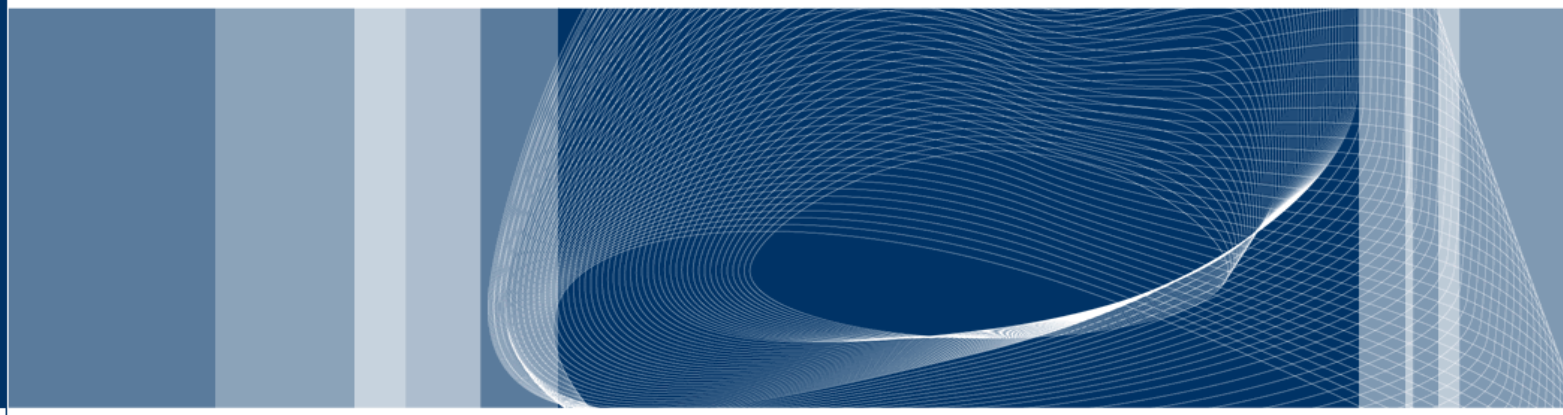
POLITECNICO DI MILANO



Basic Elements of CUDA

Algoritmi e Calcolo Parallelo

Daniele Loiacono

References

- This set of slides is mainly based on:
  - CUDA Technical Training, Dr. Antonino Tumeo, Pacific Northwest National Laboratory
  - Slide of *Applied Parallel Programming* (ECE498@UIUC) http://courses.engr.illinois.edu/ece498/al/
- Useful references
  - *Programming Massively Parallel Processors: A Hands-on Approach*, David B. Kirk and Wen-mei W. Hwu
  - http://www.gpgpu.it/ (CUDA Tutorial)
  - CUDA Programming Guide http://developer.nvidia.com/object/gpucomputing.html

Compiling the Code

- nvcc <filename>.cu [-o <executable>]
  - Builds release mode
- nvcc –g <filename>.cu
  - Builds debug (device) mode
  - Can debug host code but not device code (runs on GPU)

GPU’s Memory Management

Managing Memory

- CPU and GPU have separate memory spaces
- Host (CPU) code manages device (GPU) memory:
  - Allocate / free
  - Copy data to and from device
  - Applies to *global device memory (DRAM)*

GPU Memory Allocation / Release

- `cudaMalloc(void ** pointer, size_t nbytes)`
- `cudaMemset(void * pointer, int value, size_t count)`
- `cudaFree(void* pointer)`

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *d_a = 0;
cudaMalloc(&d_a, nbytes );
cudaMemset(d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

- `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
  - *src* is the pointer to data to be copied and *dst* is the pointer to the destination
  - blocks CPU thread (returns after the copy is complete)
  - doesn't start copying until previous CUDA calls complete
  - direction specifies locations (host or device) of *src* and *dst*:
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToHost`
    - `cudaMemcpyDeviceToDevice`

2D memory management

- `cudaMallocPitch(void** devPtr, size_t* pitch, size_t widthInBytes, size_t height)`
  - As an example, given…

    ```
    float *d_a; size_t pitch;
    cudaMallocPitch(&d_a,&pitch,16*sizeof(float),4);
    ```
  - …to access on the device to an element:

    ~~`d_a[row*16+col]`~~

    `d_a[row*pitch/sizeof(float)+col]`
- `cudaMemcpy2D(void *dst, size_t dpitch, const void* src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind direction)`

- Pitch size is in byte!
  - Remember to divide by the sizeof(typeofdata);

Kernels

Executing Code on the GPU

- Kernels are C functions with some restrictions
  - Can only access GPU memory
  - Must have void return type
  - No variable number of arguments (“varargs”)
  - Not recursive
  - No static variables
- Function arguments automatically copied from CPU to GPU memory

