Concurrent and Parallel Systems

Unit 1 -Multi-threading

1.1 -Starting a Thread

Starting a thread:
Creating a thread in C++ is simple. We need to include the thread header file

- #include <thread>

To create a thread, we then only need create a new thread object in our application, passing in the name of the function that we want the thread to run

- thread t(hello_world);

This will create a new thread that will run the function hello_world(). The thread will start executing the function while the main application continues operating on the rest of the code. Essentially the main application is a thread that is created and executed as soon as an application is launched.

Waiting for a thread to complete:
Generally we want to wait for a thread to complete its operation. We do this by using the join method on the thread

- t.join();

This will mean the currently executing code (whether it be the main application thread or another thread) will wait for the thread it joins to complete its operation.

1.2 -Multiple Tasks

Starting one thread is all well and good, but we really want to execute multiple tasks. Creating multiple threads is easy - we just create multiple thread objects. We can use the same function if we want (useful for parallelising a task), or we can run different functions (useful for tasks which block computation).
We will use a new operation during this next example;

- sleep_for(seconds(10));

This operation will allow us to put a thread to sleep for an amount of time.

To allow us to access the duration constructs (which contains seconds, etc) we use the chrono header

- #include <chrono>

1.3 -Passing Parameters to Threads

We have now seen how to create threads in C++ 11 and how to put them to sleep. The next piece of functionality we are going to use is how we pass parameters to a thread. This is actually fairly easy, and just requires us adding the passed parameters to the thread creation call. For example, given a function with the declaration:

- void task(int n, int val)

We can create a thread and pass in parameters to 'n' and 'val' as follows:

- thread t(task, 1, 20);

'n' will be assigned the value 1 and 'val' will be assigned the value 10. Our test application will use random number generation, which has also changed in C++11.

-Random Numbers in C++ 11

C++ 11 has added an entirely new random number generation mechanism which is very different, as well as having more functionality. To use random numbers we need to include the random header

- #include <random>

We then need to create a random number generation engine. There are a number of generation engines, but we will use the default one in C++11.

- default_random_engine e(seed);

'seed' is a value used to seed the random number engine (you should hopefully know by now that we cannot create truly random numbers so a seed defines the sequence the same seed will produce the same sequence of random numbers). We will use the system clock to get the current time in milliseconds to use as a seed. To get a random number from the engine we simply use the code;

- auto num = e ();

-Ranged For Loops in C++11

The other new functionality we will use in this example is a ranged for loop. You will probably be familiar with the foreach loop in C#. The ranged for loop in C++11 has the same functionality.

- for (auto &t : threads)

We can then use t as an object reference in our loop. The threads variable is a collection of some sort (here a vector, but we can use arrays, maps, etc.). This is much easier than the iterator approach in pre-C++11 code.

1.4 -Using Lambda (λ) Expressions

Lambda (λ) expressions are another new feature of C++11, and are becoming a popular feature in object-oriented languages in general (C# has them, Java 8 will have them). λ expressions come from functional style languages (for example, F#, Haskell, etc). They allow us to create function objects which we can apply parameters to and get a result.

-What is a λ Expression?

A λ expression is essentially a function. However, it has some properties that allow us to manipulate that function. For example, if we define a function as follows

- add(x,y) = x + y

We can then use this function object to create a new function object that adds 3 to any parameter:

- add3(y) = add(3)

We can then use the add3 function as a new function. We can then get the results from function calls as follows

- add(10,5) = 15
- add3(10) = 13
- add(2,0) = 2
- add4(y) = add(4)
- add4(6) = 10

-λ Expressions in C++11

One advantage of C++ λ expressions is that they allow us to create functions using fewer lines of code. Traditionally we would create an entire function / method definition with parameters and return type. In C++11, we can simplify this. For example, we can create an add function shown below:

- auto add = [] (int x, int y) { return x + y; };

The [] allows us to pass parameters into the function (we will look at this capability later). We define the parameters we want to pass to the function. Finally, in the curly brackets we define the λ expression. We can then use the add function as a normal function:

- auto x = add(10, 12);

We are brushing the surface of λ expressions here, but it is enough for what we are doing in the module.

1.5 -λ Expressions and Threads

We can use λ expressions to create threads, making our code more compact (but not necessarily easier to read).

For the rest of the module, it is up to you whether you want to use λ expressions or not. We will work interchangeably, and there is no effect on your grades.

1.6 -Gathering Data

Throughout this module, we will be gathering profiling data to allow us to analyse performance of our applications, and in particular the speed up. To do this, we will be outputting data into what is called a comma separated value (.csv) file. This is a file that we can load easily into Excel and generate a chart

from. In this first application, we will just get the average of 100 iterations, and our work will be simple (it will just spin the processor and do nothing).

-Creating a File

Creating a file in C++ is easy, and you should know this by now. As a reminder, we create an output file as shown below:

- ofstream data("data.csv", ofstream::out);

This will create an output file called data.csv which we can write to. We can treat the file just as any output stream (e.g. cout).

-Capturing Times

To capture times in C++11, we can use the new system_clock object. We have to gather a start time, and end time, and then calculate the total time by subtracting the first from the second. Typically, we will also want to cast the output to milliseconds (ms).

-Getting the Data

Once the application is complete, you should have a .csv file. Open this with Excel. Then get the mean of the 100 stored values (there is a good chance that a number of the recorded values are 0).
You also want to document the specification of your machine. This is very important. You can find this in the device manager of your computer. For example, the result from my test is 0.27ms. The hardware I ran the test on is:

- CPU - Intel i5-2500 @ 3.3GHz
- Memory - 8GB
- OS - Windows 7 64bit

This is the only pertinent information at the moment. As we progress through the module, we will find that the other pieces of information about your machine will become useful.

1.7 -Monte Carlo π

We now move onto our first case study - the approximation of π using a Monte Carlo method. A Monte Carlo method is one where we use random number generation to compute a value. Because we can just increase the number of random values we test, the problem we use can be scaled to however many computations we wish. This will allow us to actually perform a test that we can push our CPU with.

Theory

The theory behind why we can calculate π using a Monte Carlo method can best be described by figure 1.5. The radius of the circle is refined as R. We know that the area of a circle can be calculated by the following equation:

- πR^2

We can also calculate the area of a square as follows:

- 4R^2

Now let us imagine that R=1, or in other words we have a unit circle. If we pick a random point within the square, we can determine whether it is in the circle by calculating the length of the vector equivalent of the point. If the length is 1 or less, the point is within the circle. If it is greater than 1, then it is in the square but not the circle. The ratio of radom points tested to ones in the circle allows us to approximate π. This is because:

- AreaC = πR^2
- AreaS = 4R^2
- Ratio = AreaC / AreaS = π / 4

We can then create an algorithm to approximate π as shown:

Algorithm 1 Monte Carlo π

```
begin
    attempts : = N
    in_circle := M
    ratio := M/N
    π/4 ~= M/N => π ~= 4 * M/N
end
```

We can therefore approximate by generating a collection of random points (we will work in the range [0.0, 1.0] for each coordinate), checking the length and if it is less than or equal to 1 we count this as a hit in the circle.

