An Introduction to HPC and Scientific Computing

Lecture nine: Scientific Computing using the CUDA programming language.

Wes Armour

Oxford e-Research Centre,
Department of Engineering Science

Learning outcomes

In this lecture you will learn about:

- More on CUDA.
- Flow control in CUDA codes.
- Synchronisation.
- Atomic operations.
- GPU reduction.
- Monte Carlo methods.

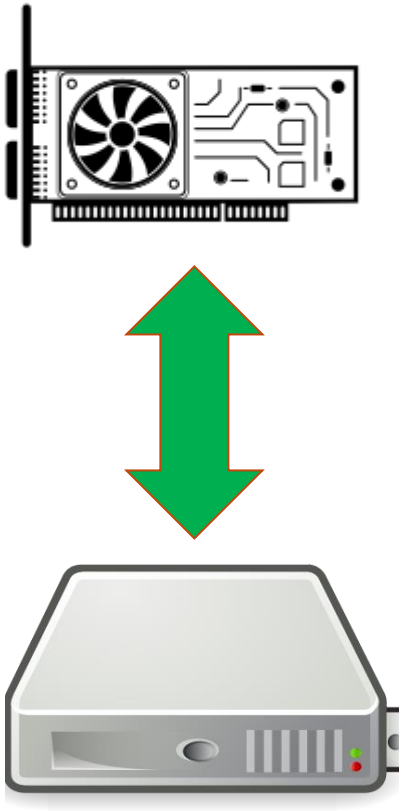
A quick recap – allocating memory

```
// Allocate pointers for device memory
float *d_input, *d_output;

// Allocate device memory input and output arrays
cudaMalloc((void**)&d_input,  mem_size);
cudaMalloc((void**)&d_output, mem_size);
```

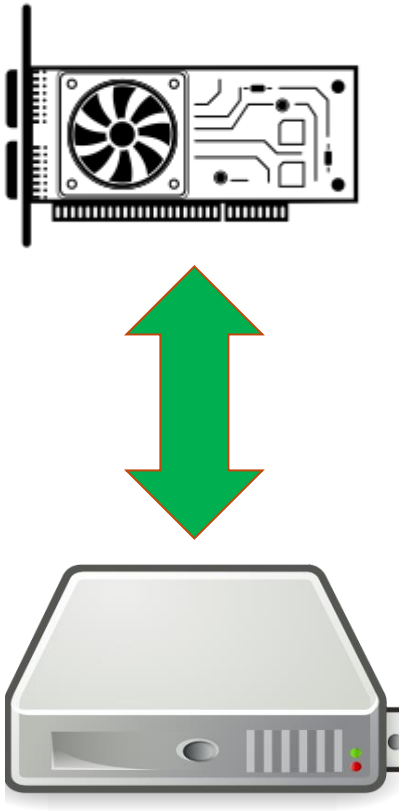
```
// Allocate pointers for host and device memory
float *h_input, *h_output;

// malloc() host memory (this is in your RAM)
h_input  = (float*) malloc(mem_size);
h_output = (float*) malloc(mem_size);
```



A quick recap – freeing memory

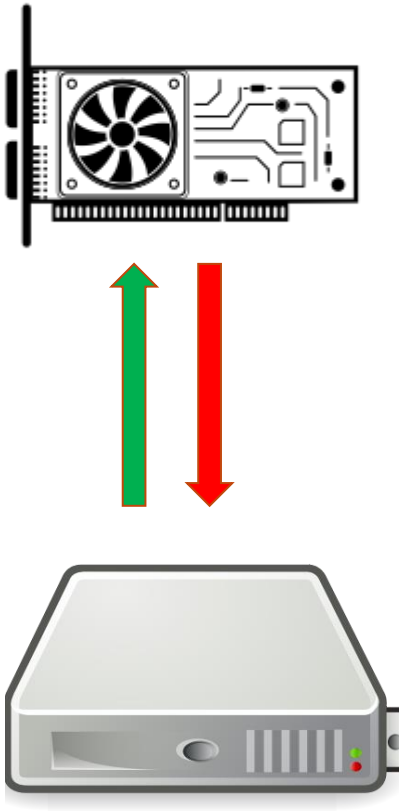
```
// Cleanup memory on the device
cudaFree(d_input);
cudaFree(d_output);
```



```
// Cleanup memory on the host
free(h_input);
free(h_output);
```

