Welcome to

Programming Massively Parallel Processors (PMPP)

Prof. Dr.-Ing. Michael Goesele
Dr. Stefan Guthe
Dominik Wodniok

Graphics, Capture and Massively Parallel Computing (GCC)
TU Darmstadt

(Preliminary) Course Schedule

TECHNISCHE UNIVERSITÄT DARMSTADT

you are here →

12.10.2015	Introduction to PMPP
13.10.2015	Lecture CUDA Programming 1
19.10.2015	Lecture CUDA Programming 2
20.10.2015	Lecture CUDA Programming 3
26.10.2015	Introduction Final Projects, Exercise 1 assigned
27.10.2015	Questions and Answers (Q&A)
2.11.2015	Lecture, Final Projects assigned, Ex. 1 due, Ex. 2 assigned
3.11.2015	Questions and Answers (Q&A)
9.11.2015	Lecture, Exercise 2 due
10.11.2015	Lecture
16.11.2015	Questions and Answers (Q&A)
17.11.2015	Questions and Answers (Q&A)
23.11.2015	1st Status Presentation Final Projects
24.11.2015	1st Status Presentation Final Projects (continued)
30.11.2015	
1.12.2015	

Review: Why PMPP?

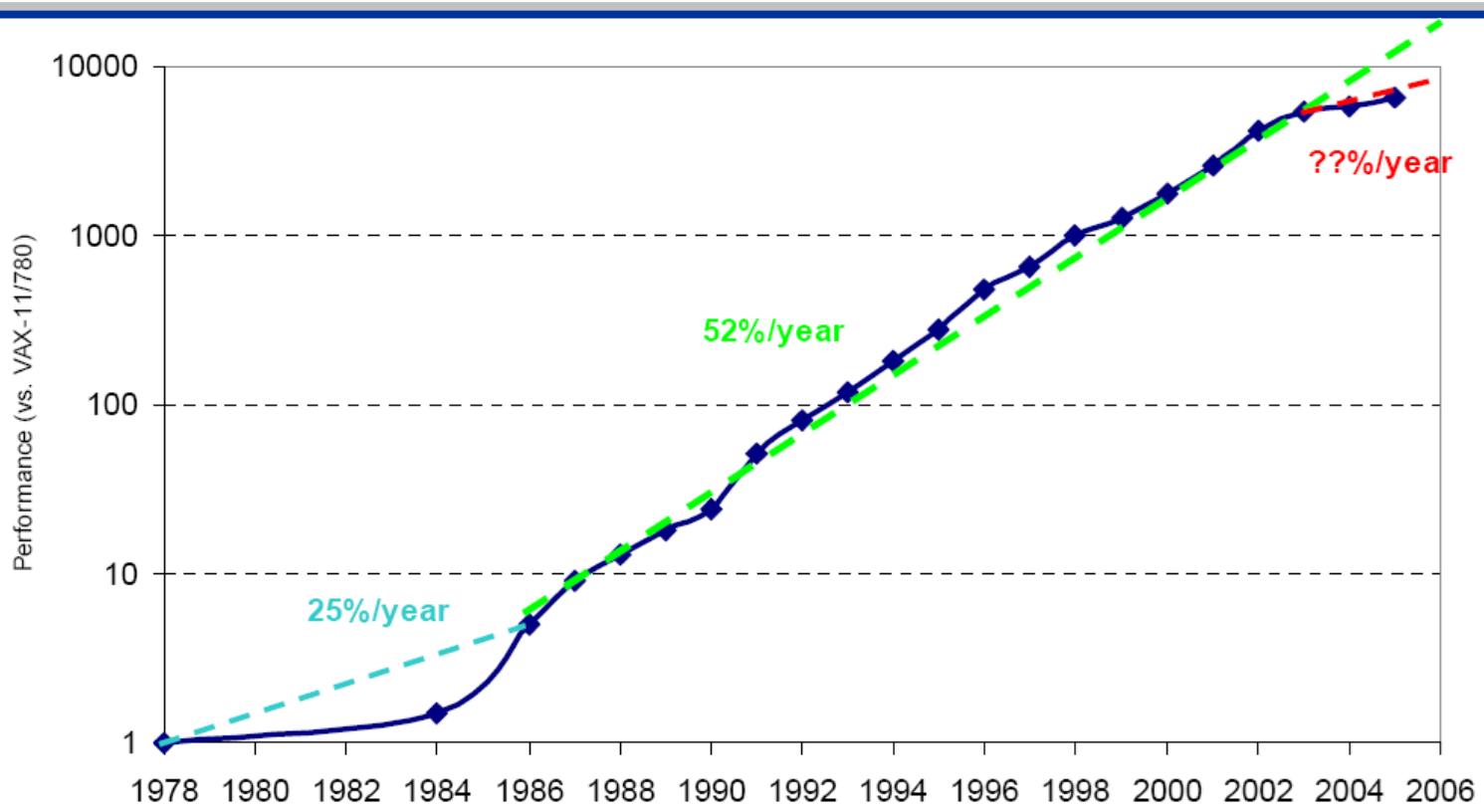
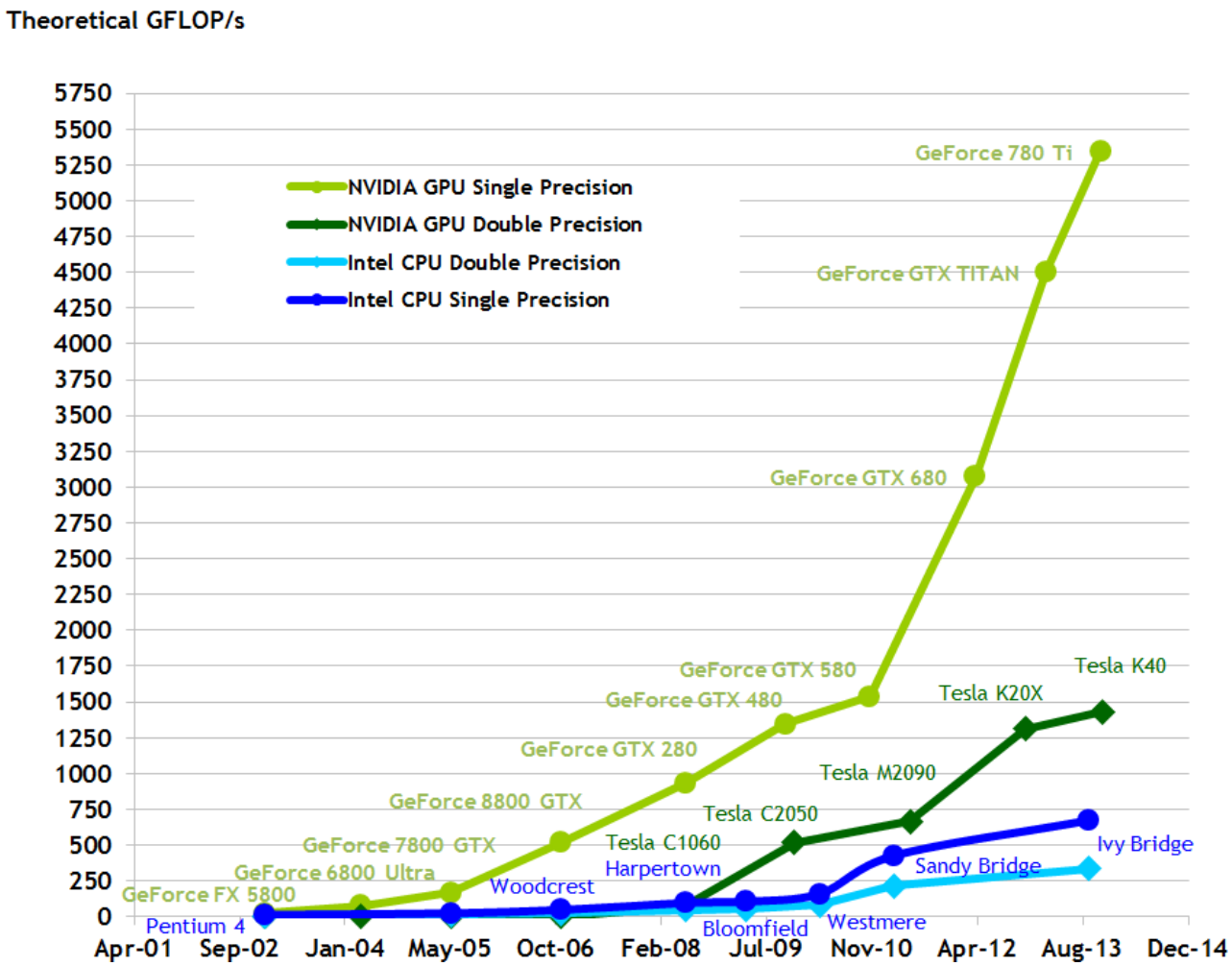


