Parallel Computing with GPUs

Introduction to CUDA Part 3 - Host Code and Memory Management

The University Of Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

- ❑CUDA Host Code and Memory Management
  - ❑State the methods in which device memory can reserved
  - ❑Demonstrate how host code can be used to move memory to and from the GPU device
  - ❑Present a complete example of a GPU program

Memory Management

- ❑GPU has separate dedicated memory from the host CPU
- ❑Data accessed in kernels must be on GPU memory
  - ❑Data must be copied and transferred
  - ❑A Unified memory approach can be used to do this transparently
- ❑**`cudaMalloc()`** is used to allocate memory on the GPU
- ❑**`cudaFree()`** releases memory

```
float *a;
cudaMalloc(&a, N*sizeof(float));
...
cudaFree(a);
```

Memory Copying

- Once memory has been allocated we need to copy data to it and from it.
- **cudaMemcpy()** transfers memory from the host to device to host and vice versa

```
cudaMemcpy(array_device, array_host,
  N*sizeof(float), cudaMemcpyHostToDevice)

cudaMemcpy(array_host, array_device,
  N*sizeof(float), cudaMemcpyDeviceToHost)
```

- First argument is always the **destination** of transfer
- Transfers are relatively slow and should be minimised where possible

```
#define N 2048
#define THREADS_PER_BLOCK 128

__global__ void vectorAdd(float *a, float *b, float *c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}

int main(void) {
     float *a, *b, *c;                // host copies of a, b, c
     float *d_a, *d_b, *d_c;          // device copies of a, b, c
     int size = N * sizeof(float);

     cudaMalloc((void **)&d_a, size);
     cudaMalloc((void **)&d_b, size);
     cudaMalloc((void **)&d_c, size);

     a = (float *)malloc(size); random_floats(a, N);
     b = (float *)malloc(size); random_floats(b, N);
     c = (float *)malloc(size);

     cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
     cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

     vectorAdd <<<N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>(d_a, d_b, d_c);

     cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

     free(a); free(b); free(c);
     cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
     return 0;
}
```

Define macros

Define kernel

Define pointer variables

Allocate GPU memory

Allocate host memory and initialise contents

Copy input data to the device

Launch the kernel

Copy data back to host

Clean up

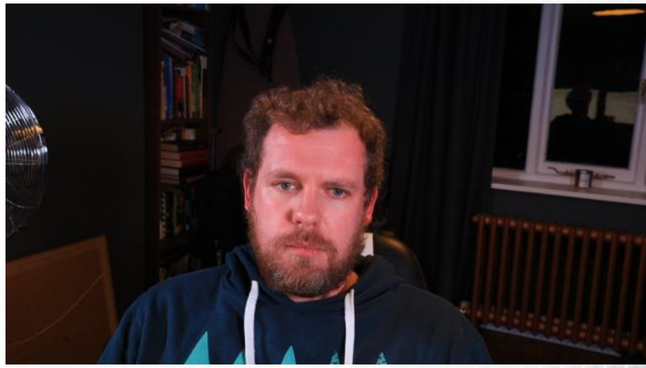
The University Of Sheffield.

Statically allocated device memory

- How do we declare a large array on the host without using malloc?
  - Statically allocate using compile time size

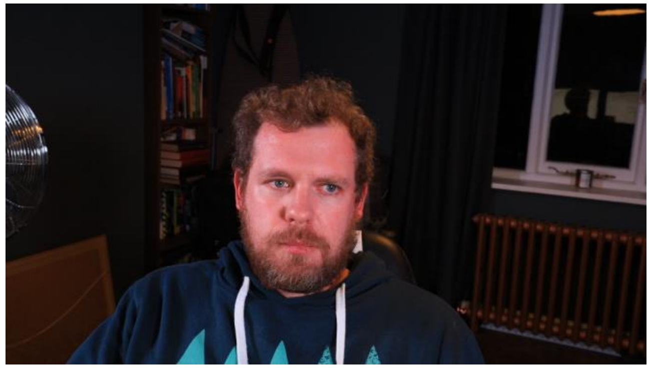
```
int array[N];
```

- We can do the same on the device. i.e.
  - Just like when applied to a function
  - Only available on the device
  - Must use `cudaMemCopyToSymbol()`
  - Must be a global variable



```
__device__ int array[N];
```

```
#define N 2048
#define THREADS_PER_BLOCK 128

__device__ float d_a[N];
__device__ float d_b[N];
__device__ float d_c[N];

__global__ void vectorAdd() {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  d_c[i] = d_a[i] + d_b[i];
}

int main(void) {
     float *a, *b, *c;                  // host copies of a, b, c
     int size = N * sizeof(float);

     a = (float *)malloc(size); random_floats(a, N);
     b = (float *)malloc(size); random_floats(b, N);
     c = (float *)malloc(size);

     cudaMemcpyToSymbol(d_a, a, size);
     cudaMemcpyToSymbol(d_b, b, size);

     vectorAdd <<<N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>();

     cudaMemcpyFromSymbol(c, d_c, size);

     free(a); free(b); free(c);
     return 0;
}
```



Define macros

Statically allocate GPU memory

Define kernel

Define pointer variables

Allocate host memory and initialise contents

Copy input data to the device

Launch the kernel

Copy data back to host

Clean up

The University Of Sheffield.

Device Synchronisation

- ❑Kernel calls are non-blocking
  - ❑Host continues after kernel launch
  - ❑Overlaps CPU and GPU execution
- ❑**cudaDeviceSynchronise()** call be called from the host to block until GPU kernels have completed

```
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
//do work on host (that doesn't depend on c)
cudaDeviceSynchronise(); //wait for kernel to finish
```

- ❑Standard **cudaMemcpy** calls are blocking
  - ❑Non-blocking variants exist

Summary

- ❑CUDA Host Code and Memory Management
  - ❑State the methods in which device memory can reserved
  - ❑Demonstrate how host code can be used to move memory to and from the GPU device
  - ❑Present a complete example of a GPU program

Acknowledgements and Further Reading

- ❑Some of the content in this lecture material has been provided by;

1. GPUComputing@Sheffield Introduction to CUDA Teaching Material
   - ❑Originally from content provided by Alan Gray at EPCC/NVIDIA
2. NVIDIA Educational Material
   - ❑Specifically Mark Harris's (Introduction to CUDA C)

- ❑**Further Reading**
  - ❑Essential Reading: CUDA C Programming Guide
    - ❑http://docs.nvidia.com/cuda/cuda-c-programming-guide/