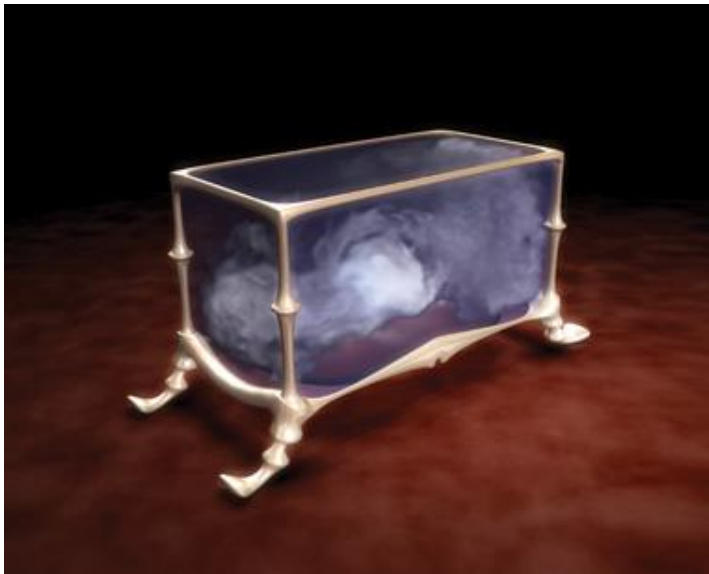


# GPU Tutorial 2: Integrating GPU Computing into a Project



# New Concepts

- ▶ Integrating CUDA with our existing code base
- ▶ Structuring data for execution on the GPU
- ▶ Sorting data for execution on the GPU
- ▶ Numerical Integration on the GPU
- ▶ API features
  - ▶ Variable declarations
  - ▶ Host Functions
  - ▶ Kernel Functions
  - ▶ Device Functions

# Integrating CUDA into our Existing Codebase

# First, BACK EVERYTHING UP

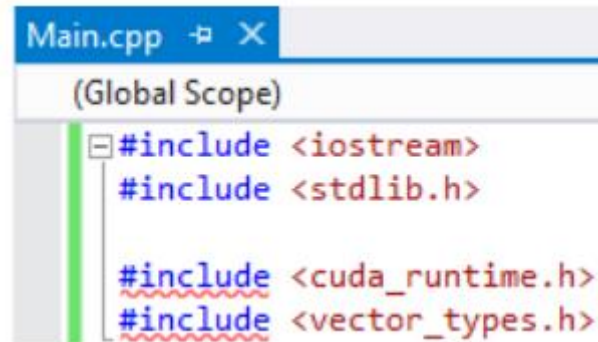
- ▶ Throughout the course, the concept of backing up your work before undertaking significant changes of direction has been suggested
- ▶ From now, and for the rest of the degree programme, managing your software is essential
- ▶ If you intend to integrate CUDA into your coursework, great! Just make sure you've got a version stored of what you had before you made the attempt

# So, what challenges are there to integration?

- ▶ Assuming you've downloaded the CUDA Toolkit and Nsight plug-in, very few
- ▶ Nsight attaches itself to Visual Studio, and uses its own compiler (nvcc) to process CUDA code into machine instructions for the GPU
- ▶ This should work seamlessly alongside Visual Studio's C++ compiler, which handles the remainder of compilation
- ▶ But we need to make sure Visual Studio knows this is meant to happen before we can get started with GPU programming

# Adding CUDA Runtime Libraries to an Existing C++ Project

- ▶ If we're going to write CUDA functions, we need to include the CUDA libraries
- ▶ `cuda_runtime.h` and `vector_types.h` are two common examples
- ▶ Problem with just using `#include` for CUDA libraries is that it normally doesn't work out the box

A screenshot of a code editor window titled 'Main.cpp'. The editor shows the following code:

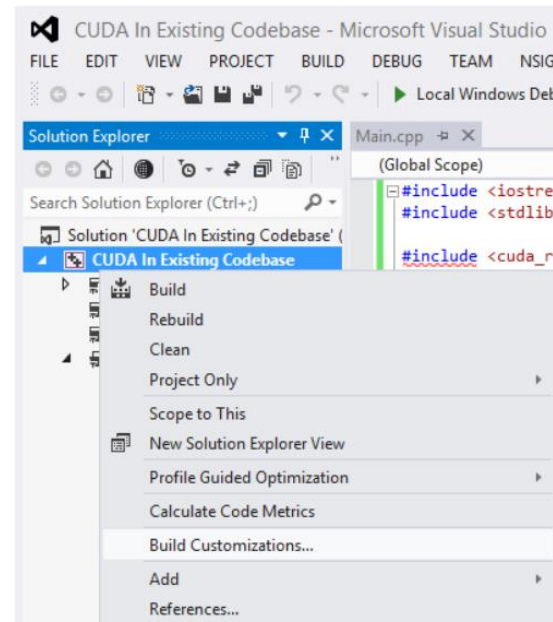
```
(Global Scope)
#include <iostream>
#include <stdlib.h>

#include <cuda_runtime.h>
#include <vector_types.h>
```