Function Qualifiers

	Executed on	Callable from
`__device__ type DeviceFunc()`	Device	Device
`__global__ void KernelFunc()`	Device	Host
`__host__ type HostFunc()`	Host	Host

- __global__ functions must return void
- __host__ and __device__ qualifiers can be combined
  - Compiler will generate both CPU and GPU code

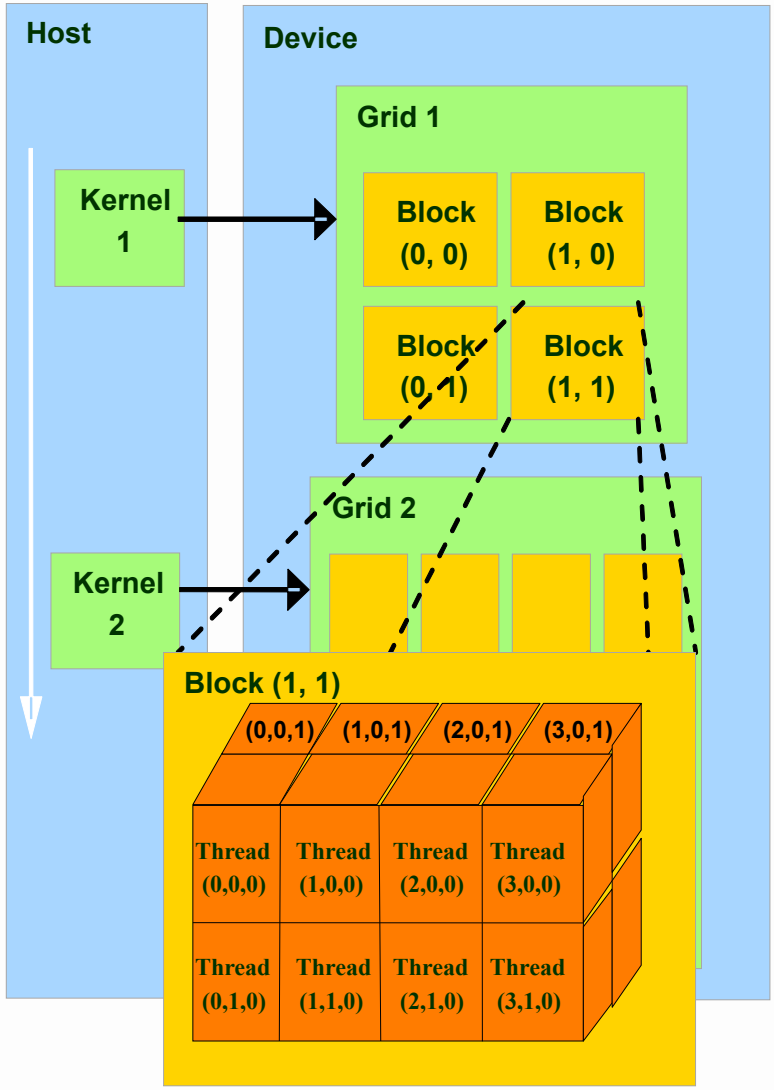
Launching Kernels

- Modified C function call syntax:
  - kernel<<<dim3 grid, dim3 block>>>(…)
- Execution Configuration ("<<< >>>"):
  - grid dimensions: x and y (2D)
  - thread-block dimensions: x, y, and z (3D)
- When grid or block have only x dimension, it is possible to define the size of the grid/blocks with an integer
- Examples:

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



CUDA Built-in Device Variables

- All __global__ and __device__ functions have access to these automatically defined variables
  - `dim3 gridDim`
    - Dimensions of the grid in blocks (at most 2D)
  - `dim3 blockDim`
    - Dimensions of the block in threads
  - `dim3 blockIdx`
    - Block index within the grid
  - `dim3 threadIdx`
    - Thread index within the block

Thread ID

- The ID of a thread within a block is

```
Tid = threadIdx.x + threadIdx.y * (blockDim.x) + threadIdx.z *
    (blockDim.y) * (blockDim.x);
```

- ThreadIdx.{x,y,z}: index of the thread in its x,y,z dimension
- BlockDim.{x,y,z}: size of the block in its x, y, z dimension