Figure 2. Processor performance improvement between 1978 and 2006 using integer SPEC [SPEC 2006] programs. RISCs helped inspire performance to improve by 52% per year between 1986 and 2002, which was much faster than the VAX minicomputer improved between 1978 and 1986. Since 2002, performance has improved less than 20% per year. By 2006, processors will be a factor of three slower than if progress had continued at 52% per year. This figure is Figure 1.1 in [Hennessy and Patterson 2007].

from [Asanovic et al. 2006]

Comparing GPU and CPU

- floating point operations per second



source: NVIDIA

Today's Topics

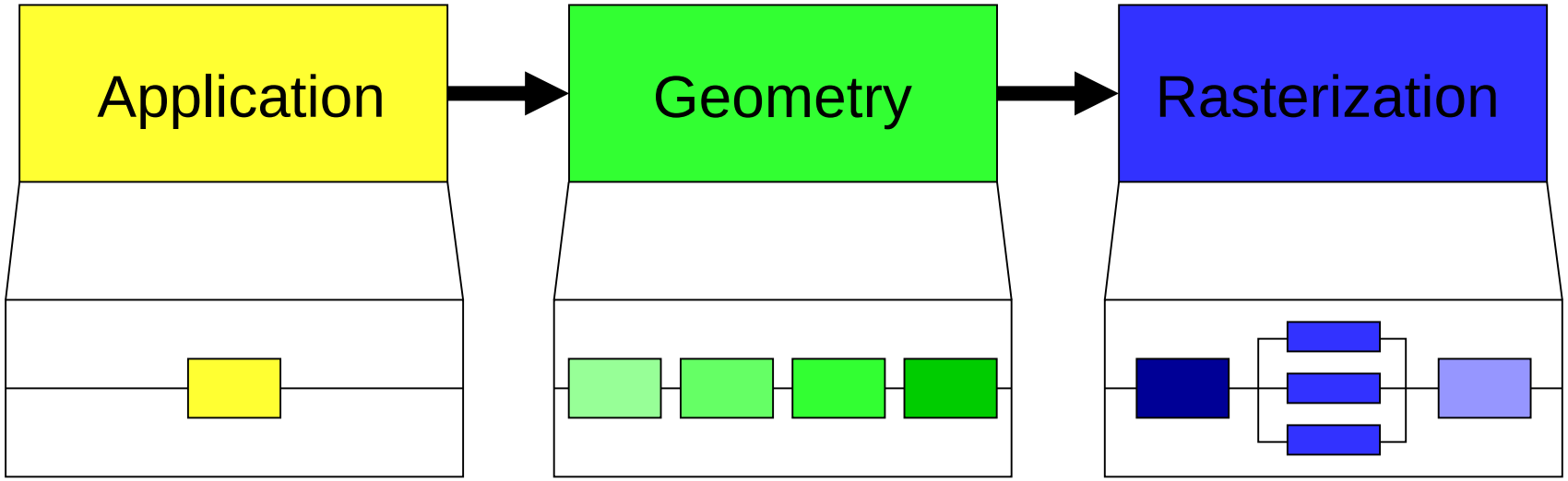
- CUDA "Hello World"
- CUDA programming model
  - basic concepts and data types
- CUDA application programming interface - basic
- simple matrix multiplication example
  - illustrate basic concepts and functionalities
  - performance features will be covered later

What is GPGPU ?

- General Purpose computation using GPU in applications other than 3D graphics
  - GPU (Graphics Processing Unit) 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 www.GPGPU.org
  - game effects (FX) physics, image processing
  - physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting, …

What is GPGPU ?

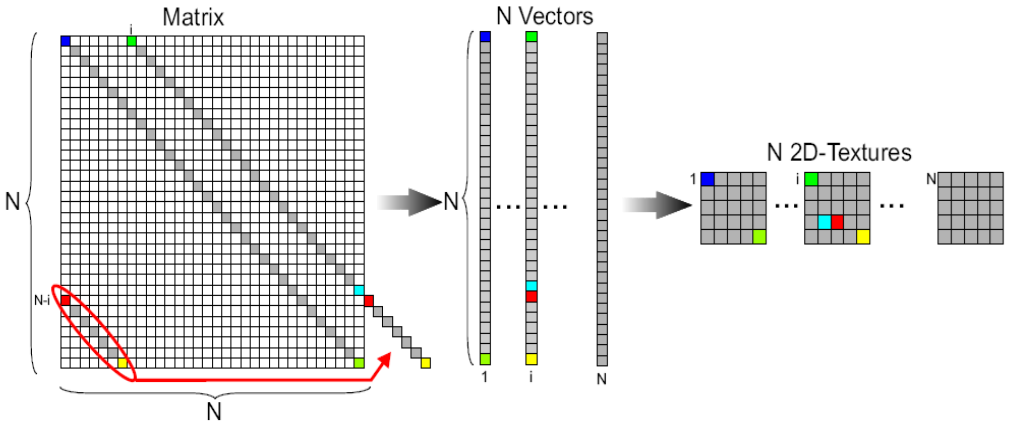
- traditional graphics pipeline
  - implemented as fixed-function hardware



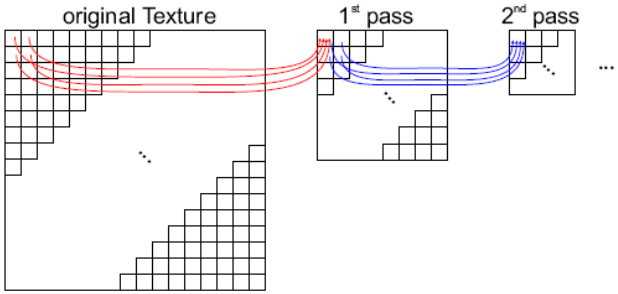
GPGPU Example

- Jens Krüger, Rüdiger Westermann: Linear algebra operators for GPU implementation of numerical algorithms. SIGGRAPH 2003, pages 908-916, 2003.

- linear algebra on GPU
- vectors, matrices mapped to textures



- algorithms implemented as rendering passes
  - e.g., reduce operation



Typical Constraints for traditional GPGPU

- dealing with graphics API
  - working with the corner cases of the graphics API
- addressing modes
  - limited texture size/dimension
- shader capabilities
  - limited outputs
- instruction sets
  - lack of integer & bit ops
- communication limited
  - between pixels
  - scatter a[i] = p

