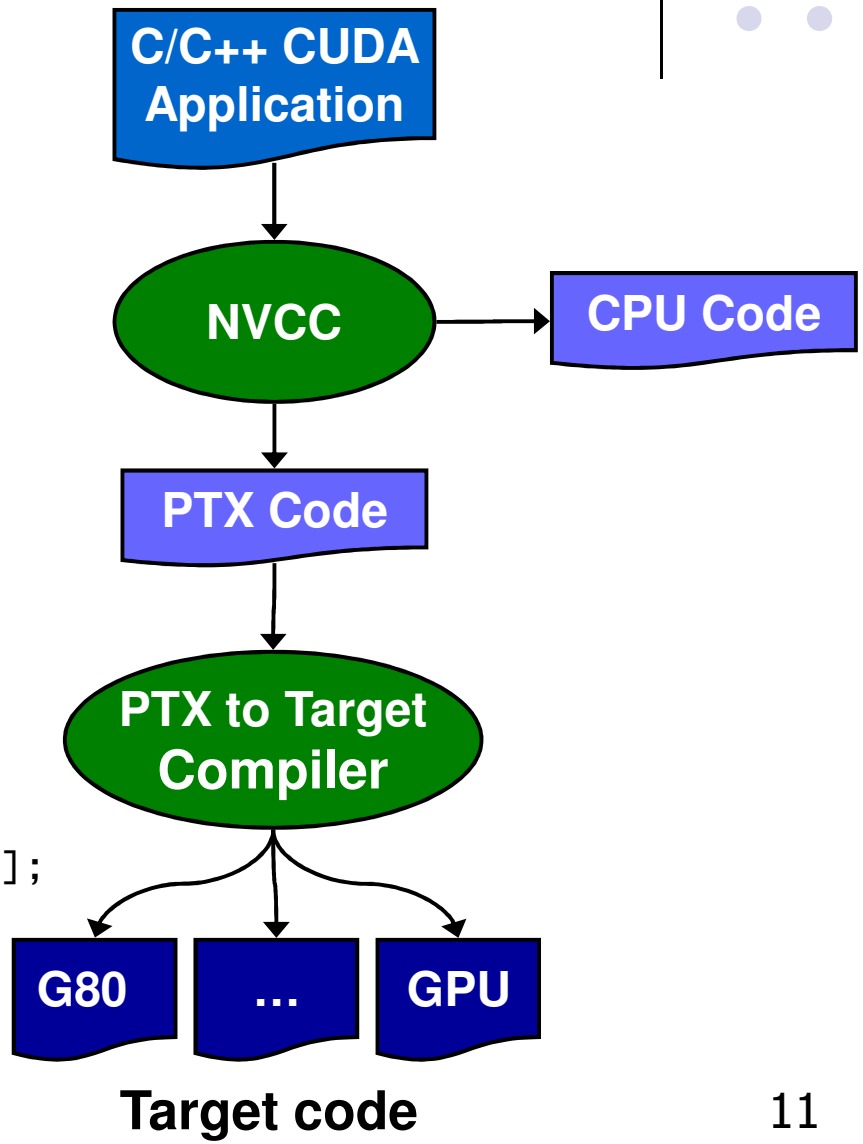


- Any source file containing CUDA language extensions must be compiled with nvcc
  - You spot such a file by its .cu suffix
- nvcc is a compile driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- nvcc can output:
  - C code
    - Must then be compiled with the rest of the application using another tool
  - ptx code (CUDA's ISA)
  - Or directly object code (cubin)

