CUDA Without a PhD

Lloyd Moore, President
Lloyd@CyberData-Robotics.com
www.CyberData-Robotics.com

Agenda:

Introduction to GPU Architecture
What is CUDA?
CUDA Setup
Problem Definition
CPU Single Threaded Solution
GPU Massively Parallel Solution
Debugging CUDA Kernels
Resources
Q & A

Disclaimer

CUDA development can be a VERY deep topic. To get the maximum performance from a GPU based application the problem needs to start with a correct formulation, considering the specific hardware being used, data access patterns, memory bandwidth, processor topology and much more.

This talk IS NOT about all of that, as there are plenty of well done presentations covering those topics already. If you want to get into the deep details NVIDA has a great starting point here: https://docs.nvidia.com/cuda/doc/index.html

This talk IS about how a developer can get a very simple start with CUDA applications and see immediate benefits without having to spend weeks of time learning all of the details. You won't be able to fully optimize an application but you will be able to quickly convert select processing patterns and see a considerable speed ups.

GPU Architecture

A modern CPU consists of a small number of complex processors that are mostly independent of one another, and perform work on generally independent tasks.

This computational pattern is generally referred to as SISD – Single Instruction, Single Data.

A modern GPU consists of hundreds to thousands of very simple processors that work together to perform the same operation on multiple pieces data in parallel.

This computational pattern is generally referred to as SIMD – Single Instruction, Multiple Data. (And SIMT – Single Instruction Multiple Thread - per NVIDIA)

The GPU “likes” to work in 32 bit floating point. 64 bit floating point is supported however you do take a time penalty for the additional precision. Of course integer math is also supported!

GPU Architecture

From this description, the GPU offers an effective speed up in the following case:

1. You have a VERY large set of data that needs to be processed
   Think 100's of MB or GB of data
2. The data format is "regular"
   For example stored in arrays or vectors
3. The same (or very similar) operations need to be performed on each element
4. The operations to be performed on each data element are independent
5. The amount of work to be performed on each element is significant enough to justify copying the data at least twice

Let's look at that last statement in more detail…..

GPU Memory Architecture

A GPU typically contains a dedicated bank of memory, independent from the normal CPU memory.

GPU memory is optimized for highly parallel access patterns.

Information to be processed by the GPU must be copied from the CPU memory, called “host memory”, to the GPU memory, called “device memory”.

Results may be used on the GPU directly or copied back to the CPU / host memory, depending on the application.

Due to the overhead of having to copy data between memories, the amount of work that needs to be done needs to be complex enough to amortize the copy overhead.

Note: “Unified Memory”, “Shared Memory” and “Texture Memory” also exist, not going to talk about those here as each has a specific use and trade offs.

What is NVIDIA CUDA?

NVIDIA CUDA is a framework and tools which allow for application development on NVIDIA GPU hardware.

Top level documentation is here: https://docs.nvidia.com/cuda/doc/index.html

Main Components:

- NVIDIA Compiler: nvcc
- CUDA API
- Debugging and Profiling Tools: Nsight Compute
- Math Libraries: cuBlas, cuFFT, cuRand, cuTensor, cuSparse, cuSolver, nvJPEG, Thrust, and many others
- Technologies: GPUDirectStorage – Direct GPU to disk access

CUDA Setup - Requirements

CUDA can run in Windows and Linux environments on PCs (x86/64) and Jetson (ARM) hardware.

For this exercise I'll use the following configuration (Note: smaller systems WILL also work fine – this is NOT a minimum recommended configuration):

CPU: AMD Ryzen 9 7950X, 16 core, 32 thread, 4.5 Ghz
Motherboard: Asus ProArt X670E-Creator
RAM: 64GB DDR5 4800
GPU: Asus GeForce RTX 4080, 16GB RAM
GeForce Game Ready Driver Version 546.33
OS: Windows 11 Pro, 64 Bit, 22H2 22621.3007
Visual Studio Community Edition 2022
CUDA: 12.2

CUDA Setup – Tool Chain

For this talk we'll focus on Visual Studio and Windows as it is the simplest to get going.

CUDA supports many other configurations on both Windows and Linux including operating through WSL2.

Install Microsoft Visual Studio Community 2022, 64 bit:
https://visualstudio.microsoft.com/vs/community/
Configure for at least C++ development

Install NVIDIA CUDA for Microsoft Visual Studio:
https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html
Don't need to worry about installing the Python tools unless you want to.

Sample Problem Definition

For this talk we'll solve a fairly simple problem showing a typical design pattern for solving many other problems, and that won't distract us with any complexity of the problem itself.

