Intermediate CUDA® Programming

Jian Tao

jtao@tamu.edu

Fall 2017 HPRC Short Course

10/31/2017

Relevant Short Courses and Workshops

Introduction to CUDA Programming

https://hprc.tamu.edu/training/intro_cuda.html

Bring-Your-Own-Code Workshop

https://coehpc.engr.tamu.edu/byoc/

Offered regularly

CUDA Programming Abstractions

Key Programming Abstractions

Three key abstractions that are exposed to CUDA programmers as a minimal set of language extensions:

- **a hierarchy of thread groups**
- **shared memories**
- **barrier synchronization**

Glossary

- **Thread** is an abstract entity that represents the execution of the kernel, which is a small program or a function.
- **Grid** is a collection of Threads. Threads in a Grid execute a Kernel Function and are divided into Thread Blocks.
- **Thread Block** is a group of threads which execute on the same multiprocessor (SMX). Threads within a Thread Block have access to shared memory and can be explicitly synchronized.

CUDA Kernels

- CUDA kernels are C functions that, when called, are executed N times in parallel by N different CUDA threads.
- A kernel is defined with `__global__` declaration specifier.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
```

Kernel Invocation

- The number of CUDA threads that execute a kernel is specified using a new `<<<...>>>`execution configuration syntax.
- Each thread that executes the kernel is given a unique **thread ID** that is accessible within the kernel through the built-in 3-component vector **`threadIdx`**.

```
// Kernel Invocation with N threads
VecAdd<<<1, N>>>(A, B, C);
```

Example 1 - Kernel Definition

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}
```

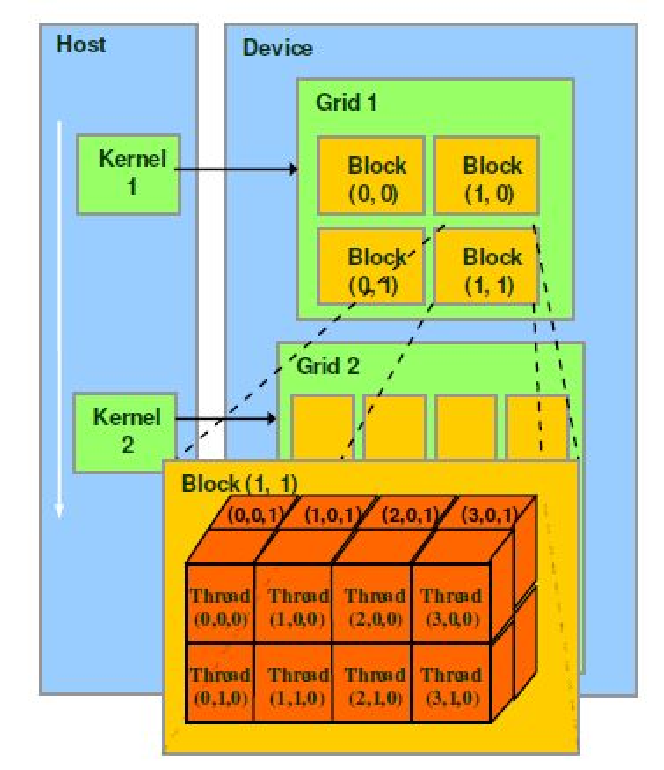
Example 1 - Kernel Invocation

```
// Kernel invocation
int main()
{
   ...
// Call kernel with one block of N * N * 1 threads
   int numBlocks = 1;
   dim3 threadsPerBlock(N, N);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
   ...
}
```