The `#include` statements for `<cuda_runtime.h>` and `<vector_types.h>` are underlined with red wavy lines, indicating compilation errors. The editor has a blue header bar with the filename 'Main.cpp' and standard window controls.

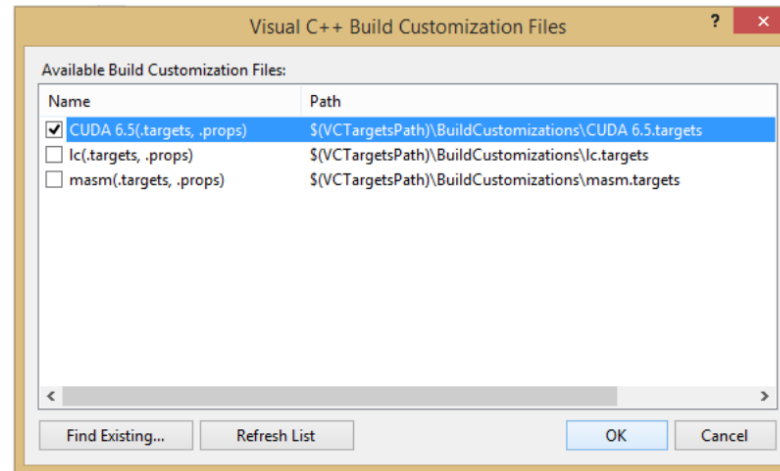
# Adding CUDA Runtime Libraries to an Existing C++ Project

- ▶ This is because even with Nsight installed, Visual Studio doesn't know we intend to use it
- ▶ To fix this, we go to Build Customisations
- ▶ This will give us a list of build customisation files to choose from



# Adding CUDA Runtime Libraries to an Existing C++ Project

- ▶ CUDA should be in there somewhere - multiple versions, if we have multiple versions installed
- ▶ Check the version we intend to use (7.5 in your case)
- ▶ Finally, click Project from the top menu, and rescan the solution. VS2013 should now see the libraries and remove include errors





# Structuring Data for GPU Computation

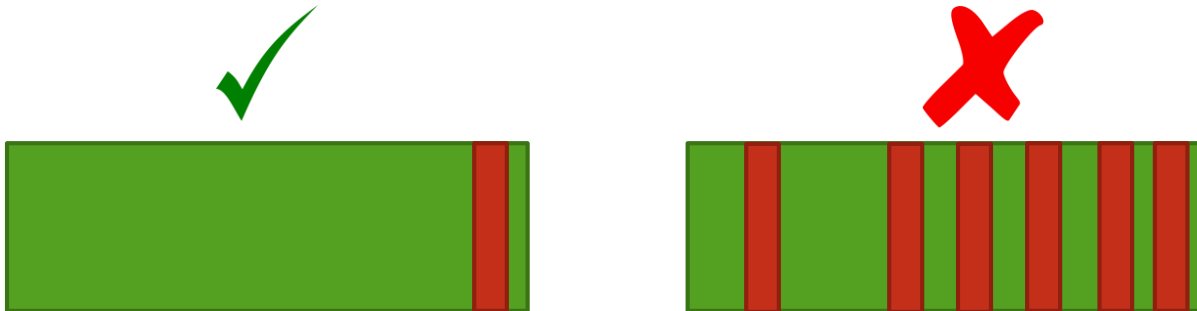


# Ancient Truths

- Fundamental maxim for GPU computing, which still largely holds true today, is:

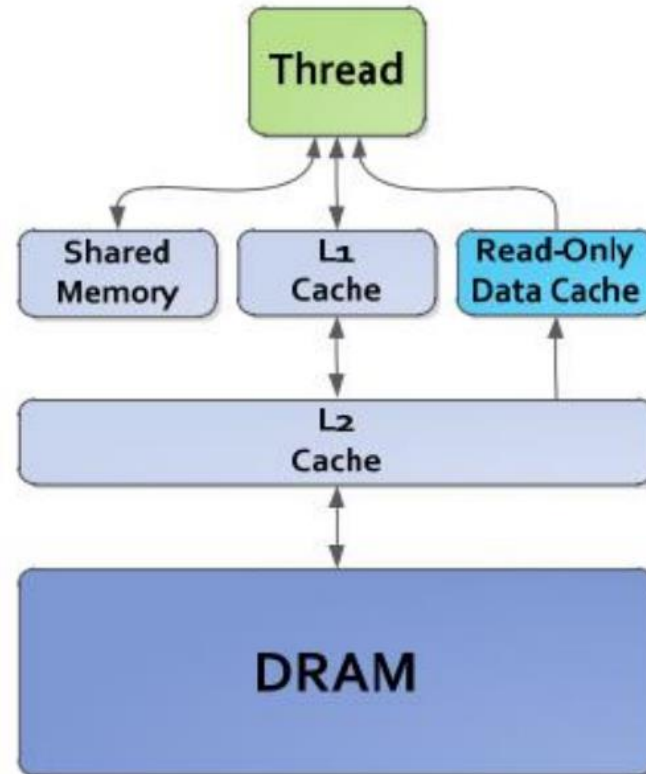
*Struct of Arrays, not Arrays of Structs*

- What this means, really, is that the GPU works at its best when all kernel instances are working on the same contiguous chunk of memory, i.e.:



# Memory Access Hierarchy Revisited

- ▶ Consider the memory access hierarchy for a single thread (kernel instance)
- ▶ A thread has access to all memory in the system, save the shared memory of other blocks
- ▶ But to access any memory not stored in a given level of cache requires it to follow the cache hierarchy for access



# Memory Access: An Example

- ▶ Simple example called TestStruct

```
struct TestStruct {  
    Vector3 a;  
    Vector3 b;  
    Vector3 c;  
} ;
```

- ▶ Contains three Vector3s
- ▶ Consider a kernel which increments the values of those Vector3 elements
- ▶ The kernel has the execution order b, a, c

# Memory Access: An Example

- ▶ Consider an array of TestStructs being sent to the GPU
- ▶ If that array is small enough (entire memory footprint smaller than L1 cache component of a single SMX) no problem
- ▶ But if the problem's that small, probably shouldn't be on the GPU in the first place

```
struct TestStruct {  
    Vector3 a;  
    Vector3 b;  
    Vector3 c;  
} ;
```

```
TestStruct TestStructArray[10];
```

# Memory Access: An Example

- ▶ For a usefully large array, we're now in a situation where for the kernel execution, two thirds of our L2 (and L1) cache data is useless for the first instruction
- ▶ If our data is more complex than a 12 byte class, should be easy to see how we can wind up with cores of SMXs doing nothing
- ▶ Worse, we can potentially create cache miss scenarios if our total data exceeds L2 cache size (again, very possible)

```
struct TestStruct {  
    Vector3 a;  
    Vector3 b;  
    Vector3 c;  
};  
  
const int N = 1000000;  
TestStruct TestStructArray[N];
```

# Memory Access: So what do we do?

- ▶ Restructure the data
- ▶ On the basis that we know what data elements our kernel instances will be accessing, in a deterministic fashion, we can create a struct of Vector3 arrays
- ▶ `b[]` is then accessed first by all cores - all cached data is relevant to the first instruction
- ▶ This allows the GPU to 'warp' the kernel - basically, runtime-optimisation of kernel deployment
- ▶ Branching if-statements complicate things, because access no longer guaranteed contiguous

# Sorting Data for the GPU



# So how does this map to big structures?

- ▶ Let's consider a class PhysicsNode, illustrated right.
- ▶ If we were try to process every element of our physics engine on the GPU, it stands to reason the GPU would need all of this data
- ▶ Bearing in mind that our struct of three Vector3s is 36 bytes in size, this >188 byte structure will cause Bad Things to happen

```
class PhysicsNode {  
public:  
    static const Vector3 gravity;  
  
protected:  
    bool useGravity;  
  
    //<-----LINEAR----->  
    Vector3    m_position;  
    Vector3    m_linearVelocity;  
    Vector3    m_force;  
    float      m_invMass;  
  
    //<-----ANGULAR----->  
    Quaternion m_orientation;  
    Vector3    m_angularVelocity;  
    Vector3    m_torque;  
    Matrix4    m_invInertia;  
  
    SceneNode* target;  
    CollisionVolume* vol;  
};
```

