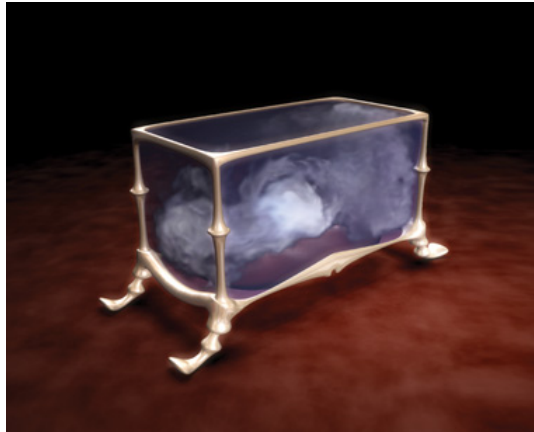


GPU Tutorial 2: Integrating GPU Computing into a Project



Summary

This tutorial extends the concept of GPU computation, introduced in the previous tutorial. We consider the specifics of integrating GPU computation with an existing project in Visual Studio, and specifically discuss the constituent elements of a CUDA program.

New Concepts

Integrating CUDA with our existing code base, Host Functions, Device Functions

Introduction

In the first tutorial in this series, we discussed the origins of GPU Computing, and the various APIs that exist to let us leverage the GPU as a powerful number-crunching processor. We discussed the nature of the problems that GPUs can solve efficiently, and the limitations of the hardware that prevent it solving more complex problems as efficiently.

We went on to discuss the CUDA architecture, paying particular attention to its memory model and multiprocessor structure, before discussing some features of the API. In terms of the API, our approach was that of an overview – we discussed the general structure of a CUDA function, some of the variable types that you will use when programming the GPU, and the pitfalls (including overheads) which can manifest when programming in CUDA. Where yesterday, our interest in the memory model focused on what sorts of functions we can deploy to the GPU, today, our interest is in how it requires us to deploy those functions.

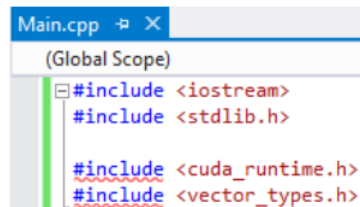
We shall begin with some housekeeping, regarding how to add CUDA functionality to an existing C++ project. Next, we will consider the manner in which data is best stored on the GPU, the reasons for this, and the consequences it has for an existing codebase. Our context for this will be the issues associated with converting our existing physics system for execution on the GPU.

We will conclude with some more, illustrative and detailed discussion of CUDA functions (the difference between device functions and kernel functions, and how these relate to each other and host functions).

Adding CUDA Runtime Libraries to an Existing C++ Project

Often, we will find ourselves in a situation where we wish to include GPU computation functionality to a code-base that was not built around that concept from the outset. While that has significant consequences for us as programmers (addressed later), the first stumbling block for many people new to CUDA is making the compiler realise that you want to include CUDA functionality in the first place.