Distributions for Random Numbers

Before looking at our C++11 implementation of the Monte Carlo π algorithm, we need to introduce the concept of distributions for random numbers. A distribution allows us to define the type of values we get from our random engine and the range of the values. There are a number of distribution types. We will be using a real distribution (i.e. for double values) and we want the numbers to be uniformly distributed (i.e. they will not tend towards any particular value). We also define the range from 0.0 to 1.0. Our call to do this is shown:

- //Create a distribution
- uniform_real_distribution<double> distribution(0.0, 1.0);
- //Use this to get a random value from our random engine e
- auto x = distribution(e);

Our algorithm implementation in C++ is shown in Listing 1.27.

We have used all the pieces of this algorithm by now, so you should understand it. Notice that we are not actually getting the return value for π at the moment you can print it out if you want to test the accuracy. We will look at how we can get the result from a task in later tutorials.

Main Application

Unit 2 -Controlling Multi-Threaded Applications

Control of multi-threaded applications is really important due to the problems encountered with shared memory. However, controlling multi-threaded applications also leads to problems. The control of multi-threaded applications is really where the concept of concurrency comes in.
The second half of this tutorial will look at some more control concepts;

- Atomics
- Futures

-Atomics are a simple method of controlling individual values which are shared between threads.
-Futures allow us to spin off some work and carry on doing other things, getting the result later.

2.1 -Shared Memory Problems

First we will see why shared memory is a problem when dealing with multi-threaded applications. We are going to look at a simple application which increments a value, using a pointer to share the value between all the threads involved. The example is somewhat contrived, but illustrates the problem.

-Shared Pointers in C++11

Smart pointers have made the world of the C++ programmer much easier in that you no longer need to worry about memory management (well, not as much anyway). In this example, we will use a shared_ptr. A shared_ptr is a pointer that is reference counted. That is, every time you create a copy of the pointer (i.e. provide access to a part of the program), a counter is incremented. Every time the pointer is not required by a part of the program, the counter is decremented. Whenever the counter hits 0, the memory allocated is automatically freed. This does have a (very slight) overhead, but will make our life easier.
For this example, we are going to use the make_shared function. make_shared will call the relevant constructor for the defined type based on the parameters passed. For example:

- auto value = make_shared<int>(0);

The type of value (automatically determined by the compiler) is

shared_prt<int>. We can now happily share this value without our program (with multi-threaded risks) by passing the value around.
To use smart pointers, we need to include the memory header:

- #include <memory>

-Application

Our application is quite simple - it will increment our shared value. Listing 2.3 illustrates.

Notice what we do on line 6. We dereference the pointer (this still works for shared_ptr) to get the value stored in the shared_ptr and add one to the value, setting the shared_ptr value with this new incremented value. Our main application is shown in listing 2.4.

Notice on line 7 we use a call to thread::hardware_concurrency. This value is the number of threads that your hardware can handle natively - a useful value if you understand our results from the last tutorial.
As increment adds 1 million to the shared value, depending on your hardware configuration you might expect a printed value of 1 million, 2 million, 4 million, maybe even 8 million. However, running this application on my hardware I get the result shown in figure 2.1. Somehow I lost almost 2 million increments. So where did they go?

This is the problem of shared resources and multi-threading. We do not actually know which order the threads are accessing the resource. For example, if we consider a 4 thread application, something like Figure 2.2 might happen.

Even though each thread has incremented the value twice (you can check) and the value should be 8, the actual value is 2. Each thread is competing for the resource. This is what we call a RACE CONDITION (or race hazard). This is an important concept to grasp and you need to understand that sharing resources in multi-threaded applications is inherently dangerous if you do not add some control (control does bring its own problems).

2.2 -Mutex

The first approach to controlling multi-threaded applications is using what's called a mutex. A mutex (mutual expression) allows us to guard particular pieces of code so that only one thread can operate within it at a time. If we share the mutex in some manner (mutexes cannot be copied so have to be shared in some manner) then we can have different sections of code which we protect, ensuring that only one of these sections is running at a time.
To use a mutex, we need to include the mutex header.

- #include <mutex>

We will also need a mutex as a global variable for our program

- mutex mut;

We can then simply update our increment method from the previous application to use the mutex. The two important methods of the mutex we will be using are lock (to lock the mutex thus not allowing any other thread to successfully lock) and unlock (freeing the mutex, allowing another thread to call lock). Any thread which cannot lock the mutex must wait until the mutex is unlocked and it successfully acquires the lock (there might be a queue of waiting threads).
With a mutex in place, we can update our increment function to that shown in Listing 2.6. If you run this application (and note that it takes a little longer) your result should be as expected (4000000).

2.3 -Lock Guards

Remembering to lock and unlock a mutex can cause problems (especially when you forget to lock and unlock) and does not necessarily make your code simple to follow. Other problems that can occur come from methods having multiple exit points (including exceptions) which means you might miss when to unlock a mutex. Thankfully, thanks to C++s object deconstruction at the end of scopes, we can utilise what is known as a lock guard to automatically lock and unlock a mutex for us.
Modifying our application to use lock guards is very simple. A change to the increment function is required as shown in Listing 2.7.
Using mutexes and lock guards is the simplest method of protecting sections of code to ensure shared resources are protected. However, their simplicity does lead to other problems (such as dealock) which we need to overcome.

2.4 -Condition Variables

One limitation when using mutexes is that we are only really controlling access to certain sections of the application to try and protect shared resources. You can think of it as having a gate. We let one person in through the gate at any one time, and do not let anyone else enter until the person has left. This is all well and good, but we have no control over what happens outside the gate (where arguments over who is next in line may occur).
Another approach we might want to take is waiting for a signal - a sign to state that we can perform some action. We could happily wait, and when the signal is activated we stop waiting and carry on doing some work. This is a technique originally defined using what was called a semaphore.
Consier what happens are a set of traffic lights at a crossroads. We have two streams of traffic that wish to use the intersection at the same time. By having a set of lights (think of red as meaning wait and green meaning signal), we control access to the crossroads. A semaphore (and in C++11 a condition variable) allows us to control access in a similar manner to a set of traffic lights at a crossroads.
To use condition variables, we need to include the condition_variable header

- #include <condition_variable>

We are going to look at three operations to use with a condition variable. First there is wait

- condition.wait(unique_lock<mutex>(mut));

This will enable us to wait until a signal has been received. We have to pass a lock to the wait method, and as we are not using the lock anywhere else we use unique_lock. This ensures that we are waiting on the mutex.
The next method we are interested in is notify_one

- condition.notify_one();

This will notify one thread that is currently waiting on the condition variable. There is a similar method called notify_all which will signal all waiting threads.
The final method we are interested in is wait_for

- if (condition.wait_for(unique_lock<mutex>(mut), seconds(3)))

As you can see, wait_for returns a true or false value. If we are signalled (notified) before the time runs out, then wait_for returns true. Otherwise, it returns false. There is a similar method called wait_until which allows you to set the absolute time to stop waiting at.

-Application

We are going to create 2 threads which wait and signal each other, allowing interaction between the threads. Our first thread runs the code in Listing 2.12, and our second will run the code in listing 2.13. Finally, our main is given in listing 2.14.

Note the use of the ref function on lines 7 and 8. This is how we create a reference to pass into our thread functions. The interactions between these threads have been organised so that you will get output shown in Figure 2.4.