# Sorting vs. Refactoring

- ▶ We have two options, then, to process a data structure of this type
- ▶ First, we could refactor our entire codebase to operate solely in arrays for physical objects
- ▶ Can be costly in development time, and anything which costs development time increases the project budget
- ▶ Second, might be infeasible depending on how other elements of the software are engineered

```
class PhysicsNode {  
public:  
    static const Vector3 gravity;  
  
protected:  
    bool useGravity;  
  
    //<-----LINEAR----->  
    Vector3    m_position;  
    Vector3    m_linearVelocity;  
    Vector3    m_force;  
    float      m_invMass;  
  
    //<-----ANGULAR----->  
    Quaternion m_orientation;  
    Vector3    m_angularVelocity;  
    Vector3    m_torque;  
    Matrix4    m_invInertia;  
  
    SceneNode* target;  
    CollisionVolume* vol;  
};
```

# Sorting vs. Refactoring

- ▶ The alternative is sorting the data prior to the point we memcpy it to the GPU -OR- sorting it after we copy it back -OR- both
- ▶ Sorting operations are costly - in the best case, assuming we're not trying to optimise anything, we're at least going to iterate through every object in our environment just to copy every element of data from the class and insert it into the appropriate array

```
class PhysicsNode {  
public:  
    static const Vector3 gravity;  
  
protected:  
    bool useGravity;  
  
    //<-----LINEAR----->  
    Vector3    m_position;  
    Vector3    m_linearVelocity;  
    Vector3    m_force;  
    float      m_invMass;  
  
    //<-----ANGULAR----->  
    Quaternion m_orientation;  
    Vector3    m_angularVelocity;  
    Vector3    m_torque;  
    Matrix4    m_invInertia;  
  
    SceneNode* target;  
    CollisionVolume* vol;  
};
```

# Sorting vs. Refactoring

- ▶ This becomes even more expensive if we try to subdivide the actual data structures within the Class (i.e. breaking down the Quaternion into its component elements)
- ▶ This all adds into the overhead we talked about previously - and it becomes a balancing act.
- ▶ For everything we do to try and accelerate our GPU kernel, we need to know that acceleration will lead to net performance gain
- ▶ Difficult to know that in PCs, because platform variety

```
class PhysicsNode {  
public:  
    static const Vector3 gravity;  
  
protected:  
    bool useGravity;  
  
    //<-----LINEAR----->  
    Vector3    m_position;  
    Vector3    m_linearVelocity;  
    Vector3    m_force;  
    float      m_invMass;  
  
    //<-----ANGULAR----->  
    Quaternion m_orientation;  
    Vector3    m_angularVelocity;  
    Vector3    m_torque;  
    Matrix4    m_invInertia;  
  
    SceneNode* target;  
    CollisionVolume* vol;  
};
```

# But this just applies to CUDA, right?

- ▶ No.
- ▶ It applies to all GPUs, to a greater or lesser extent
- ▶ In fact, hypothetically, it applies to all architectures - what we're talking about is cache coherency
- ▶ The GPU is just **much more vulnerable** to the problem than modern CPUs

# But this just applies to CUDA, right?

- ▶ Remember, Skylake CPUs have four levels of cache - and the last level is 128MB in size - you just won't ever make data structures big enough for a four-core CPU with that sort of memory architecture to ever encounter these problems
- ▶ By contrast, writing for the GPU is like writing for a thousand particularly dumb ZX81s who can't talk to each other without asking everyone around them to shut up
- ▶ Again, you're largely being taught principles, not an API; while the end of this tutorial does present API information for CUDA, the premises (kernels, functions that execute on the Device, functions that execute on the Host) map to most GPU computing APIs

# Numerical Integration on the GPU

# Let's Consider the Problem

- ▶ Which integration methods, of those you've considered, would function best on the GPU?



# Well, all of them, really

- ▶ Updating position and orientation of our objects - the first stage of our physics update - is an embarrassingly parallel problem
- ▶ Objects don't interact with one another until the subsequent steps
- ▶ All we need know are the forces acting on our individual entities, which we compute either in the collision resolution stage or as a function of environment
- ▶ And we can integrate for each object
- ▶ If there's an interface generated, that's detected over the next two stages of the process (which aren't integration)

# Overhead

- ▶ That, of course, is the problem
- ▶ We need to memcpy the integration result back to the Host
- ▶ Update forces based on collision data
- ▶ Then copy force data back to the GPU for the next update
- ▶ Even more expensive if our system automatically corrects positions for interfacing objects, rather than relying on constraints - means recopying the entire position array

# Not true in a collisionless system, though!

- ▶ In a particle system, there is no collision possibility to consider
- ▶ As such, don't need to copy data back from the CPU to the GPU - the latest position data computed by the GPU will always be right
- ▶ Can be made even cheaper by oscillating between two kernel calls which flip the input and output array references
  - ▶ Frame 1: Array1 = Input, Array2 = Output
  - ▶ Frame 2: Array2 = Input, Array 1 = Output
  - ▶ Etc.
- ▶ Might not even need to copy data from the GPU to the CPU, if the GPU is rendering the particles automatically (or has enough data already to know how to render them appropriately - it already knows where to render them!)