Thread ID

- Considering a block with

  blockDim.x = 2

  blockDim.y = 4

  blockDim.z = 6

- Thre thread with

  ThreadIdx.x = 1

  ThreadIdx.y = 2

  ThreadIdx.z = 5

- Has the ID

  1 + 2 * (2) + 5 * (2 * 4) = 1 + 4 + 40 = 45

A simple kernel

```
__global__ void assign( int* d_a, int value)
{
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   d_a[idx] = value;
}
```

```
float* h_A = new float[N];
float* d_A;

cudaMalloc(&d_A, N*sizeof(float));
cudaMemcpy(d_A, h_A, N*sizeof(float), cudaMemcpyHostToDevice);
increment_gpu<<< ceil(N/512), 512>>>(d_A, 3);
cudaMemcpy(h_A, d_A, N*sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(d_A);
```

Example of kernel for 2D data

```
// Host code
…
int width = 64, height = 64;
float* devPtr; size_t pitch;
cudaMallocPitch(&devPtr,&pitch,width * sizeof(float), height);
…
MyKernel<<<grid, block>>>(devPtr, pitch, width, height);
…
// Device code
__global__ void MyKernel(float* devPtr, size_t pitch, int
                                        width, int height){
  …
      float element = devPtr[r*pitch/sizeof(float)+c];
  …
  }
}
```

Example: Increment Array Elements

- Increment N-element vector a by scalar b

- Let's assume N=16, blockDim=4 -> 4 blocks

blockIdx.x=0
blockDim.x=4
threadIdx.x=
0,1,2,3
idx=0,1,2,3

blockIdx.x=1
blockDim.x=4
threadIdx.x=
0,1,2,3
idx=4,5,6,7

blockIdx.x=2
blockDim.x=4
threadIdx.x=
0,1,2,3
idx=8,9,10,11

blockIdx.x=3
blockDim.x=4
threadIdx.x=0,1,2,3
idx=12,13,14,15

- int idx = blockDim.x * blockId.x + threadIdx.x;
  - will map from local index threadIdx to global index
- NB: blockDim should be >= 32 in real code, this is just an example

Example: Increment Array Elements

- CPU implementation

```
void increment_cpu(float *a, float b, int N)
{
   for (int idx = 0; idx<N; idx++)
   a[idx] = a[idx] + b;
}
void main()
{
.....
   increment_cpu(a, b, N);
}
```

Example: Increment Array Elements

- CUDA implementation

```
__global__ void increment_gpu(float *a, float b, int N)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if (idx < N)
       a[idx] = a[idx] + b;
}

void main()
{
…
      dim3 dimBlock (blocksize);
      dim3 dimGrid( ceil( N / (float)blocksize) );
      increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Synchronization

Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed
- `cudaMemcpy()` is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed
- `cudaThreadSynchronize()`
  - blocks until all previous CUDA calls complete

Example: Host Code

```
// allocate host memory
int numBytes = N * sizeof(float)
float* h_A = new float[N];
// allocate device memory
float* d_A = 0;
cudaMalloc(&d_A, numbytes);
// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);
// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);
// copy data from device back to host
// --> implicit synchronization
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);
// free device memory
cudaFree(d_A);
```

GPU Thread Synchronization

- void __syncthreads();
- Synchronizes all threads in a block
  - Generates barrier synchronization instruction
  - No thread can pass this barrier until all threads in the block reach it
  - Used to sync when accessing shared memory
- Allowed in conditional code only if the conditional is uniform across the entire thread block

Additional features

Variable Qualifiers (GPU code)

- __device__
  - Stored in device memory (large, high latency, no cache)
  - Allocated with cudaMalloc (__device__ qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- __shared__
  - Stored in on-chip shared memory (very low latency)
  - Allocated by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: kernel execution
- Unqualified variables:
  - Scalars and built-in vector types are stored in registers
  - Arrays of more than 4 elements stored in device memory

Using shared memory

Size known at compile time

```
__global__ void kernel(…)
{
    …
    __shared__ float sData[256];
    …
}

int main(void)
{
…
    kernel<<<nBlocks,blockSize>>>(…);
…
}
```

Size known at kernel launch