2.5 -Guarded Objects

Now that we know how to protect sections of code, and how to signal threads, let us consider how we go about doing thread safe(ish) code in an object-oriented manner. We are going to modify our increment example so that an object controls the counter. First of all we need to define a header file

- #include “guarded.h”

Listing 2.15 provides the contents.
Notice that we provide the object with its own mutex. This is to protect access to our value. We then use a lock_guard to control access to the increment method. This goes in our guarded.cpp file. The contents of which can be found in Listing 2.16.
The method is somewhat contrived to force multiple operations within the method. Finally, our main method is shown in Listing 2.17.
Your output window should state that value equals 4 million. The use of a lock_guard and an object level mutex is the best method to control access to an object. This will ensure that methods are only called when permitted by competing threads.

2.6 -Thread Safe Data Structures

To end the first part of this tutorial, we will look at how we can implement a thread safe stack. Data structures are the normal method of storing data, but

are the most susceptible to multi-threading problems. This example is taken from C++ Concurrency in Action (slightly modified).

-Overview

A stack is one of the simplest data structures. We simply have a stack of values which we can add to the top of (push) or remove from the top of (pop). Our implementation will be very primitive, and will just wrap a standard stack in an object with thread safe operations.
You will need to create a new header file

- threadsafe_stack.h

First we declare our class and constructures as in Listing 2.18.

Notice that we have a copy constructor. When it is copying it must lock the other stack to ensure its copy is correct. Also note the use of the keyword mutable on line 15. This keyword indicates that the mutex can be modified in const methods (where normally we do not mutate an object's state). This is a convenience as our mutex does not affect other classes, but we still want to have const methods.

-Push

Our push method is quite trivial - all we need to do is lock the stack for usage. The code is in Listing 2.19.

-Pop

Pop is a little bit more involved. We still have to lock the object, but we must also check if the stack is empty before attempting to return a value. The code is provided in Listing 2.20.

On line 7, we check if the internal stack is empty, and if so throw an exception. There is a very good chance you will have never worked with exceptions in C++ before (it is newish but prior to C++11). They essentially work the same as Java and C#.

-Empty

Our final method allows us to check if the stack is empty (Listing 2.21).

-Tasks

Our test application is going to have one thread add 1 million values to the stack, and the other extract 1 million values from the stack. The approach is not the most efficient (using exceptions to determine if the stack is empty is not a good idea really), but will provide you with an example of exception handling in C++.
Our first task is called pusher - it's job is to push values onto the stack (Listing 2.22).
Notice the use of yield on line 8. This means that the thread will let another

thread in front of it if one is waiting. We are adding this to make the pusher yield to our other task (popper), meaning the stack will appear empty sometimes. Popper is defined in Listing 2.23.
Popper will try and pop a value from the stack. If it is empty, it will catch an exception and print it. The try-catch construct is similar to Java and C#.

-Main Application

Our main application just needs to create our resources, start the two tasks, and then check if the stack is empty at the end (1 million values pushed minus 1 million values popped). It is shown in Listing 2.24.

Remember that 1 = true, so the stack is empty.

2.8 -Atomics

Atomics in C++ are simple data types and related operations - actually operations that are of interest here. An atomic operation is one that cannot be split apart.

In our first example in this tutorial, we invoked a (very contrived) data race condition. The problem we had was that our actual increment operation performed the following internal operations:

1. Get Value
2. Add 1 to Value
3. Store Value

You can make a simple observation that there are two “holes” within our increment (assuming serial computation) between 1 and 2, and 2 and 3. Within these holes, anything could be happening on the machine (another increment for example).

An atomic operation has no holes. For example, an increment operation acts as a single indivisible operation - there is only before the operation and after, not in the middle of. This means we do not need to worry about data race conditions (to a certain extent).

Limiting factor is that they only provide simple operations such as add, load, store, swap, etc. These operations are simple enough to be used with atomic types.

A CPU may also natively support some atomic operations, making like even easier. However, in some cases, there may be a lock involved internally (and hidden away) to perform the atomic action.

-Sample Atomic Application

We are going to recreate our increment application, but this time using atomics

rather than a mutex to protect the data. Our application does not change too much from our original non-mutex version, apart from the use of the atomic value.

#include <atomic>

An atomic itself is just a template value in our application. For example we can define an atomic int as follows

atomic<int> value;

Depending on the type of atomic, we have a number of different operations defined.

In many regards, we can treat integral (ie number) atomic values as normal integral values. We merely state that they are atomic.

As we are using a shared_ptr, we must dereference it it to increment (line 6). You might notice that this version of increment is faster than the mutex approach (measure to check).

-atomic_flag

Another (slightly unique) atomic construct is something called an atomic_flag. It can be considered like a boolean value, although it is more akin to having a signal (either it is set or it is not). An atomic flag provides two methods of interest:
-test_and_set tests if the flag is set, if it is, then returns false. If it is not, sets the flag and returns true.
-clear clears the set flag

These operations can be very fast on supported CPUs and are therefore very good for situations where we do not want to put a thread to sleep while we perform an operation, but rather would like the thread to keep "spinning" (doing something - maybe even testing the flag again).

The interesting code is line 8 - notice that our while loop will keep spinning until the flag can be set by the calling thread.

The threads interleave so that only one has the flag at any one time. However, if you look at your CPU utilisation in the Task Manager, you will get something similar to figure 2.7.

89% CPU utilisation for an application that essentially does nothing. This is because we are using what is known as a spin lock, or busy wait, approach. Putting a thread to sleep does have an overhead, and so a busy wait might be more efficient (particularly for real-time applications). However, it comes with

a CPU overhead.

2.9 -Futures

The final concurrency construct we will look at is futures. They are a great way of starting some work, going of to do something else, and then retrieving the result when we are ready. This makes futures a useful method for background processing or for spinning off tasks (such as when we build GUI applications).

#include <future>

There are a number of ways to create a future, but for our purposes we are going to use the async function:
auto f = async(find_max);

This will create a future using the given function (in this case find_max). We can also pass in parameters as we do with thread creation.

To access the result of a future we use
auto result = f.get();

WARNING
You can only get the result of a future once, so you want to store it when you do. Calling get() more than once throws an exception.

-Example Future Application

We will create a very simple example of using futures. Our application will attempt to find the maximum value in a vector of random values by splitting the work across the CPU. To do this, we get the supported hardware concurrency (n) and create n-1 futures. The main application will execute on the eft over hardware thread.

Think of it this way: If we have 4 hardware threads and are searching a vector with 16 values, we get the following configuration:

1. Future 1 searches elements 0 to 3
2. Future 2 searches elements 4 to 7
3. Future 3 searches elements 8 to 11
4. Main thread searches elements 12 to 15

Our find_max function should be straightforward to understand. Our main application simply creates our futures and gets the maximum from each, displaying the overall maximum.

Line 18 is where we create our futures, and line 26 where we get the result. The structure of the application is similar to how we have been creating threads, although now we also perform some work in the main application.

2.10 -Fractals

To provide a better understanding of futures, we are going to introduce another common problem that can be parallelised - the generation of fractals. A fractal (from a simple POV) can be used to generate an image that is self-similar at different scales. That is, a part of the image can be zoomed in upon and the image still looks the same. Zooming further the image still looks the same, and so on. The actual definition is more complicated than this, but for our purposes we are using a function that will provide us with the values we need at (x, y) coordinates of an image to produce a fractal image. This week, we are going to generate the Mandelbrot fractal.