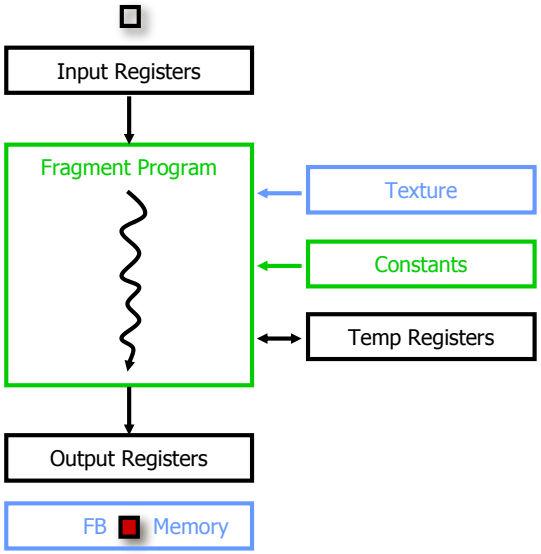


image source: NVIDIA

Modern GPU Architecture

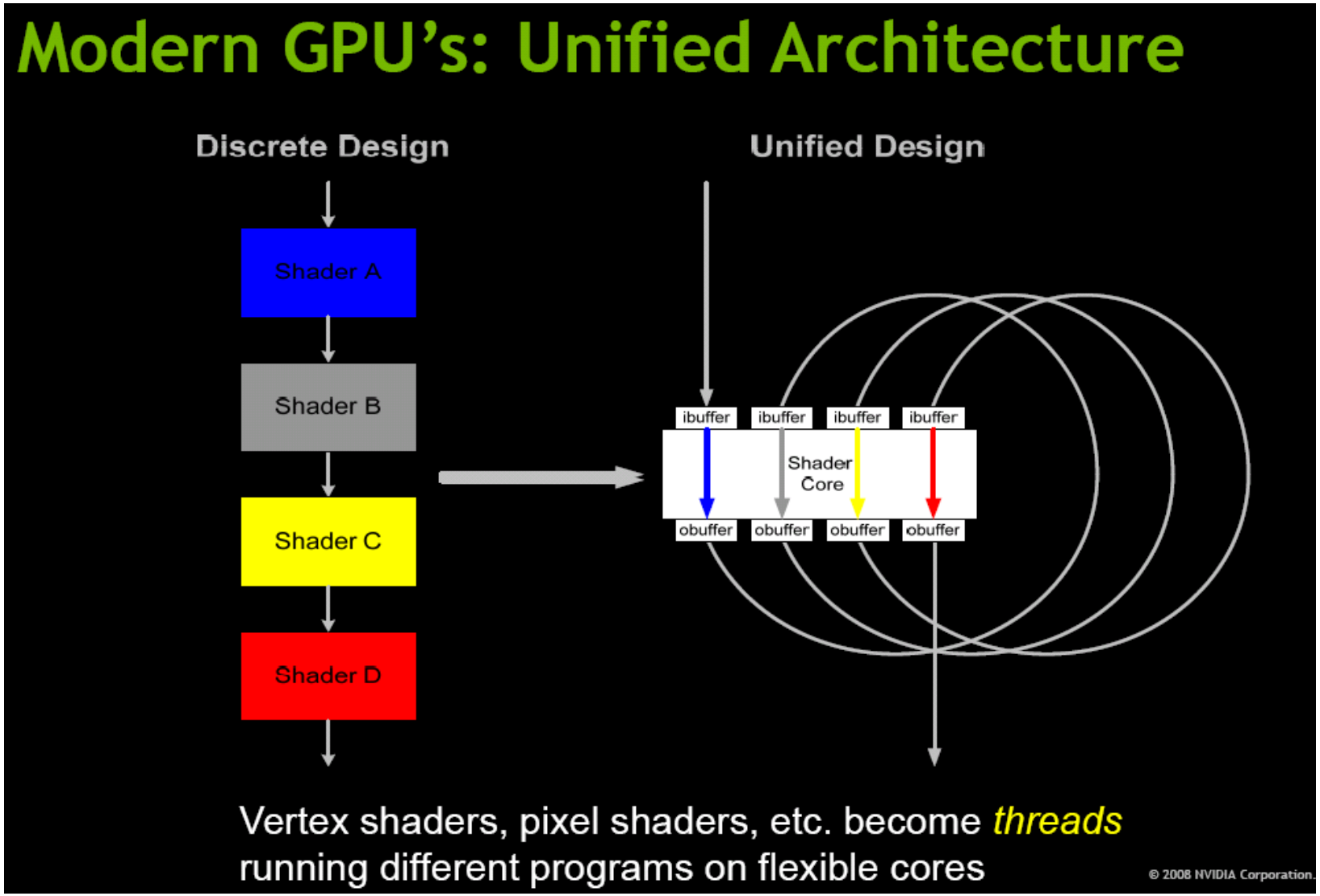


image source: NVIDIA

CUDA

- "Compute Unified Device Architecture"
- general purpose programming model
  - user kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- targeted software stack
  - compute-oriented drivers, language, and tools
- driver for loading computation programs into GPU
  - standalone driver – optimized for computation
  - interface designed for compute – graphics free API
  - data sharing with OpenGL buffer objects
  - guaranteed maximum download & readback speeds
  - explicit GPU memory management

image source: NVIDIA

High Level View

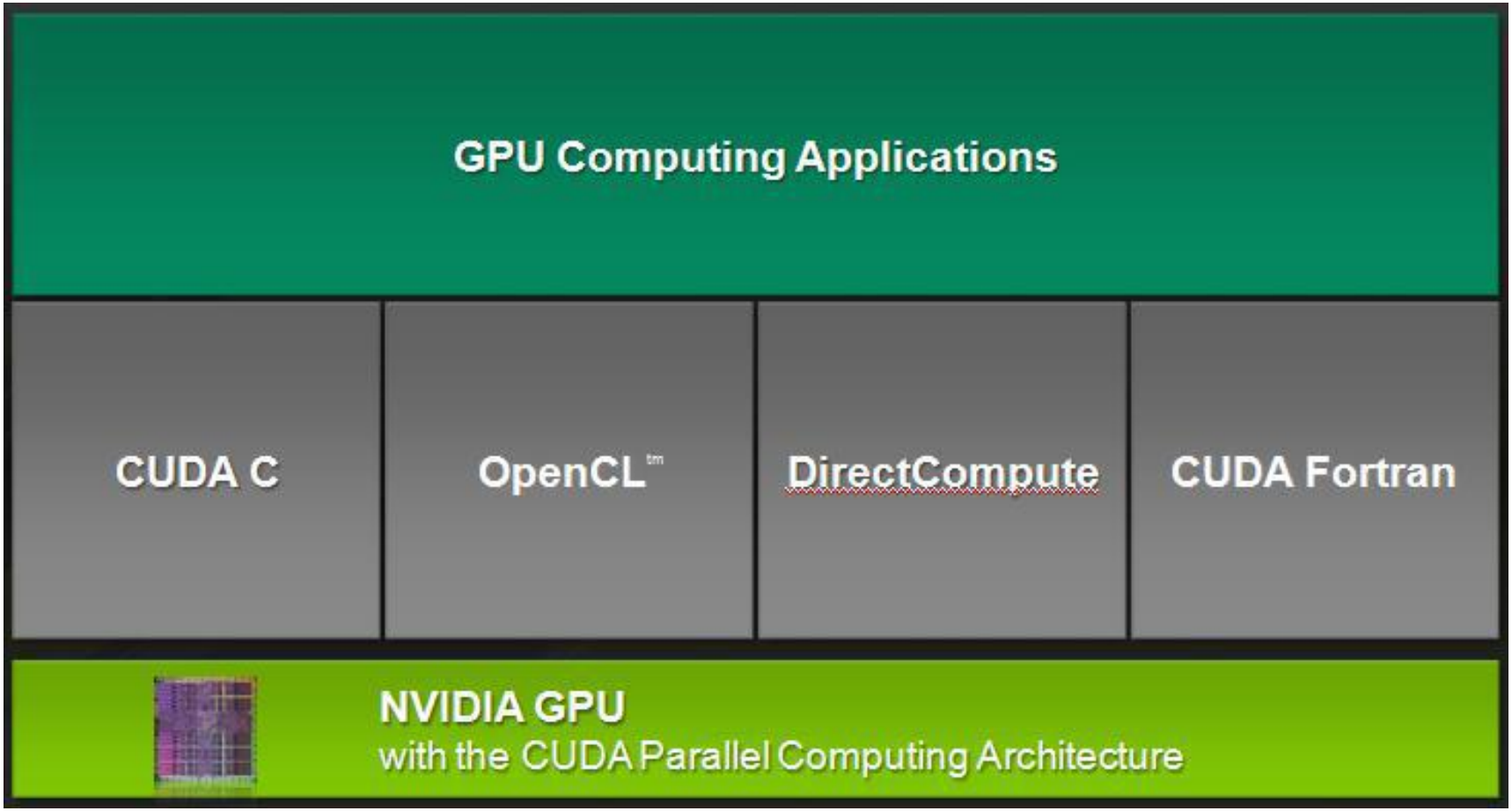


image source: NVIDIA

Extended C

- **declspecs**
  - **global**
  - **device**
  - **shared**
  - **local**
  - **constant**

- **keywords**
  - **threadIdx**
  - **blockIdx**

- **intrinsics**
  - **__syncthreads**

```
__device__ float filter[N];
__global__ void convolve (float *image) {
  __shared__ float region[M];
  [...]

  region[threadIdx.x] = image[i];

  __syncthreads()
  [...]
  image[j] = result;
}
```

Extended C

- **runtime API**
  - **memory management**
  - **symbol management**
  - **execution management**

```
// Allocate GPU memory
void *myimage = cudaMalloc(bytes)
```

- **function launch**

```
// 100 blocks
// 10 threads per block
convolve<<<100, 10>>> (myimage);
```

CUDA Programming Model

- the GPU (graphics processing unit) is viewed as a compute device that
  - is a coprocessor to the CPU or host
  - has its own DRAM (device memory)
  - runs many threads in parallel
- data-parallel portions of an application are executed on the device as kernels running in parallel on many threads
- differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - multi-core CPU needs only a few

helloWorld.cu

```
#include <helper_cuda.h>
#include "helloWorld_kernel.cu.h"
```

various definitions, macros, …
include the code actually executed on the GPU

```
#define BLOCK_SIZE 16
#define GW (5 * BLOCK_SIZE)
```

defining constants

```
int main(int argc, char** argv) {
    printf("block size: %u \n", BLOCK_SIZE);
    printf("grid dimension: %u\n", GW/BLOCK_SIZE);
```

the main program
debug output on host

```
    dim3 threads(BLOCK_SIZE);
    dim3 grid(GW / BLOCK_SIZE);
```

number of threads per block
number of blocks in grid

```
    helloWorld<<< grid, threads >>>();
```

start kernel (executed on the device)

```
    getLastCudaError("Kernel execution failed");
}
```

check for success

helloWorld_kernel.cu.h

```
#pragma once

#include <stdio.h>


__global__ void helloWorld() {
    if ( blockIdx.x==0 &&  threadIdx.x==0) {
    printf("Welcome to the Hello World test program for CUDA.\n");
  }


  // output per thread
  printf("Hello World!\n");
  printf("Block index:  x: %u \n",  blockIdx.x);
  printf("Thread index:  x: %u\n",  threadIdx.x);
}
```

avoid multiple reads and interpretation of file

definition of a global function, callable from host

executed only once

welcome message

print message per thread

print block ID and thread ID (in block)