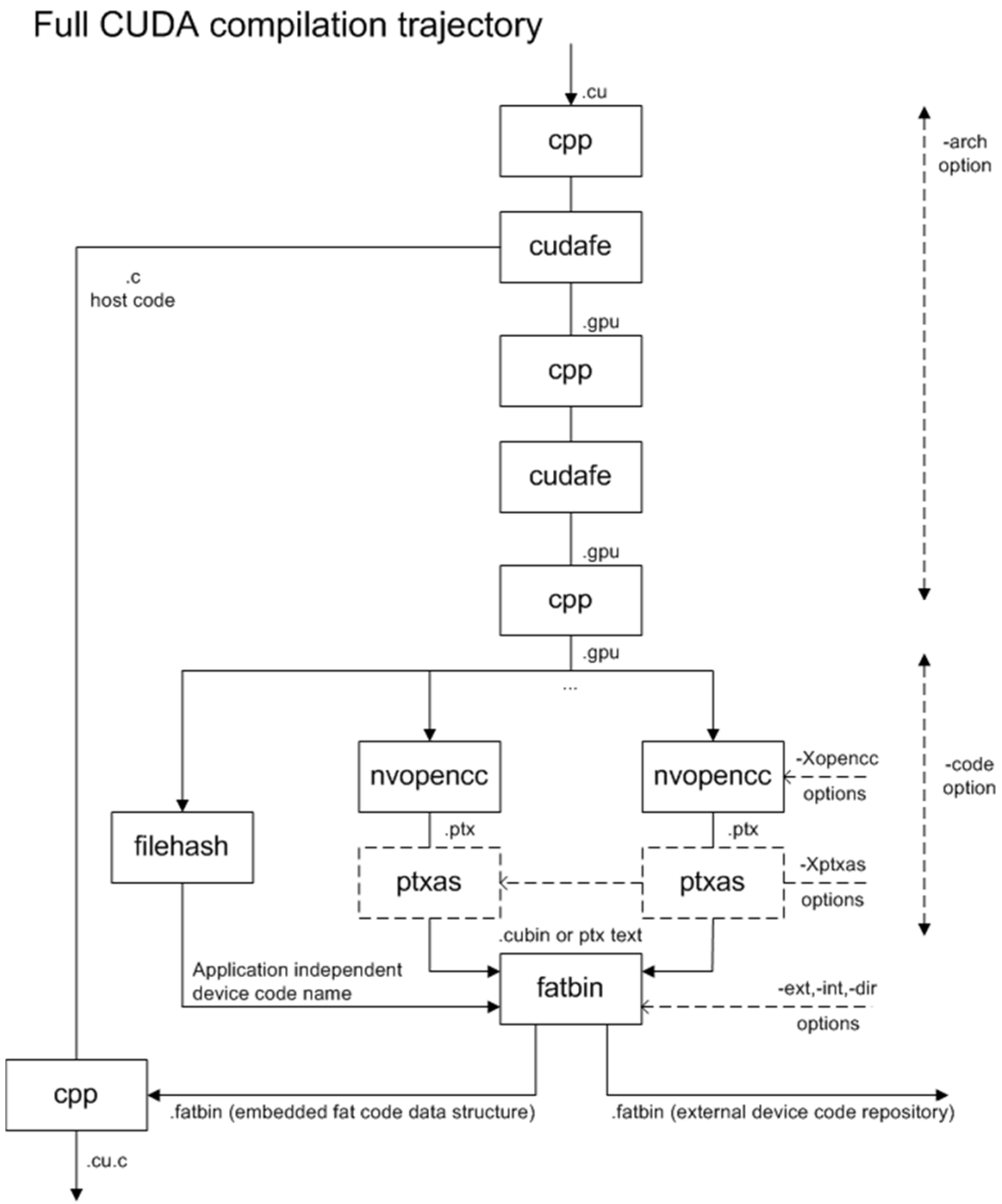
Compiling CUDA

- nvcc
  - Compile driver
  - Invokes cudacc, gcc, cl, etc.
- PTX
  - Parallel Thread eXecution
  - Like assembly language
  - NVIDIA's ISA