-Mandelbrot

The Mandelbrot set (which defines the Mandelbrot fractal) is perhaps the most famous fractal. The Mandelbrot set can be determined on a per pixel basis. This means that we can distribute the pixels to separate threads and therefore parallelise the algorithm. We can also scale the problem by increasing the resolution of the image. This makes Mandelbrot fractal generation ideal for parallel work.

For our approach we are going to split the Mandelbrot image into strips.

-Mandelbrot Algorithm

There are a few methods to calculate the Mandelbrot set - we are going to use one that relies on the escape value of a loop. That is, based on the number of iterations we perform up to a maximum provides us with the pixel value (0.0 to 1.0). We first need some global values for our application.

Our algorithm uses these values to determine our Mandelbrot value. The interesting part is the while loop starting on line 20 of listing 2.35.

The Mandelbrot set works on real numbers in the range [-2.1, 1.0] on the x dimension and [-1.3, 1.3] on the y dimension. Each individual pixel is converted to be within this range (the x and y values in the algorithm). Based on the x and y value, the loop runs up to max_iterations times. We then use the number of iterations to determine the escape factor of the loop (line 28) to determine the individual pixel value, pushing this onto the vector.

To store the results, our main application must create a number of futures, and then gather the results. We will create num_threads futures in this example.

Again, this follows our standard approach to spreading work across our CPU. This application will take a bit of time to complete, so be patient. To check your result, you'll need to save the image (see the exercises)

Unit 3 -OpenMP

OpenMP is an API that supports shared-memory parallel programming, so can

enable us to work with our CPU as a multi-core device.

OpenMP provides some different constructs to normal C++11 concurrency, and is itself quite a mature platform. We will look at some example applications using OpenMP that investigates these constructs before doing some analysis work.

REMEMBER: Before building an OpenMP application, you need to make sure that your compiler supports it.
-Project -> Properties -> C/C++ -> Language ->Open MP Support -> Yes(/openmp)

3.1 -First OpenMP Application

#include <omp.h>

Hello is our operation we are running multiple times. The #pragma statement shows is how we do this.

We are using a pre-processor to tell the compiler that OpenMP code should be generated here. We are running the operation in parallel, and using num_threads to tell OpenMP how many copies to run.

As we are not controlling the output, there is some conflict with the threads trying to output at the same time.

3.2 -Parallel For

OpenMP tries to extract away from the idea of having threads. They still exist but they are hidden from the application developer. One of the powerful constructs OpenMP provides is parallel for. This allows us to create threads by executing a for loop. Each iteration uses a thread to compute thus providing a speedup. You have to think a little when using parallel for but it can be useful.

3.2.1 -Calculating π (not using Monte Carlo Simulation)

Using Monte Carlo simulation to calculate π is great for testing performance and speedup, but is not really the most efficient method of calculating π. A better method to approximate π is using the following formula.

$$4\sum_{k=0}^{\infty}\frac{-1^{k}}{(2k+1)} = 4(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots)$$

We are going to use this method in a parallel for to see how we can calculate π.

3.2.2 -Parallel For Implementation of π Approximation

We are using quite a bit of new ideas in the pre-processor comment. First of all, the general parallel for looks very similar to a standard parallel but with the keyword ‘for’ added. The two new parts are at the end of the pre-processor.

reduction:
We will be covering reduction in more detail when we look at map-reduce in MPI later in the module. What we are saying here is that addition on the pi variable should be controlled to add all the for loops together.

private:
This indicates that each for loop has a private copy of the factor value. Each for loop can modify the value independently and not cause corruption to another for loop.

3.3 -Bubble Sort

We are now going to diverge for a bit and look at sequential sorting using a bubble sort. The reason we are doing this is because we are going to build a parallel sorting mechanism and then compare the performance. You should hopefully all be familiar with what is meant by a bubble sort by now.

generate_values() will generate a vector full of values using a random engine

bubble_sort is a straight forward algorithm. We bubble up through the values, swapping them as we go to move a value towards the top.

You should be able to implement this algorithm in c++ by now.

Algorithm 2 Bubble Sort Algorithm

```
var values
  begin
    for count := values.size() to 2 step  − 1 do
        for i := 0 to count − 1 step 1 do
            if values[i] > values[i + 1]
              then
                    tmp := values[i]
                    values[i] := values[i + 1]
                    values[i + 1] := tmp
            fi
        od
    od
  end
```

3.3.3 -Main Application

Our main application will time the implementation of bubble_sort using vectors of different sizes.

If you run this application you should be able to produce a graph. Change the y-axis scale to se a log2 scale. This give a nice straight line.

3.4 -Parallel Sort

Notice that for small vector sizes, throwing parallelism at the problem has not given us a performance boost but in fact slowed us down. Granted we are using a slightly different algorithm, but hopefully you can see that the problem set is too small to get any speed up - in fact the setup and control of the OpenMP program is having an effect Once our sort space is large enough we gain performance - 3+ times as much (the CPU is dual core with 4 hardware threads so this seems reasonable).

3.5 -The Trapezoidal Rule

Our next use of OpenMP will look at something called the trapezoidal rule. This technique can be used to approximate the area under a curve.

We select a number of points on the curve and measure their value. We then use this to generate a number of trapezoids. We can then calculate the area of the trapezoids and get an approximate value for the area under the curve. The more points we can use on the curve, the better the result.

The incoming parameters are as follows:

f the function that we are using to generate the curve

start the starting value we will place in the function

end the end value we will place in the function

iterations the number of iterations (or trapezoids) we will generate

p a shared piece of data to store the result

You should be able to follow the algorithm using the comments The new part we have introduced from OpenMP is line 23 - a critical section. A critical section is just a piece of code that only one thread can access at a time - it is controlled by a mutex. We use the critical section to control the adding of the local result to the global result.

[This isn't working cause it's bitching about M_PI so gonna skip for now] 🤷‍♀️

3.6 -Scheduling

Scheduling involves us telling OpenMP how to divide up the work in a parallel for. At the moment, each thread is given a chunk of work in order. For example, if we have 1024 iterations and we have 4 threads, our work is divided as follows:

Thread 1: 0, 4, 8, 12, ...
Thread 2: 1, 5, 9, 13, ...
Thread 3: 2, 6, 10, 14, ...
Thread 4: 3, 7, 11, 15, ...

Using a schedule of 2 allocates work to threads in blocks of 2:

Thread 1: 0, 1, 8, 9, ...
Thread 2: 2, 3, 10, 11, ...
Thread 3: 4, 5, 12, 13, ...
Thread 4: 6, 7, 14, 15, ...

-Main Application

We use the schedule function in the pre-processor argument to control the division of work. Your task here is to manipulate the schedule value and see the effect.

Running the application will output a timing value - test the scheduling value and chart the difference in performance.

3.7 -Concurrency Visualizer

We will end this practical by looking at the concurrency visualizer in visual studio. In Visual Studio 2013, you will need to install the concurrency visualizer via extensions and updates.

To run it, select analyse in the visual studio menu and select concurrency visualizer, then select current project.

Once run, you will get a report. This view looks at the utilization of your process across the processor. If you click on threads, you can view the amount of work a thread does.

One particular interest is the amound of work the threads are doing - shown by the coloured blocks. Notice that in the example 4% of the time was spent on pre-emption. This means that 4% of our application runtime was taken up by the threads being switched. Not too bad, but illustrates the problem with pre-emptive scheduling.