Hierarchy of Threads

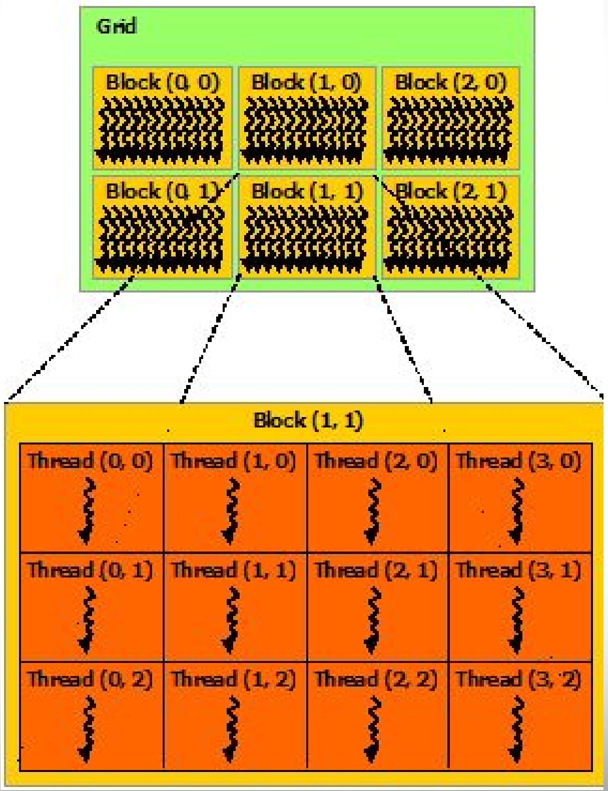
Thread Hierarchy - I

- 1D, 2D, or 3D threads can form 1D, 2D, or 3D **thread blocks**.
- 1D, 2D, or 3D blocks can form 1D, 2D, or 3D **grid of thread blocks**
- The number of threads per block and the number of blocks per grid are specified in the <<<. . . .>>> syntax.



Thread Hierarchy - II

- Each block within the grid can be identified by an index accessible within the kernel through the built-in 3-component vector **`blockIdx`**.
- The dimension of the thread block is accessible within the kernel through the built-in 3-component vector **`blockDim`**.

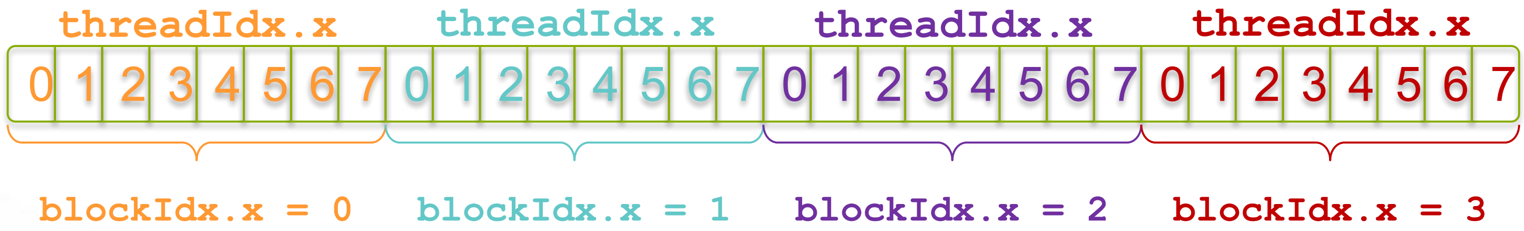


Thread Index and Thread ID

- **1D**
  thread ID is the same as the index of a thread
- **2D**
  for a two-dimensional block of size **(blockDim.x, blockDim.y)**, the thread ID of a thread of index **(x, y)** is **(x + y * blockDim.x)**
- **3D**
  for a three-dimensional block of size **(blockDim.x, blockDim.y, blockDim.z)**, the thread ID of a thread of index **(x, y, z)** is **(x + y * blockDim.x + z * blockDim.x * blockDim.y)**

Indexing Arrays with Blocks and Threads

- Consider indexing an array with one element per thread (8 threads/block)