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```

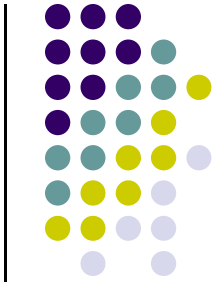


Compiling CUDA extended C



http://sbel.wisc.edu/Courses/ME964/2008/Documents/nvccCompilerInfo.pdf

The nvcc Compiler – Suffix Info

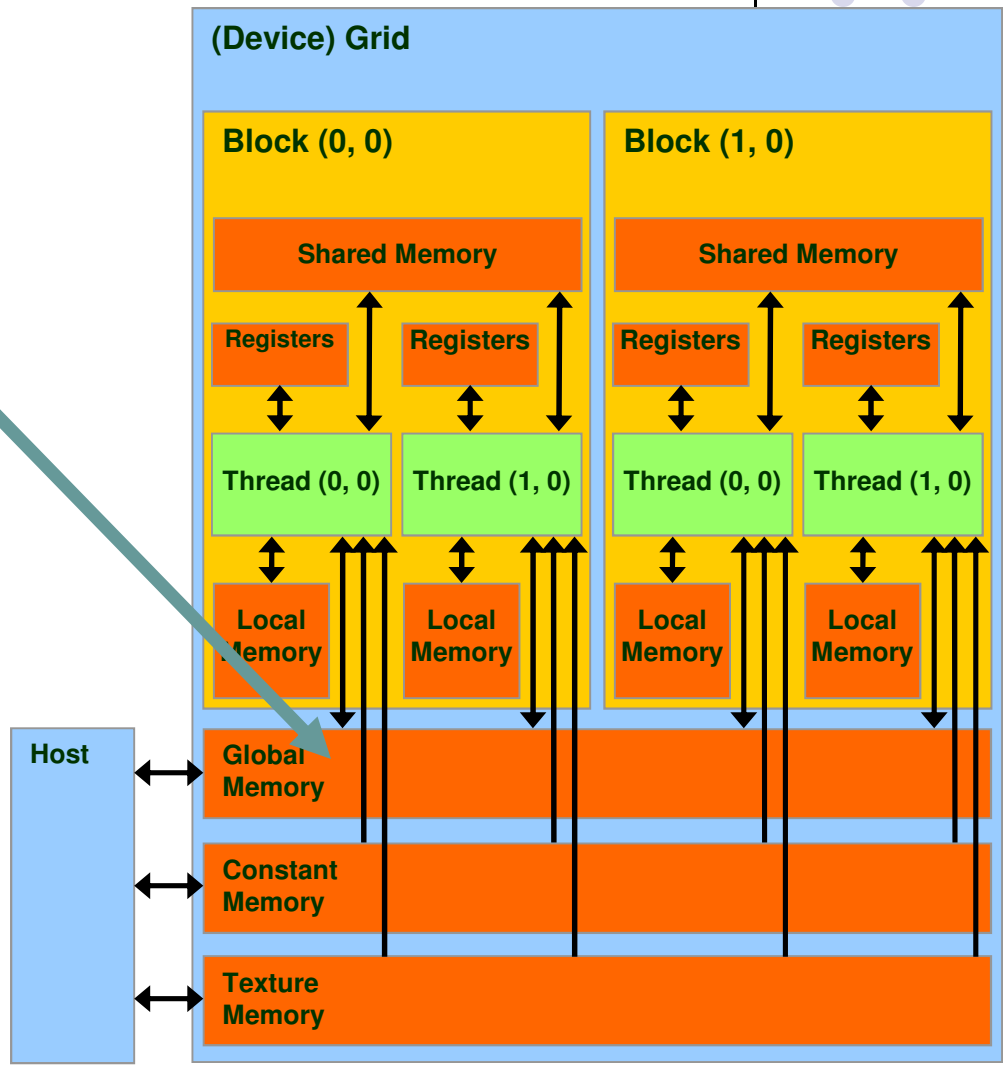


File suffix	How the nvcc compiler interprets the file
.cu	CUDA source file, containing host and device code
.cup	Preprocessed CUDA source file, containing host code and device functions
.c	'C' source file
.cc, .cxx, .cpp	C++ source file
.gpu	GPU intermediate file (device code only)
.ptx	PTX intermediate assembly file (device code only)
.cubin	CUDA device only binary file

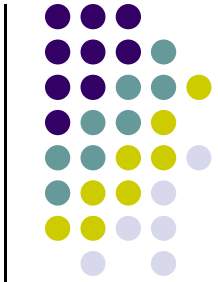
CUDA API: Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** allocated object

- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



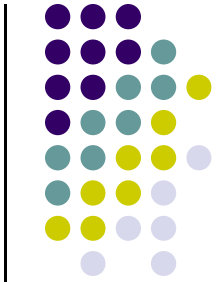
Example Use: A Matrix Data Type



- NOT part of CUDA API
- It will be frequently used in many code examples
  - 2 D matrix
  - Single precision float elements
  - Width * height elements
  - Matrix entries attached to the pointer-to-float member called “elements”
  - Matrix is stored row-wise

```
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```

CUDA Device Memory Allocation (cont.)

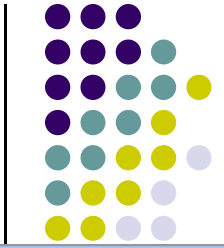


- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md.elements
  - "d" is often used to indicate a device data structure

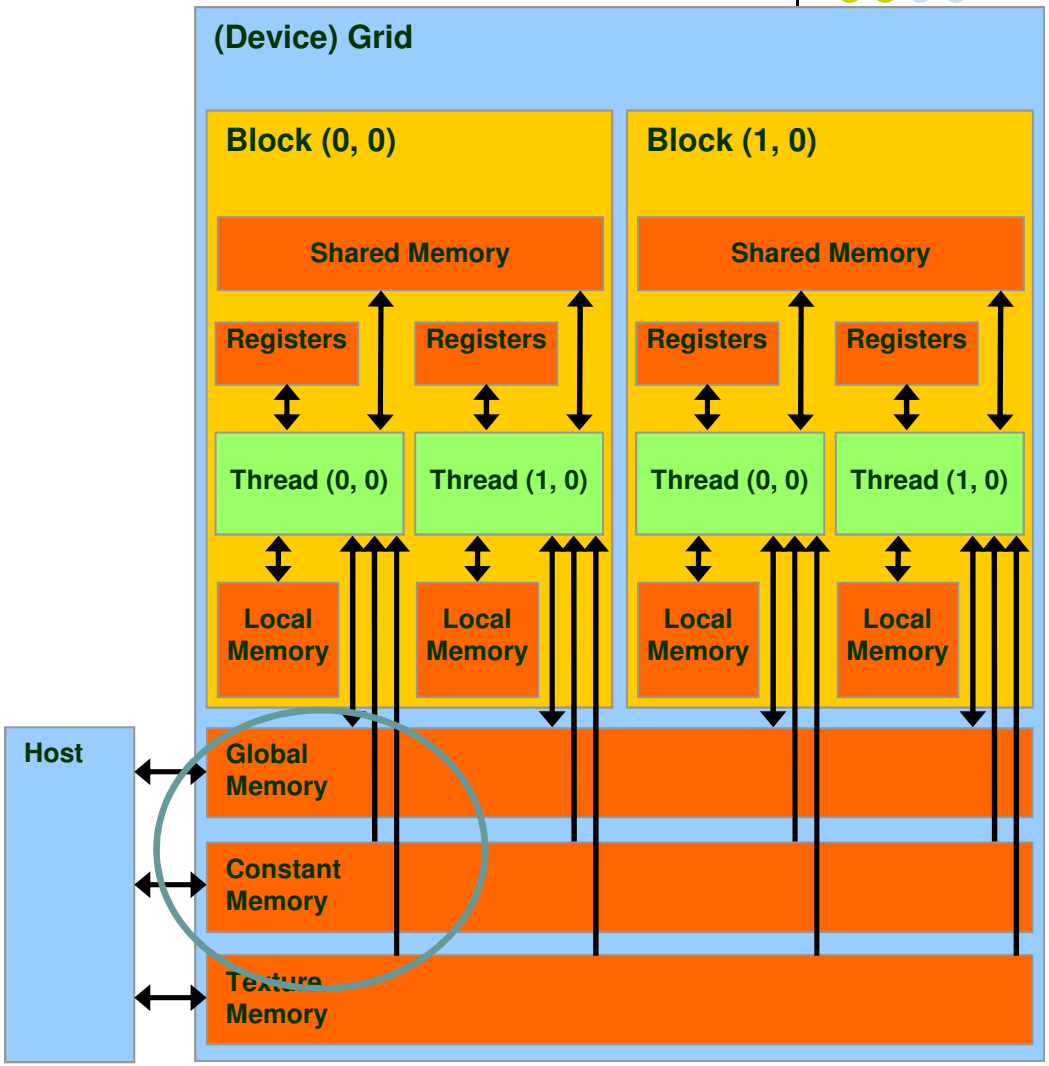
```
BLOCK_SIZE = 64;
Matrix Md;
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**)&Md.elements, size);
…
//use it for what you need, then free the device memory
cudaFree(Md.elements);
```

All the details are spelled out in the CUDA Programming Guide 3.2 (see the resources section of the class website)

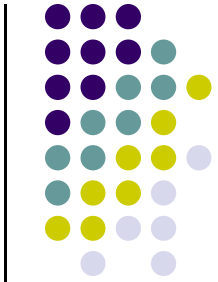
CUDA Host-Device Data Transfer



- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device



CUDA Host-Device Data Transfer (cont.)

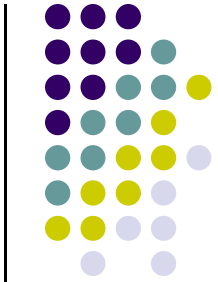


- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