Selecting the cores view allows us to view how our work was divided aongst the cores.

Our threads are being switched across the four logical cores - which will lead to some of the pre-emption. Logical Core 3 looks like it has the best division of work (long chunks of work). Playing around with the schedule clause could improve things.

Unit 4 -CPU Instructions

We are now going very low level, looking at how we can get the CPU to utilise its internal 128-bit registers to calculate values for us more efficiently.

A WORD OF WARNING:
The techniques we are going to use are supported on Intel and AMD based hardware, and some of the approaches are specific to Microsoft's C++ compiler.
VS also turns on some optimisations by default - you will need to disable them to see a difference in performance (VS enables some automatic usage of the techniques)

TO DISABLE:
Project -> Properties -> C++ -> Optimization -> Optimization -> Disabled (/Od)

4.1 -Memory Alignment

Our first application illustrates how we can align memory correctly in C++ (MS C++).

We declare a new type of variable - __m128.

This means that the value takes up 128-bits of memory. Notice the declaration from the start.
This is used to ensure that the data is aligned in memory in a 16 byte block - or 128 bits.

We can see the 128-bit value as a collection of floats - this can be useful, although we will just treat it as a pointer to float data.

Another method is the _aligned_malloc function

We can also use this technique to create an array of memory aligned 128-bit values. The SIZE value can be set to any number for the size of our array.

Finally, any alloced memory must be freed using the _align_free function.

Running this application will not do much - just print out a couple of values. The point is to introduce the methods. We are going quite low level here, so are working at a C-based level.

4.2 -SIMD Operations

Allocating memory is just one step of our work with the processor. The next stage is to use SIMD based operations. We will only use a couple of these for illustration purposes.

Our first test application will simply add 4 to a collection of floating point values. We will create an array of floating point values, assign them the value of 1.0f and then add 4.0f to them. We will do this using a SIMD operation which will allow us to add 4 floats at a time (128-bits).

We use a particular method to add two 4-value vectors at once:
_mm_add_ps

There are instructions for adding, multiplying, etc.

As a comparison, we do the same operation using standard floats. You will need to ensure that optimisation is off when you run the application to get a result similar to that in figure 4.1.

We are getting a more than 4 times speedup using this approach. this is the power of SIMD operations - we can use them to speed up all sorts of applications.

4.3 -Normalizing a Vector

We are now going to use a SIMD approach to normalizing some 4D vectors -

the same technique could be used for 3D vectors easily enough. To do this we are going to use SIMD instructions to parallelise the process.

Algorithm 3 shows the pseudocode, AKA I haven't done it, cba.

You will notice that the values do not quite match but the results are close enough.

Unit 5 -Distributed Parallelism with MPI

For the next two practicals it might be useful to work with a partner so you can get work on the distributed work we are undertaking. There is quite a bit of setup to do in this practical, so take your time and ensure everything is done correctly.

5.1 -Installing MPI

We are going to use Microsoft's HOC SDK to support our MPI work. To do this you need the following three items installed:
-Microsoft HPC Pack 2012 Client Components
-Microsoft HPC Pack 2012 SDK
-Microsoft HPC Pack 2012 MS-MPI Redistributable Pack

You will need to install these on every machine you plan to use in your application. You will also need to add the relevant include and library folders to your project - these will be found in the Program Files folder. The library that you need to link against is called:

msmpi.lib

5.2 -First MPI Application

Our first application will just initialise MPI, display some local information, and then shutdown.

The methods of interest are:
MPI_Init (initialises MPI)
MPI_Comm_size (gets the number of processes in the application)
MPI_Comm_rank (gets the ID of this process)
MPI_Finalize (shuts down MPI)

At the moment you should just build this application - running an MPI application takes a bit more work.

5.3 -Running an MPI Application

You will need to open a command prompt in the directory where your built application is. Once you have done this, you can run the following command to execute the application locally in parallel:

mpiexec /np 4 “exe_name.exe”

/np denotes the number of processes to use (here I use 4 - the number of logical cores on my machine).

5.4 -Using a Remote Host

Running an MPI application in this method is all well and good, but we are not really doing any distributed parallelism. What we want to do is use one or more remote machines to do our processing.

First you will need to find the IPv4 address of the machine you want to use as the remote node. We do this using the command:
ipconfig

Next you want to run the following command on the remote machine:
smpd -d

This is called a Single-Program Multiple-Data task. It will listen on the machine and wait for us to allocate a job. To do this, we run mpiexec with a few more commands:
mpiexec /np 4 /host <ip-address> <application>

We now tell MPI which host to run on (we can also define multiple hosts).

You will also need to copy the application to the other machine.

5.5 -Sending and Receiving

The next few short examples will look at the different methods for communication. First of all we will use the standard send and receive messages.

Here we are using the process rank to determine which process does what. The process with rank 0 we consider the main process, and its job is to receive messages.
Each other process will just send a message to the main process.

Here we are using two new commands:
-MPI_Send
requires the data to be sent, the size of data (we make sure we send an extra byte for a string (the null terminator)), the type of data, the destination (0, the main process), a tag (we will not be using tags), and the communicator.
-MPI_Recv
requires a buffer to store the message, the maximum size of the buffer, the process to receive from, the tag, the communicator to use, and status conditions

There are a number of different data types MPI can use beyond MPI_CHAR - the Introduction to Parallel Programming book will explain these further.

5.6 -Map Reduce

Another approach to communication we can use is map-reduce. For this you will have to use a Monte-Carlo π application.

MPI_Reduce takes the following values:
-The value to send
-The value to reduce into
_The number of elements in the send buffer
-The type of the send buffer
-The reduction operation - we are using MPI_SUM to sum
-The rank of the process that collects the reduction operation - we use rank 0 as the main process
-The communicator used

Notice That we only use our main application to calculate π.

5.7 -Scatter-Gather

Scatter-gather involves taking an array of data and distributing it evenly amongst the processes.

Gathering involves gathering the results back again at the end.

For scatter-gather we are going to implement our vector normalization application.

The MPI_Scatter command has the following parameters:
-The data to scatter - only relevant on the root process
-The count of data to send to each process
-The type of data sent
-The memory to receiver the data into on each process
-The count of data to receiver at each process
-The type of data received
-The root process (sender)
-The communicator used

MPI_Gather is essentially the reverse:
-The local data to send
-The count of data to send from each process
-The type of data sent
-The memory to gather results into
-The count of data to receieve from each process
-The type of data received
-The root process

-The communicator used

5.8 -Broadcast

The final communication type we will look at is broadcast. Broadcasting just allows us to send a message from one source to all processes on the cummunicator.

MPI_Bcast takes the following parameters:
-Data to broadcast (sender sends, receiver reads into this data)
-Count of data to send / receive
-Data type sent / received
-Root node
-Communicator

Unit 6 -More MPI

Our second tutorial on MPI will focus on particular examples of using MPI from our previous work. We will just focus on the examples rather than go into any new MPI as such.

6.1 -Mandelbrot

For Mandelbrot, our task is quite easy - the implementation we had was designed to run using rank based execution. We just need to modify it to use MPI ranks.

We use gather here to gather the final results. This is the point where the results are gathered back to the host machine.

6.2 -Parallel Sort

For parallel sort we have to do quite a lot more work to get things to work.

