ME964
High Performance Computing for Engineering Applications

The CUDA API wrap up

Memory Layout in CUDA

February 10, 2011

In theory there is no difference between theory and practice. In practice there is.
Yogi Berra

Before We Get Started…

- Last time
  - API related issues
  - Memory allocation, copying, freeing, etc.
  - Simple matrix multiplication example
  - Discussed the typical kernel invocation sequence
- Today
  - Wrap up CUDA API discussion
  - Start discussion of memory hierarchy in NVIDIA's GPU and CUDA support
- HW
  - HW2: due today at 23:59 PM
  - HW3 has been posted. Due date: 02/15
  - There is assigned reading

Application Programming Interface (API) ~Taking a Step Back~

- CUDA runtime API: exposes a set of extensions to the C language
  - See **Section 3.2** and **Appendix B** of "NVIDIA CUDA C Programming Guide"
    - Keep in mind the 20/80 rule

- It consists of:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library split into:
    - A common component providing built-in vector types and a subset of the C runtime library in both host and device codes
      - Callable both from device and host
    - A host component to control and access one or more devices from the host
      - Callable from the host only
    - A device component providing device-specific functions
      - Callable from the device only

Language Extensions: Variable Type Qualifiers

	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	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 `__local__`, `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except arrays, which reside in local memory (unless they are small and of known constant size)

Common Runtime Component

- "Common" above refers to functionality that is provided by the CUDA API and is common both to the device and host

- Provides:
  - Built-in vector types
  - A subset of the C runtime library supported in both host and device codes

Common Runtime Component: Built-in Vector Types

- `[u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4], double[1..2]`
  - Structures accessed with `x`, `y`, `z`, `w` fields:

```
uint4 param;
int dummy = param.y;
```

- `dim3`
  - Based on `uint3`
  - Used to specify dimensions
  - You see a lot of it when defining the execution configuration of a kernel (any component left uninitialized assumes default value 1)

See Appendix B in
"NVIDIA CUDA C Programming Guide"

Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- `etc.`
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

Host Runtime Component

- Provides functions available only to the host to deal with:
    - Device management (including multi-device systems)
    - Memory management
    - Error handling

- Examples:
    - cudaHostAlloc, cudaHostFree, cudaMemcpyAsync, etc.

- Quick Remark, Device Management
    - A host thread can invoke device code on only one device
        - Multiple host threads required to run on multiple devices

Host Runtime Component: Memory Management

- Device memory allocation
  - `cudaMalloc(), cudaFree()`

- Memory copy from host to device, device to host, device to device
  - `cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()`

- Memory addressing – returns the address of a device variable
  - `cudaGetSymbolAddress()`

Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. `sin(x)`) have a less accurate, but faster device-only version (e.g. `__sin(x)`)

  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`

End API discussion
…… transitioning into…
Memory Layout discussion

Terminology Review

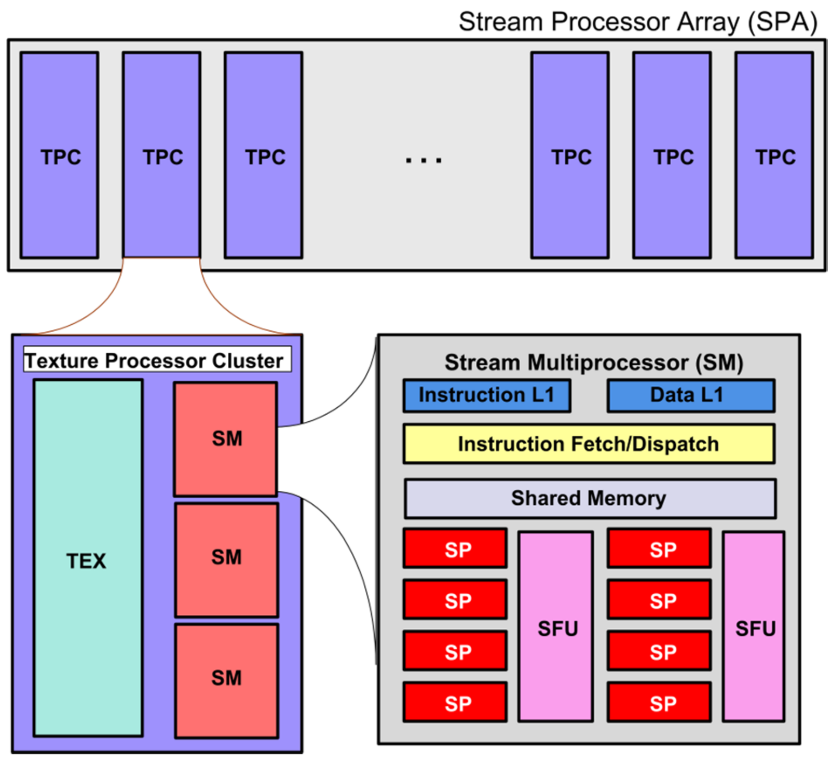
- **Kernel** = GPU program executed by each parallel thread in a block
- **Block** = a 3D collection of threads that can access the block's shared memory and can synchronize during execution
- **Grid** = 2D array of blocks of threads that execute a kernel
- **Device** = GPU = set of stream multiprocessors
- **Stream Multiprocessor** (SM) = set of scalar processors & shared memory
- **Scalar Processor** (SP) = also called CUDA processor, shader processor is where instructions are executed