```
cudaMemcpy(Md.elements, M.elements, size,
                        cudaMemcpyHostToDevice);

cudaMemcpy(M.elements, Md.elements, size,
                        cudaMemcpyDeviceToHost);
```

Assignment 2 Pseudocode

[short detour, helpful with assignment]



Problem 2 can be implemented as follows (four steps):

Step 1: Allocate memory on the device (see cudaMalloc)

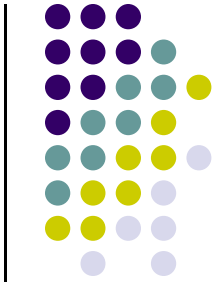
Step 2: Invoke kernel with one block, the block has four threads (see vector add example for passing the device pointer to the kernel)

NOTE: each of the four threads populates the allocated device memory with the result it computes

Step 3: Copy back to host the data in the device array (see cudaMemcpy)

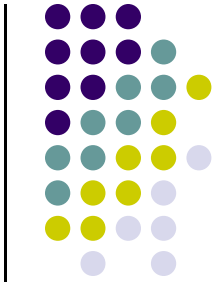
Step 4: Free the memory allocated on the device (see cudaFree)

Simple Example: Matrix Multiplication

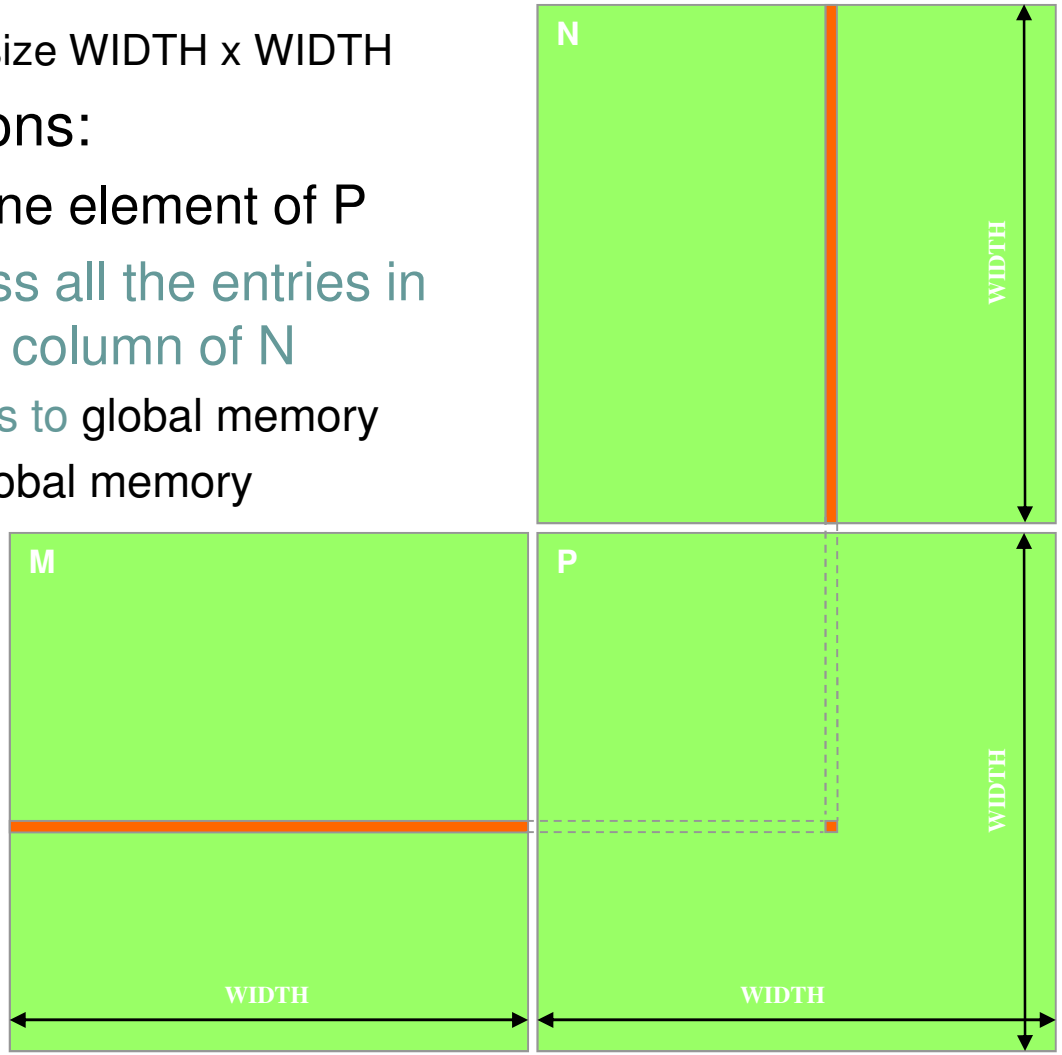


- A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - For now, concentrate on
    - Local variable and register usage
    - Thread ID usage
    - Memory data transfer API between host and device

Square Matrix Multiplication Example



- Compute P = M * N
