CUDA

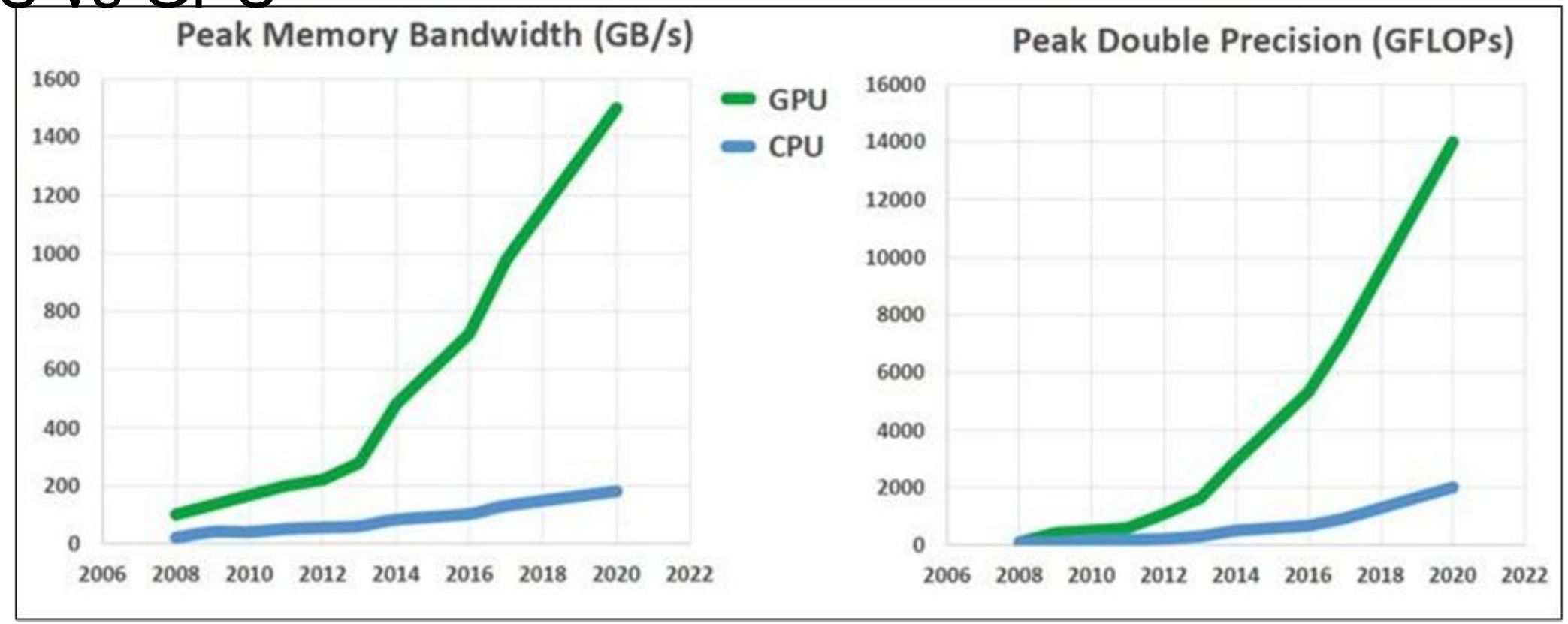
Elias Farhan

Frédéric Dubouchet

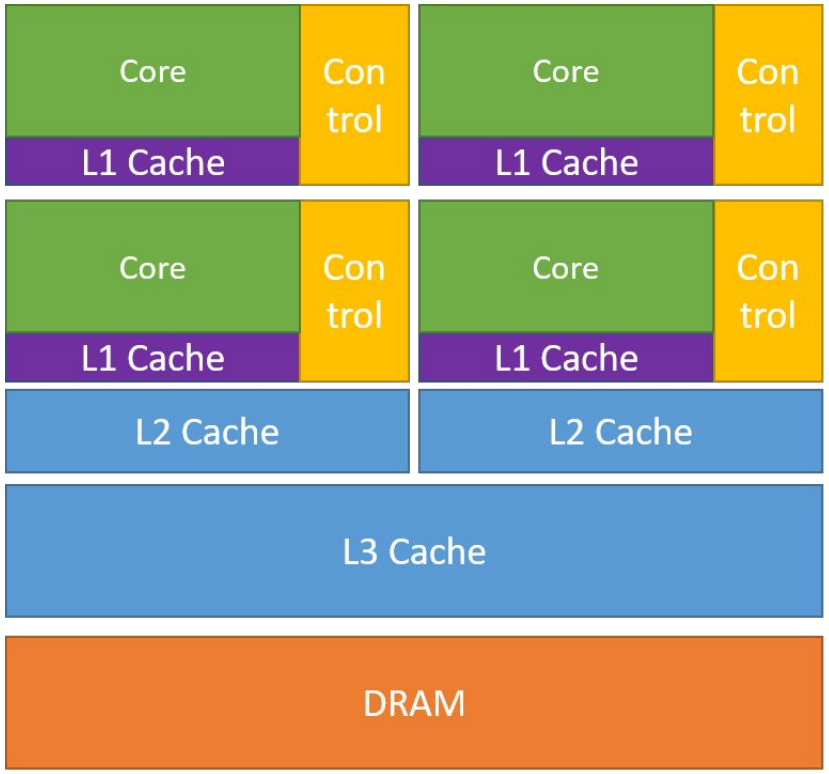
Disclaimer

Even if the GPU computing sounds sexy. It is not always the best solution for your problem.

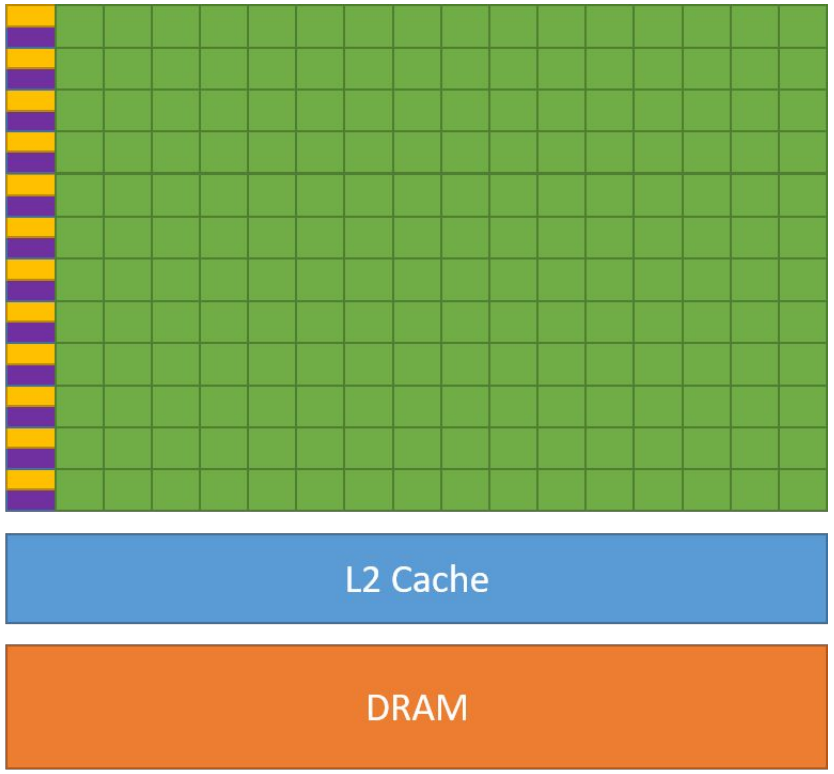
CPU vs GPU


Peak Memory Bandwidth (GB/s)
1600
1400
1200
1000
800
600
400
200
0
2006
2008
2010
2012
2014
2016
2018
2020
2022
GPU
CPU
Peak Double Precision (GFLOPs)
16000
14000
12000
10000
8000
6000
4000
2000
0
2006
2008
2010
2012
2014
2016
2018
2020
2022