The first step to introducing CUDA to our program (after installing the Visual Studio plug-in from NVIDIA's developer website) is to include the CUDA runtime libraries. Let us assume a simple program, where we intend to add CUDA somewhere in our main function loop. In this function, we wish to include the CUDA runtime libraries and some data types which CUDA defines. If we simply include these libraries, we'll generally see something close to Figure 1.



```
Main.cpp [X]
(Global Scope)
#include <iostream>
#include <stdlib.h>

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

Figure 1: First Hurdle

The compiler does not know that the project is going to use CUDA, and Visual Studio won't look in the directory containing the CUDA libraries unless we tell it that it needs to. To fix this, we need to right-click on our project name and select Build Customizations (Figure 2).

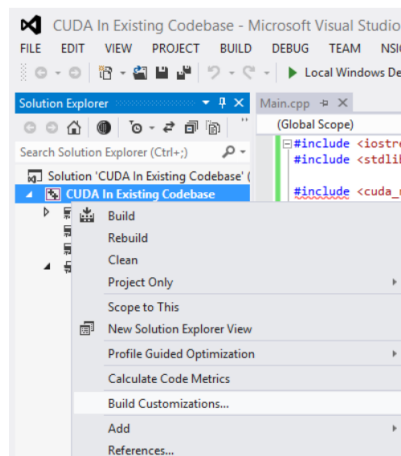


Figure 2: Step One

In the Build Customizations menu, we should ensure the box next to the CUDA version we intend to use (in the example illustrated in Figure 3, 6.5) is checked, and then click OK.

If we then click PROJECT from the top menu, and select Rescan Solution, Visual Studio should now see the libraries and remove the error. From here, the addition of a CUDA function to our program is relatively straightforward, and shall be addressed in detail later on.

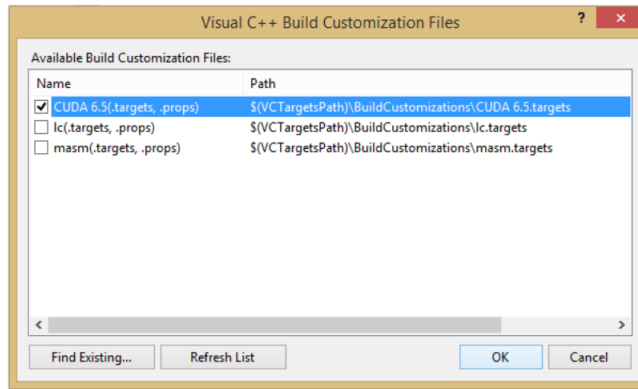


Figure 3: Step Two

Structuring Data for GPU Computation

An early maxim for GPU computation in CUDA, which still holds true today, is Structs of Arrays, not Arrays of Structs. What this means, essentially, is that CUDA operates best if its threads are accessing a continuous block of memory. We remember from the last tutorial the memory hierarchy of the CUDA architecture we will be developing for, illustrated for only a single thread in Figure 4.

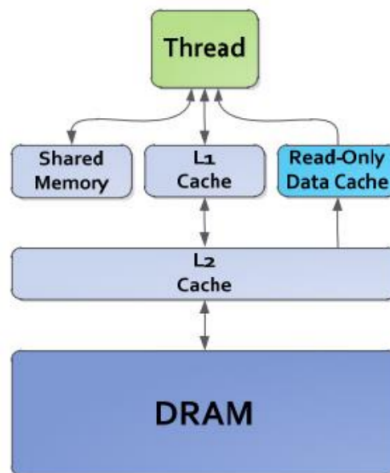


Figure 4: Memory Access Hierarchy: Thread

Similarly, we recall that CUDA threads execute in parallel warps. This has consequences in terms of how we should store data. Let us consider a simple example: a structure (called `TestStruct`) featuring three, three-element vectors (your `Vector3` class from the Graphics tutorials, or the `float3` struct). In our example kernel, a mathematical operation is performed on all elements of all three vectors. The order of execution is `Vector3 B`, `Vector3 A`, `Vector3 C`.

If we pass an array (called `testStructArray`) of `TestStructs` to the GPU, and execute the kernel, two thirds of L2 cache will feature data that is not needed for the first instruction. This is due to the fact that the data is stored as a structure, and structures are continuous in memory. As a result, `A` and `C` for the first batch of threads must be pulled into cache, along with the (needed) `B`.

For a small enough array (assume `testStructArray`'s entire memory footprint is smaller than the L1 cache of a single SMX), this will have no real-terms impact on performance. But if the number of elements in our execution block is *that* small, what are we doing deploying it to the GPU in the first place?

A better approach for us to take, if we want to maximise the efficiency of the GPU, is to extract the data contained in each element of `testStructArray`, and form three arrays of `Vector3s` (called A, B and C). We then send those three arrays to the GPU, instead of the one array of structures, and call the same kernel.

Arrays, like structures, are continuous in memory. This means that when our kernel begins to operate on the large number of `Vector3` Bs, only elements of array B fetched into L2 cache, and so all data being stored in the cache is useful for the current executed task. When enough of the array B is processed to leverage warping, array A is fetched, and so on.

There are further optimisations one can do along this line of thought for example, we are using `Vector3s` or `float3s` which are a `class` and a `struct`, respectively. We could, therefore, break those down into arrays of their own. The reason we might not choose to do this is simple: overhead. We will discuss the specifics of that overhead below, in the context of a physics system.

Sorting Data for GPU Computation

Consider a class `PhysicsNode`, illustrated in Figure 5.