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

Off-chip means on-device; i.e., slow access time.

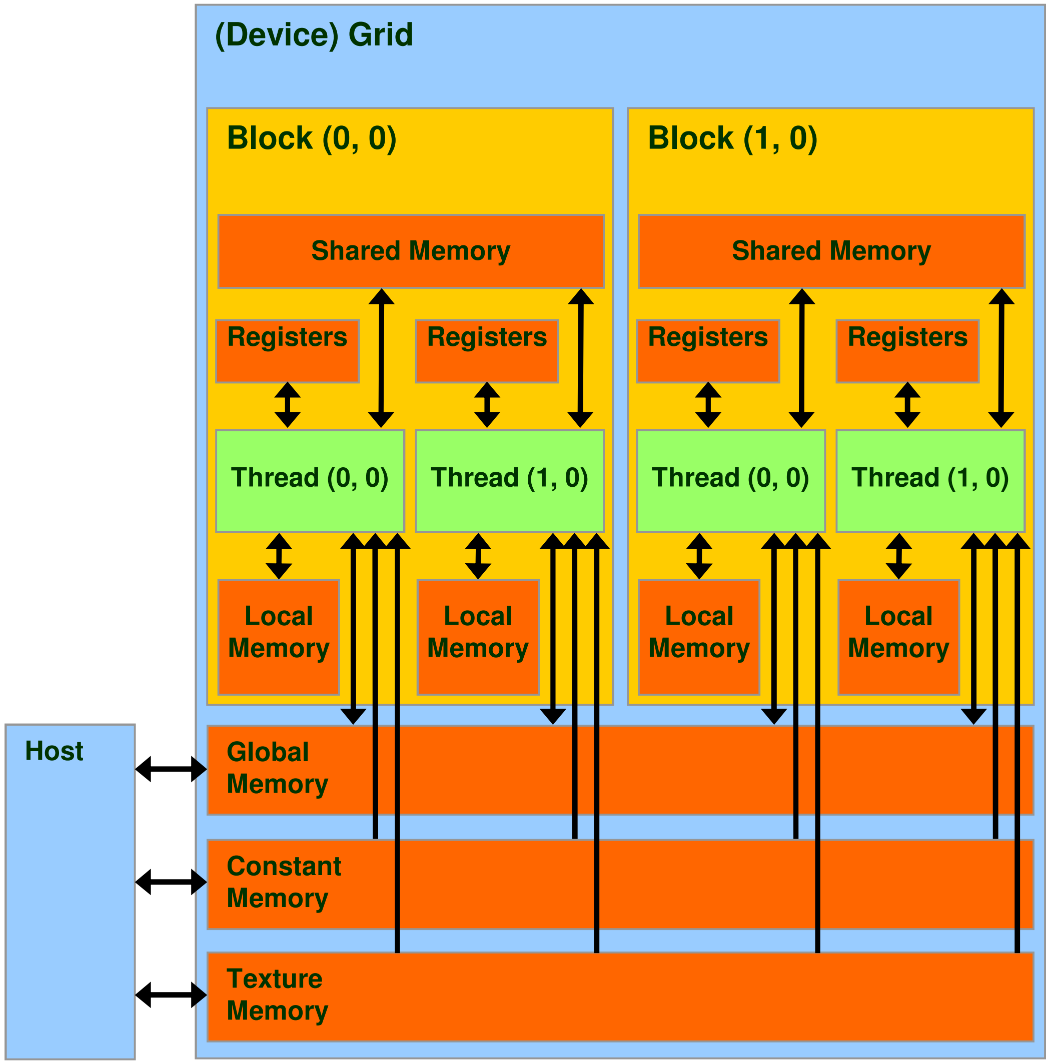
GPU: Underlying Hardware
[Tesla C1060]