Problem:

Compute the hypotenuse of a large quantity of triangles given the lengths of the sides of the triangles, using the Pythagorean Theorem: h = sqrt(a^2 + b^2)

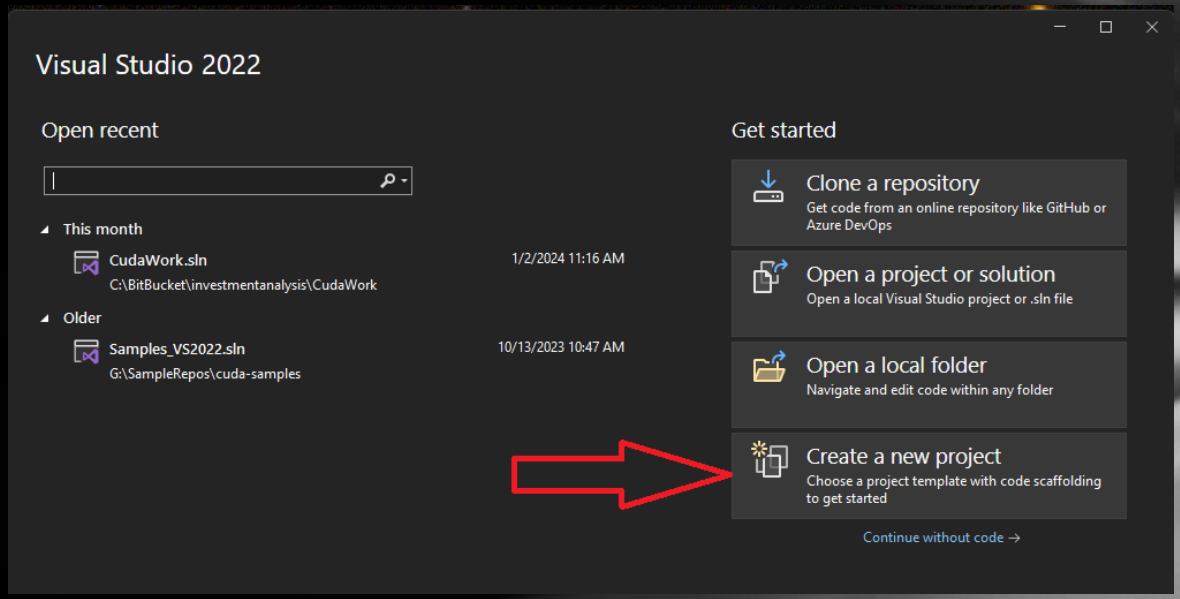
For this example we'll create two vectors of random numbers for 'a' and 'b'.

We'll compute the results using a single threaded approach on the CPU and then convert the code to use the GPU and compare the execution times.

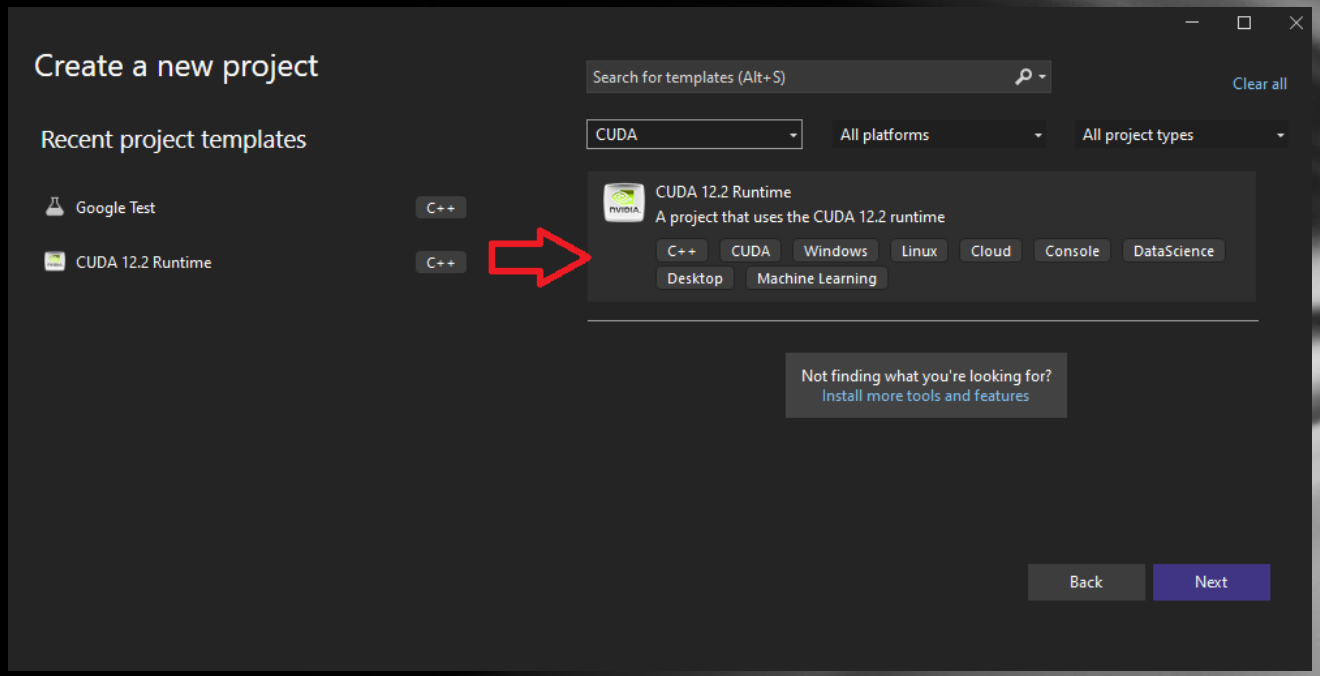
Finally we'll compare the results between the CPU and GPU and make sure they match.