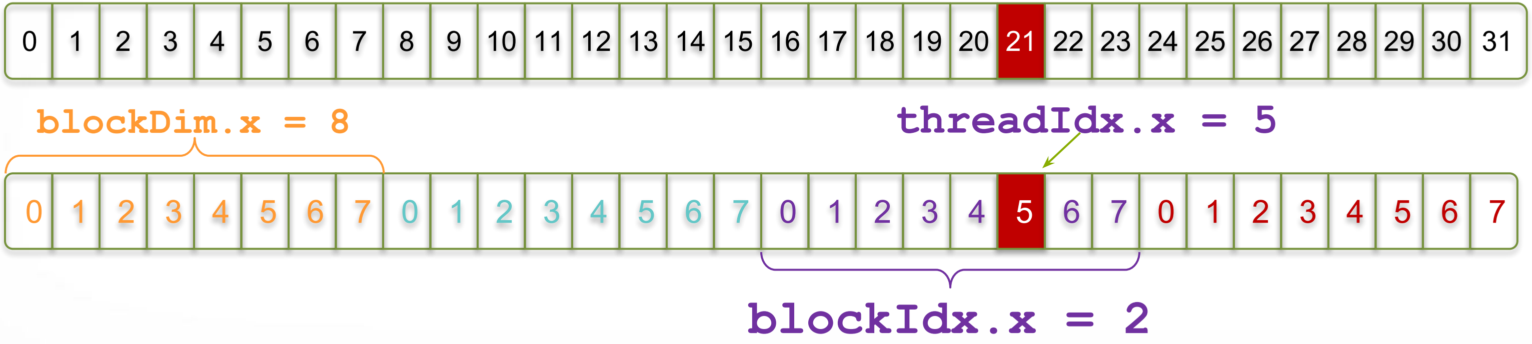


- With **`blockDim.x`** threads/block, the thread is given by:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Indexing Arrays: Example

- Which thread will operate on the red element?




```
int index = threadIdx.x + blockIdx.x * blockDim.x;
          =      5      +     2      *     8
          = 21
```

Example 2 - Kernel Definition

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < N && j < N)
    C[i][j] = A[i][j] + B[i][j];
}
```

Example 2 - Kernel Invocation

```
// Kernel invocation
int main()
{
   ...
// run kernel with multiple blocks of 16*16*1 threads
   dim3 threadsPerBlock(16, 16);
   dim3 numBlocks(N / threadsPerBlock.x, N /
threadsPerBlock.y);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
   ...
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void VecAdd(int *A, int *B, int *C, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        C[index] = A[index] + B[index];
}
```

Update the kernel launch: **M = blockDim.x**