CPU vs GPU



CPU

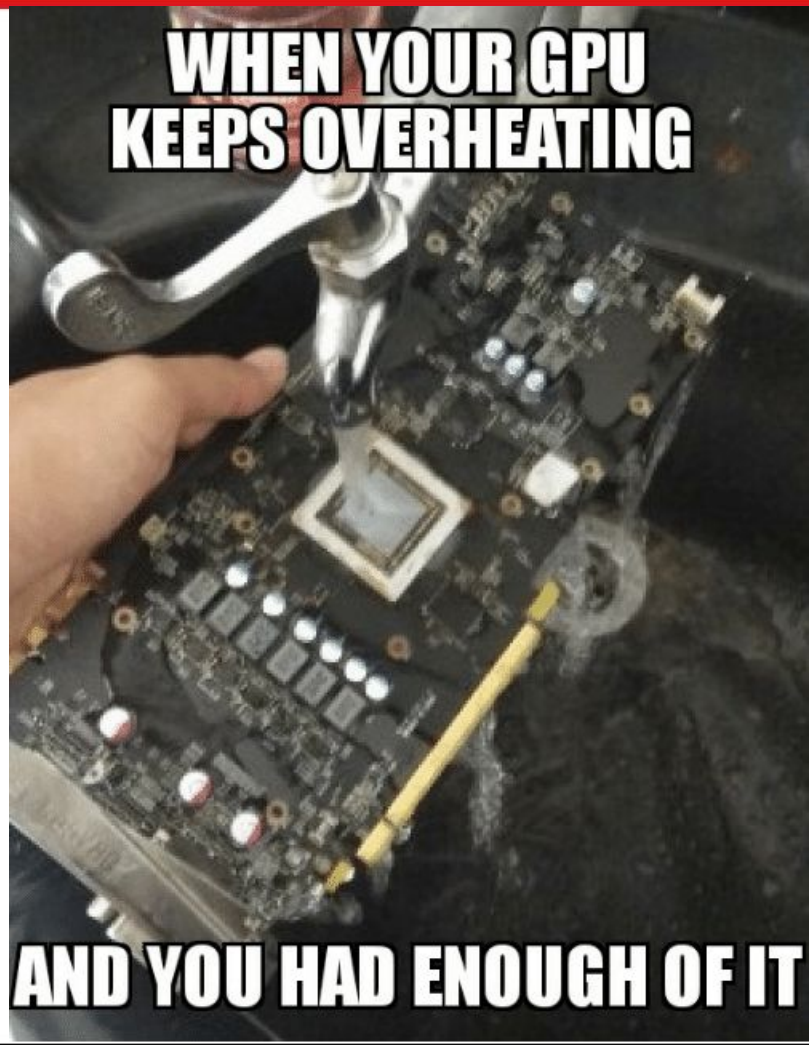


GPU

CPU vs GPU

GPU are specialized in highly parallel computation and devote more transistors to data processing, rather than data caching and flow control.

The GPU can hide memory access latencies with computation instead of avoiding memory access latencies through large data caches and flow control with highly parallel computations



CUDA

"Compute Unified Device Architecture" is a general purpose parallel computing platform and programming model for the most modern Nvidia cards.

It is a proprietary software, but it is one of the simplest to use, compare to OpenCL and others.

The CPU program is a ***host*** and the GPU code is running on the ***device***.

CUDA

GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
				

CUDA kernels

CUDA extends C++ by allowing programmer to define special C++ functions, called kernels.

Let's see how it looks like.

A kernel is defined by the __global__ word.