  - The matrices P, M, N are of size WIDTH x WIDTH
- Software Design Decisions:
  - One thread handles one element of P
  - Each thread will access all the entries in one row of M and one column of N
    - 2*WIDTH read accesses to global memory
    - One write access to global memory

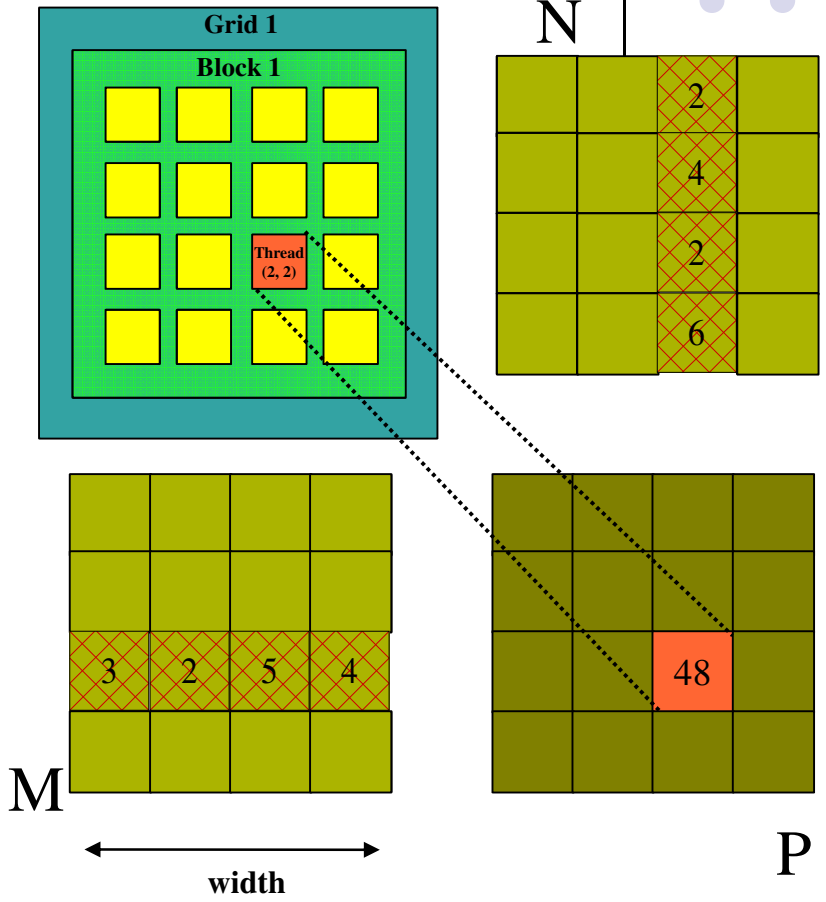


Multiply Using One Thread Block

- One Block of threads computes matrix P
  - Each thread computes one element of P

- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1
    - Not that good, acceptable for now…

- Size of matrix limited by the number of threads allowed in a thread block

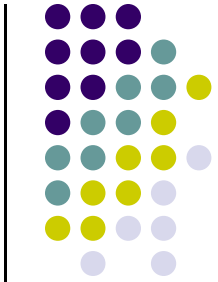


Step 1: Matrix Multiplication
A Simple Host Code in C

// Matrix multiplication on the (CPU) host in double precision;

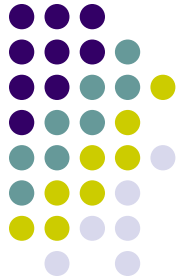
```
void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i) {
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k]; //you'll see a lot of this…
                double b = N.elements[k * N.width + j];  // and of this as well…
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
    }
}
```

Step 2: Matrix Multiplication, Host-side. Main Program Code