Note: For this example we'll use 32 bit floating point.

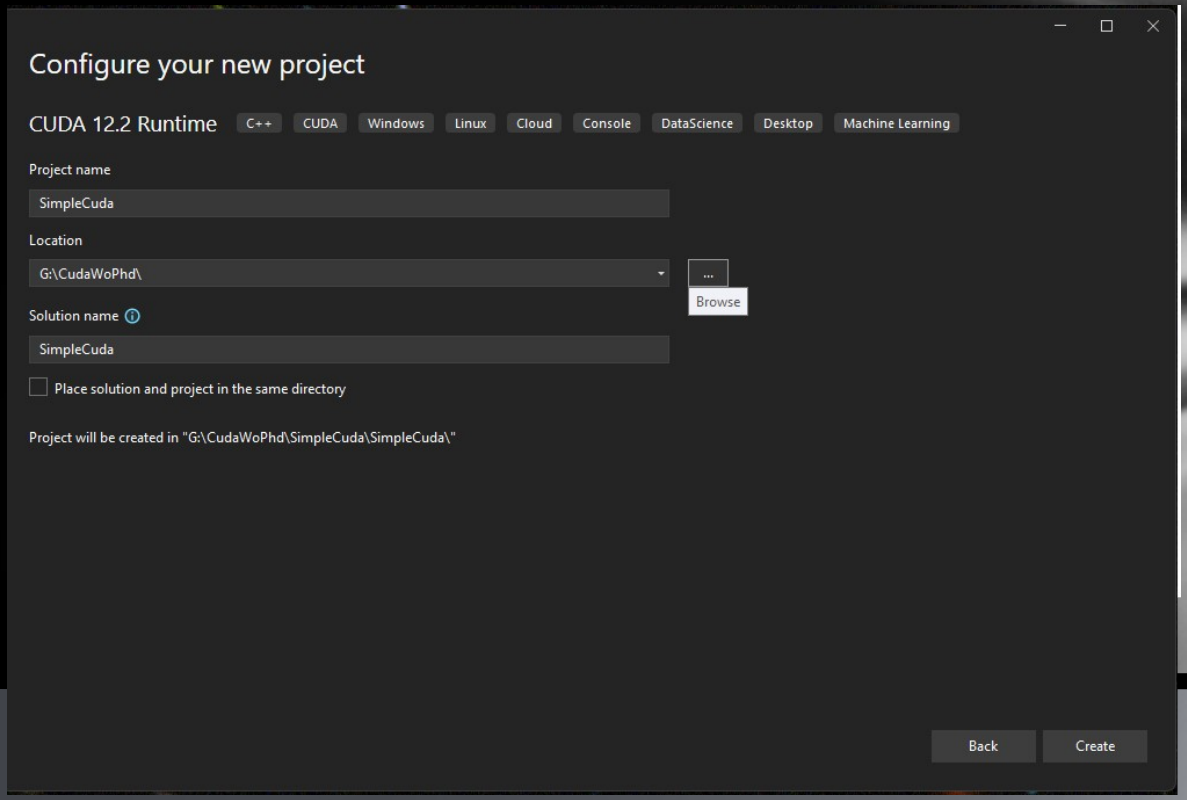
Creating the Project in VS



Creating the Project in VS



Creating the Project in VS



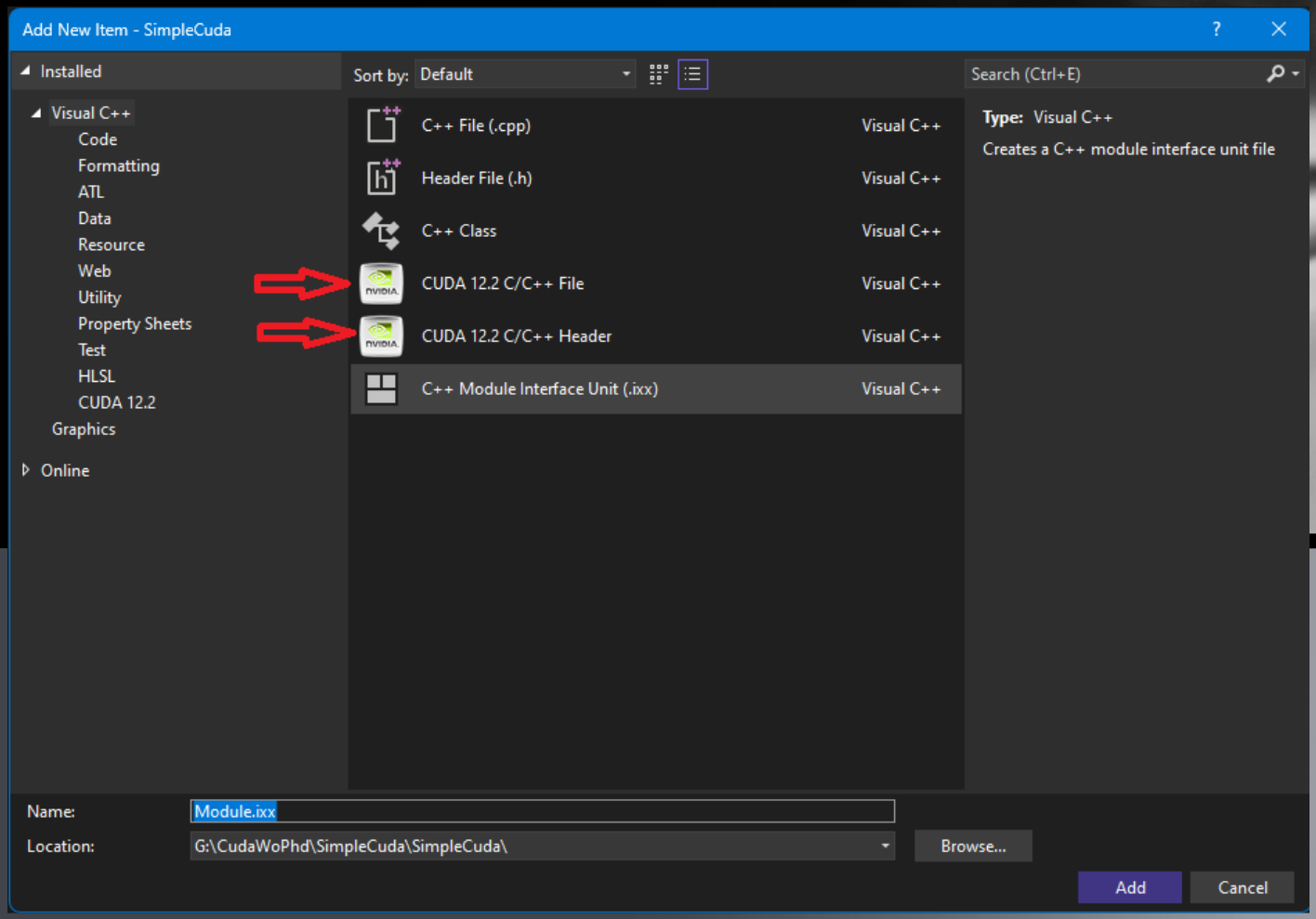
Creating the Project in VS

```cpp
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size);

__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

int main()
{
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };

    // Add vectors in parallel.
    cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addWithCuda failed!");
        return 1;
    }

    printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
        c[0], c[1], c[2], c[3], c[4]);

    // cudaDeviceReset must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }

    return 0;
}
```

Note: Full sample truncated to fit on this slide!

Creating the Project in VS

When adding files for CUDA use the “Add” → “Module…” option.
CUDA files are named *.ccuh for header files and *.cu for C++ files.