To call the kernel, we call the function with this weird:

<<<1,1>>> notation.

```
__global__
void add(int n, float* x, float* y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
```

```
// Run kernel on 1M elements on the GPU
add <<<1, 1>>> (N, x, y);
```

CUDA high-latency

Sending data and command to the GPU is slow…
Very slow... ~10-100ms

So using the GPU for kernels that are of order below 1 second makes no sense, because of the latency issue.

CUDA thread hierarchy

We actually used only 1 thread on the previous example.

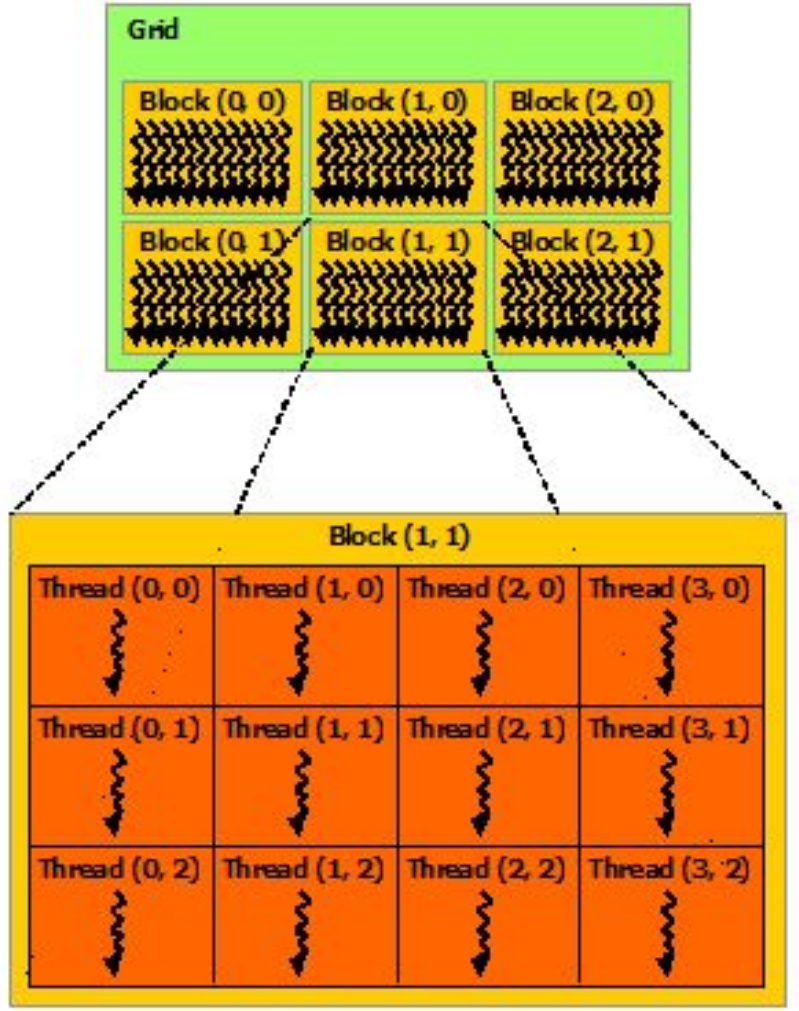
We can use several threads pretty easily, we need to change the second argument:

```
add << <1, 256 >> > (N, x, y);
```

CUDA thread hierarchy

And we need to take account for it in the kernel as well:

```
__global__
void add(int n, float* x, float* y)
{
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}
```



CUDA thread hierarchy

For convenience, threadIdx is a 3-component vector (convenient for big 2-dimension matrix for example Leontief matrix).

What was this blockDim.x? It is the size of the thread block (here 256) and can contain up to 1024 threads.

Can we have multiple blocks? Yes, of course!

CUDA thread hierarchy

For convenience, threadIdx is a 3-component vector (convenient for big 2-dimension matrix for example Leontief matrix).

What was this blockDim.x? It is the size of the thread block (here 256) and can contain up to 1024 threads.