The parallel sort we used was something called an odd-even sort. This works in a number of phases that requires sharing between processes. The general algorithm is as follows:

1. Sort local data
2. For number of phases
   (a) Exchange data with phase partner
   (b) Merge
3. Gather results

Because of this we have a number of new operations we have to define.

First, let us define two merge methods - one to merge at the top of the list and one at the bottom.
We call these merges during an odd-even iteration where we execute data between partners.

Depending on the phase and whether we have a partner to the left or right, we exchange data accordingly. We then merge data with our results, resulting in us having either the sorted upper portion of the two processes data or the sorted lower portion.
Eventually each process will end up with its relevant sorted portion which we can send back to the main process.

The sort method merely sorts the local data section before performing the necessary number of phases (which is equal to the number of processes involved).

All we need is to call the sort method after scattering out the data.
We then gather the data at the end.

6.3 -Trapezoidal Rule

For the trapezoidal rule we will use a barrier to synchronise our work.

6.4 -Performance Evaluation of MPI

We will now look at how we can measure latency and bandwidth using MPI.

These values can be useful if you are undertaking any serious distribution of tasks and data communication. However, you will likely find that the stated network speed is what we hit.

6.4.1 -Measuring Latency

For latency all you need is in the application in Listing 6.7

6.4.2 -Measuring Bandwidth

For bandwidth you just need to change the latency application so that it has bigger data sizes.

Use powers of two as normal, and range from approximately 1K to 1MB.
You will have to convert from the time taken to send the message to the actual MBit/s
You should also measure broadcast to see the performance there as well.

REMEMBER:
Create the charts and look at performance.
You should be able to predict the performance of an application purely by sequential computation time plus communication time.

Unit 7 -GPU Programming with OpenCL

We are now going to move onto programming the GPU to perform data parallel processing.

Using the GPU in this manner is a relatively new concept. However, the principals are essentially the same as shader programming.

Before getting started you will need to make sure you have the relevant SDK installed on the machine you are using.
This will depend on the hardware you are using.
Intel, Nvidia and AMD each provide an SDK (the Intel and AMD ones also support the CPU as an OpenCL device).

You will have to work out the setup of your OpenCL projects in Visual Studio.

The header we will be using is:
CL/cl.h

There is a C++ header (cl.hpp) in some installations but not all - so we will work at C level rather than C++.

The library is:
OpenCL.lib

7.1 -Getting Started with OpenCL

Our first application is purely about setting up OpenCL.

The first thing we do is get the number of platforms and use this to resize a vector to store the platform information.
A platform in OpenCL is a different OpenCL runtime - for example your machine could have both Intel and Nvidia platforms.

Next we get the devices supported by Platform 0 - this assumes that Platform 0 is you GPU platform so you might have to modify this code.
As we also want to only work with the GPI we use the device type:
CL_DEVICE_TYPE_GPU.

You can use the following device types:

-CL_DEVICE_TYPE_CPU	as a CPU device
-CL_DEVICE_TYPE_GPU	as a GPU device
-CL_DEVICE_TYPE_ACCELERATOR	a custom OpenCL device
-CL_DEVICE_TYPE_DEFAULT	the default OpenCL device
-CL_DEVICE_TYPE_ALL	all devices on the platform

The final two steps are the creation of a cl_context (allows creation of command queues, kernels, and memory) and a cl_command_queue (allows sending of commands to the OpeCL device).

All we are doing at the moment is calling the initialise method and then cleaning up the resources.

Running this application will not do anything, but it will allow you to check your OpenCL setup seems to be working.

7.2 -Getting OpenCL Info

Our next application will print out the information for our OpenCL devices.
To do this, we just need to grab the info using some of the OpenCL functions - again it is much like doing so from OpenGL.

This is fairly straightforward and you should know how to update your main application to use it.

Running this will give you the information about your OpenCL devices.

Modifying this to get all the devices for the platform might change your output.

If you are using AMD or Intel hardware, you can also retrieve the GPU values.

[Check workbook for Kevin's hardware stuff]

7.3 -Loading an OpenCL Kernel

We are now going to look at how we have a basic method of setting up OpenCL and displaying the information we require.

This involves us loading what are called kernels.
You can think of this a little bit like loading a shader, but we are doing less specialised programming and more general purpose (hence the name General Purpose GPU programming)

The kernel we are using is given in Listing 7.4.
You should save this in a file called:
kernel.cl

We will look at this code in a bit of detail. First, if our function is a kernel we use the keyword
__kernel
at the start of the declaration.
Kernels do not return values, so our return value is void.

The parameters for our kernel all are declared as __global.
This means that they are accessible to all the cores when the kernel executes - it is global memory.
We declare these as pointers as they will be blocks of memory that we will be accessing and writing to.

Our kernel is adding two vectors - or two arrays of a particular size (we will add two 2048 element vectors together). The

get_global_id
function allows us to get the index of the current executing thread.
A thread can have various dimensions for the index - so we can get the id for 0, 1, 2, etc.
We can also get the local id for work groups.

As our kernel adds two ID vectors, we only need to use the 0 dimension.
The final line of the kernel just stores the value.

To load a kernel, we use Listing 7.5
This method works much how we would load a shader.
We read in the file contents and then create a
cl_program
object and build it. If the build fails, we print out the build log.

Now that we are loading a program, we need to choose which kernel to use.
We can do this by updating the main function.

We load the program, passing in the name of our file.
We then select the kernel we want to use - notice as well that we are now releasing our kernel and program.

Running this application should still print out your OpenCL device properties - however it should not print out an error log.

7.4 -Passing Data to OpenCL Kernels

We are now ready to send data to our OpenCL kernel.
This involves us creating memory buffers on the GPU and coping the data to the GPU.

The first thing we need to do is create the “host” memory - that is the memory that sits in main memory accessible by the CPU.

For our application we will create two arrays of 2048 elements.

Next, we need to create our buffers on the OpenCL device.
Notice that we have to tell OpenCL the type of buffer we are creating (read, write, etc.) and the size of the buffer.
This is just creating a buffer on the GPU - in our instance it is allocating memory.

All we need to do is copy our host data to our device buffers.

We use
clEnqueueWriteBuffer
to write our data to the relevant buffers.

The third parameter is an interesting one as it tells OpenCL whether the application should wait for the write to complete.
Typically, we probably do not worry about a write completing - but we normally do for reading.

The final stage is setting the kernel arguments.
Remember our arguments for the kernel are as follows:
A - this is an input vector
B - this is an input vector
C - this is an output vector

We can set the kernel arguments as shown in Listing 7.10

We need the
cl_kernel
object, the parameter (0 indexed), the size of the parameter, and a pointer to the parameter value.

We also need to make sure we free our buffers. We do this at the end of the main application.

Running this application still will not do anything - we have not executed the kernel and got our results back.

7.5 -Running and Getting Results from OpenCL Kernels

We are finally ready to run our OpenCL kernel. To do this we need to define a new piece of information - the dimensions of the work.
In our instance we have a single dimension arrays with elements items.
We can therefore set up our work size as shown in listing 7.12

We can now run our kernel, as shown in Listing 7.13

At the moment we are only interested in some of these parameters. These are:

cmd_queue	the command queue we are using to enqueue the work
kernel	the kernel we are executing
1	the number of dimensions for the work
global_work_size.data()	the number of elements per dimension

We will look at some of the other parameters as we work through the rest of the module.