CPU Single Threaded

To start we'll create a little C++ class to hold all of our data and algorithms called CudaWorker:

```cpp
#pragma once

#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

constexpr size_t VECTOR_SIZE = 100000000;   // Number of elements to compute

class CudaWorker
{
public:
    CudaWorker();
    ~CudaWorker();

    // Disable special member functions
    CudaWorker(const CudaWorker& other) = delete;
    CudaWorker(CudaWorker&& other) noexcept = delete;
    CudaWorker& operator=(const CudaWorker& other) = delete;
    CudaWorker& operator=(CudaWorker&& other) = delete;

    // Perform the computation using a single thread on the CPU
    void CpuCompute();

    // Perform the computation using the GPU in parallel
    void GpuCompute();

    // Verify that the answers match between the CPU and GPU
    bool Verify();

private:
    // Storage on the host
    thrust::host_vector<float> a;               // Values for one one side of the triangle
    thrust::host_vector<float> b;               // Values for the other side of the triangle
    thrust::host_vector<float> cpu_result;      // Result values from CPU computation
    thrust::host_vector<float> gpu_result;      // Result values from GPU computation

    // Storgage on the gpu
    thrust::device_vector<float> device_a;
    thrust::device_vector<float> device_b;
    thrust::device_vector<float> device_result;
};
```

CPU Single Threaded

I am using a library called Thrust. Thrust is a C++ template library for CUDA, based on the std::vector that works for both ‘host’ and ‘device’ memory.

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
```

```
// Storage on the host
thrust::host_vector<float> a;                // Values for one one side of the triangle
thrust::host_vector<float> b;                // Values for the other side of the triangle
thrust::host_vector<float> cpu_result;       // Result values from CPU computation
thrust::host_vector<float> gpu_result;       // Result values from GPU computation

// Storgage on the gpu
thrust::device_vector<float> device_a;
thrust::device_vector<float> device_b;
thrust::device_vector<float> device_result;
```

CPU Single Threaded

The constructor simply fills the a & b vectors with random data, we also prep the ‘device’ vectors for the GPU at the same time:

```
CudaWorker::CudaWorker() {
    // Preallocate the vector storage
    a.reserve(VECTOR_SIZE);
    b.reserve(VECTOR_SIZE);
    cpu_result.reserve(VECTOR_SIZE);
    gpu_result.reserve(VECTOR_SIZE);

    // Initialize a random number generator
    std::default_random_engine rgen(83903);
    std::uniform_real_distribution<float> fran(0.0, 100.0);

    // Fill the two vectors with some random numbers
    for (size_t i = 0; i < VECTOR_SIZE; ++i) {
        a.push_back(fran(rgen));
        b.push_back(fran(rgen));
    }

    // Setup the same values on the GPU
    device_a = a;
    device_b = b;
    device_result.reserve(VECTOR_SIZE);
}
```

Note that with Thrust copying the vector from host memory to device memory is a simple assignment!

CPU Single Threaded

CpuCompute() simply consists of a loop that calls a common math routine for each set of data:

```
void CudaWorker::CpuCompute() {
    for (size_t i = 0; i < VECTOR_SIZE; ++i) {
        float* ptr_to_a = &a[i];
        float* ptr_to_b = &b[i];
        float* ptr_to_result = &cpu_result[i];
        do_pythagorean(ptr_to_a, ptr_to_b, ptr_to_result);
    }
}
```

```
__device__ __host__ void do_pythagorean(const float* p_a, const float* p_b, float* p_result) {
    float a = *p_a;
    float b = *p_b;
    *p_result = sqrt(a * a + b * b);
}
```

Note that do_pythagorean() is tagged with “__device__” and “__host__”. These are attributes to the NVCC compiler to build the function so it can run on both the CPU and GPU.

We will also see “__global__” which is an attribute flagging a CUDA Kernel – we’ll talk about that more when we convert this code for the GPU.

CPU Single Threaded

And of course we need a main() to instantiate and call CudaWorker. This also has the GPU code present…….

