

Learn CUDA in an Afternoon

Alan Gray

EPCC

The University of Edinburgh

Overview

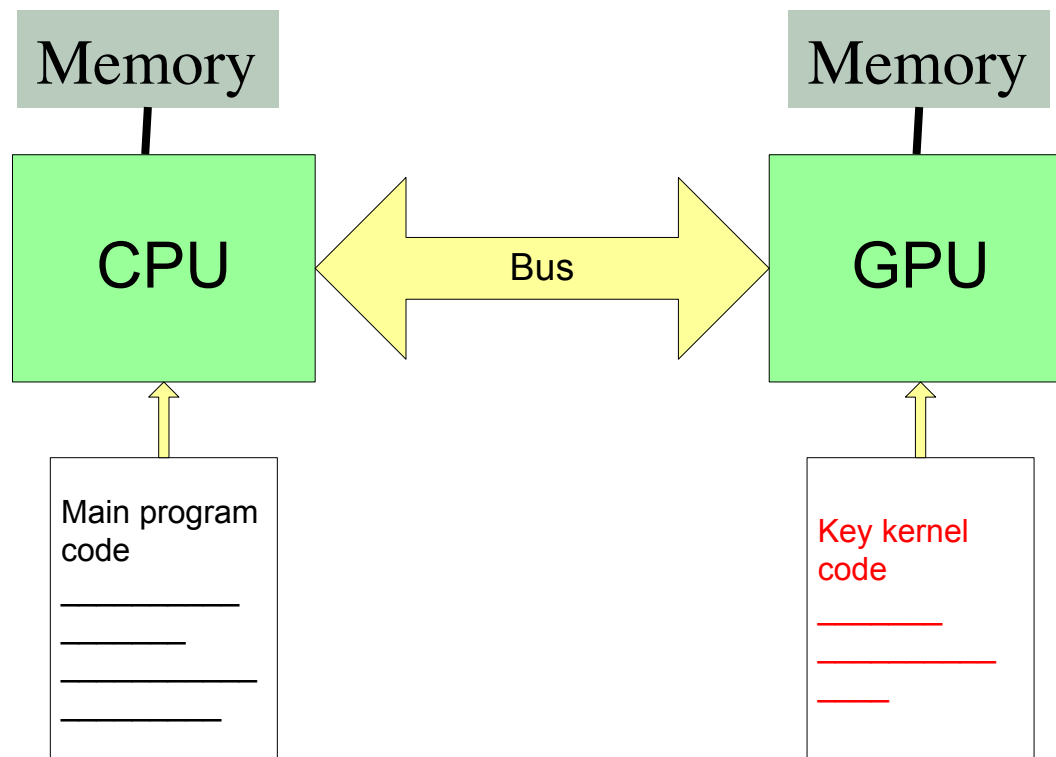
- Introduction to CUDA
- *Practical Exercise 1: Getting started with CUDA*
- GPU Optimisation
- *Practical Exercise 2: Optimising a CUDA Application*

Overview

- Introduction to CUDA
- *Practical Exercise 1: Getting started with CUDA*
- GPU Optimisation
- *Practical Exercise 2: Optimising a CUDA Application*

