Why do we care about GPU Programming?
A History of GPU vs. CPU Performance

There have been three ways:
1. Write a graphics display program (≥ 1985)
2. Write an application that looks like a graphics display program, but uses the fragment shader to do some per-node computation (≥ 2002)
3. Write in OpenCL or CUDA, which looks like C++ (≥ 2006)

Due to the nature of graphics computations, GPU chips are customized to handle streaming data.

Another reason is that GPU chips do not need the significant amount of cache space that occupies much of the real estate on general-purpose CPU chips. The GPU die real estate can then be re-targeted to hold more cores and thus to produce more processing power.
Why have GPUs Been Outpacing CPUs in Performance?

Another reason is that general CPU chips contain on-chip logic to do branch prediction and out-of-order execution. This, too, takes up chip die space.

But, GPU chips can handle more general-purpose computing tasks. So, which is better, a CPU or a GPU? It depends on what you are trying to do!

Originally, GPU Devices were very task-specific

Consider the architecture of the NVIDIA Tesla V100’s that we have in our GDX System

What is a “Core” in the GPU Sense?

Look closely, and you’ll see that NVIDIA really calls these “CUDA Cores”

Look even more closely and you’ll see that these CUDA Cores have no control logic — they are pure compute units. (The surrounding SM has the control logic.)

Other vendors refer to these as “Lanes”. You might also think of them as 64-way SIMD.
How Many Robots Do You See Here?

12? 72? Depends what you count as a “robot”.

The Bottom Line is This
So, the Titan Xp has 30 processors per chip, each of which is optimized to do 128-way SIMD. This is an amazing achievement in computing power. But, it is obvious that it is difficult to directly compare a CPU with a GPU. They are optimized to do different things.

So, let’s use the information about the architecture as a way to consider what CPUs should be good at and what GPUs should be good at

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>General purpose programming</td>
<td>Data parallel programming</td>
</tr>
<tr>
<td>Multi-core under user control</td>
<td>Little user control</td>
</tr>
<tr>
<td>Irregular data structures</td>
<td>Regular data structures</td>
</tr>
<tr>
<td>Irregular flow control</td>
<td>Regular Flow Control</td>
</tr>
</tbody>
</table>

So, the Trick is to Break your Problem into Many, Many Small Pieces

1. Have one thread per each particle.
2. Put all of the initial parameters into an array in GPU memory.
3. Tell each thread what the current Time is.
4. Each thread then computes its particle’s position, color, etc. and writes it into arrays in GPU memory.
5. The CPU program then initiates OpenGL drawing of the information in those arrays.

Note: once setup, the data never leaves GPU memory!
What is Fused Multiply-Add?

Many scientific and engineering computations take the form:

\[ D = A + (B \cdot C); \]

A “normal” multiply-add would likely handle this as:

\[ \text{tmp} = B \cdot C; \]
\[ D = A + \text{tmp}; \]

A “fused” multiply-add does it all at once, that is, when the low-order bits of \( B \cdot C \) are ready, they are immediately added into the low-order bits of \( A \) at the same time the higher-order bits of \( B \cdot C \) are being multiplied.

Consider a Base 10 example: \( 789 + (123 \times 456) \)

<table>
<thead>
<tr>
<th>123</th>
<th>738</th>
<th>615</th>
<th>492</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Can start adding the 9 the moment the 8 is produced!

Note: “Normal” \( A+(B \cdot C) \) ≠ “FMA” \( A+(B \cdot C) \)

There are Two Approaches to Combining CPU and GPU Programs

1. Combine both the CPU and GPU code in the same file. The CPU compiler compiles its part of that file. The GPU compiles just its part of that file.
2. Have two separate programs: a .cpp and a .somethingelse that get compiled separately.

**Advantages of Each**

1. The CPU and GPU sections of the code know about each others' intents. Also, they can share common structs, #define's, etc.
2. It's potentially cleaner to look at each section by itself. Also, the GPU code can be easily used in combination with other CPU programs.

Who are we Talking About Here?

We will talk about each of these separately – stay tuned!
### GPU Spec Sheet

<table>
<thead>
<tr>
<th>GPU</th>
<th>Kepler GK100</th>
<th>Maxwell GM200</th>
<th>Pascal GP100</th>
<th>Volta GV100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>3.5</td>
<td>3.5</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Threads / Warps</td>
<td>92</td>
<td>92</td>
<td>92</td>
<td>92</td>
</tr>
<tr>
<td>Max Threads / SM</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Threads / SM</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Threads Block / SM</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Max L1 Instruction Cache / SM</td>
<td>96K</td>
<td>96K</td>
<td>96K</td>
<td>96K</td>
</tr>
<tr>
<td>Max Registers / SM</td>
<td>9655M</td>
<td>9766M</td>
<td>9555M</td>
<td>9555M</td>
</tr>
<tr>
<td>Max Registers / SM</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Max Texture Units</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Pixel Shader Units</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Texture Units</td>
<td>480</td>
<td>480</td>
<td>480</td>
<td>480</td>
</tr>
<tr>
<td>Shared Memory Size / SM</td>
<td>16KB/33KB/48KB</td>
<td>64KB</td>
<td>64KB</td>
<td>Configurable up to 512KB</td>
</tr>
</tbody>
</table>

### Additional Notes
- **Compute Capability**: Indicates the level of compute performance, with higher numbers indicating more powerful capabilities.
- **Threads / Warps**: Shows the maximum number of threads per SM or warp, which is a grouping of threads used for parallel processing.
- **Max Threads / SM**: The maximum number of threads a single SM can handle.
- **Max Texture Units**: The number of texture units, which are used for image processing.
- **Max Pixel Shader Units**: The number of pixel shader units, which are used for rendering pixels on the screen.
- **Texture Units**: A measure of the texture memory bandwidth.
- **Shared Memory Size / SM**: Shows the size of shared memory, which is used for efficient data sharing between threads on the same SM.