```cpp
#include <chrono>
#include <iostream>

using namespace std;
using std::chrono::high_resolution_clock;
using std::chrono::duration_cast;
using std::chrono::duration;
using std::chrono::microseconds;

#include "CudaWorker.cuh"
#include "Gpu.h"

int main()
{
    Gpu gpu;            // Singleton to mange the GPU, initializes the first card
    CudaWorker worker;  // A instance of the work we want to accomplish

    cout << endl;

    // Perform the work on the CPU and report the time
    auto start_cpu_time = high_resolution_clock::now();
    worker.CpuCompute();
    auto end_cpu_time = high_resolution_clock::now();
    cout << "CpuCompute took " << duration_cast<microseconds>(end_cpu_time - start_cpu_time).count() << "us" << endl;

    // Perform the work on the GPU and report the time
    auto start_gpu_time = high_resolution_clock::now();
    worker.GpuCompute();
    auto end_gpu_time = high_resolution_clock::now();
    cout << "GpuCompute took " << duration_cast<microseconds>(end_gpu_time - start_gpu_time).count() << "us" << endl;

    // Make sure the answers match between the CPU and GPU
    if (worker.Verify()) {
        cout << "Answers match!" << endl;
    }
    else {
        cout << "Answers DO NOT match!" << endl;
    }

    return 0;
}
```

GPU Massively Parallel

Next we'll convert this solution to run on the GPU. The first thing we need to do is initialize the GPU. I have a singleton class called Gpu for this:

```
#pragma once

// A helper class to interact with the GPU. Currently this class is basically implemented as a singleton and assumed
// It will always initialize to the first GPU enumerated if more than one GPU is present.

#include "cuda_runtime.h"

class Gpu
{
public:
    Gpu();
    ~Gpu();

    static int ComputeCapabilityMajor();
    static int ComputeCapabilityMinor();
    static int CudaCoreCountPerMultiProcessor();
    static int CudaCores();
    static int MaxThreadsPerBlock();
    static int MaxThreadsPerMultiProcessor();
    static int MultiProcessorCount();

private:
    static bool initialized;
    static cudaDeviceProp deviceProp;
};
```

GPU Massively Parallel

The constructor initializes the GPU and prints out some of the GPU parameters:

```cpp
#include <iostream>

#include "Gpu.h"

using namespace std;

bool Gpu::initialized {false};
cudaDeviceProp Gpu::deviceProp;

const int DEFAULT_GPU = 0;

Gpu::Gpu() {
    if (!initialized) {
        cudaError_t cudaStatus = cudaSetDevice(DEFAULT_GPU);
        if (cudaStatus != cudaSuccess) {
            cout << "cudaSetDevice failed with error: " << cudaGetErrorString(cudaStatus) << endl;
        }
        else
        {
            cudaGetDeviceProperties(&deviceProp, DEFAULT_GPU);
            initialized = true;
        }

       cout << "GPU Stats:" << endl;
       cout << "    Compute Capability: " << ComputeCapabilityMajor() << "." << ComputeCapabilityMinor() << endl;
       cout << "    " << MultiProcessorCount() << " Multiprocessors, " << CudaCoreCountPerMultiProcessor() << " CUDA
       cout << "    MaxThreadsPerMultiProcessor: " << MaxThreadsPerMultiProcessor() << endl;
       cout << "    MaxThreadsPerBlock: " << MaxThreadsPerBlock() << endl;
    }
}
```

GPU Massively Parallel

The destructor “cleans up” the GPU with a cudaDeviceReset():

```
Gpu::~Gpu() {
    if (initialized) {
        initialized = false;
        // cudaDeviceReset must be called before exiting in order for profiling and
        // tracing tools such as Nsight and Visual Profiler to show complete traces.
        cudaError_t cudaStatus = cudaDeviceReset();
        if (cudaStatus != cudaSuccess) {
            cout << "cudaDeviceReset failed with error: " << cudaGetErrorString(cudaStatus) << endl;
        }
    }
}
```

GPU Massively Parallel

A block of code that runs on the GPU is called a “kernel”

An instance of the “kernel” is run on each core of the processor as an independent thread.

From the developer’s point of view the “kernel” is just a function call, however under the covers this function call will be instantiated on each core in parallel, and have access to information uniquely identifying each instance. This is called a “thread address”.

In traditional graphics processing each pixel displayed is assigned to a thread. See https://www.shadertoy.com/ to play with this concept!

For the current problem we have vectors of data so we’ll simply assign each element / index of the vector to a thread.

GPU Massively Parallel

GpuCompute() is the member function that configures and launches a “kernel” on the GPU, at this point it is assumed the data has already been copied to the GPU memory, the CudaWorker constructor did that:

```
void CudaWorker::GpuCompute() {
    const size_t num_threads = Gpu::MaxThreadsPerBlock();
    const size_t num_blocks = VECTOR_SIZE / num_threads + 1;    // Simple ceiling function
    pythagorean_kernel<<<num_blocks, num_threads>>>(device_a.data().get(), device_b.data().get(),
        device_result.data().get());

    cudaError_t cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        cout << "CudaWorker::GpuCompute() kernel failed with" << cudaGetErrorString(cudaStatus) << endl;
        return;
    }
    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
        cout << "CudaWorker::GpuCompute() synchronize failed with " << cudaGetErrorString(cudaStatus) << endl;
        return;
    }
}
```

cudaGetLastError() will return an error code if the “kernel” was not launched successfully.

cudaDeviceSynchronize() will wait for the work on the GPU to be completed and return an error code if anything went wrong.

GPU Massively Parallel

The GPU hardware only allows so many threads in a “block” so the work must be partitioned into blocks.

Invoking a kernel looks just like a function call with some extra annotation:

```
const size_t num_threads = Gpu::MaxThreadsPerBlock();
const size_t num_blocks = VECTOR_SIZE / num_threads + 1;    // Simple ceiling function
pythagorean_kernel<<<num_blocks, num_threads>>>(device_a.data().get(), device_b.data().get(),
    device_result.data().get());
```

The “<<<blocks, threads>>>” annotation is picked up by NVCC and converted into a kernel invocation matching the given geometry

Under the covers CUDA maps the given geometry to the hardware geometry and launches as many threads in parallel as the hardware allows. If there are more threads than actual hardware, multiple launches are serialized until all the work is done.

GPU Massively Parallel

```
__global__ void pythagorean_kernel(const float* a, const float* b, float* result) {
    const size_t index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < VECTOR_SIZE) {
        do_pythagorean(a+index, b+index, result+index);
    }
}
```

Function annotations tell the compiler how to build and call the code:

__global__ : Runs on the GPU, called from either CPU or GPU
__device__ : Runs on the GPU, called from the GPU
__host__ : Runs on the CPU, called from the CPU

Annotations can be combined.

Each GPU thread needs to do two things:

Identify the data elements that it is to work on
Perform the specified work on those data elements

GPU Massively Parallel

Each kernel is invoked with variables for “thread addressing”:

threadIdx : Contains the “address” of current thread

blockIdx : Contains the “address” of the current block

blockDim : Contains the geometry of the block sizes

Currently we use only the X dimension, in reality these values have 3 dimensions allowing for easy mapping to real word 3D spaces.

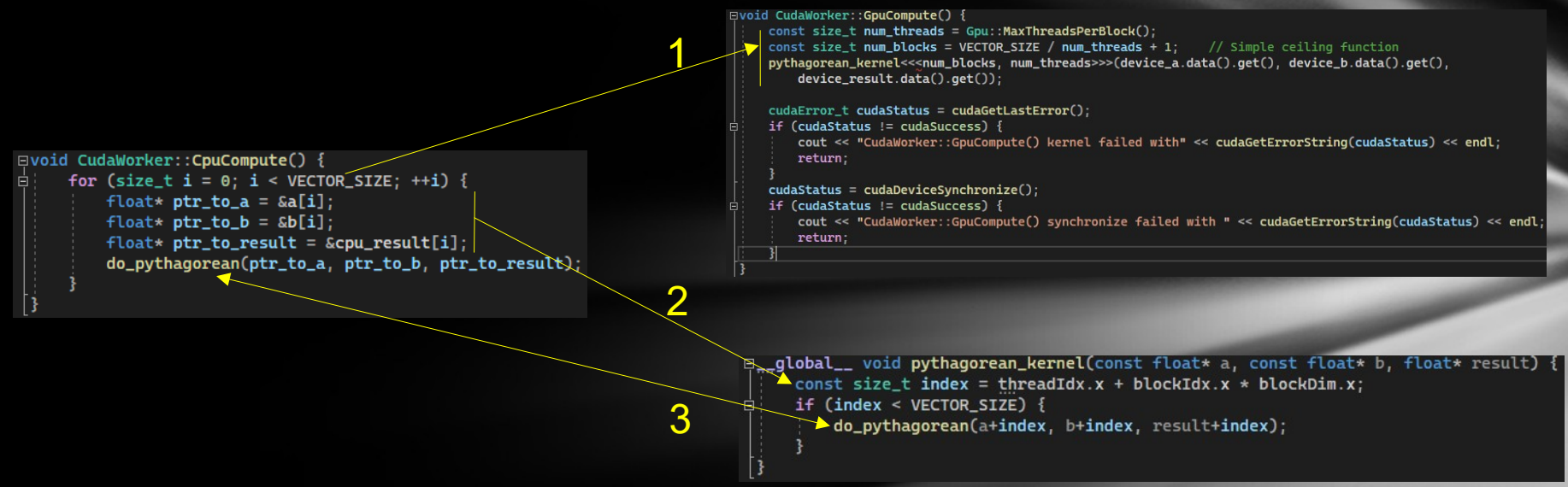
```
__global__ void pythagorean_kernel(const float* a, const float* b, float* result) {
    const size_t index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < VECTOR_SIZE) {
        do_pythagorean(a+index, b+index, result+index);
    }
}
```

For each dimension combine the threadIdx, blockIdx and blockDim as shown to create a fully unique ID for the kernel invocation.

For this problem data was up such that the “thread address” directly maps to the index of the data – this is very common and very simple!

There may be more “thread addresses” than data, mask with an “if”.

CPU vs. GPU Code



Key Conversion Points:

1. The sequential ‘for’ statement becomes a CUDA kernel invocation
2. The address calculations become a thread address calculation
3. The “work” ends up being done by exactly the same function!

Once you get familiar with this conversion technique you can generally apply it in about 30 to 60 minutes! (Faster if you plan for it in advance!)

Verification