```
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;
};
```

Figure 5: Physics Node Class

If we wish to process all physics updates for our physics system on the GPU, we will need to transfer this data to the GPU. Bearing in mind what we just observed regarding the parallelisability of a structure of three `Vector3s` (a 36 byte structure), we can infer with a degree of certainty that a `PhysicsNodeData` structure of at least 188 bytes will be suboptimal if processed as-is.

As such, if we are to process any of the data from our physics system on the GPU (let alone all of it), we need to either refactor our entire physics system to operate on arrays (one array per data type, and one element per array per entity), or perform a sorting operation to generate these arrays once per update prior to calling our CUDA kernel.

These operations themselves are significant in terms of their overhead. They would become even more significant if we attempted to break down the individual structures within the class (the `Vector3s`, `Quaternion`, and `Matrix4`) into arrays of their own which is one reason, along with the GPUs optimisations for working on three- and four-float data-types, that we do not do this.

When you factor in this overhead, along with the overhead discussed in the last tutorial regarding copying data from the Host to the Device and back again, it is relatively clear that GPU computation for physics in your system is only truly viable if your system is very complex (has a great many

entities). Which is not to say that it should never be done - only that it must always be balanced to ensure all overhead, including sorting overhead, is worthwhile in terms of total performance.

Numerical Integration on the GPU

When implementing physics on the GPU, you can opt to have the GPU only perform certain physical tasks. A good example of this is water representation, the generation of ripples based on a point of contact at the centre of a continuous object. If you have particle systems within your physics which do not need to collide with other moving entities within the environment, there are optimisations you can perform to reduce your computational overhead based on your chosen numerical integration method.

You will recall that in both Euler and Verlet integration, we are required to update an object based on its position (or past positions, in the case of Verlet), and some updated value of acceleration. Based on this principle, we can flip the order of the parameters in our kernel (have two kernel calls, which oscillate between one another) to remove the need to copy the next frames current position, as it will always be the last frames next position (in collision-less systems, or colliding systems which do not employ projection to account for overlap).

In the case of Verlet, this is slightly more complicated as you might have several frames worth of position data, and wind up cycling through them in a First-in, First-out sense. The trade-off with this complexity is that Verlet benefits far more from the optimisation, as without it you might be transferring several previous frames position data arrays every update of the physics system.

This optimisation will not work for systems where narrowphase collision checks automatically correct for object overlap (as that will make position data stored on the GPU invalid). Similarly, it will not work for any system where position can be changed host-side inbetween GPU kernel executions. Correcting this requires copying the updated position data in any case, which can negate the benefit (the GPUs analogues for writing updates to specific memory addresses say, only updating the positions of the entities which have had collision-based corrections to position are cumbersome).

It would not be appropriate to adapt the existing physics framework to operate on the GPU, but it would be appropriate to introduce another physics system - for example, a particle-based vortex - to your environment. This could be computed in CUDA and, as it is not a colliding scene element, would not impact your other physical behaviours. Other scene elements you could add which might be suitable for GPU solution include:

- Motion updates for large numbers of non-colliding objects
- Fluid motion modelling for limited-collision fluids
- Flocking Boids

CUDA API Features

In this section, we introduce three key language features which underpin the CUDA API, in the order in which you would call them:

Host Functions

These are forward declared in a `.cpp` file where they will be called. That forward-declaration takes the form:

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

Host Function Forward Declaration

In this example, a host function called `cudaTest`, accepting pointers to arrays of `floats` and a number of entities, is forward declared at the start of a `.cpp`. The function itself is defined in a `.cu` file, if we intend to compartmentalise our CUDA code away from the rest of our program.

Kernel Functions

Kernel functions in the GPU are written in your `.cu` file (see the sample project), and called from within a CUDA host function (also written in the `.cu` file). A kernel function is declared using the following syntax:

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

Kernel Function Declaration

The `__global__` tag informs the CUDA compiler (`nvcc`) that this is a kernel, which shall be called from within a CUDA host function (or, in later versions of CUDA, within another CUDA kernel - note that this functionality is not available on all machines). The call within a CUDA host function which initiates a kernel takes the form:

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

Kernel Call

All kernels should be considered void. Your data will be returned by writing to memory you include within the kernel call (either overwriting input data, or writing to a specific output array). We reiterate from earlier that a kernel function call is actually a trigger for an instance of that function to be called for every element in the execution grid.

In the example above, `testKernel` is accepting three arrays of floats, and an integer. These arrays, importantly, are on the GPU. Following our execution through from the `cudaTest` function, `cuda_a` and `cuda_b` might well be duplicates of arrays `a` and `b`, with `cuda_x` serving as an output array to be copied back to the Host after execution completes.

Device Functions

Device functions are a means of writing efficient code, and are analogous to functions in C/C++. They can be used to negate the need for classes in CUDA (implementing class functions in CUDA as properties of a class can be cumbersome, and requires additional compiler definitions). There are declared in the form:

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

Device Function Declaration

As with kernel functions, the `__device__` tag informs the compiler of the function's nature. Device functions are highly versatile; in the example above, `testFunction` might be receiving a memory address for a single float currently being processed, or an entire array.

Moreover, device functions enable you to maintain tidy kernels, and can provide additional compiler optimisations as a function of the GPUs instruction cache, depending on the manner in which they are employed. Device functions can ONLY be called by other device functions, or a kernel; a device function can *not* be called by any host code.

Implementation

You are encouraged to introduce CUDA functionality to your projects, but ensure to back your projects up beforehand - dabbling with the GPU can be problematic, and you should always maintain a version of your software prior to introducing CUDA libraries.

Check the Practical Tasks for today. Look into a particle swarm or vortex, and how you might implement that on the GPU for inclusion in your scene. Consider flocking boids following a mobile element of your scene.