```
int main(void) {
    // Allocate and initialize the matrices.
    // The last argument in AllocateMatrix: should an initialization with
    // random numbers be done? Yes: 1.  No: 0 (everything is set to zero)
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);

   return 0;
}
```

Step 3: Matrix Multiplication Host-side code



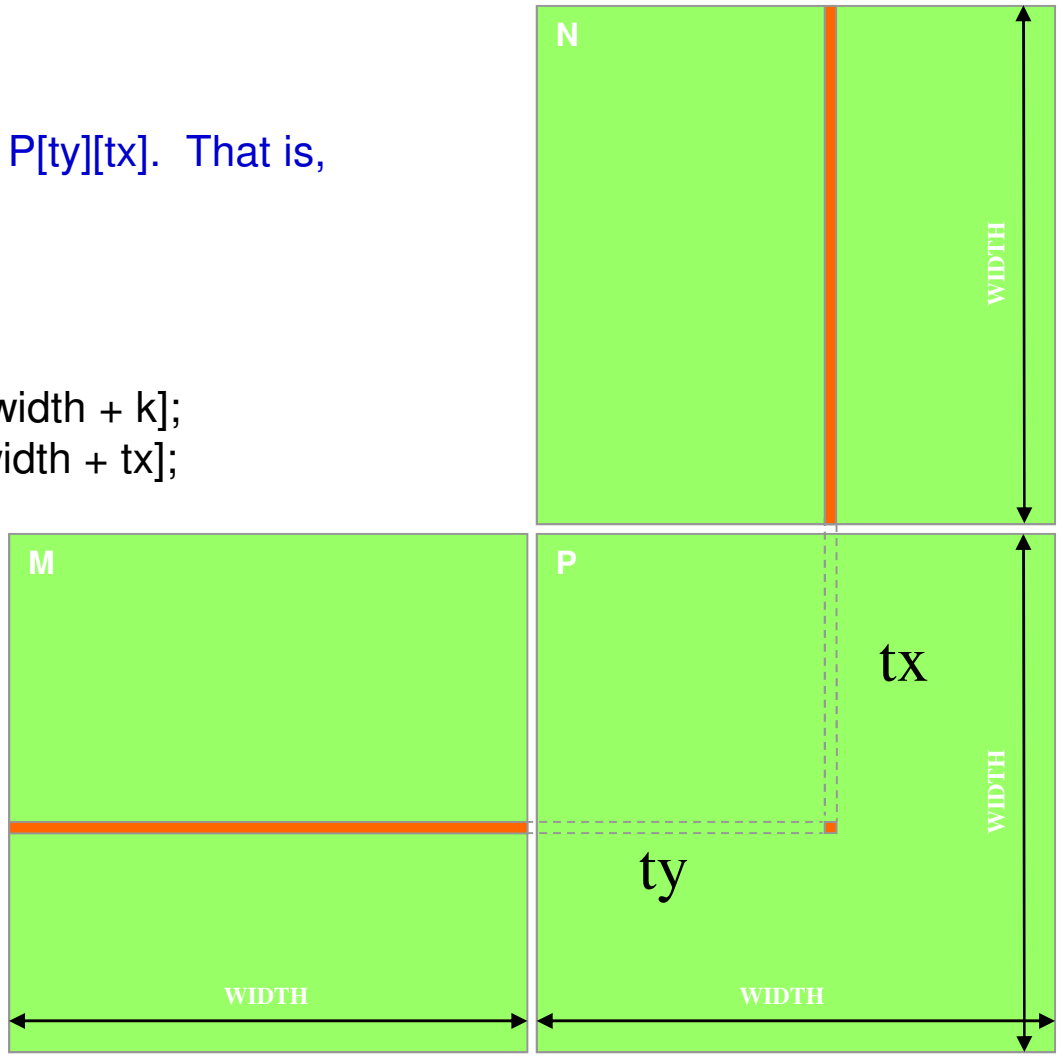
```
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const
Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);
```

Continue here…

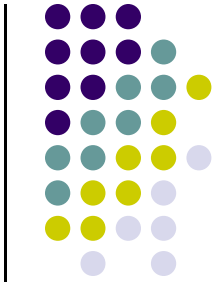
```
    // Setup the execution configuration
    dim3 dimGrid(1, 1);
    dim3 dimBlock(WIDTH, WIDTH);