```
__global__ void kernel(…)
{
…
extern __shared__ float sData[];
…
}

int main(void)
{
…
  smBytes = blockSize*sizeof(float);
  kernel<<<nBlocks, blockSize,
                          smBytes>>>(…);
…
}
```

Built-in Vector Types

- Can be used in GPU and CPU code
  - [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]
  - Structures accessed with x, y, z, w fields:
    - uint4 param;
    - int y = param.y;
- dim3
  - Based on uint3
  - Used to specify dimensions
  - Default value (1,1,1)

GPU Atomic Integer Operations

- Requires hardware with compute capability 1.1
  - G80 = Compute capability 1.0
  - G84/G86/G92 = Compute capability 1.1
  - GT200 = Compute capability 1.3
- Atomic operations on integers in global memory:
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap

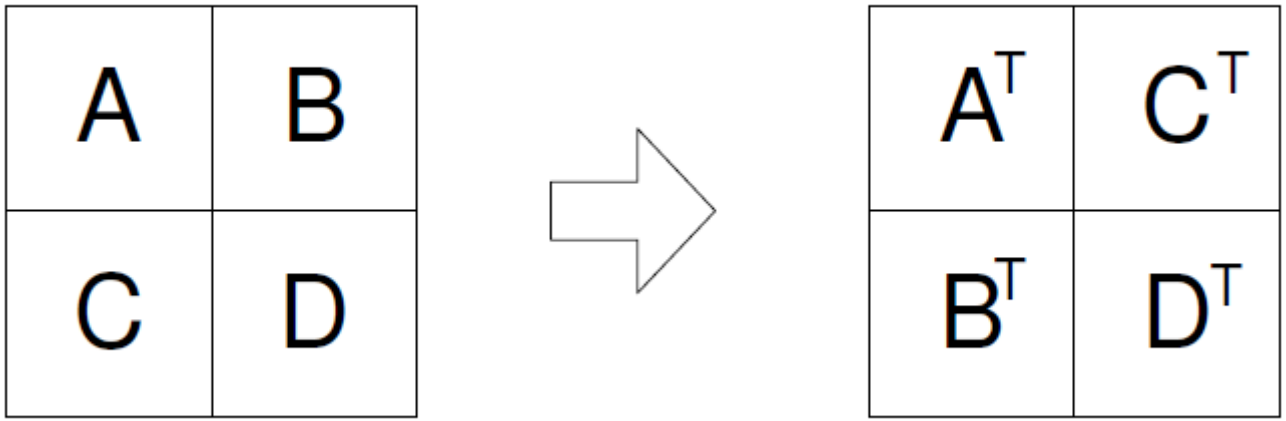
CUDA Error Reporting to CPU

- All CUDA calls return error code:
  - Except for kernel launches
  - cudaError_t type
- cudaError_t cudaGetLastError(void)
  - Returns the code for the last error (no error has a code)
  - Can be used to get error from kernel execution
- char* cudaGetErrorString(cudaError_t code)
  - Returns a null-terminated character string describing the error
- printf("%s\n", cudaGetErrorString(cudaGetLastError()));

Examples

Matrix Transpose

- Each thread block transposes an equal sized block of matrix M
- Assume M is square (n x n)
- What is a good blocksize?
- CUDA places limitations on number of threads per block
  - 512 threads per block is the maximum allowed by CUDA



Solution: host code


```
int main(int args, char** vargs) {
  const int HEIGHT = 1024; const int WIDTH = 1024;
  const int SIZE = WIDTH * HEIGHT * sizeof(float);
  dim3 bDim(16, 16);
  dim3 gDim(WIDTH / bDim.x, HEIGHT / bDim.y);
  float* M = new float[WIDTH*HEIGHT];
  for (int i = 0; i < HEIGHT * WIDTH; i++) { M[i] = i; }
  float* Md = NULL;
  cudaMalloc(&Md, SIZE);
  cudaMemcpy(Md, M, SIZE, cudaMemcpyHostToDevice);
  float* Bd = NULL;
  cudaMalloc(&Bd, SIZE);
  transpose<<<gDim, bDim>>>(Md, Bd, WIDTH);
  cudaMemcpy(M,Bd, SIZE, cudaMemcpyDeviceToHost);
  return 0;
}
```

Solution: kernel


```
__global__
void transpose(float* in, float* out, uint width) {
  uint tx = blockIdx.x * blockDim.x + threadIdx.x;
  uint ty = blockIdx.y * blockDim.y + threadIdx.y;
  out[tx * width + ty] = in[ty * width + tx];
}
```

Sum of vectors

- Given two real vectors A and B of size N
- Compute vector C = A+B

Solution: Host Code (1)


```
int main(){
  int N = ...;
  size_t size = N * sizeof(float);
  // Allocate input vectors h_A and h_B in host memory
  float* h_A = (float*)malloc(size);
  float* h_B = (float*)malloc(size);
  // Initialize input vectors
  ...
  // Allocate vectors in device memory
  float* d_A;
  cudaMalloc(&d_A, size);
  float* d_B;
  cudaMalloc(&d_B, size);
  float* d_C;
  cudaMalloc(&d_C, size);
```

Solution: Host Code (2)


```
  // Copy vectors from host memory to device memory
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
  // Invoke kernel
  int threadsPerBlock = 256;
  int blocksPerGrid = (N+threadsPerBlock-1)/ threadsPerBlock;
  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
  // Copy result from device memory to host memory
  // h_C contains the result in host memory
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  // Free device memory
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
  // Free host memory
  ...
}
```

Solution: Device Code


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