A quick recap – transferring data

```
// Copy result from device to host
cudaMemcpy(host, device, mem_size, cudaMemcpyDeviceToHost);
```



```
// Copy host memory to device input array
cudaMemcpy(device, host, mem_size, cudaMemcpyHostToDevice);
```

A quick recap – special variables

`gridDim`	`gridDim.x`	Size (or dimensions) of grid of blocks
`blockDim`	`blockDim.y`	Size (or dimensions) of each block
`blockIdx`	`blockIdx.z`	Index (or 2D/3D indices) of block
`threadIdx`	`threadIdx.y`	Index (or 2D/3D indices) of thread
`warpSize`		Currently 32 lanes (and has been so far)
`kernel<<<…>>>(args);`		Kernel launch

A quick recap – a simple code

```
int main() {

    float *h_x, *d_x;
    int nblocks=2, nthreads=8, nsize=2*8;

    h_x = (float *)malloc(nsize*sizeof(float));
    cudaMalloc((void **)&d_x, nsize*sizeof(float));

    my_kernel<<<nblocks,nthreads>>>(d_x);

    cudaMemcpy(h_x,d_x,nsize*sizeof(float),cudaMemcpyDeviceToHost);

    for (int n=0; n<nsize; n++) printf(" n, x = %d %f \n",n,h_x[n]);

    cudaFree(d_x); free(h_x);
}
```

A quick recap – a simple code

This simple code allocates some memory on the host and device.

It launches a kernel that has 2 blocks of 8 threads.

```
blockDim.x  = 8
blockIdx.x  = 0 or 1
threadIdx.x = 0…7
```

The kernel *typecasts* the integer index `tid` to a floating point number.

```
__global__ void my_first_kernel(float *x)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    x[tid]  = (float) threadIdx.x;
}
```

https://www.tutorialspoint.com/cprogramming/c_type_casting.htm

Flow control

Different warps can execute different code (for example warp one calculates a +, warp three calculates a *).

This has no impact on performance.

How about if we have an `if` statement within a warp? (i.e. we have different possible code paths)??

```
if(radius <= 10.0) {
    area = PI * radius * radius;
} else {
    area = 0.0;
}
```

Flow control

In this case if all threads *in a warp* have a radius less than or equal to 10 then the execution path for all threads follows the same direction. This is also true if all threads in the warp have a radius greater than 10.

What if some threads in the warp have a radius less than 10 and some more than 10?

In this case both execution paths for the warp are executed but threads that don't participate in a particular execution path are masked.

```
if(radius <= 10.0) {
    area = PI * radius * radius;
} else {
    area = 0.0;
}
```

Warp divergence

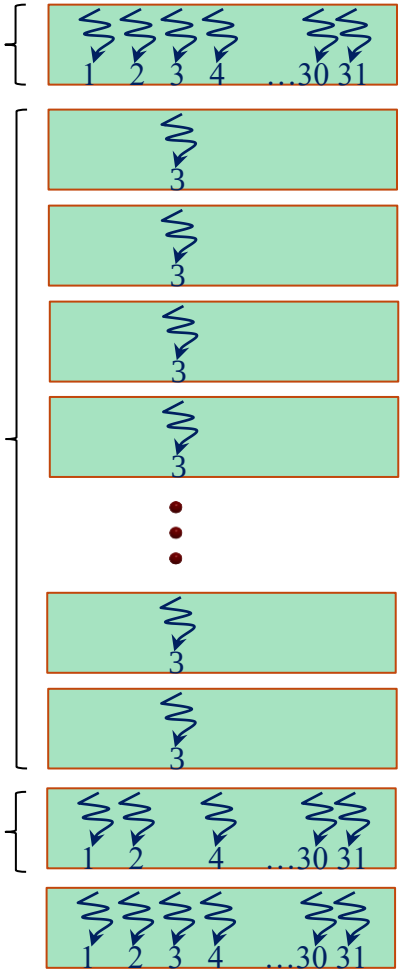
This is called *warp divergence*

Warp divergence can cause significant slow down in code execution. In the worst case this can cause a loss of 32x in performance if only one thread needs to follow a computationally expensive branch and the rest of the threads do nothing.

```
array[threadIdx.x] = 0.0;

if(threadIdx.x == 3) {
  for(i=0; i < 1000; i++) {
    array[threadIdx.x]++;
  }
} else {
  array[threadIdx.x] = 3.14;
}

else {
  array[threadIdx.x] = 3.14;
}
```



Synchronisation

As we've mentioned before all threads within a warp execute in lock-step. They are explicitly synchronised.