Hello (CUDA) World! - Output

```
block size: 16
grid dimension: 5
Welcome to the Hello World test program for CUDA.
Hello World!
Block index:  x: 0
Thread index:  x: 0
Hello World!
Block index:  x: 0
Thread index:  x: 1
Hello World!
Block index:  x: 0
Thread index:  x: 2

[…]

Block index:  x: 4
Thread index:  x: 14
Hello World!
Block index:  x: 4
Thread index:  x: 15
```

Blocks and Grids of Threads

- thread batching
- a kernel is executed as a grid of thread blocks
  - all threads share data memory space (global memory)
- a thread block is a batch of threads that can cooperate
  - synchronizing their execution
  - efficiently sharing data through shared memory
- two threads from two different blocks cannot cooperate

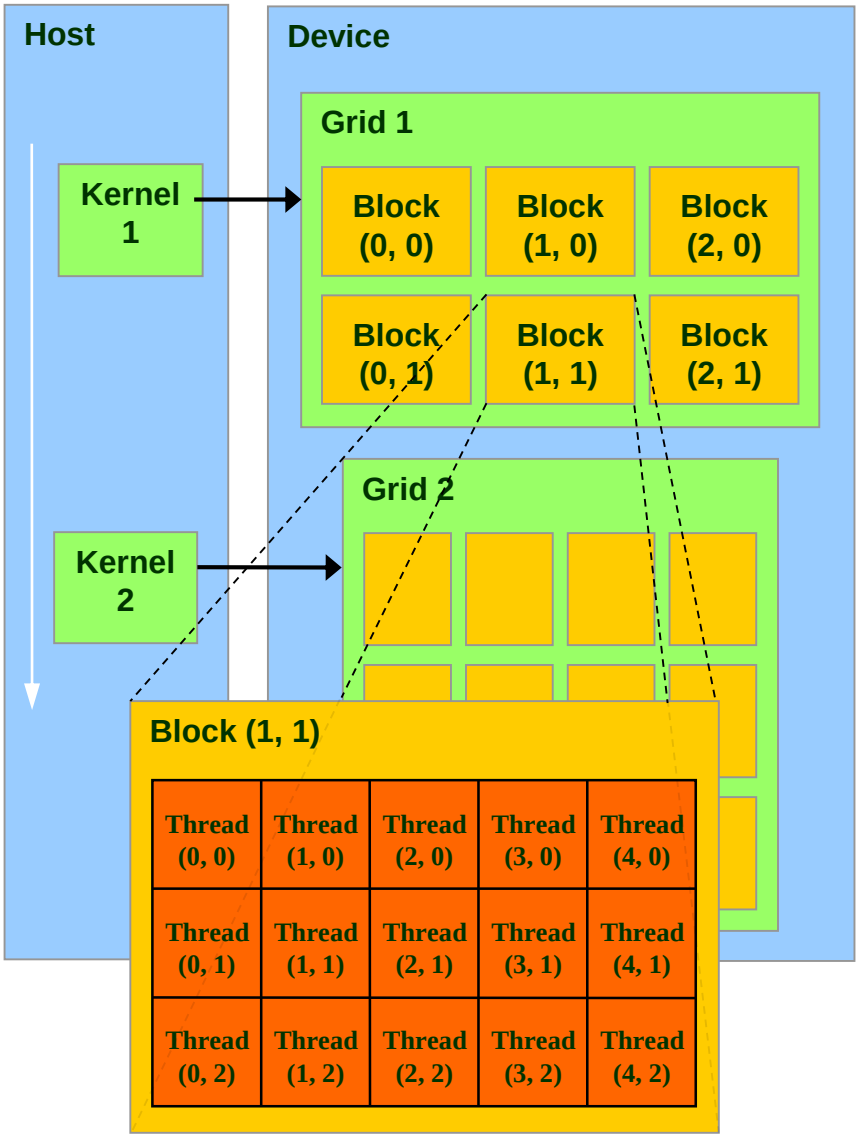


image source: NVIDIA

Blocks and Grids of Threads

- corresponding code in helloWorld.cu

```
#define BLOCK_SIZE 16
#define GW (5 * BLOCK_SIZE)

dim3 threads(BLOCK_SIZE);
dim3 grid(GW /BLOCK_SIZE);

helloWorld<<< grid, threads >>>();
```

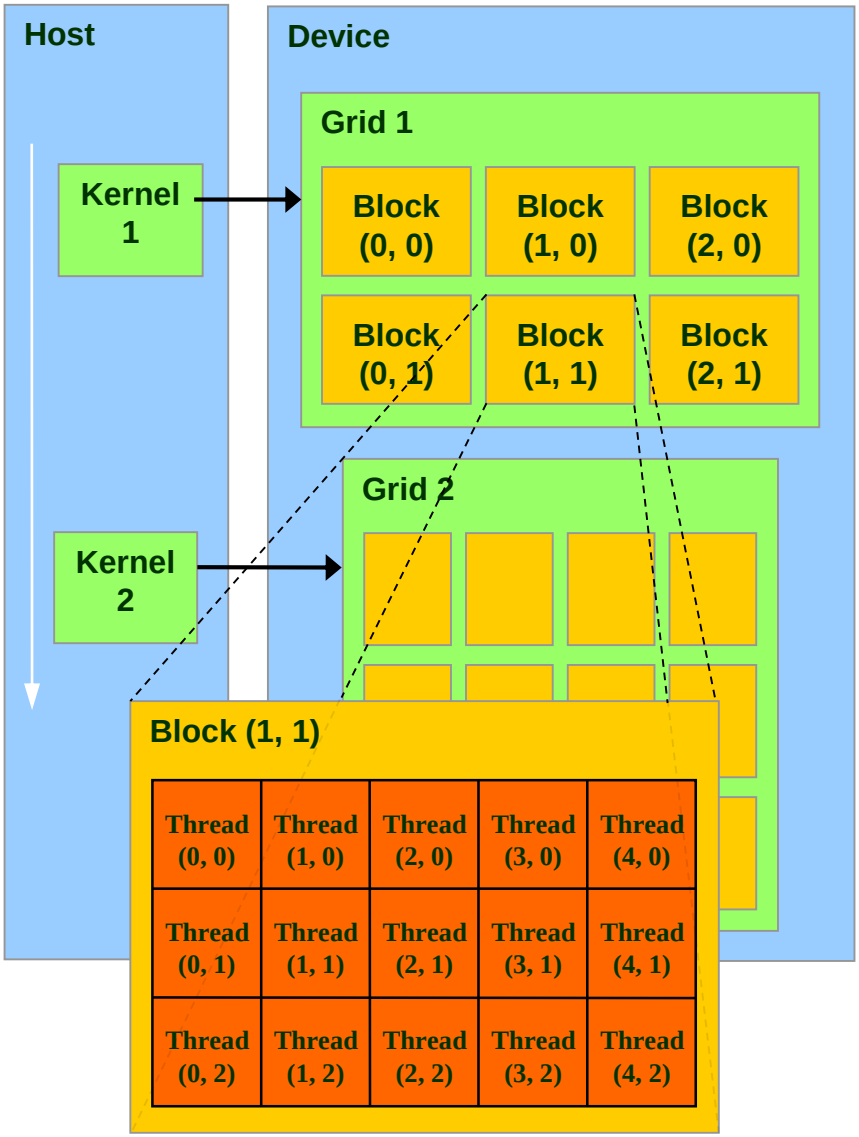


image source: NVIDIA

Block and Thread IDs

- threads and blocks have IDs
  - so each thread can decide what data to work on
  - block ID: 1D, 2D or 3D (>SM2.0)
  - thread ID: 1D, 2D, or 3D
  - data type dim3
- simplifies memory addressing when processing multidimensional data
  - image processing
  - solving PDEs on volumes
  - …