```
VecAdd<<<(N + M-1) / M, M>>>(A, B, C, N);
```

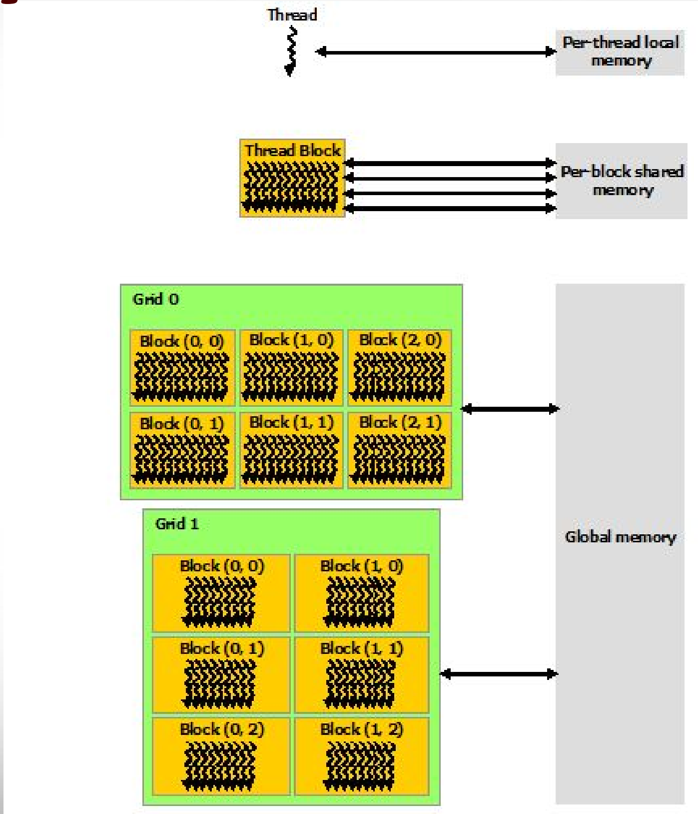
Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Threads within a block can cooperate by sharing data through some shared memory
- by synchronizing their execution to coordinate memory accesses with `__syncthreads()`

Memory Hierarchy

Hierarchical Memory Structure

- Each thread has access to **registers** and **private local memory**.
- Each thread block has **shared memory** visible to all threads of the block and with the same lifetime as the block.
- All threads have access to **global memory**.



Memory Spaces

- **Register, local**, **shared**, **global**, **constant (read only)**, and **texture (read only)** memory are the memory spaces available.
- Only register and shared memory reside on GPU.
- The **global, constant,** and **texture memory spaces** are cached and persistent across kernel launches by the same application.

Memory: Scope and Performance

- Data in **register memory** is visible only to the thread and lasts only for the lifetime of that thread.
- **Local memory** has the same scope rules as register memory, but performs slower.
- Data stored in **shared memory** is visible to all threads within that block and lasts for the duration of the block.
- Data stored in **global memory** is visible to all threads within the application (including the host), and lasts for the duration of the host allocation.
- **Constant memory** is used for data that will not change over the course of a kernel execution and is read only.
- **Texture memory** is another variety of read-only memory on the device.

Using Global Memory

- Linear memory is typically allocated using **`cudaMalloc()`** and freed using **`cudaFree()`** and data transfer between host and device is done using **`cudaMemcpy()`**.
- Linear memory can also be allocated through **`cudaMallocPitch()`** and **`cudaMalloc3D()`** and transferred using **`cudaMemcpy2D()`** and **`cudaMemcpy3D()`** with better memory alignment.

Using Shared Memory

- Much faster than global memory.
- Allocated using the **`__shared__`** memory space specifier.

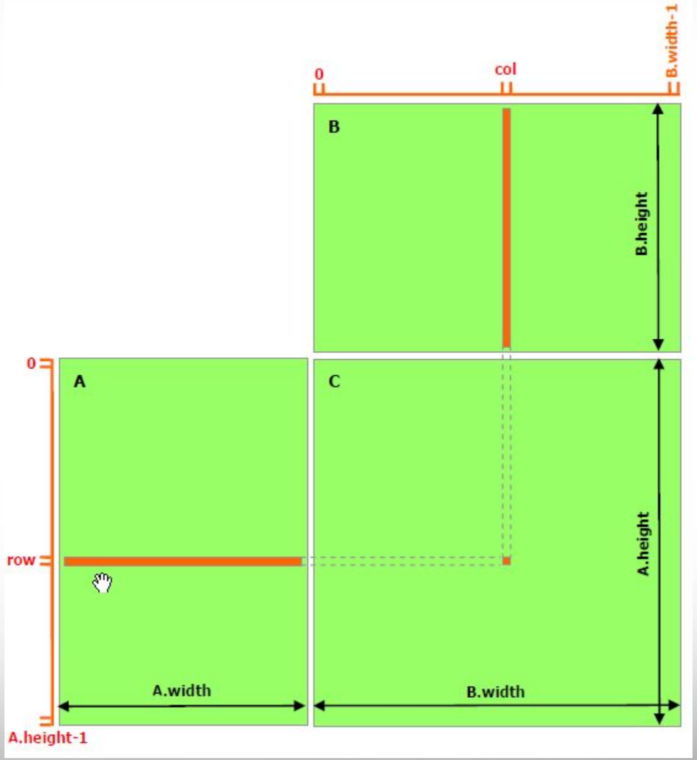
```
__shared__ float A[BLOCK_SIZE][BLOCK_SIZE];
```

- Shared memory shall be used as a cache for global memory to exploit locality of the code.

Example 3 - Matrix Multiplication w/o SM

Each thread computes one element of C by accumulating results into Cvalue.

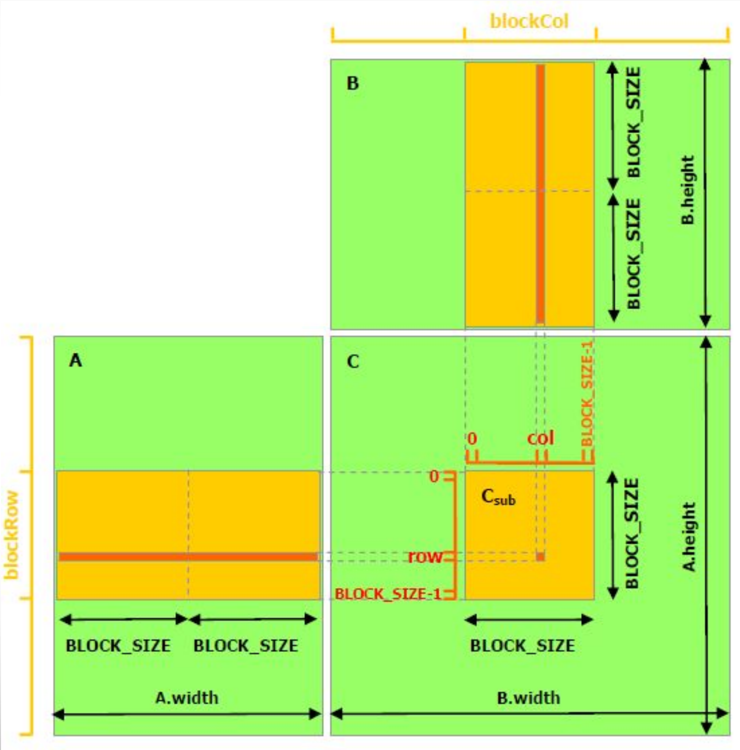
```
__global__ void MatMulKernel(Matrix A, Matrix B,
                              Matrix C)
{
  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e] *
              B.elements[e * B.width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```