Introduction

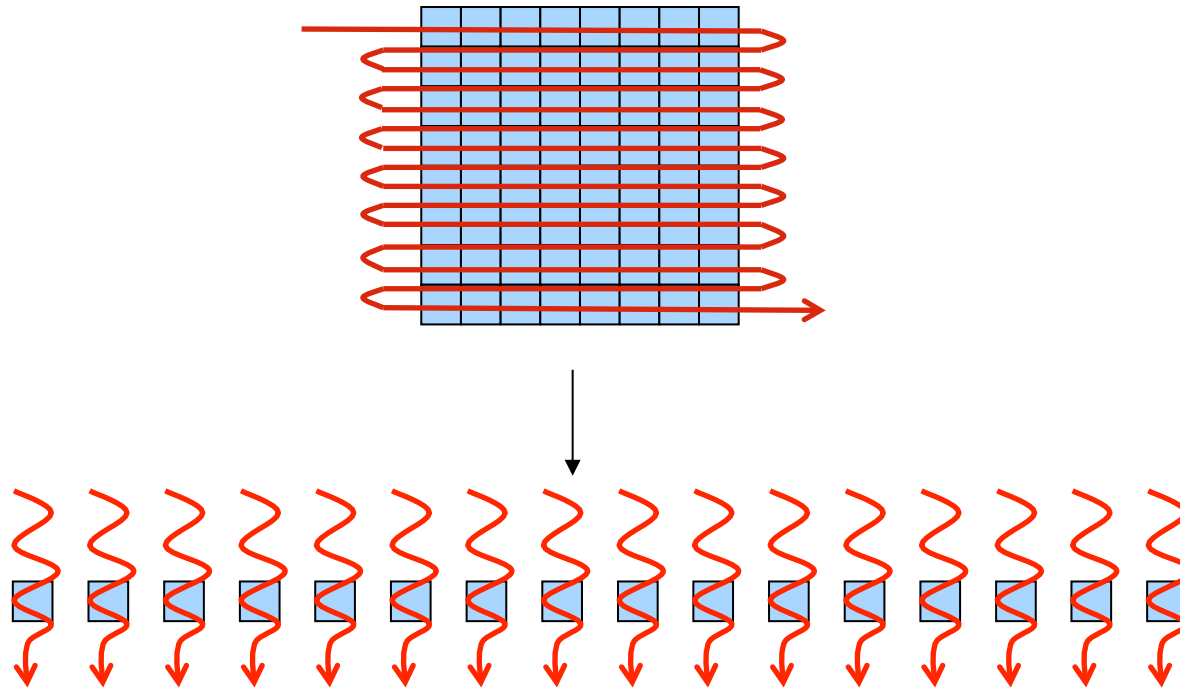
- Graphics Processing Units (GPUs) offer higher performance than CPUs
 - More of the silicon on the chip is dedicated to computation: many cores in each GPU chip
 - Graphics memory (GDRAM) has higher bandwidth than the DRAM memory used by CPUs
- GPUs can not be used alone, but must be used in combination with a CPU
 - GPUs accelerate those computationally demanding sections of code (which we call *kernels*).