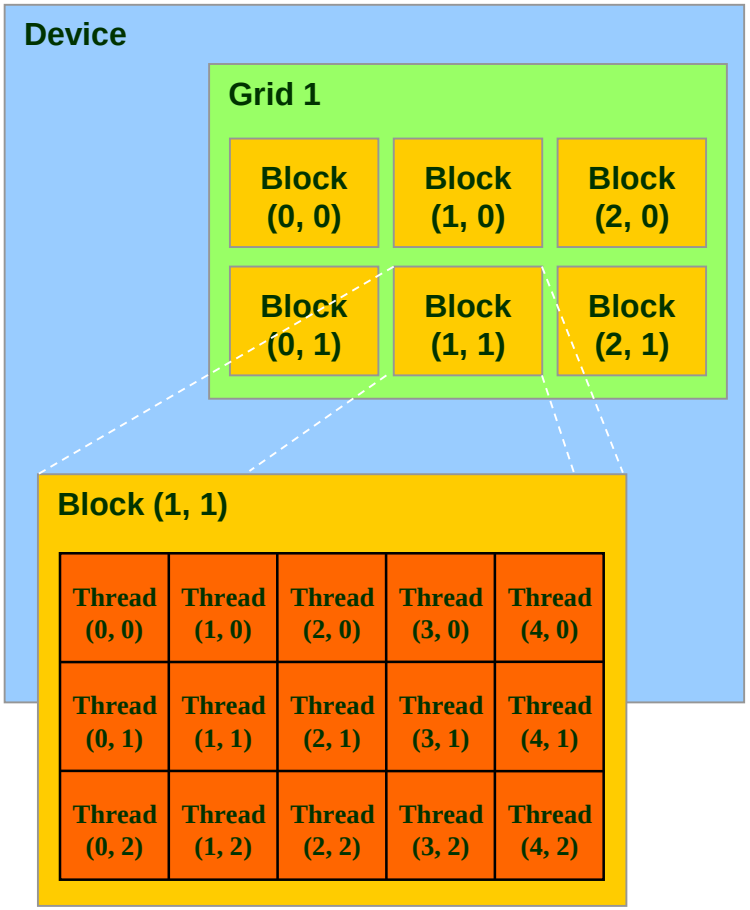


image source: NVIDIA

Block and Thread IDs

- corresponding code in helloWorld_kernel.cu.h

```
if ( blockIdx.x==0 &&  threadIdx.x==0) {
   printf("Welcome to the Hello …\n");
}

// output per thread
printf("Hello World!\n");
printf("Block index:  x: %u \n",
      blockIdx.x);
printf("Thread index:  x: %u\n",
      threadIdx.x);
```

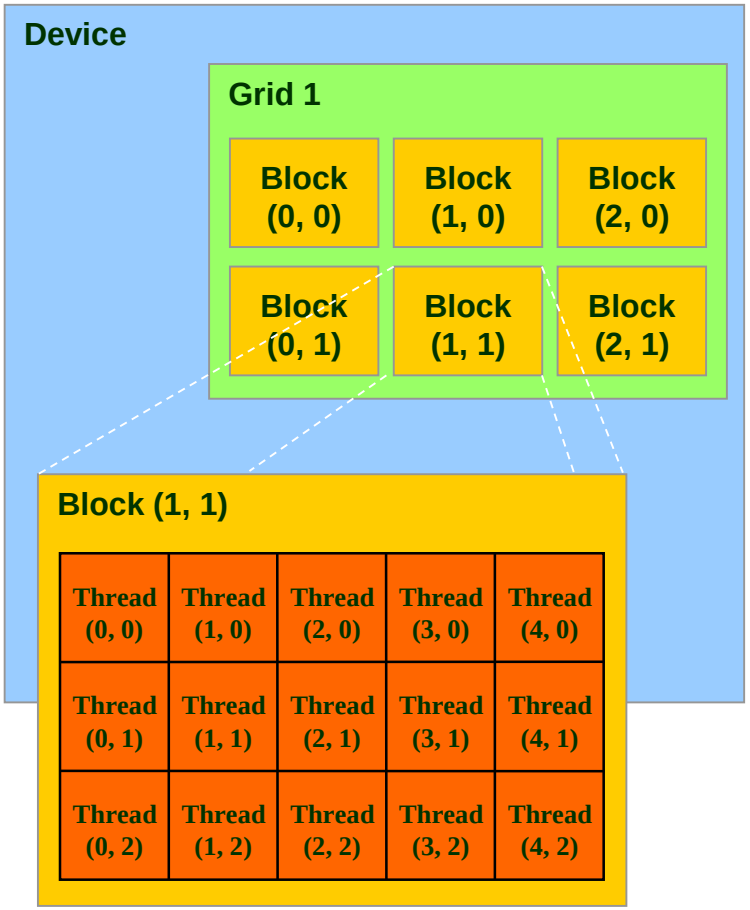
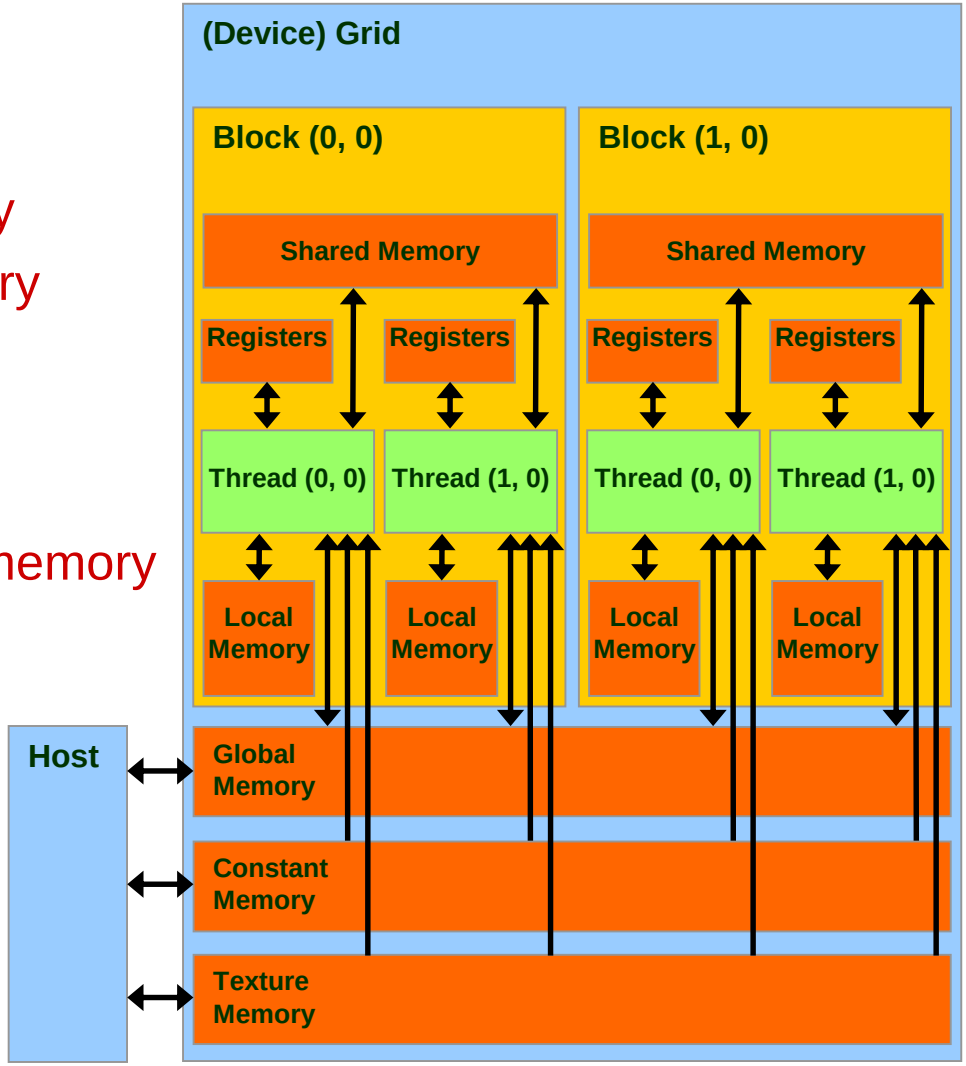


image source: NVIDIA

CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only* per-grid texture memory
- The host can
  - R/W global memory
  - R/W constant memory
  - R/W texture memory



* Can use surface memory for R/W access

image source: NVIDIA

Global, Constant, and Texture Memories

- long latency access
  - 400-600 clock cycles
- global memory
  - main means of communicating R/W data between host and device
  - contents visible to all threads
- texture and constant memories
  - constants initialized by host
  - contents visible to all threads
  - cached access
  - texture has built-in interpolation