Can we have multiple blocks? Yes, of course! How much at max? (2^31)-1 O_O

CUDA memory

We are currently using *cudaMallocManaged* to allocate memory for our CUDA program.

Managed memory are available from the *host* (CPU) and the *device* (GPU). But you can make memory available only to the *device* and copy data from the *host*.

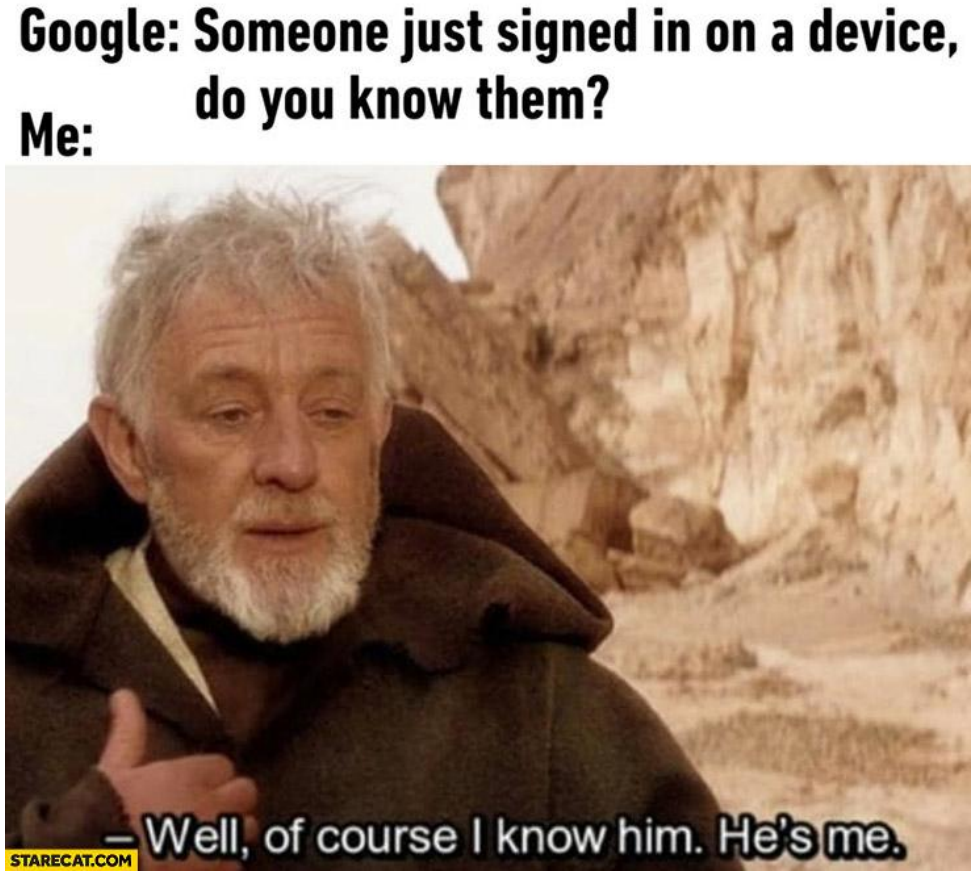
```cpp
//Host memory
float* h_A = (float*)malloc(size);

//Device memory
float* d_A;
cudaMalloc(&d_A, size);
//copy from host to device
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

//Do something on device
...

//Copy back to host from device
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

//Free device memory
cudaFree(d_A);
```



CUDA Download

1. Check you have an Nvidia chipset!
2. https://developer.nvidia.com/ you may need to log-in first
3. download and install CUDA https://developer.nvidia.com/cuda-downloads
   a. Took me 20 min from the mountains ;P