- The hardware organized as follows:
  - One Stream Processor Array (SPA)…
    - … has a collection of Texture Processor Clusters (TPC, ten of them on C1060) …
      - …and each TPC has three Stream Multiprocessors (SM) …
        - …and each SM is made up of eight Stream or Scalar Processor (SP)



$$\text{SPA} \xrightarrow[10]{has} \text{TPC} \xrightarrow[3]{each\ has} \text{SM} \xrightarrow[8]{each\ has} \text{SP}$$

The CUDA Memory Ecosystem



Access Times [Tesla C1060]

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

Matrix Multiplication Example, Revisited

- Purpose
  - See an example where the use of multiple blocks of threads plays a central role
  - Emphasize the role of the shared memory
  - Emphasize the need for the _syncthreads() function call

Why Revisiting the Matrix Multiplication Example?

- In the naïve first implementation the ratio of arithmetic computation to memory transaction very low
  - Each arithmetic computation required one fetch from global memory
  - The matrix M (its entries) is copied from global memory to the device N.width times
  - The matrix N (its entries) is copied from global memory to the device M.height times
- What matters when you implement the solution of a numerical problem is going through the chain of computations as fast as possible
  - You don't get ahead moving data around but only computing things

A Common Programming Pattern
BRINGING THE SHARED MEMORY INTO THE PICTURE

- Local and global memory reside in device memory (DRAM) - much slower access than shared memory

- An advantageous way of performing computation on the device is to partition (“tile”) data to take advantage of fast shared memory:

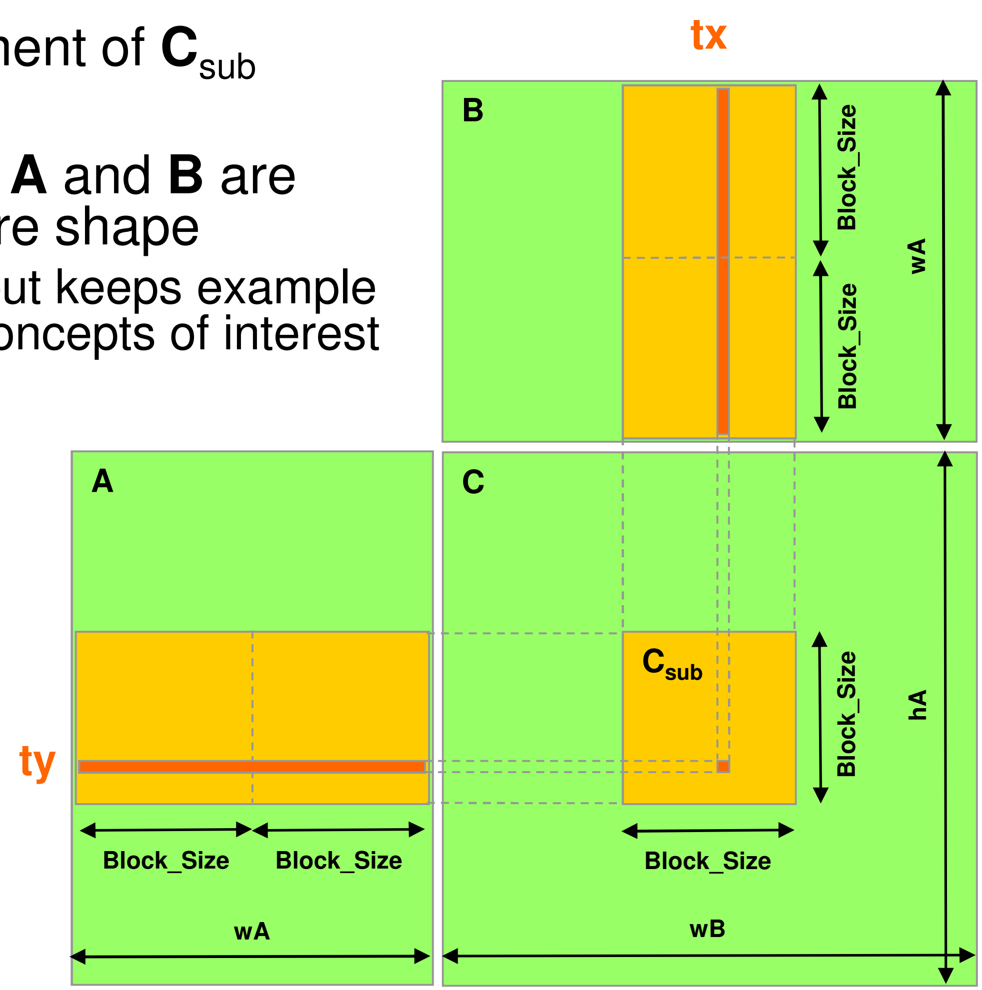
  - Partition data into data subsets (tiles) that each fits into shared memory