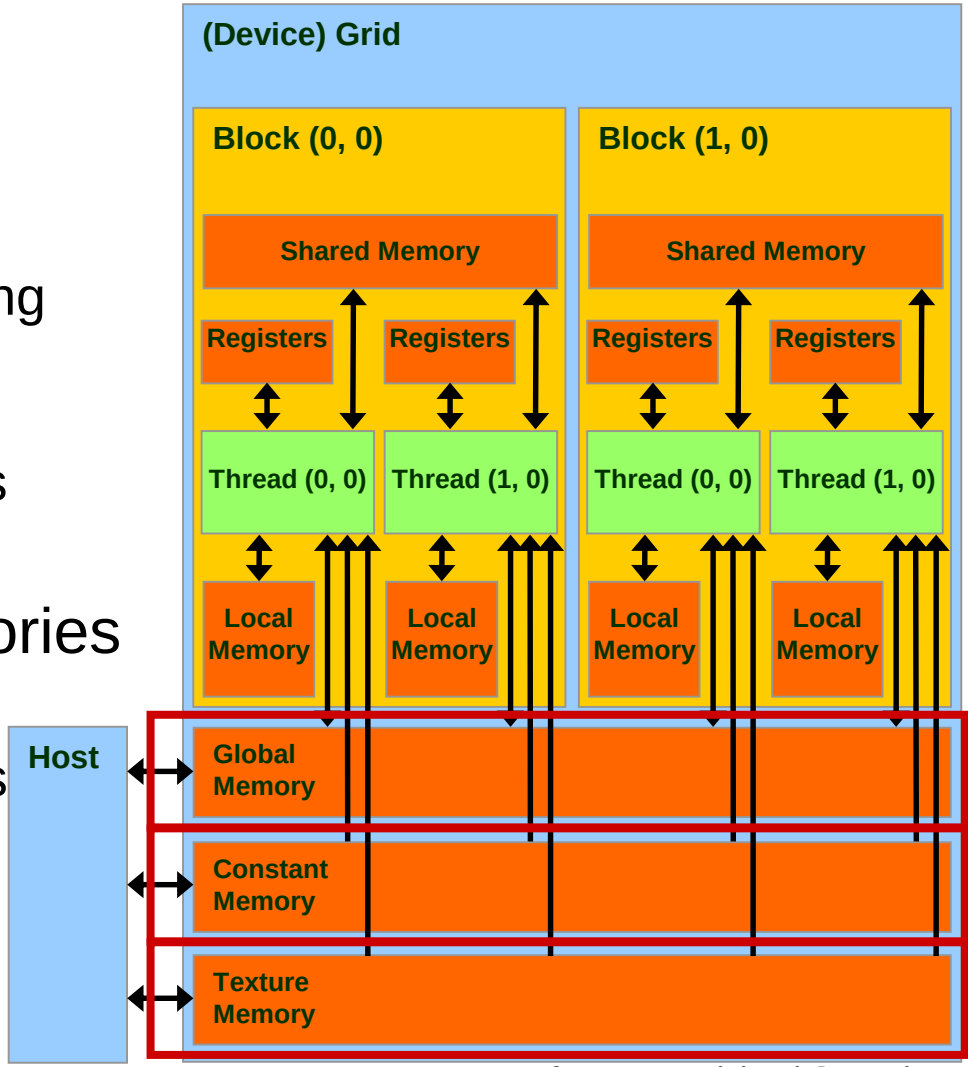


image source: NVIDIA

The CUDA API

- the API is an extension to the C/C++ programming language
  - ➔ low learning curve

- the hardware is designed to enable a lightweight runtime environment and driver
  - ➔ high performance

A Small Detour: A Matrix Data Type

- NOT part of CUDA
- It will be frequently used in many code examples
  - 2D matrix
  - single precision float elements
  - width * height elements
  - pitch meaningful for alignment and if the matrix is actually a sub-matrix of another matrix
  - data elements allocated and attached to elements

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

CUDA Device Memory Allocation

- cudaMalloc()
  - allocates object in the device global memory
  - requires two parameters
    - address of a pointer to the allocated object
    - size of allocated object
- cudaFree()
  - frees object from device global memory
    - pointer to freed object

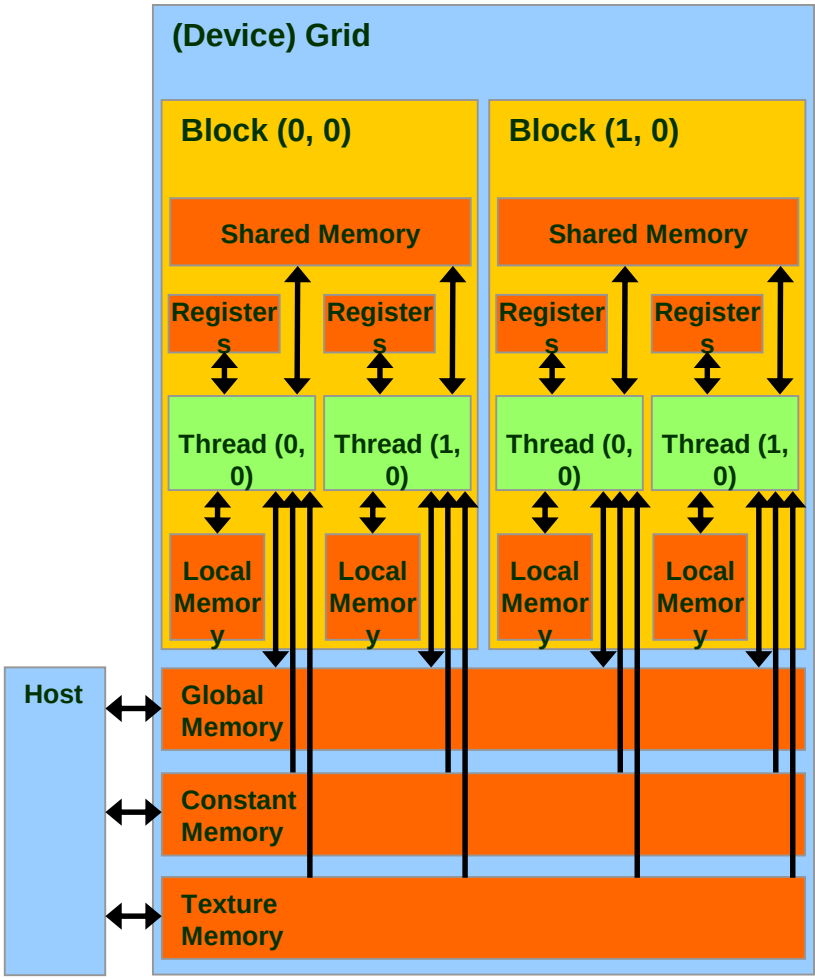


image source: NVIDIA

CUDA Device Memory Allocation

- code example:
  - allocate a 64*64 float array
  - attach the allocated storage to Md.elements
  - “d” is often used to indicate a device data structure

```
#define MATRIX_WIDTH 64
Matrix Md;
int size =
    MATRIX_WIDTH * MATRIX_WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);
cudaFree(Md.elements);
```

CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - requires four parameters
    - pointer to destination
    - pointer to source
    - number of bytes copied
    - type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- asynchronous since CUDA 1.1
  - cudaMemcpyAsync()