```
bool CudaWorker::Verify() {
    // Copy the GPU / device results back to the host
    gpu_result = device_result;

    // Make sure the size and contents of the CPU results matches the GPU results
    bool verification_result = cpu_result.size() == gpu_result.size();
    verification_result &= std::equal(cpu_result.begin(), cpu_result.end(), gpu_result.begin());

    return verification_result;
}
```

You don’t normally need to include a verification routine, but is helpful:

- Math processing on the GPU is different than on the CPU
- Nice sanity check to convince yourself this really works

Results

```
GPU Stats:
    Compute Capability: 8.9
    76 Multiprocessors, 128 CUDA Cores / MP, 9728 CUDA Cores
    MaxThreadsPerMultiProcessor: 1536
    MaxThreadsPerBlock: 1024

CpuCompute took 107394us
GpuCompute took 2047us
Answers match!
```

Speed up: 107394us / 2047us = 52.46x (for this one case, clearly run more!)

This speed up DOES NOT include the copy overhead of the data to and from the GPU. This will impact the results considerably, however the “work” we are also doing is pretty simple. It does include kernel invocation, which is nontrivial.

This is a VERY unoptimized solution! With a full effort you can get 1000x improvements for very well formed and well fitting cases.

Debugging CUDA Kernels

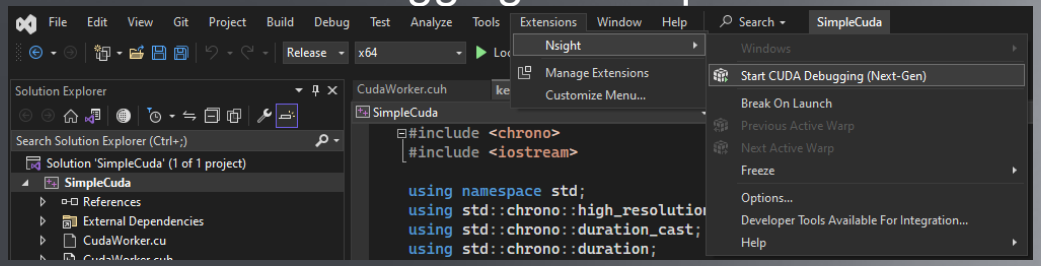
Debugging kernels can be a bit more challenging due to the following:

- Typically there are THOUSANDS of instances running
- Access to the GPU memory is more restricted

Simple guidelines to get started:

- Place the “work” to be done in function, like was done here
  - Debug the “work” on the CPU as you normally would
  - Once this is done all that is left is the data mapping
- Reduce the kernel invocation to a single thread
  - function<<<1,1>>>(a, b, c);
  - Gets around thousands of invocations running
  - Also helpful to test with two invocations: <<<1,2>>>
- In Visual Studio, printf() works just as you expect inside kernels
  - Combine with reducing the kernel invocations

Breakpoints and visual debugging techniques ARE available!



Additional Resources

This talk has barely scratched the surface of what can be done. The goal was to provide a simple, effective solution to a common problem, and is the beginning of a journey!

Official Documentation:

CUDA Main Docs: https://docs.nvidia.com/cuda/doc/index.html
CUDA Dev Tools: https://developer.nvidia.com/tools-overview
Thrust Library: https://developer.nvidia.com/thrust

Good Books:

Programming in Parallel with CUDA
Richard Anderson; ISBN: 978-1108479530
Programming Massively Parallel Processors
Hwu, Kirk, Jajj; ISBN: 978-0323912310

Shader Toy: https://www.shadertoy.com/

Open Discussion
&
Q & A