    // Launch the kernel on the device
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```

Step 4: Matrix Multiplication- Device-side Kernel Function

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 2D Thread Index.  In the business of computing P[ty][tx]…
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue will end up storing the value of P[ty][tx].  That is,
    // P.elements[ty * P. width + tx] = Pvalue
    float Pvalue = 0;

    for (int k = 0; k < M.width; ++k)  {
         float Melement = M.elements[ty * M.width + k];
         float Nelement = N.elements[k * N. width + tx];
         Pvalue += Melement * Nelement;
    }

    // Write the matrix to device memory;
    // each thread writes one element
    P.elements[ty * P. width + tx] = Pvalue;
}
```



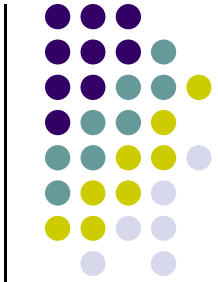
Step 5: Some Loose Ends



```
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M)
{
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}


// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost)
{
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size,
               cudaMemcpyHostToDevice);
}


// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice)
{
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size,
               cudaMemcpyDeviceToHost);
}
```

```
// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}


void FreeMatrix(Matrix M) {
    free(M.elements);
}
```

The Common Pattern to CUDA Programming

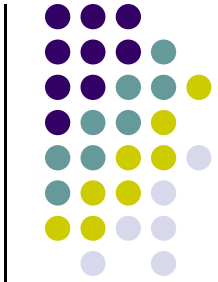


- **Phase 1**: Allocate memory on the device and copy to the device the data required to carry out computation on the GPU

- **Phase 2**: Let the GPU crunch the numbers based on the kernel that you defined

- **Phase 3**: Bring back the results from the GPU. Free memory on the device (clean up…). You're done.

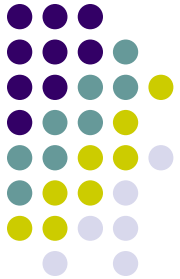
Timing Your Application



- Timing support – part of the API
  - You pick it up as soon as you include <cuda.h>
  - Why is good to use
    - Provides cross-platform compatibility
    - Deals with the asynchronous nature of the device calls by relying on events and forced synchronization
  - Resolution: miliseconds.
    - From NVIDIA CUDA Library Documentation:
      - Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). If either event has not been recorded yet, this function returns cudaErrorInvalidValue. If either event has been recorded with a non-zero stream, the result is undefined.

Timing Example

Timing a query of device 0 properties



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

int main() {
        cudaEvent_t startEvent, stopEvent;
        cudaEventCreate(&startEvent);
        cudaEventCreate(&stopEvent);

        cudaEventRecord(startEvent, 0);

        cudaDeviceProp deviceProp;
        const int currentDevice = 0;
        if (cudaGetDeviceProperties(&deviceProp, currentDevice) == cudaSuccess)
                printf("Device %d: %s\n", currentDevice, deviceProp.name);

        cudaEventRecord(stopEvent, 0);
        cudaEventSynchronize(stopEvent);
        float elapsedTime;
        cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);
        std::cout << "Time to get device properties: " << elapsedTime << " ms\n";

        cudaEventDestroy(startEvent);
        cudaEventDestroy(stopEvent);
        return 0;
}
```