Finally we need to copy our data back from the kernel at the end.
Again, we are only interested in a few parameters:

cmd_queue	the command queue

buffer_C	the buffer we are reading from
CL_TRUE	we will wait for the read to complete
O	the offset of the buffer to read from
data_size	the amount of data to read from the buffer
C.data()	pointer to the host memory to copy the device buffer to

Now all we want to do is validate that the data read back is correct.

We can break down our application as follows:

1. Initialise
2. Load kernel
3. Create host memory
4. Create device memory and copy host memory to device
5. Set kernel arguments
6. Run kernel - remember to set the work dimensions
7. Copy device memory back to host to get the results
8. Clean up resources

This will be our quite standard approach to working with OpenCL.
We might iterate through some of these stages (running kernels and setting / getting results), but typically the process is the same.

7.6 -Matrix Multiplication

This is more of an exercise than a straight tutorial.

Enough said Kevzo, GG brah no hapnin

Unit 8 -Programming with CUDA

OpenCL is the equivalent of OpenGL for GPU programming.
It runs on pretty much all hardware and has tools provided by a number of vendors.
This generality can come at a cost of simplicity and performance.

CUDA can be considered the DirectX of GPU programming.
It is provided by Nvidia to run on Nvidia hardware. Tools are from Nvidia.
However, there is a simplicity and performance benefit because of this.

You will find the general ideas are the same, however our setup is a bit different.

8.1 -Getting Started with CUDA

If you have the CUDA SDK installed, you should have Nvidia Nsight added to Visual Studio.
This being the case, you will be able to create a new CUDA application by

selecting it during the project creation (under NVIDIA in the templates).

Visual studio will provide an initial kernel.
You should delete this code and start from an empty file.

First we need to write an application to initialise CUDA (figure 8.1)
And that's it. Much simpler than OpenCL.

We only need to select a device. If your machine only has one Nvidia device, this is just 0.
You can run this application to test it just to ensure that everything is set up properly.

8.2 -Getting CUDA Info

Getting information from CUDA is also fairly trivial (figure 8.2).

You will need to call the operation
cuda_info
from the main application.

This is the same information achieved from calling OpenCL.

[Check workbook for more theory stuff]

8.3 -CUDA Kernels

One of the main differences CUDA provides from OpenCL is that we are using a single file solution.
That means that we will write our CUDA kernel in the same file as our main method.
This keeps us closer to standard C++ development, and we do not need to load and compile external files.

We will get to how we launch the kernel soon.
First let us look at memory management between the CPU and GPU.

8.4 -Passing Data to CUDA

As with OpenCL, we have to work between host memory (main memory) and device memory (GPU memory).
If you remember with OpenCL we first declared and initialised some host memory. We will do the same this time.

We also need to initialise memory on our device.
Notice that we don't need any special types with CUDA. We simply declare an int pointer as standard.

The only difference is in how we allocate memory.

You may be familiar with malloc from standard C.
cudaMalloc
undertakes the same functionality but for allocating memory on the GPU.

All we need to do now is copy the memory from the host to the device. We do this using the cudaMemcpy operation:
cudaMemcpy(dest, src, size, direction);

The direction value is used to tell CUDA which way the data is being copied (host to device, device to host, device to device).

8.5 -Running and Getting Results from CUDA

Now we just need to run the kernel.

Notice that it looks very similar to running the operation normally.
The only difference is that we are defining the number of blocks (ELEMENTS / 1024) and the number of threads per block (1024).

The call to
cudaDeviceSynchronize
means that we wait for the kernel to complete executing before continuing.

To get our results back, we simply call
cudaMemcpy
again.

You should write the code to check that the input is correct. It is the same code as in the OpenCL example.

8.6 -Freeing Resources

The last thing our application has to do is free any resources used by CUDA.
As we only have allocated memory, this requires only a few calls.

Running this application should give you the same output as that in the OpenCL version.

8.7 -Matrix Multiplication

As with OpenCL, your task here is to write and test the application required to multiply two matrices using CUDA.

[naw]

Unit 9 -Programming Examples

In this unit we are going to look at some GPU samples.
We will look at OpenCL and CUDA examples where possible.

The general approaches are the same.

We will start by looking at some of the problems we have already looked at, before moving onto some other ideas. Most of these examples will only be briefly discussed, as you should be able to implement the necessary applications by now.

You should also be taking timings to compare performance with sequential, multi-threaded, MPI, and other solutions.

9.1 -Monte Carlo π

Monte Carlo π was our first problem examined, so you should have a good understanding of the principle by now.

We will look at one OpenCL and a number of CUDA solutions.
OpenCL can handle most of the CUDA approaches as well, so you can implement these too if you wish [lol okay Kevin]

9.1.1 -OpenCL Monte Carlo π

For this application you will have to generate some random values and pass them to the GPU.
OpenCL comes with a type that can be used to represent a 2D position in space - float2.
This is the type used in the kernel.

For our main application, the type is cl_float2. The general approach we are taking is as follows:

1. Generate random points on CPU - one point for each thread
2. Allocate memory for points and a result value (0 or 1) for each thread
3. Copy points to GPU
4. Run kernel
5. Get results back from GPU
6. Sum results
7. Calculate π

You are only provided with the kernel. It is up to you to implement the necessary main application.

9.1.2 -CUDA Monte Carlo π

For CUDA, we will look at a few different approaches which allow us to examine potential performance gains.
We will implement the same solution as OpenCL first, then we will look at a solution using a for loop within a kernel, generating random values using CUDA, and finally summing on the GPU rather than the CPU.

STANDARD APPROACH
Our standard approach follows the same method we undertook for OpencL. Therefore, the kernel is just provided.
Notice that CUDA does not provide a length function like OpenCL, so we have to calculate the length manually.

USING A FOR LOOP IN THE KERNEL
Our next implementation is going to move towards our CPU implementation of Monte Carlo π.
We do this by undertaking a number of iterations on each thread rather than a thread calculating a single point.

For example, if we consider a solution where we have 2^24 points to check, we need to run 2^24 threads and copy back 2^24 bytes (16Mbytes) of data to main memory to sum the result.

If we allow each thread to calculate a number of iterations, we increase the amount of work a single thread does, while reducing the number of threads run and decreasing the memory usage.

For example, if we have 2^24 points to calculate and let each thread run 2^16 iterations (so each thread calculates 2^16 points), then we only run 2^8 threads, and only copy back 2^8 x 4 (we need an int now) bytes (1024 bytes or 0.001 MBytes).

The same approach is required as before, except you have to consider the amount of data required to copy back and the number of threads against the iterations per thread. You should experiment with different configurations to explore the different performance characteristics.
Also try different thread to block ratios to further analyse performance. You will be surprised how some tweaks can alter performance.

USING CUDA TO GENERATE RANDOM POINTS
We can also get CUDA to generate our random values for us directly on the GPU.
To do this we will need to use
cuRand (http://docs.nvidia.com/cuda/curand/)

You will need to read a little bit about how to use cuRand, but example code for generating random values (in your main application, not the kernel) is in listing 9.4.

Note that this approach generates random values across 2 dimensions (since we have 2D values).
Other approaches could also be used.

Read more on cuRand to find out.

You can use the same kernel as previously. All you are changing is how you are generating random values.