 - Kernels are decomposed to run in parallel on the multiple cores
- The CPU and GPU each have their own memory spaces



NVIDIA CUDA

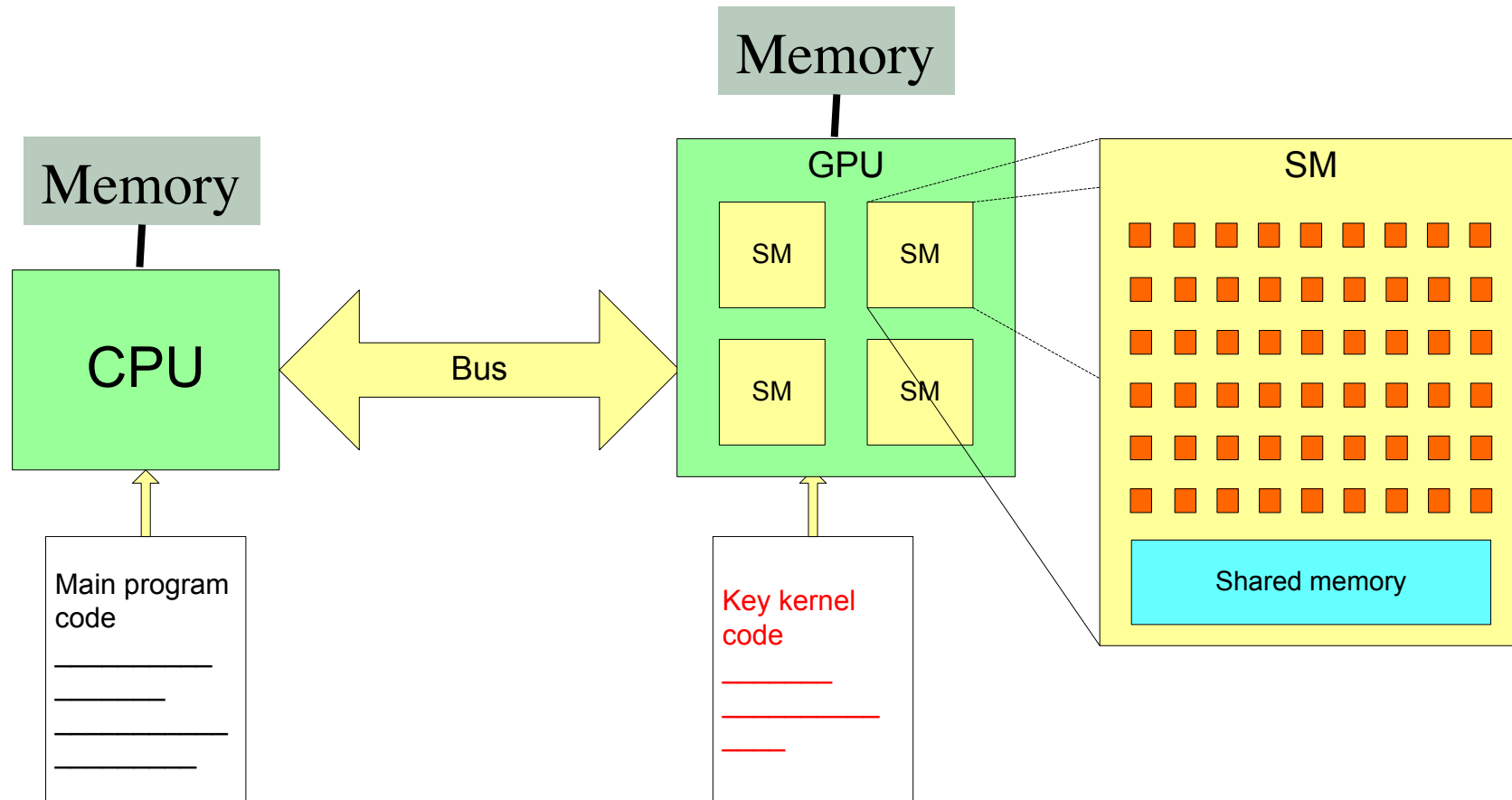
- Traditional languages alone are not sufficient for programming GPUs
- CUDA is an extension to C/C++ that allows programming of NVIDIA GPUs
 - language extensions for defining kernels
 - API functions for memory management