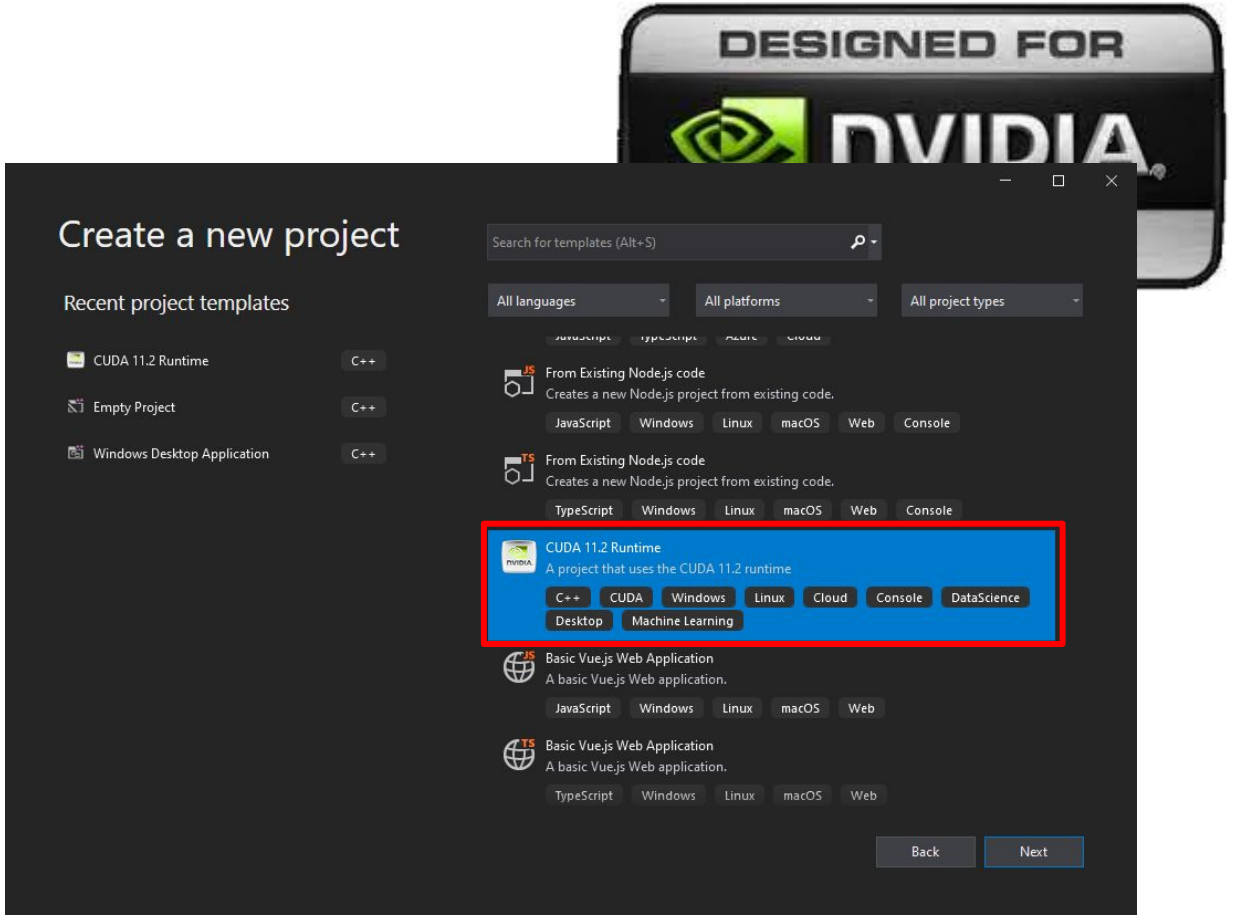
Pause

- Question time!
- You can also play with VS2022!
- Or have a pause

  in case your brain is melting...