What if we need to synchronise threads within a thread block?

CUDA provides a barrier function for this called:

```
__syncthreads();
```

This causes all threads within a thread block to wait until every thread in that thread block has reached this point.

Synchronisation

What if we want to synchronise **everything** – *in our main() code we want to ensure that we reach a point where everything on the host and device is synchronised?*

```
cudaDeviceSynchronize();
```

Ensures that all asynchronous (sometimes called non-blocking) tasks are finished before code execution proceeds further.

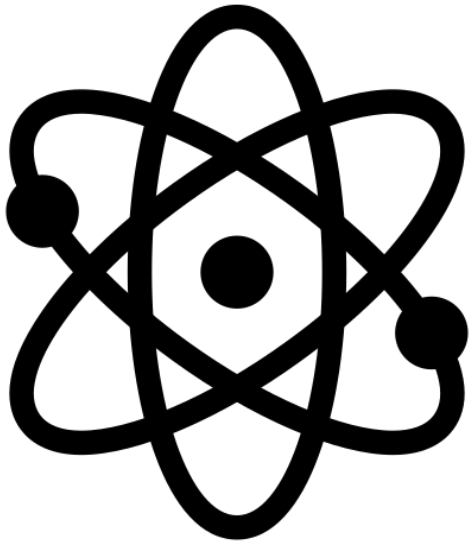
This is a function used in host code.

Atomics

There will be instances where you need two or more threads to update a single variable.

Obviously due to the undefined scheduling of threads, several threads could try to update the variable at the same time leading to an undefined result.

To get around this issue, CUDA provides several atomic operations to do this for you, these guarantee that only one thread at a time can update the variable.



https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Atomics

```
type atomicAdd(type* address, type val);
```

type can be `int, unsigned int, unsigned long long int, float, double`

The function reads the value of “old” located at address (in global or shared memory), then computes (old + val). The result is placed back to memory at the same address.

The function returns old.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Atomics

There are other atomic functions that are all useful too.

There is a subtraction version (like addition).

There is a maximum and minimum, these compare the value at address with val and store the maximum or minimum.