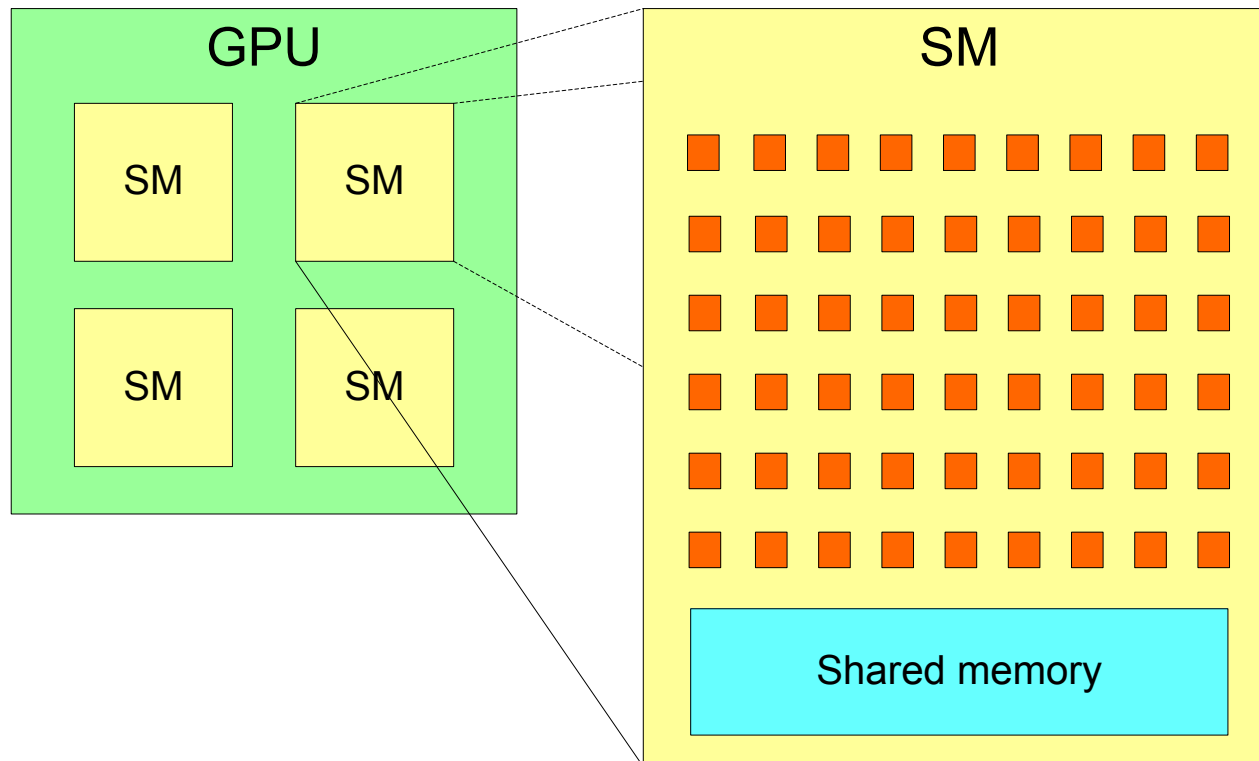
Stream Computing



- Data set decomposed into a *stream* of elements
- A single computational function operates on each element
 - “thread” defined as execution of kernel on one data element
- Multiple cores can process multiple elements in parallel
 - i.e. many threads running in parallel
- Suitable for data-parallel problems

Hardware





- NVIDIA GPUs have a 2-level hierarchy:
 - Multiple *Streaming Multiprocessors* (SMs), each with multiple *cores*
- The number of SMs, and cores per SM, varies across generations

-
- In CUDA, this is abstracted as *Grid of Thread Blocks*
 - The multiple **blocks** in a grid map onto the multiple **SMs**
 - Each block in a grid contains multiple **threads**, mapping onto the **cores** in an SM
 - We don't need to know the exact details of the hardware (number of SMs, cores per SM).
 - Instead, *oversubscribe*, and system will perform scheduling automatically
 - Use more blocks than SMs, and more threads than cores
 - Same code will be portable and efficient across different GPU versions.

CUDA dim3 type

- CUDA introduces a new `dim3` type
 - Simply contains a collection of 3 integers, corresponding to each of X,Y and Z directions.

```
dim3 my_xyz_values(xvalue,yvalue,zvalue);
```

-
- X component can be accessed as follows:

```
my_xyz_values.x
```

And similar for Y and Z

- E.g. for

```
dim3 my_xyz_values(6,4,12);
```

then `my_xyz_values.z` has value 12



Analogy

- You check in to the hotel, as do your classmates
 - Rooms allocated in order
- Receptionist realises hotel is less than half full
 - Decides you should all move from your room number i to room number $2i$
 - so that no-one has a neighbour to disturb them

-
- **Serial Solution:**
 - Receptionist works out each new number in turn



- Parallel Solution:



“Everybody: check your room number. Multiply it by 2, and move to that room.”

- **Serial solution:**

```
for (i=0; i<N; i++) {  
    result[i] = 2*i;  
}
```

- We can parallelise by assigning each iteration to a separate CUDA thread.

CUDA C Example

```
__global__ void myKernel(int *result)
{
    int i = threadIdx.x;
    result[i] = 2*i;
}
```

- Replace loop with function
- Add **__global__** specifier
 - To specify this function is to form a GPU kernel
- Use internal CUDA variables to specify array indices
 - **threadIdx.x** is an internal variable unique to each thread in a block.
 - X component of dim3 type. Since our problem is 1D, we are not using the Y or Z components (more later)

CUDA C Example

- And launch this kernel by calling the function
 - *on multiple CUDA threads using <<<...>>> syntax*

```
dim3 blocksPerGrid(1,1,1); //use only one block
dim3 threadsPerBlock(N,1,1); //use N threads in the block
```

```
myKernel<<<blocksPerGrid, threadsPerBlock>>>(result);
```

CUDA Example

- The previous example only uses 1 block, i.e. only 1 SM on the GPU, so performance will be very poor. In practice, we need to use multiple blocks to utilise all SMs, e.g.:

```
__global__ void myKernel(int *result)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    result[i] = 2*i;
}

...
dim3 blocksPerGrid(N/256,1,1); //assuming 256 divides N exactly
dim3 threadsPerBlock(256,1,1);

myKernel<<<blocksPerGrid, threadsPerBlock>>>(result);
...
```

- We have chosen to use 256 threads per block, which is typically a good number (see practical).