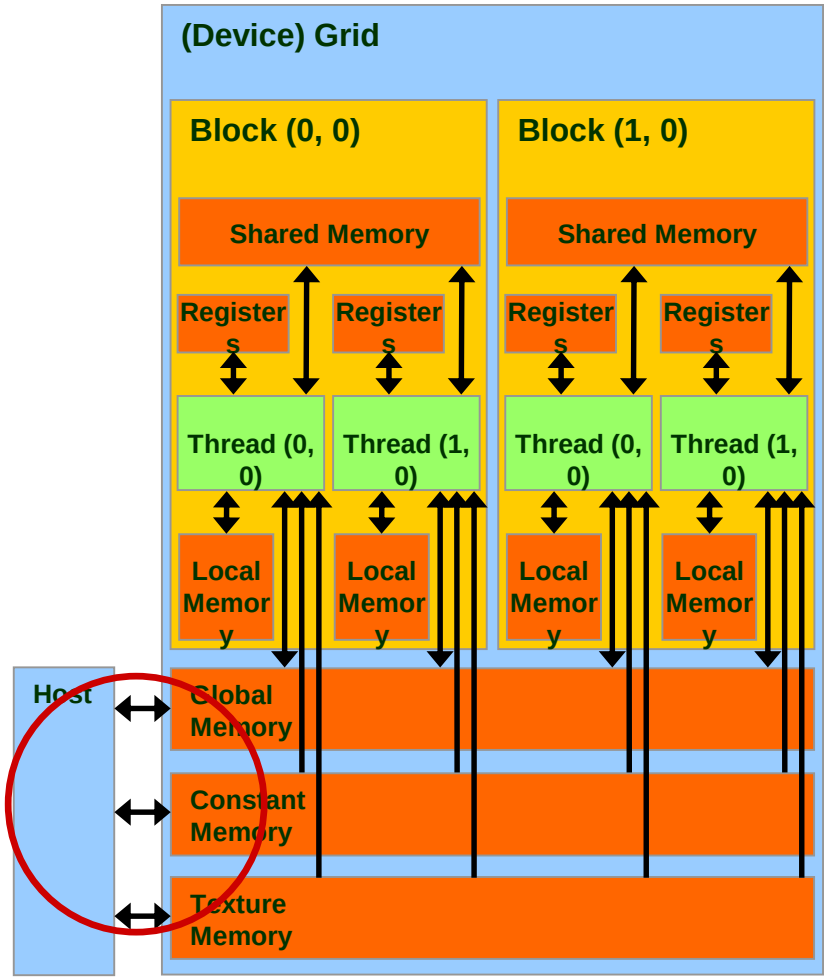


image source: NVIDIA

CUDA Host-Device Data Transfer

- 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);
```

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`**
  - can access **`blockIdx`** and **`threadIdx`**
- **`__device__`** and **`__host__`** can be used together
  - two versions of the function will then be compiled: one for the host and one for the device

CUDA Function Declarations

- `__device__` functions cannot have their address taken
- for functions executed on the device:
  - recursion only on cards with compute capability >= 2.x
  - no static variable declarations inside the function
  - no variable number of arguments

Calling a Kernel Function

- thread creation
- kernel function must be called with an execution configuration

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);     // 5000 thread blocks
dim3   DimBlock(4, 8, 8);    // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- any call to a kernel function is asynchronous, explicit synch needed for blocking

Simple Example: Matrix Multiplication

- straightforward matrix multiplication example
- illustrates the basic features of memory and thread management in CUDA programs
  - leave shared memory usage until later
  - local, register usage
  - thread ID usage
  - memory data transfer API between host and device

Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- without tiling
  - one thread handles one element of P
  - M and N are loaded WIDTH times from global memory

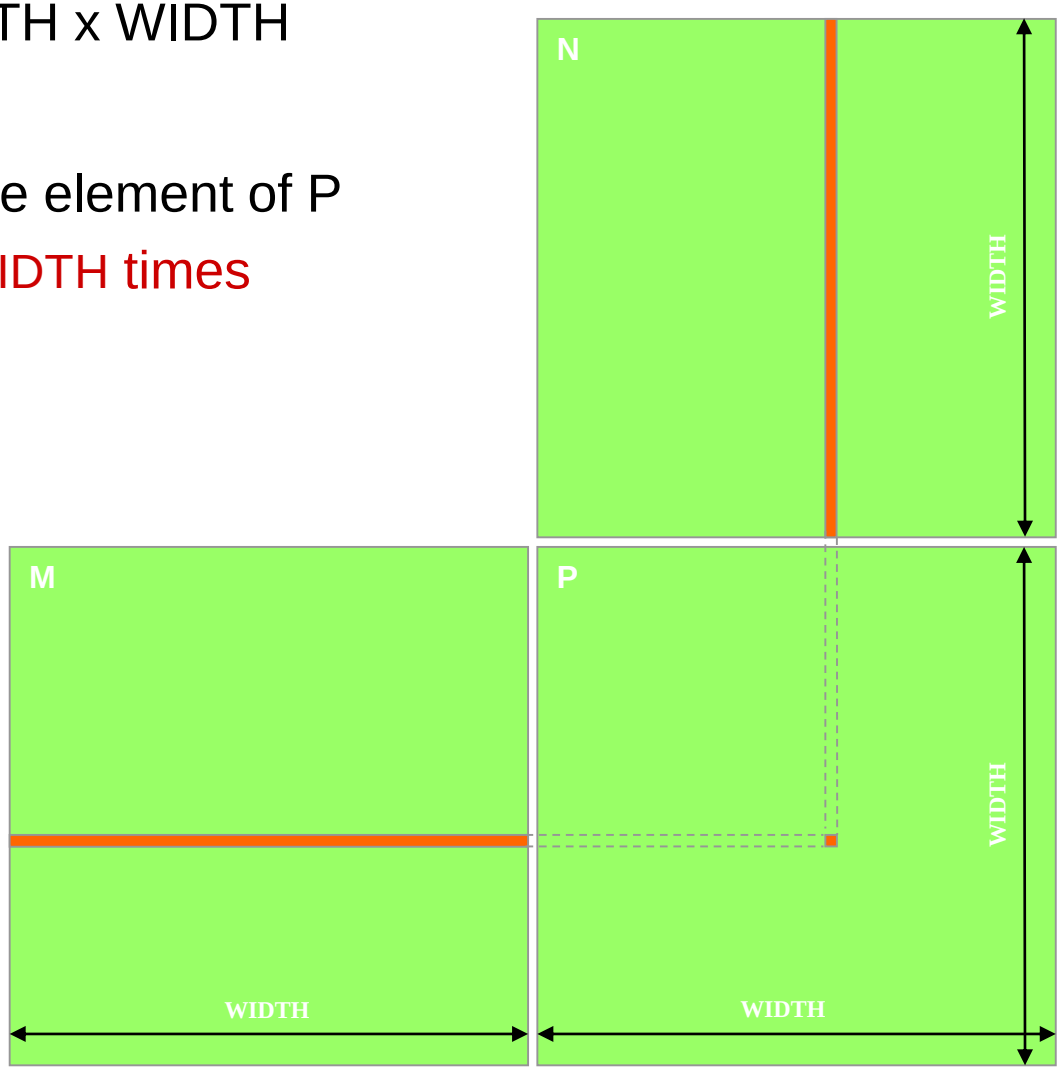


image source: NVIDIA

Step 1: Matrix Data Transfers

```
// Allocate the device memory where we will copy M to
Matrix Md;
Md.width  = WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);

// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);

...

// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size, cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(Md.elements);
```

Step 2: Matrix Multiplication

- simple host code in C
  - multiplication on the CPU host in double precision
  - for simplicity, we will assume that all dimensions are equal

```
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];
                double b = N.elements[k * N.width + j];
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
}
```

Multiply Using One Thread Block

- one block of threads compute matrix P
  - one element of P per thread
- each thread
  - loads a row of matrix M
  - loads a column of matrix N
  - perform one multiply and addition per pair of M and N elements
  - compute to off-chip memory access ratio close to 1:1 (not very high)
- size of matrix limited by the number of threads allowed in a thread block

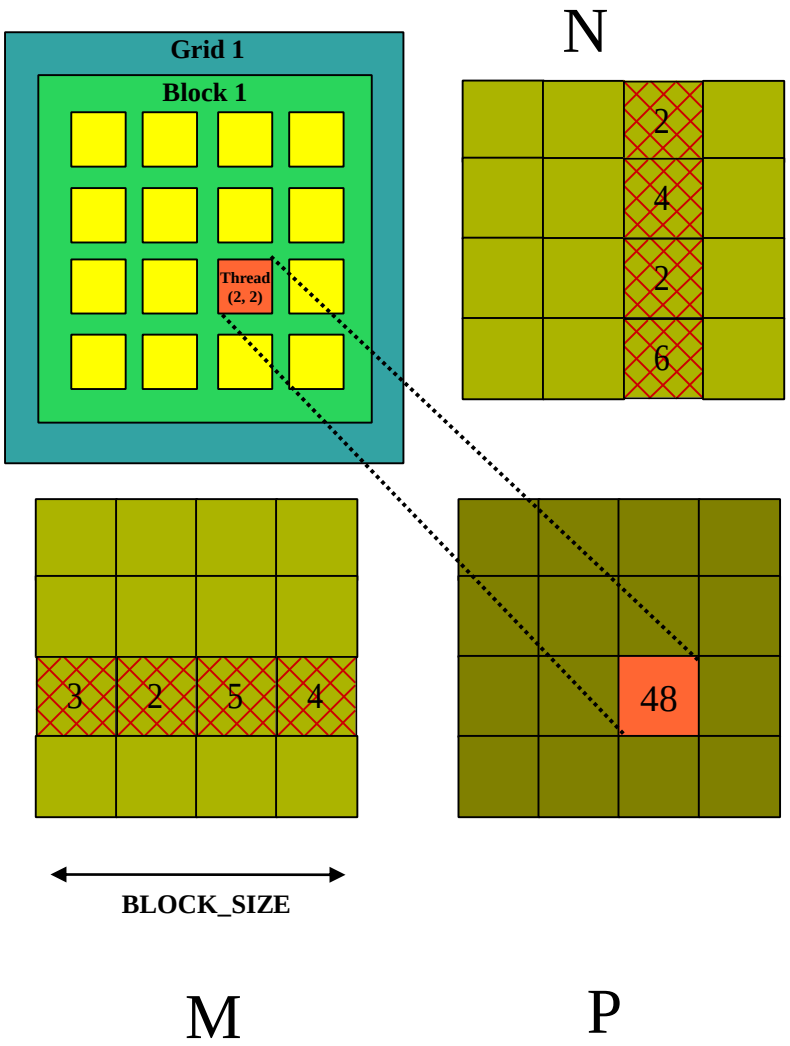


image source: NVIDIA

Step 3: Matrix Mul. – Host-Side Code 1