There is also a function for exchange, so the val is stored in place of the value at address.

```
type atomicSub(type* address, type val);

type atomicExch(type* address, type val);

type atomicMin(type* address, type val);

type atomicMax(type* address, type val);
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Atomics

On older hardware atomics can be verrryyyyy slooooooowwww!!!

On newer hardware atomics are quicker, but I'd recommend using them only when you really need to.

https://www.flickr.com/photos/fatboyke/2668411239

Reduction

Now let's look at a slightly harder example of using a GPU to solve a problem.

There are many times in scientific computing where you will be required to perform a simple operation such as summing a series of numbers. The classic example that you will all be familiar with is when calculating a mean (μ).

But this is a sequential sum.

We've seen that sequential programs wont work well on GPUs, a single GPU thread (which would be mapped to a single GPU core) would perform the entire sum.

This means that on the biggest GPUs only 1/5120th (one core / all cores) would be used!

$$\mu = \frac{1}{n}\sum_{i=0}^{n-1} x_i$$

Reduction

Lets think about the sum that represents the mean in a little more detail.

We could break the sum down into sub-sums by summing nearest neighbour elements.

Taking this approach further we could then sum nearest neighbours of the resulting sum. And so on…

$$n\mu = x_0 + x_1 + x_2 + x_3 + x_4 + x_5 \ldots x_{n-1}$$

$$n\mu = (x_0 + x_1) + (x_2 + x_3) + (x_4 + x_5) \ldots x_n$$

$$\text{Let } x'_0 = (x_0 + x_1) \ldots$$

$$n\mu = x'_0 + x'_1 + x'_2 + x'_3 \ldots x'_{\frac{n}{2}}$$

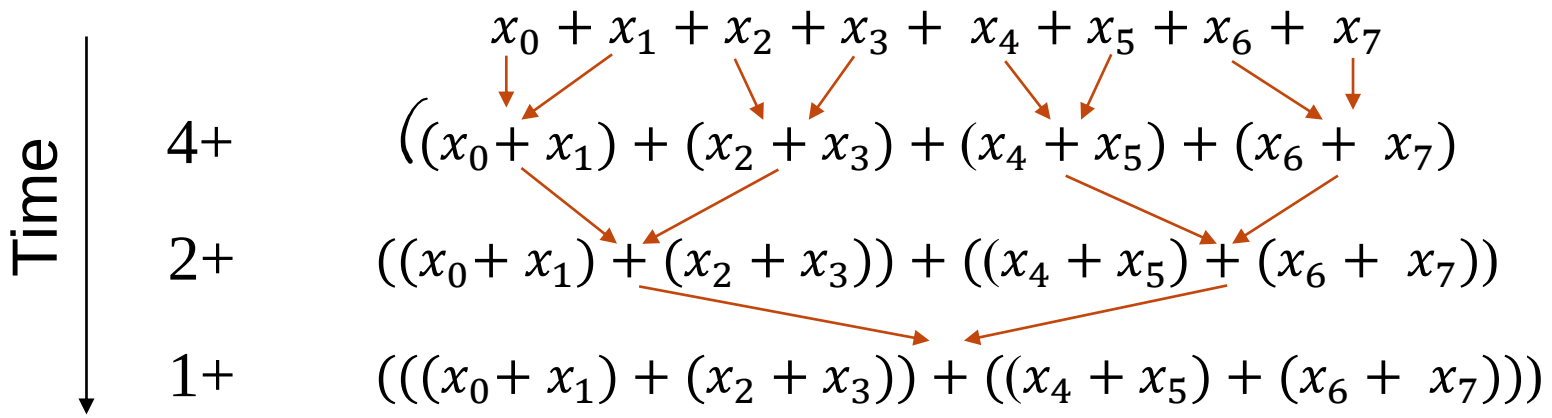
$$\text{Let } x''_0 = (x'_0 + x'_1) \ldots$$

$$n\mu = x''_0 + x''_1 + x''_2 + x''_3 \ldots x''_{\frac{n}{4}}$$

$$\vdots$$

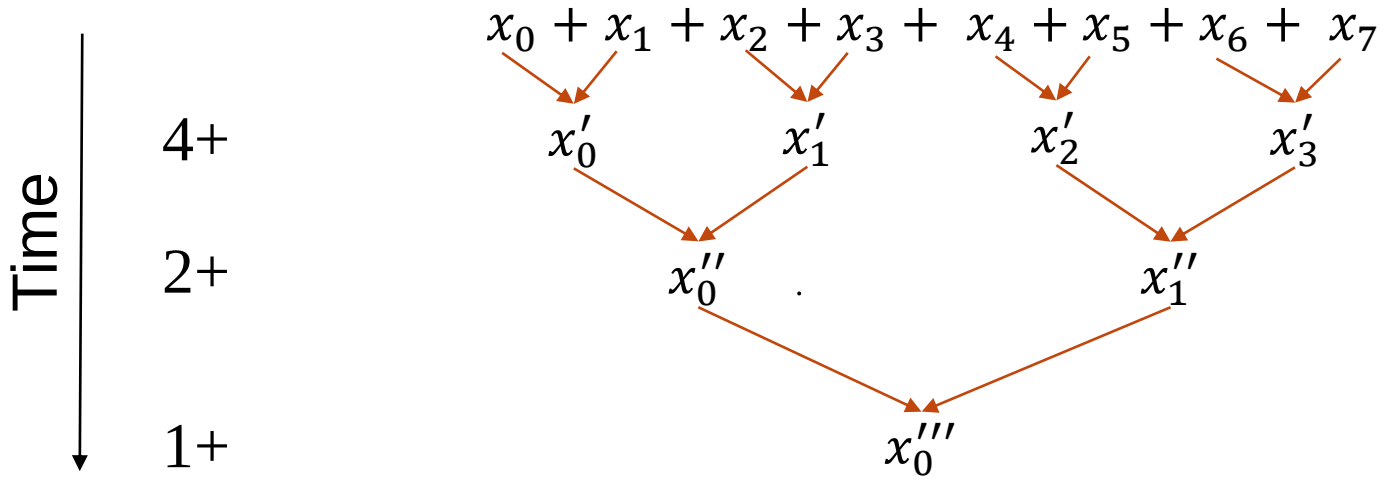
Reduction

The schematic below gives an example of the algorithm that we outlined on the previous slide for eight elements.
We see that 4 + 2 + 1 = 7 additions are required to reach the final summation. However these can all be performed in parallel.



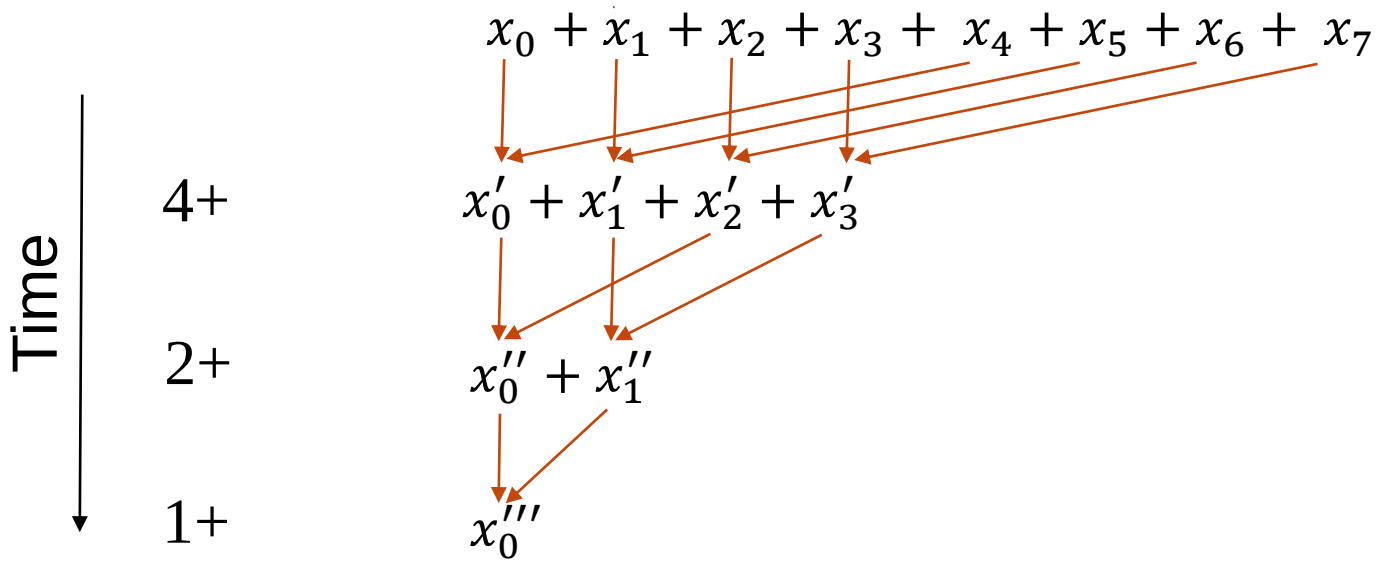
Reduction

Making the schematic a little bit clearer gives us some indication of how data would be accessed in a cache line.
We see that this doesn't really look like the coalesced memory access pattern that we covered in our first CUDA lecture slides.



Reduction

We can change the way in which we construct the partial sums (we would sum two elements that aren't nearest neighbours).
The schematic below gives another example of how to perform the reduction in parallel.



Reduction

Now lets look at a standard serial C code and think about how we might change this into a fast CUDA code.