CUDA C Example

- More realistic 1D example: vector addition

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

...
dim3 blocksPerGrid(N/256,1,1); //assuming 256 divides N exactly
dim3 threadsPerBlock(256,1,1);

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
...
```

CUDA C Internal Variables

For a 1D decomposition (e.g. the previous examples)

- `blockDim.x`: Number of threads per block
 - Takes value 256 in previous example
- `threadIdx.x`: unique to each thread in a block
 - Ranges from 0 to 255 in previous example
- `blockIdx.x`: Unique to every block in the grid
 - Ranges from 0 to $(N/256 - 1)$ in previous example

2D Example

- 2D or 3D CUDA decompositions also possible, e.g. for matrix addition (2D):

```
__global__ void matrixAdd(float a[N][N], float b[N][N], float c[N][N])
{
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    c[i][j] = a[i][j] + b[i][j];
}

int main()
{
    dim3 blocksPerGrid(N/16,N/16,1); // (N/16)x(N/16) blocks/grid (2D)
    dim3 threadsPerBlock(16,16,1); // 16x16=256 threads/block (2D)
    matrixAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
}
```

Memory Management - allocation

- The GPU has a separate memory space from the host CPU
- Data accessed in kernels must be on GPU memory
- Need to manage GPU memory and copy data to and from it explicitly
- `cudaMalloc` is used to allocate GPU memory
- `cudaFree` releases it again

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


Memory Management - cudaMemcpy

- Once we've allocated GPU memory, we need to be able to copy data to and from it
- cudaMemcpy does this:

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

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

- The first argument always corresponds to the *destination* of the transfer.
- Transfers between host and device memory are relatively slow and can become a bottleneck, so should be minimised when possible

Synchronisation between host and device

- Kernel calls are *non-blocking*. This means that the host program continues immediately after it calls the kernel
 - Allows overlap of computation on CPU and GPU
- Use `cudaThreadSynchronize()` to wait for kernel to finish

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

- Standard `cudaMemcpy` calls are blocking
 - Non-blocking variants exist

Synchronisation between CUDA threads

- Within a kernel, to synchronise between threads in the same block use the `syncthreads()` call
- Therefore, threads in the same block can communicate through memory spaces that they share, e.g. assuming `x` local to each thread and `array` in a shared memory space

```
if (threadIdx.x == 0) array[0]=x;
syncthreads();
if (threadIdx.x == 1) x=array[0];
```

- It is *not possible* to communicate between different blocks in a kernel: must instead exit kernel and start a new one

Compiling CUDA Code

- CUDA code is compiled using `nvcc`:

```
nvcc -o example example.cu
```

Overview

- Introduction to CUDA
- *Practical Exercise 1: Getting started with CUDA*
 - *See Practical PDF document*
- GPU Optimisation
- *Practical Exercise 2: Optimising a CUDA Application*

Overview

- Introduction to CUDA
- *Practical Exercise 1: Getting started with CUDA*
- **GPU Optimisation**
- *Practical Exercise 2: Optimising a CUDA Application*

GPU performance inhibitors

- Copying data to/from device
- Device under-utilisation/ GPU memory latency
- GPU memory bandwidth
- Code branching

This lecture will address each of these

- And advise how to maximise performance
- Concentrating on NVIDIA, but many concepts will be transferable to e.g. AMD

Host – Device Data Copy

- CPU (host) and GPU (device) have separate memories.
- All data read/written on the device must be copied to/from the device (over PCIe bus).
 - This very expensive
- Must try to minimise copies
 - Keep data resident on device
 - May involve porting more routines to device, even if they are not computationally expensive
 - Might be quicker to calculate something from scratch on device instead of copying from host

Data copy optimisation example

```
Loop over timesteps
  inexpensive_routine_on_host(data_on_host)
  copy data from host to device
  expensive_routine_on_device(data_on_device)
  copy data from device to host
End loop over timesteps
```

- **Port inexpensive routine to device and move data copies outside of loop**

```
copy data from host to device
Loop over timesteps
  inexpensive_routine_on_device(data_on_device)
  expensive_routine_on_device(data_on_device)
End loop over timesteps
copy data from device to host
```

Exposing parallelism

- GPU performance relies on parallel use of many threads
 - Degree of parallelism much higher than a CPU
- Effort must be made to expose as much parallelism as possible within application
 - May involve rewriting/refactoring
- If significant sections of code remain serial, effectiveness of GPU acceleration will be limited (Amdahl's law)

Occupancy and Memory Latency hiding

- Programmer decomposes loops in code to threads
 - Obviously, there must be at least as many total threads as cores, otherwise cores will be left idle.
- For best performance, actually want
#threads >> #cores
- Accesses to GPU memory have several hundred cycles latency
 - When a thread stalls waiting for data, if another thread can switch in this latency can be hidden.
- NVIDIA GPUs have very fast thread switching, and support many concurrent threads

Exposing parallelism example