# Verlet

- ▶ Special case of Verlet
- ▶ The lack of Velocity computation means no need to store velocity variable
- ▶ The nature of the GPU's memory model means we can essentially store a large FIFO (first in, first out) array to access historical position data
- ▶ Means we can consider Verlet in higher orders for greater accuracy, and we can roll back in time relatively easily

# Suitable Uses?

- ▶ Not really suitable to go through and change the physics engine to operate on the GPU - not enough time
- ▶ But you could add other effects through GPU computation
- ▶ Boids implementation - that's just a bunch of forces
  - ▶ Requires some thoughtful sorting of nearest neighbours to be optimal on the GPU
- ▶ Fluid motion simulation
  - ▶ The more complex partial-diff methods are more appropriate for summer projects, but something simple like interfacing sine and cosine waves updating a heightmap should be fairly simple
- ▶ Motion updates for a particle system (non-colliding physics objects)

# CUDA API Features

# Important Variables and Declarations

- ▶ There are a couple of ways we can declare variables to reside in specific areas of Device memory, and a few things we should know out the gate.
- ▶ There's a CUDA variable `threadIdx` which is used to identify the kernel instance. In general, the first act of a kernel is to determine the instance's thread ID (but not always!).
- ▶ `threadIdx` has dimensionality determined by your block dimensions (check CUDA SDK documentation) - in a one-dimensional block, `threadIdx.x` defines the thread ID, but most blocks aren't 1D

# Important Variable Types

- We recall that the GPU has a memory space for Constant variables which is readable by all threads - we basically declare constants globally using the `__constant__` tag, e.g.:

```
__constant__ float accel_g = 9.81;
```

- We also recall that there's shared memory we can declare variables or arrays to be resident in. These are also declared globally or within a kernel **prior to determining thread ID**, using the `__shared__` keyword, e.g.:

```
__shared__ int terrainType[1024];
```



# Host Functions

- ▶ Host functions are generally forward declared in a .cpp file in which they'll be called, then written explicitly in a .cu file (as a means of isolating the GPU-related portions of the program)

- ▶ That declaration will take a form similar to:

```
extern "C" void cudaTest(float* a, float* b, int entities)
```

- ▶ It is this function which identifies the Host data we expect the GPU to process, and generally passes information regarding just how many threads we expect to create. It can also pass in block/grid dimensions, if we're computing those reactively.

# Kernel Functions

- ▶ These will generally be written in our .cu file, and are called within our Host function. They are **called** by the Host, but **executed** on the Device.
- ▶ They're declared using the following syntax

```
__global__ void testKernel(float* x, float* a, float* b, int entities)
```

- ▶ And called within the host function using this syntax

```
testKernel<<<1, entities>>>(cuda_x, cuda_a, cuda_b, aL)
```

- ▶ Where the values between the angle brackets are customising the dimensionality of grids and blocks, and the variables are the memcpy'd data now resident on the GPU's VRAM

# Device Functions

- ▶ Device functions are a means of keeping your code tidy and modular (just as functions are in C++ and C more generally)
- ▶ Device functions are only **called** and **executed** on the Device. As such, they can only be called by a kernel or by other Device functions - a Device function can **never** be called by Host code - if you want Host code to perform a similar function, you need a normal C++ version of the function.
- ▶ The syntax for a Device function is

```
__device__ void testFunction(float* x, float* a, float* b)
```

- ▶ A Device function can see any constant that its calling thread can see. It can only see shared memory data if that data is declared globally

# Summary

- ▶ Discussed how we go about integrating CUDA into an existing project
- ▶ Considered the consequences for data arrangement of the GPU's memory architecture
- ▶ Discussed numerical integration as a problem applied to the GPU
- ▶ Explored some API features of GPU Computation, using CUDA as a basis

# Implementation

- ▶ Consider the N-body problem for a very busy solar system in 3 dimensions
- ▶ If you set the GPU to favour Shared Memory over cache, you can store the position and mass data for around 3000 objects as a large array which every thread can access
- ▶ Pick a number of objects and give every object an initial and appropriate velocity tangential to the line between it and the massive object at the centre (your 'Sun'), update based on an integration scheme and computed forces between objects of a given mass, and see if you can render the output
- ▶ Have some fun with it! See what you can do to improve the efficiency of the approach - look up bank conflicts to see what might be causing slowdown at high numbers of entities