```
#include <stdio.h>
#include <stdlib.h>

#define NUM_ELS 1024

int main() {

    float random_array[NUM_ELS];
    float sum=0;

    for(int i=0; i<NUM_ELS; i++) {
        float x=((float)rand())/((float)RAND_MAX);
        random_array[i]=x;
    }

    for(int i=0; i<NUM_ELS; i++) {
        sum+=random_array[i];
    }

    printf("\nAverage:\t%f\n", sum/(float)NUM_ELS);

    return(0);
}
```

Reduction

The kernel on the right demonstrates how to perform the reduction for a single thread block. The block has NUM_ELS elements.

The code loads data from global memory into shared memory. It syncthreads to ensure that all warps in the threadblock have loaded there memory.

It then steps through the binary tree performing the reduction as we outlined in slide 22.

Finally the code stores element zero located in the smem array (which contains the final value) into global memory.

```
__global__ void reduction(float *d_input, float *d_output)
{
    // Allocate shared memory

    __shared__  float smem_array[NUM_ELS];

    int tid = threadIdx.x + blockDim.x*blockIdx.x;

    // first, each thread loads data into shared memory

    smem_array[tid] = d_input[tid];
    __syncthreads();

    // next, we perform binary tree reduction

    for (int d = blockDim.x/2; d > 0; d /= 2) {
      __syncthreads();  // ensure previous step completed
      if (tid<d)  smem_array[tid] += smem_array[tid+d];
    }

    // finally, first thread puts result into global memory

    if (tid==0) d_output[0] = smem_array[0];
}
```

π

Our lecture *Introduction to the C programming language* gave examples of calculating the area of a circle.

Lets imagine that an alien lands on Earth. He's interested in finding the area of a circle too.

He knows the equation to do this. He also knows the equation to calculate the area of a square. But he doesn't know the value of the constant π

(Quite surprising because it's managed to travel half way across the universe. -- This seems very suspicious).

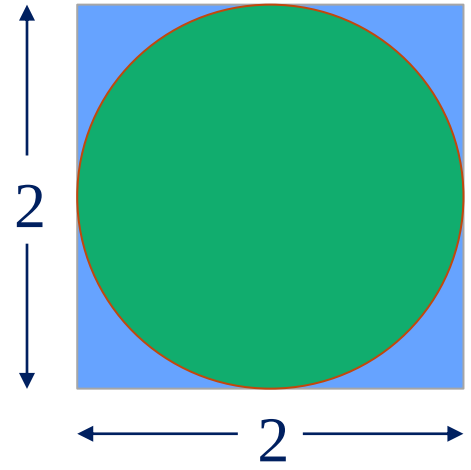
$$area\ of\ a\ circle = \pi\, r^2 = \pi(x^2 + y^2)$$

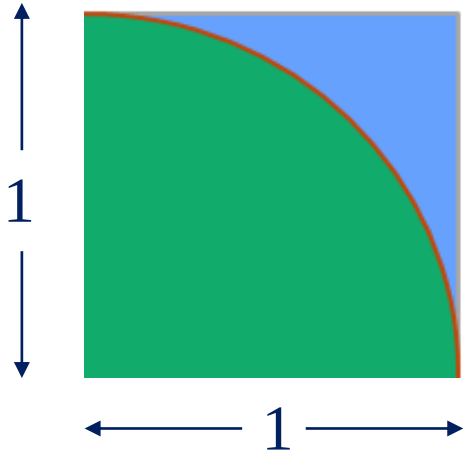
$$area\ of\ a\ square = x\ * y$$

Monte Carlo methods

Our alien “friend” can approximate π in the following way.

1. First it draws a 2 x 2 square.
2. Then it draws a circle in the square, the radius of the circle is 1.
3. Finally it concentrates on a single quadrant of the square.





Monte Carlo methods

Our alien then works out the following…

$$area\ of\ the\ quadrant = \frac{\pi\ 1^2}{4} = \frac{\pi}{4}$$

$$area\ of\ the\ quater\ square = \frac{2\ *2}{4} = 1$$

$$\pi\ = 4\ * \frac{area\ of\ quadrant}{area\ of\ a\ quater\ square}$$

https://www.flickr.com/photos/wwarby/11457324843

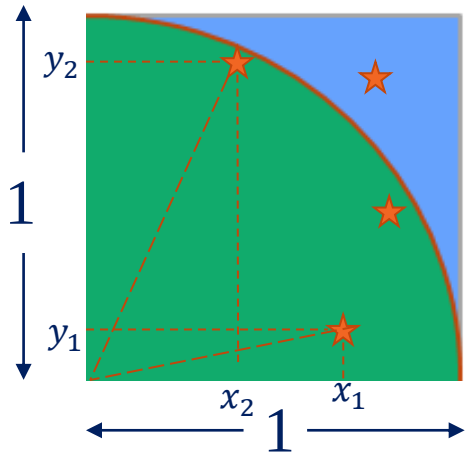
William Warby

Monte Carlo methods

Next it picks two random numbers, both between zero and one. Lets call these (x_1, y_1). It then picks another pair of numbers (x_2, y_2), and so on…

If we look at the example on the right we see that there are four random points, three have landed in the circle.

Using the previous formula our alien can estimate π



$$\pi = 4 * \frac{3}{4} = 3$$

Monte Carlo methods

The code on the right is an example serial implementation of the algorithm that we've just worked through.

The code loops over pairs of uniformly distributed random numbers (having a value between 0 and 1).

The code then calculates the distance of the point from the origin and compares this to the radius of our unit circle.

If the point lands within the quadrant then it adds one to the estimate of the area of the quadrant.

Once the loop completes π is estimated using our previous formula.