Example 4 - Matrix Multiplication with SM

Each thread computes one element of Csub // by accumulating results into Cvalue

```
...
for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        Matrix Bsub = GetSubMatrix(B, m, blockCol);
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
        __syncthreads();
         for (int e = 0; e < BLOCK_SIZE; ++e)
             Cvalue += As[row][e] * Bs[e][col];
         __syncthreads();
    }
 ...
```



Review - 1

- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N/M,M>>>(…);`
  - Use `blockIdx.x` to access block index
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Review - 2

- Launching parallel threads
  - Launch `N` blocks with `blockDim.x` threads per block with `kernel<<<N, blockDim.x>>>(…);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Review - 3

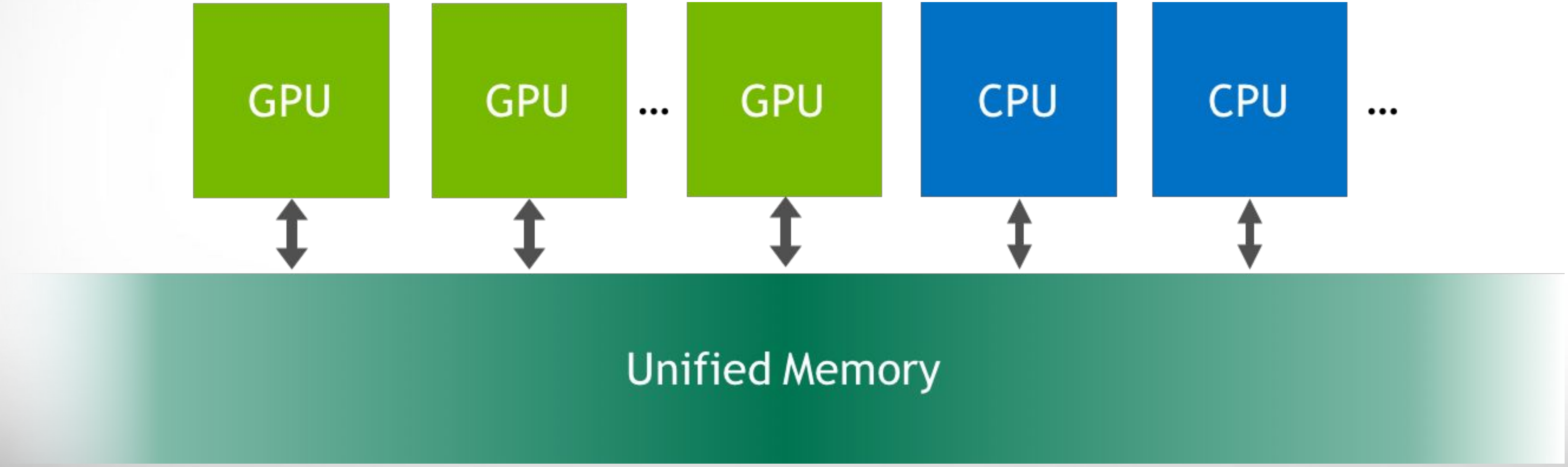
- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

Unified Memory Programming

Unified Memory

Software: CUDA 6.0 in 2014

Hardware: Pascal GPU in 2016



Unified Memory

- A managed memory space where all processors see a single coherent memory image with a common address space.
- Eliminates the need for `cudaMemcpy()`.
- Enables simpler code.
- Equipped with hardware support since Pascal.