```
Loop over i from 1 to 512  
  Loop over j from 1 to 512  
    independent iteration
```

Original code


1D decomposition

```
Calc i from thread/block ID  
  Loop over j from 1 to 512  
    independent iteration
```

 512 threads

2D decomposition

```
Calc i & j from thread/block ID  
  independent iteration
```

 262,144 threads

Memory coalescing

- GPUs have high *peak* memory bandwidth
- Maximum memory bandwidth is only achieved when data is accessed for multiple threads in a single transaction: *memory coalescing*
- To achieve this, ensure that **consecutive threads access consecutive memory locations**
- Otherwise, memory accesses are serialised, significantly degrading performance
 - Adapting code to allow coalescing can dramatically improve performance

Memory coalescing example

- *consecutive threads* are those with consecutive `threadIdx.x` values
- Do consecutive threads access consecutive memory locations?

```
index = blockIdx.x*blockDim.x + threadIdx.x;  
output[index] = 2*input[index];
```



Coalesced. Consecutive `threadIdx` values correspond to consecutive `index` values

Memory coalescing examples

- Do consecutive threads read consecutive memory locations?
- In C, outermost index runs fastest: j here

```
i = blockIdx.x*blockDim.x + threadIdx.x;  
for (j=0; j<N; j++)  
    output[i][j]=2*input[i][j];
```

 Not Coalesced. Consecutive `threadIdx.x` corresponds to consecutive `i` values

```
j = blockIdx.x*blockDim.x + threadIdx.x;  
for (i=0; i<N; i++)  
    output[i][j]=2*input[i][j];
```

 Coalesced. Consecutive `threadIdx.x` corresponds to consecutive `j` values

Memory coalescing examples

- What about when using 2D or 3D CUDA decompositions?
 - Same procedure. X component of `threadIdx` is always that which increments with consecutive threads
 - E.g., for matrix addition, coalescing achieved as follows:

```
int j = blockIdx.x * blockDim.x + threadIdx.x;  
int i = blockIdx.y * blockDim.y + threadIdx.y;  
  
c[i][j] = a[i][j] + b[i][j];
```


Code Branching

- On NVIDIA GPUs, there are less instruction scheduling units than cores
- Threads are scheduled in groups of 32, called a *warp*
- Threads within a warp must execute the same instruction in lock-step (on different data elements)
- The CUDA programming allows branching, but this results in all cores following all branches
 - With only the required results saved
 - This is obviously suboptimal
- Must avoid intra-warp branching wherever possible (especially in key computational sections)

Branching example

- E.g you want to split your threads into 2 groups:

```
i = blockIdx.x*blockDim.x + threadIdx.x;  
if (i%2 == 0)  
    ...  
else  
    ...
```



Threads within warp diverge

```
i = blockIdx.x*blockDim.x + threadIdx.x;  
if ((i/32)%2 == 0)  
    ...  
else  
    ...
```



Threads within warp follow same path

CUDA Profiling

- Simply set COMPUTE_PROFILE environment variable to 1
- Log file, e.g. cuda_profile_0.log created at runtime: timing information for kernels and data transfer

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla M1060
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6e2e9ee8858
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 37.952 ] cputime=[ 86.000 ]
method=[ memcpyHtoD ] gputime=[ 37.376 ] cputime=[ 71.000 ]
method=[ memcpyHtoD ] gputime=[ 37.184 ] cputime=[ 57.000 ]
method=[ _Z23inverseEdgeDetect1D_colPfS_S_ ] gputime=[ 253.536 ] cputime=[ 13.00
0 ] occupancy=[ 0.250 ]
...
```

- Possible to output more metrics (cache misses etc)
 - See doc/Compute_Profiler.txt file in main CUDA installation

Overview

- Introduction to CUDA
- *Practical Exercise 1: Getting started with CUDA*
- GPU Optimisation
- *Practical Exercise 2: Optimising a CUDA Application*
 - *See Practical PDF document*