```
int main(void) {
  // Allocate and initialize the matrices
  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 Mul. – Host-Side Code 2

```
// 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);
  CopyToDeviceMatrix(Pd, P); // Clear memory
```

Step 3: Matrix Mul. – Host-Side Code 3

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

  // Launch the device computation threads!
  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: Device-side Kernel Function 1

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N,
                                Matrix P) {
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue is used to store the element of the matrix
  // that is computed by the thread
  double Pvalue = 0;
```

Step 4: Device-side Kernel Function 2

```
  for (int k = 0; k < M.width; ++k) {
    float Melement =
      M.elements[ty * M.pitch + k];
    float Nelement =
      N.elements[k * N.pitch + tx];
    Pvalue += Melement * Nelement;
  }
  // Write the matrix
  // to device memory;
  // each thread writes
  // one element
  P.elements[ty * P.pitch + tx] = Pvalue;
}
```

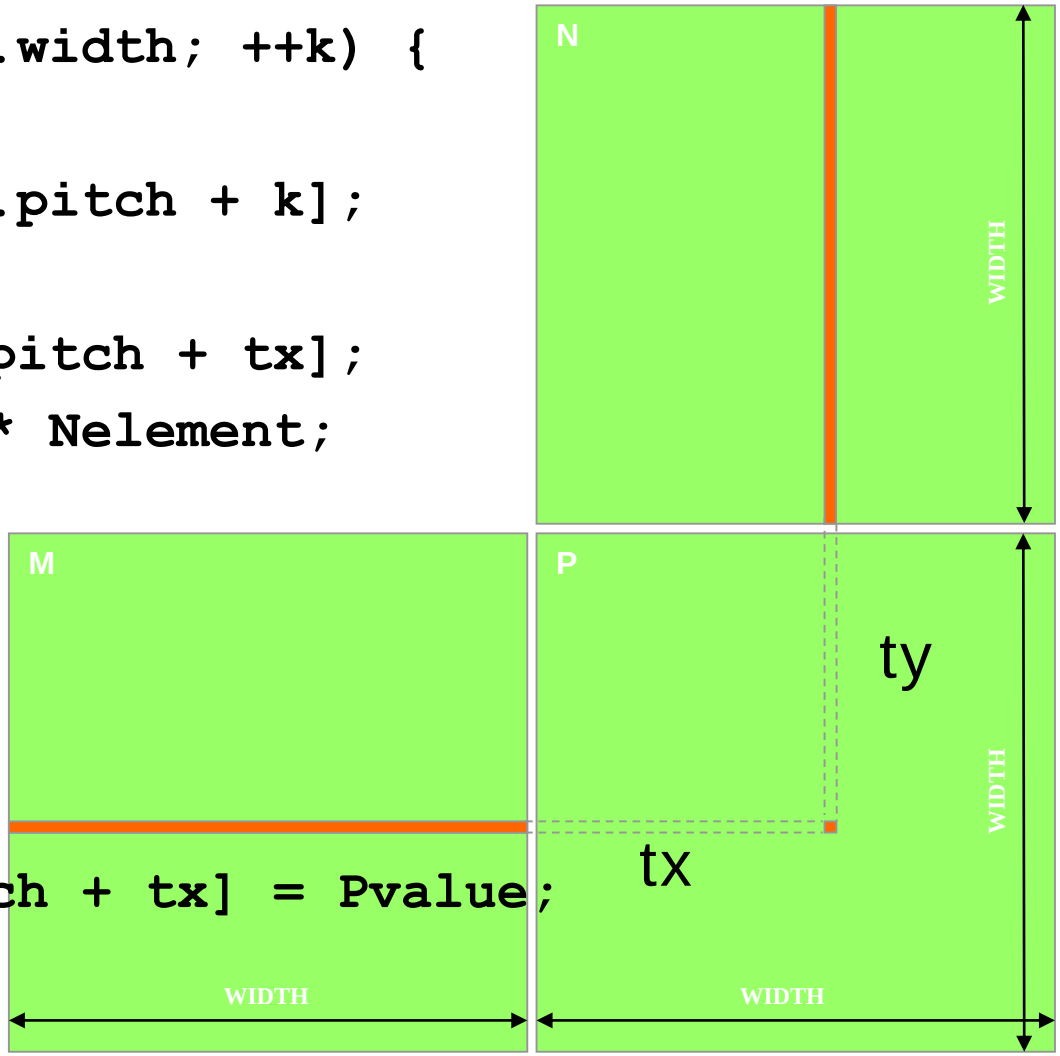


image source: NVIDIA

Step 5: Some Loose Ends 1

```
// 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;
}

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

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

Step 5: Some Loose Ends 2

```
// 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);
}
```

Step 6: Arbitrary Sized Square Matrices

- each 2D thread block computes a $(BLOCK\_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - each has $(BLOCK\_WIDTH)^2$ threads
- 2D grid
  - $(WIDTH/BLOCK\_WIDTH)^2$ blocks
- need to put a loop around the kernel call for cases where WIDTH is greater than max grid size!

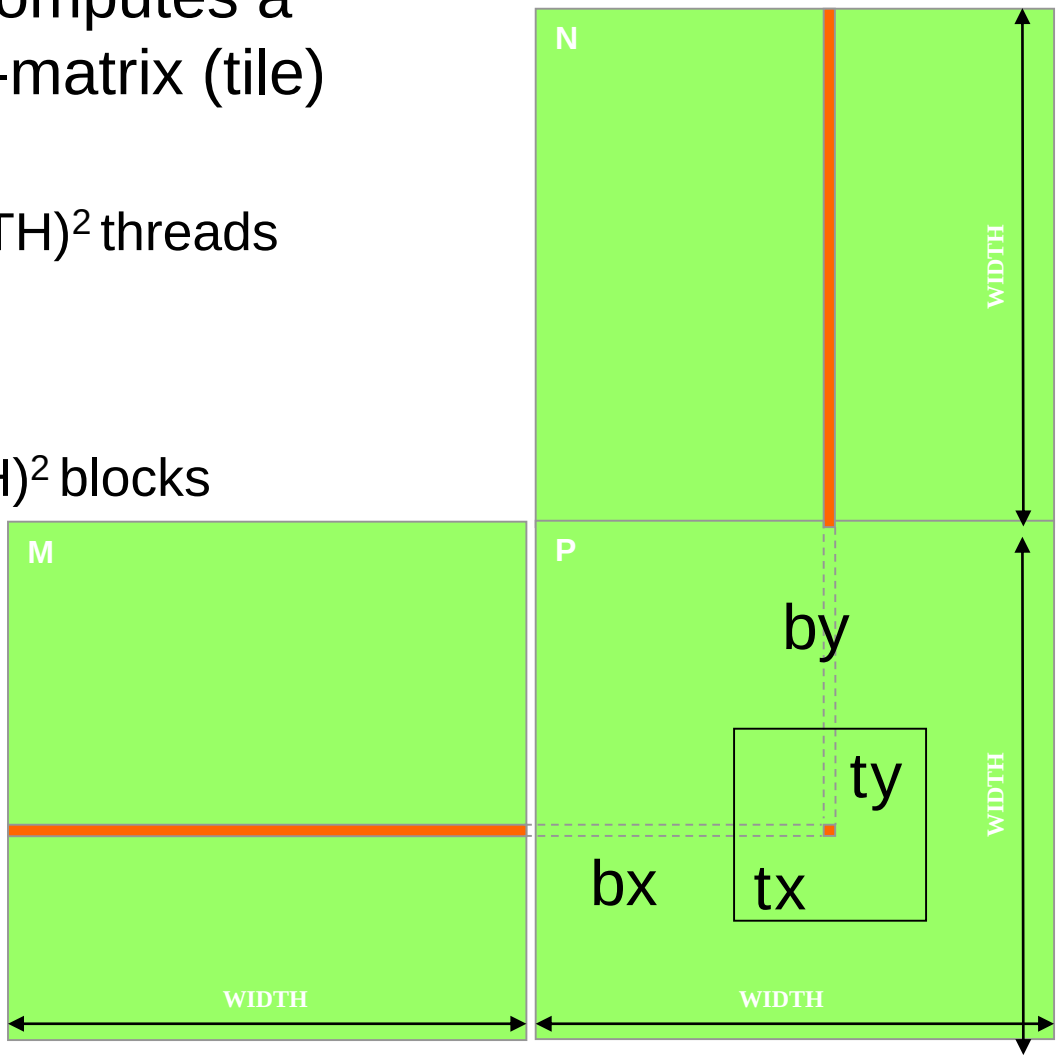


image source: NVIDIA

Assignments and Accounts

- Exercise 1 – due Monday 2.11.2015
- read Chapters 1+2 of the CUDA 7.5 Programming Guide
  - available on NVIDIA's web page http://docs.nvidia.com/
- accounts will be handed out to you ASAP