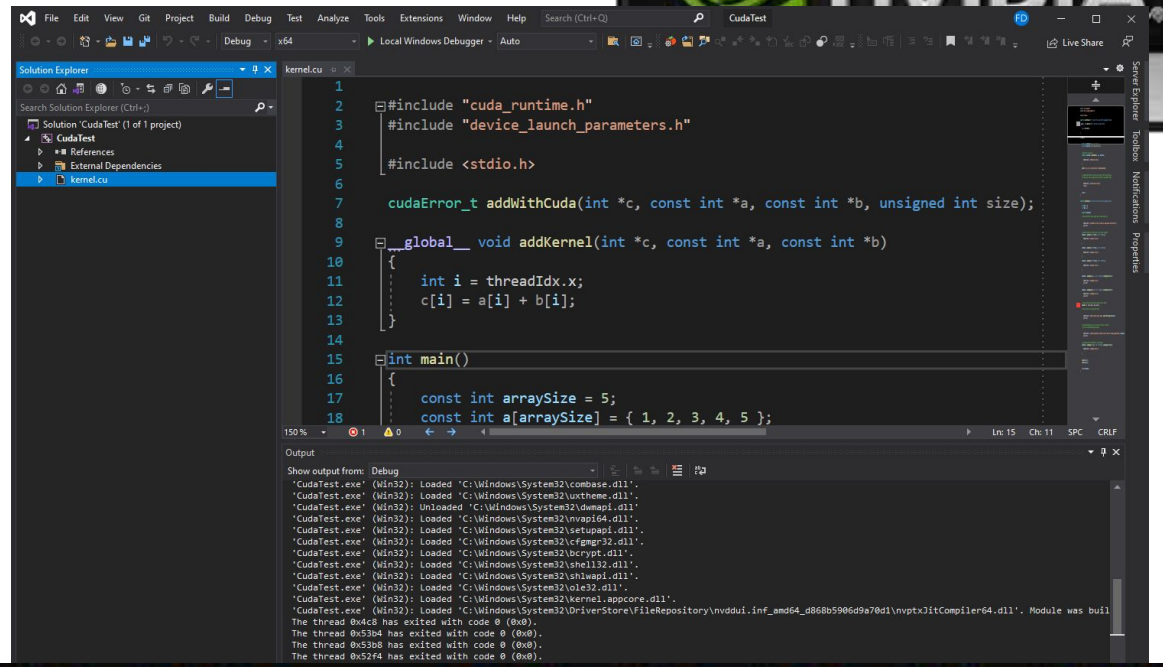
VS 2019 1st test!

1. Open VS 2019
2. Create a new project
3. Select Cuda as default
4. Select a stupid name
5. Validate!



VS 2019 1st test!

1. Build
2. Run!
3. Check the result?!



{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}

CUDA

We have some example running!

- You can inspire yourself from it to create new interesting things!
- We have access to the whole world of CUDA!

VS 2022 Cuda Crash Course

Let's restart from the beginning!

Go to the following repo:

https://github.com/anirul/CUDA_Crash_Course/

VS 2022 CUDA Crash Course

After installing all the required dependencies you can start cloning the
- Clone it to the same repo than the one you installed VCPKG!

```
PS C:\GitHub> dir


    Directory: C:\GitHub


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----           15/01/2025    07:06                CUDA_Crash_Course
d----           15/01/2025    08:11                vcpkg
```

VS 2022 Cuda Crash Course


```
...> git clone
https://github.com/anirul/CUDA_Crash_Course.git
...> cd CUDA_Crash_Course
.../CUDA_Crash_Course> cd Examples
.../Examples> mkdir build
.../Examples> cd build
.../build> cmake ..
-DCMAKE_TOOLCHAIN_FILE="../../../vcpkg/scripts/bui
ldsystems/vcpkg.cmake"
```

VS 2022 Cuda Crash Course

```
-- Generating done (0.1s)
-- Build files have been written to:
C:/GitHub/CUDA_Crash_Course/Examples/build
```

VS 2022 Cuda Crash Course


```
> cmake --build . --config Debug
[...]
  Video.vcxproj ->
C:\GitHub\CUDA_Crash_Course\Examples\build\Video\Release\Video.exe
>
```

Simple

Now let's see the difference between OpenCL and CUDA!