SUMMING ON THE KERNEL
Our final approach to Monte Carlo π involves us performing the final stage of the computation (the calculation of π) on the GPU.
To do this, we will use a new function within our kernel:
__syncthreads

This function allows us to stop all threads within a block at a particular point in our code.
Note that only threads in the same block will synchronise.
This means that if you have more than one block, not all threads will synchronise.

For our implementation to work then you can only have one block.

Note that you only have to copy back one value - the float containing π.

Our host actually needs no buffers of data allocated if you are generating random values using CUDA as well.
All data is just allocated on the GPU.

9.2 -Mandelbrot Fractal

The mandelbrot fractal we introduced when working with futures used a particular technique called the Escape Time Algorithm.
We can also implement this on the GPU.

9.2.1 -OpenCL Mandelbrot

Our kernel this time involves loops and branching statements - which might be an issue for performance.

Our main application just needs to declare our data that we read back into and use for our image.

SAVING AN IMAGE USING FreeImage
You can save the data generated by the Mandelbrot kernel by using the code given in listing 9.8.
You will have to download and setup FreeImage to achieve this.

9.2.2 -Exercise

You should be able to convert the OpenCL version of Mandelbrot to a CUDA version fairly trivially.
Do this now to ensure you understand the basic concepts behind making a

CUDA application.

9.3 -Trapezoidal Rule

Our next example returns to the trapezoidal rule application we looked at with OpenMP.
This time, we will use CUDA.

Remember that the trapezoidal rule works by calling a particular function that we sample based on a value based on the thread ID.
To do this, we need to introduce another idea for CUDA - device callable functions.

Up until now with CUDA we have defined functons using
__global__.

This defines code that runs on the GPU and is callable from our main application. For code that runs on the GPU and is callable from other GPU code we use
__device__.

You will need to write the rest of the CUDA code yourself, but you should be able to do this easily enough.

9.3.1 -Exercise

This time write the equivalent in OpenCL.
This again should be fairly trivial.

9.4 -Image Rotation

Image rotation involves us working on a large buffer of data that represents our image, creating another large buffer of the same size, and setting the values in this buffer based on the following calculation:

$$destination(x, y) = source(x' \times \cos\theta + y' \times \sin\theta + x', y' \times \cos\theta - x' \times \sin\theta + y')$$

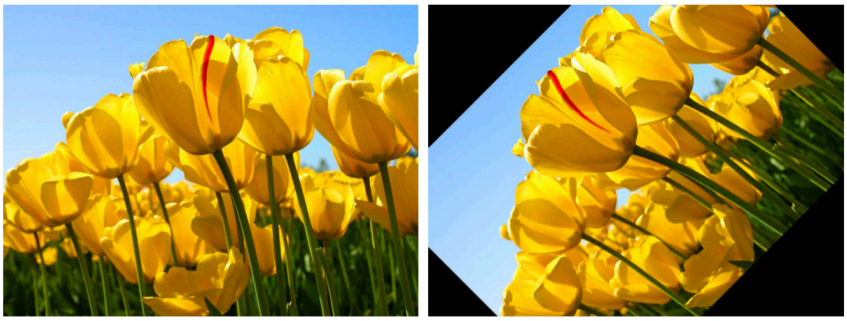
where

$$x' = x - \frac{width}{2}$$

$$y' = y - \frac{height}{2}$$

$\theta$is the angle of rotation

An example of the output from the application is shown in Figure 9.1.



(a) Original Image (b) Rotated Image

In OpenCL our kernel is provided. It just calculates the pixel colour accordingly.

To use this kernel we need to load in the image.
Here we use FreeImage as shown in Listing 9.11

Finally, the rest of our main code just sets up the various buffers and parameters for the kernel to execute.

9.4.1 -Exercises

1. We have used standard buffers with OpenCL here. However, OpenCL does support texture buffer objects. Investigate these and try to modify the application to utilise these concepts.
2. Implement the image rotation example using CUDA and textures. You will have to do some research into this since we are using a number of new concepts.

9.5 -Profiling and Debugging

There are a number of different tools you can use to analyse GPU performance.

Here are a few links to get started:
-AMD CodeXL (OpenCL profiling on AMD hardware)
[http://developer.amd.com/wordpress/media/2012/10/CodeXL_Quick_Start_Guide.pdf]
-Nvidia Nsight (CUDA and OpenCL profiling on Nvidia hardware)
[http://developer.download.nvidia.com/ParallelNsight/2.1/Documentation/

UserGuide/HTML/Content/Using_CUDA_Debugger.htm]
-Intel Graphics Performance Analyzer (Intel profiling)
[https://software.intel.com/sites/default/files/m/d/4/1/d/8/Collecting_Important_OpenCL-related_metrics_with_GPA.pdf]

Unit 10 -C++ AMP

Here we focus on another GPU programming technology. C++ AMP is a technology developed by Microsoft to support GPU programming in standard C++ code.
It is an open standard, although only Microsoft compilers really support it at present.

The main difference with C++ AMP is that we are only going to write C++ code. We will compile it as standard with no additional libraries and runtimes.
We achieve this using function objects and λ-expressions

10.1 -Getting C++ AMP Information

```
#include <amp.h>
```

That is all we need to start working with C++ AMP. There are other headers we can add, but to get a basic application started we just need the header file.

Listing 10.1 provides the code for getting information from C++ AMP.
Notice that there isn't much we can get.

The accelerator variable allows us to access details on the current device.
Notice as well that we use wcout rather than cout to print details.
wcout prints wide character (16-bit char) strings.

The accelerator type and other C++ AMP operations and data types are part of the concurrency namespace.
We will use this operation in our main code.

10.2 -Vector Addition Using C++ AMP

Using C++ AMP, our vector addition example becomes very simple. This is due to our GPU code being written in our standard C++ code.
We still need to go through the main steps when working with the GPU, but things are far simpler.

10.2.1 -Declaring Host Memory

For our vector addition example, declaring host memory is no different than in our other GPU examples.
Note that we have to state we are using the array type from the standard library (std::).

This is because C++ AMP also declares an array type (just to confuse things)

10.2.2 -Creating Device Memory

Device memory setup in C++ AMP is much easier. We just need to create an array_view object that contains our host memory.
The copying and the allocation is handled in this data type.

The second value of the template (the number) provides the dimensions of the data.
In our instance we only have one - a single vector.
We can create different configurations as standard GPU code.

10.2.3 -Vector Addition Function in C++ AMP

Our kernel code is shown in listing 10.4.
Notice the similarity to our other approaches, although now we use a function object.

There are a few parts that need explaining:
-[=]
means that the function should automatically copy the required variables into the function scope. This means that we don't have to pass any parameters to call the function. They are automatically copied across.
-index<l>
provides us with the id of the particular thread or work item. Here, we only have one dimension of work, but we could have more
-restrict(amp)
is an instruction to the compiler to ensure that the defined function meets the requirements of C++ AMP code.

10.2.4 -Running the vecadd Function

To actually run the function we use the code in 10.5

And that is it. We run the function, giving it the work dimensions (the buffer_C.extent part) and the function to run.
We then wait on buffer_C to fill

10.2.5 -Complete Main for C++ AMP vecadd Application

The complete C++ AMP main code is shown in listing 10.6.
It is short but performs the same operation as our previous two examples.

10.3 -Matrix Multiplication Using C++ AMP

[GG brah the rest of the workbook is just unfinished / code listings]