```
#include <stdio.h>
#include <stdlib.h>

int main() {

    int N=10000000;
    int area=0;

    for(int i=0; i<N; i++) {
        float x = ((float)rand())/RAND_MAX;
        float y = ((float)rand())/RAND_MAX;
        if(x*x + y*y <= 1.0f) area++;
    }
    printf("\nPi:\t%f\n", (4.0*area)/(float)N);

    return(0);
}
```

Monte Carlo methods

Things to consider when converting this into a CUDA code:

- What libraries to use and include?
- Warp divergence?
- Use cuRAND to generate random numbers?
- How to we ensure good coalesced access to data?
- How do we sensibly make the update to the variable area?
- How do we increase accuracy?

```
#include <stdio.h>
#include <stdlib.h>

int main() {

    int N=10000000;
    int area=0;

    for(int i=0; i<N; i++) {
        float x = ((float)rand())/RAND_MAX;
        float y = ((float)rand())/RAND_MAX;
        if(x*x + y*y <= 1.0f) area++;
    }
    printf("\nPi:\t%f\n", (4.0*area)/(float)N);

    return(0);
}
```

What have we learnt?

In this CWM you have learnt about computer architectures, CPUs and GPUs. You've learnt the C programming language and are now able to write your own codes.

You've learnt about Linux, compilers and build systems. Along with good coding practices.

Finally you've learnt about CUDA and the associated tools and libraries.

Lastly you've learnt - don't trust aliens.

Further reading

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

https://devblogs.nvidia.com/even-easier-introduction-cuda/

https://developer.nvidia.com/about-cuda

https://developer.nvidia.com/cuda-zone

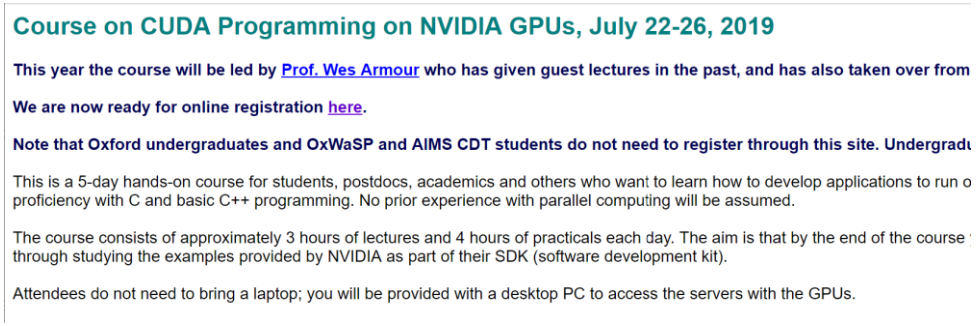
Further work

If you've enjoyed learning about CUDA and would like to develop your knowledge and skills we offer a course on CUDA programming which is free to Oxford undergraduates.

There is no assessment, however you do receive a certificate on completing the course, which is great to add to your CV.

If you are interested contact:

wes.armour@eng.ox.ac.uk



Course on CUDA Programming on NVIDIA GPUs, July 22-26, 2019

This year the course will be led by Prof. Wes Armour who has given guest lectures in the past, and has also taken over from

We are now ready for online registration here.

Note that Oxford undergraduates and OxWaSP and AIMS CDT students do not need to register through this site. Undergradu

This is a 5-day hands-on course for students, postdocs, academics and others who want to learn how to develop applications to run o proficiency with C and basic C++ programming. No prior experience with parallel computing will be assumed.

The course consists of approximately 3 hours of lectures and 4 hours of practicals each day. The aim is that by the end of the course through studying the examples provided by NVIDIA as part of their SDK (software development kit).

Attendees do not need to bring a laptop; you will be provided with a desktop PC to access the servers with the GPUs.

https://people.maths.ox.ac.uk/gilesm/cuda/

UNIVERSITY OF OXFORD

Oxford e-Research Centre