  - Handle each data subset (tile) with one thread block by:
    - Loading the tile from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the tile from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory back to global memory

Multiply Using Several Blocks

- One block computes one square sub-matrix C_{sub} of size Block_Size
- One thread computes one element of C_{sub}
- Assume that the dimensions of **A** and **B** are multiples of Block_Size and square shape
  - Doesn't have to be like this, but keeps example simpler and focused on the concepts of interest

NOTE: Similar example provided in the CUDA Programming Guide 3.2
• Available on the class website



```
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int
wB, float* C)
{
  int size;

  // Load A and B to the device
  float* Ad;
  size = hA * wA * sizeof(float);
  cudaMalloc((void**)&Ad, size);
  cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
  float* Bd;
  size = wA * wB * sizeof(float);
  cudaMalloc((void**)&Bd, size);
  cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
```

(continues with next block…)

(continues below…)

```
  // Allocate C on the device
  float* Cd;
  size = hA * wB * sizeof(float);
  cudaMalloc((void**)&Cd, size);

  // Compute the execution configuration assuming
  // the matrix dimensions are multiples of BLOCK_SIZE
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid( wB/dimBlock.x , hA/dimBlock.y );

  // Launch the device computation
  Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

  // Read C from the device
  cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

  // Free device memory
  cudaFree(Ad);
  cudaFree(Bd);
  cudaFree(Cd);
}
```

```
// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
  // Block index
  int bx = blockIdx.x;  // the B (and C) matrix sub-block column index
  int by = blockIdx.y;  // the A (and C) matrix sub-block row index

  // Thread index
  int tx = threadIdx.x; // the column index in the sub-block
  int ty = threadIdx.y; // the row index in the sub-block

  // Index of the first sub-matrix of A processed by the block
  int aBegin = wA * BLOCK_SIZE * by;

  // Index of the last sub-matrix of A processed by the block
  int aEnd = aBegin + wA - 1;

  // Step size used to iterate through the sub-matrices of A
  int aStep = BLOCK_SIZE;

  // Index of the first sub-matrix of B processed by the block
  int bBegin = BLOCK_SIZE * bx;

  // Step size used to iterate through the sub-matrices of B
  int bStep = BLOCK_SIZE * wB;

  // The element of the block sub-matrix that is computed
  // by the thread
  float Csub = 0;
```

(continues with next block…)

```
 // Loop over all the sub-matrices of A and B required to
 // compute the block sub-matrix
 for (int a = aBegin, b = bBegin;
   a <= aEnd;
   a += aStep, b += bStep) {

     // Shared memory for the sub-matrix of A
     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

     // Shared memory for the sub-matrix of B
     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

     // Load the matrices from global memory to shared memory;
     // each thread loads one element of each matrix
     As[ty][tx] = A[a + wA * ty + tx];
     Bs[ty][tx] = B[b + wB * ty + tx];

     // Synchronize to make sure the matrices are loaded
     __syncthreads();

     // Multiply the two matrices together;
     // each thread computes one element
     // of the block sub-matrix
     for (int k = 0; k < BLOCK_SIZE; ++k)
       Csub += As[ty][k] * Bs[k][tx];

     // Synchronize to make sure that the preceding
     // computation is done before loading two new
     // sub-matrices of A and B in the next iteration
     __syncthreads();
 }
 // Write the block sub-matrix to global memory;
 // each thread writes one element
 int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
 C[c + wB * ty + tx] = Csub;
}
```