The simple program is kite simple. It just list the number of device and there name and send a program to the device.

Simple

Now let's see the difference between OpenCL and CUDA!


```
int device_count = 0;
cudaGetDeviceCount(&device_count);
[...]
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, d);
[...]
cudaSetDevice(d);
```

Simple

Malloc the memory and copy it to the input buffers


```
cudaMalloc(&d_in1, vector_size * sizeof(float));
cudaMalloc(&d_in2, vector_size * sizeof(float));
cudaMalloc(&d_out, vector_size * sizeof(float));

// 5. Copy data from host to device
cudaMemcpy(d_in1, in1.data(), vector_size * sizeof(float),
   cudaMemcpyHostToDevice);
cudaMemcpy(d_in2, in2.data(), vector_size * sizeof(float),
   cudaMemcpyHostToDevice);
```

Simple

Send the kernel to the device!


```
int blockSize = 256; // typical block size
int gridSize = (vector_size + blockSize - 1) / blockSize;

simpleKernel<<<gridSize, blockSize>>>(
   d_in1, d_in2, d_out, vector_size);

cudaDeviceSynchronize();
```

Simple

Then Free the memory

```
cudaMemcpy(out.data(), d_out, vector_size * sizeof(float),
cudaMemcpyDeviceToHost);

cudaFree(d_in1);
cudaFree(d_in2);
cudaFree(d_out);
```

Histogram

This histogram is mostly the same you split the histogram computation by chunks and make the reduce after by pieces.

Histogram

The interesting part is the kernel calls

```
// 1) Convert to luminosity
{
    dim3 block(blockSize_);
    dim3 grid(gridSize_);
    kernelLuminosity<<<grid, block>>>(d_bgra_, d_lum_, width_, height_);
    CUDA_CHECK(cudaGetLastError());
    CUDA_CHECK(cudaDeviceSynchronize());
}
```

Histogram

The interesting part is the kernel calls


```
// 2) Init partial hist
{
    int length = 256 * numGroups_;
    int initGrid = (length + blockSize_ - 1) / blockSize_;
    kernelInit<<<initGrid, blockSize_>>>(d_part_, length);
    CUDA_CHECK(cudaGetLastError());
}
```

Histogram

The interesting part is the kernel calls


```
// 3) Build partial hist
{
    dim3 block(blockSize_);
    dim3 grid(gridSize_);
    kernelPartial<<<grid, block>>>(
      d_lum_, d_part_, totalSize_, numGroups_);
    CUDA_CHECK(cudaGetLastError());
}
```

Histogram

The interesting part is the kernel calls


```
// 4) Reduce partials into final hist
{
    // 256 threads in one block
    kernelReduce<<<1, 256>>>(d_part_, numGroups_, d_final_);
    CUDA_CHECK(cudaGetLastError());
}
```

Floyd Warshall

I changed the way the kernel is handled, it seams it worked (not fully tested).

```
for (unsigned int k = 0; k < n_; ++k)
{
    dim3 block(blockSizeX_, blockSizeY_);
    dim3 grid((n_ + blockSizeX_ - 1) / blockSizeX_,
              (n_ + blockSizeY_ - 1) / blockSizeY_);

    fw_iteration_kernel<<<grid, block>>>(d_mat_, k, n_);
    CUDA_CHECK(cudaDeviceSynchronize()); // Sync each iteration
}
```

Video

This is a basic reconvertion from the CL path.

No major changes except the use of std::chrono instead of boost also use of abseil instead of boost::program_options.

I even kept the unused variable and member calls.

Conclusion

General purpose programming on the GPU can be useful to solve very big highly parallelizable problem that can compensate the high latency to work with the GPU.

Further exercices

Now we know how to use CUDA to make computing on the GPU!

We can now use it to make Various implementations:

- A game of life
- Julia
- Raytracing (or Ray marching)
- Marching cubes