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

Note that the top of the graph on the previous page is here.

How Can You Gain Access to GPU Power?

There are 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)

The “Core-Score”. How can this be?
Why have GPUs Been Outpacing CPUs in Performance?

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

Another reason is that general CPU chips contain on-chip logic to process instructions out-of-order if the CPU is blocked and is waiting on something (e.g., a memory fetch). This, too, takes up chip die space.

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

But, CPU 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

Today’s GPU Devices are much less task-specific

Consider the architecture of the NVIDIA Titan Black that we have in our rabbit System

15 Streaming Multiprocessors (SMs) / chip
192 cores / SM
Wow, 2880 cores / chip? Really?
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 192-way SIMD.

A Mechanical Equivalent...

Streaming Multiprocessors

128 CUDA Cores

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

How Many Robots Do You See Here?

A Spec Sheet Example

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.

The Bottom Line is This

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>

BTW, The general term in the OpenCL world for an SM is a Compute Unit. The general term in the OpenCL world for a CUDA Core is a Processing Element.

Compute Units and Processing Elements are Arranged in Grids

A GPU Platform can have one or more Devices. A GPU Device is organized as a grid of Compute Units.

Each Compute Unit is organized as a grid of Processing Elements.

So in NVIDIA terms, their new Turing GPU has 68 Compute Units, each of which has 64 Processing Elements, for a grand total of 4,352 Processing Elements.
Thinking ahead to OpenCL…

How can GPUs execute General C Code Efficiently?

- Ask them to do what they do best. Unless you have a very intense Data Parallel application, don’t even think about using GPUs for computing.
- GPU programs expect you to not just have a few threads, but to have thousands of them!
- Each thread executes the same program (called the kernel), but operates on a different small piece of the overall data
- Thus, you have many, many threads, all waking up at about the same time, all executing the same kernel program, all hoping to work on a small piece of the overall problem.
- OpenCL has built-in functions so that each thread can figure out which thread number it is, and thus can figure out what part of the overall job it’s supposed to do.
- When a thread gets blocked somehow (a memory access, waiting for information from another thread, etc.), the processor switches to executing another thread to work on.

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

Particle Systems are a great example.
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!

Something New – Tensor Cores

Tensor Cores Accelerate Fused-Multiply-Add Arithmetic

NVIDIA Volta

• GPUs were originally designed for the streaming-ness of computer graphics
• Now, GPUs are also used for the streaming-ness of data parallel numeric computing
• GPUs are better for some things. CPUs are better for others.