Example 5 - Vector Addition w/o UM

```
__global__  void  VecAdd( int  *ret,  int  a,  int  b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMalloc(&ret, 1000 * sizeof(int));
    VecAdd<<< 1, 1000 >>>(ret, 10, 100);
    int *host_ret = (int *)malloc(1000 * sizeof(int));
    cudaMemcpy(host_ret, ret, 1000 * sizeof(int), cudaMemcpyDefault);
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, host_ret[i]);
    free(host_ret);
    cudaFree(ret);
    return  0;
}
```

Example 6 - Vector Addition with UM

```
__global__ void VecAdd(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));
    VecAdd<<< 1, 1000 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
    return  0;
}
```

Example 7 - Vector Addition with Managed Global Memory

```
__device__ __managed__  int  ret[1000];

__global__ void VecAdd(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    VecAdd<<< 1, 1000 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return  0;
}
```

Managing Device

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

`cudaMemcpy()`	Blocks the CPU until the copy is complete. Copy begins when all preceding CUDA calls have completed
`cudaMemcpyAsync()`	Asynchronous, does not block the CPU
`cudaDeviceSynchronize()`	Blocks the CPU until all preceding CUDA calls have completed

Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself or
  - Error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
printf("%s\n",cudaGetErrorString(cudaGetLastError())
);
```

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple threads can share a device
- A single thread can manage multiple devices

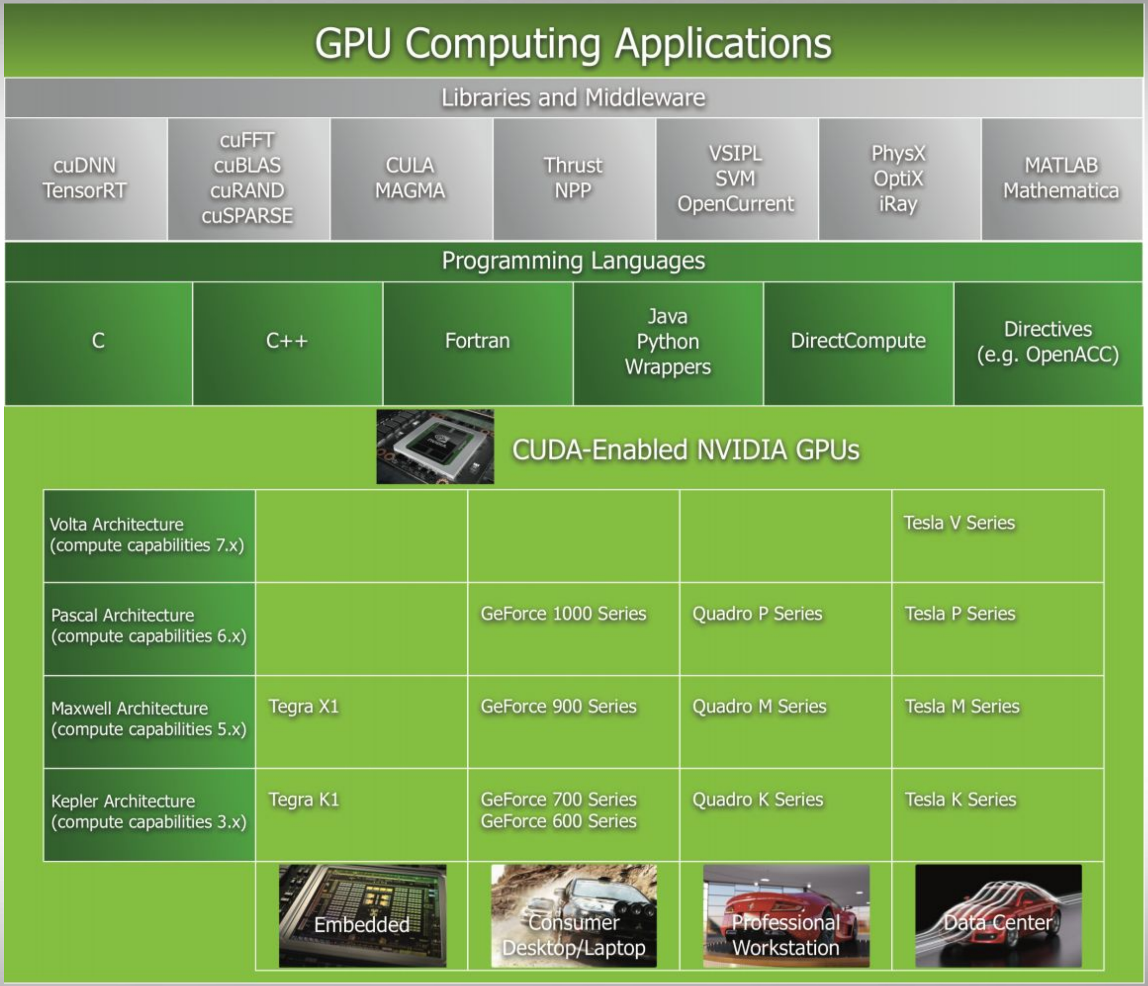
  Select current device: `cudaSetDevice(i)`

  For peer-to-peer copies: `cudaMemcpy(…)`

† requires OS and device support

GPU Computing Capability

The compute capability of a device is represented by a version number that identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.



More Resources

You can learn more about CUDA at

- CUDA Programming Guide (docs.nvidia.com/cuda)
- CUDA Zone – tools, training, etc. (developer.nvidia.com/cuda-zone)
- Download CUDA Toolkit & SDK (www.nvidia.com/getcuda)
- Nsight IDE (Eclipse or Visual Studio) (www.nvidia.com/nsight)

Acknowledgements

- Educational materials from NVIDIA via its Academic Programs.
- Supports from Texas A&M Engineering Experiment Station (TEES) and High Performance Research Computing (HPRC).

Appendix

1D Grid of Blocks in 1D, 2D, and 3D

```
__device__ int getGlobalIdx_1D_1D ()
{
  return blockIdx.x * blockDim.x + threadIdx.x;
}

__device__ int getGlobalIdx_1D_2D ()
{
  return blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x +
    threadIdx.x;
}

__device__ int getGlobalIdx_1D_3D ()
{
  return blockIdx.x * blockDim.x * blockDim.y * blockDim.z
    + threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x +
    threadIdx.x;
}
```

2D Grid of Blocks in 1D, 2D, and 3D

```
__device__ int getGlobalIdx_2D_1D ()
{
  int blockId = blockIdx.y * gridDim.x + blockIdx.x;
  int threadId = blockId * blockDim.x + threadIdx.x;
  return threadId;
}

__device__ int getGlobalIdx_2D_2D ()
{
  int blockId = blockIdx.x + blockIdx.y * gridDim.x;
  int threadId =
    blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x;
  return threadId;
}

__device__ int getGlobalIdx_2D_3D ()
{
  int blockId = blockIdx.x + blockIdx.y * gridDim.x;
  int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
    + (threadIdx.z * (blockDim.x * blockDim.y))
    + (threadIdx.y * blockDim.x) + threadIdx.x;
  return threadId;
}
```

3D Grid of Blocks in 1D, 2D, and 3D

```
__device__ int getGlobalIdx_3D_1D ()
{
  int blockId = blockIdx.x
    + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;
  int threadId = blockId * blockDim.x + threadIdx.x;
  return threadId;
}

__device__ int getGlobalIdx_3D_2D ()
{
  int blockId = blockIdx.x
    + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;
  int threadId = blockId * (blockDim.x * blockDim.y)
    + (threadIdx.y * blockDim.x) + threadIdx.x;
  return threadId;
}

__device__ int getGlobalIdx_3D_3D ()
{
  int blockId = blockIdx.x
    + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;
  int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
    + (threadIdx.z * (blockDim.x * blockDim.y))
    + (threadIdx.y * blockDim.x) + threadIdx.x;
  return threadId;
}
```