How Have You Been Able to Gain Access to GPU Power?

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)

Why do we care about GPU Programming?

A History of GPU vs. CPU Performance

Due to the nature of graphics computations, GPU chips are customized to stream regular data. General CPU chips must be able to handle irregular 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.
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, 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. Consider the architecture of the NVIDIA Tesla V100's Streaming Multiprocessors (SMs) / chip. 64 cores / SM. Wow, 5,396 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 64-way SIMD.

A Mechanical Equivalent…
How Many Robots Do You See Here?

12? 72? Depends what you count as a "robot".

A Spec Sheet Example

<table>
<thead>
<tr>
<th>Node Product</th>
<th>CUDA Cores per SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla V100</td>
<td>84</td>
</tr>
<tr>
<td>Tesla K80</td>
<td>70</td>
</tr>
<tr>
<td>Tesla K20X</td>
<td>82</td>
</tr>
<tr>
<td>Tesla K20</td>
<td>84</td>
</tr>
<tr>
<td>Tesla K40</td>
<td>84</td>
</tr>
<tr>
<td>Tesla K20X</td>
<td>82</td>
</tr>
<tr>
<td>Tesla K80</td>
<td>70</td>
</tr>
<tr>
<td>Tesla V100</td>
<td>84</td>
</tr>
</tbody>
</table>

The Bottom Line is This

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>

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.

Thinking ahead to CUDA and 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.

• CUDA and OpenCL have 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

D = A + (B*C);
A “normal” multiply-add would likely handle this as:
tmp = B*C;
D = A + tmp;
A “fused” multiply-add does it all at once, that is, when the low-order bits of B*C are ready, they are immediately added into the low-order bits of A at the same time the higher-order bits of B*C are being multiplied.

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

<table>
<thead>
<tr>
<th>123</th>
<th>x 456</th>
</tr>
</thead>
<tbody>
<tr>
<td>738</td>
<td></td>
</tr>
<tr>
<td>615</td>
<td></td>
</tr>
<tr>
<td>492</td>
<td></td>
</tr>
<tr>
<td>+ 789</td>
<td></td>
</tr>
<tr>
<td>56,877</td>
<td></td>
</tr>
</tbody>
</table>

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

Note: “Normal” A+(B*C) ≠ “FMA” A+(B*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 compiler 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?
1 = NVIDIA’s CUDA
2 = Khronos’s OpenCL

Looking ahead: If threads all execute the same program, what happens on flow divergence?

if( a > b )
Do This;
else
Do That;
1. The line “if( a > b )” creates a vector of Boolean values giving the results of the if-statement for each thread. This becomes a “mask”.
2. Then, the GPU executes all parts of the divergence:
   Do This:
   Do That;
3. During that execution, anytime a value wants to be stored, the mask is consulted and the storage only happens if that thread’s location in the mask is the right value.
- GPUs were originally designed for the streaming-ness of computer graphics
- Now, GPUs are also used for the streaming-ness of data-parallel computing
- GPUs are better for some things. CPUs are better for others.

Dismantling a Graphics Card

This is an Nvidia 1080 Ti card – one that died on us. It willed its body to education.

Removing the covers:

Removing the heat sink:

This transfers heat from the GPU Chip to the cooling fins

Removing the fan assembly reveals the board:

Power half of the board:
Dismantling a Graphics Card

Graphics half of the board:

- **GPU Chip**
  - This one contains 7.2 billion transistors!
  - (Thank you, Moore’s Law)

Underside of the board:

- Video out

Here is a fun video of someone explaining the different parts of this same card: https://www.youtube.com/watch?v=dSCNf9DIBGE

Bonus – Looking at More Complete GPU Spec Sheet

<table>
<thead>
<tr>
<th>GPU</th>
<th>Kepler GTX80</th>
<th>Maxwell-GK200</th>
<th>Pascal-GP100</th>
<th>Volta-GV100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>3.5</td>
<td>3.2</td>
<td>2.0</td>
<td>7.0</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Wgls / 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 Shad Regs / SM</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Regs / Block</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Thread</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Max Thread Blocks / SM</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Ratio of SM to FP32 Cores</td>
<td>1.25</td>
<td>1.25</td>
<td>1.25</td>
<td>1.25</td>
</tr>
<tr>
<td>Shared Memory Size / SM</td>
<td>8 KB</td>
<td>8 KB</td>
<td>8 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td>Configurable up to 512 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bonus – Looking at More Complete GPU Spec Sheet

In addition to the provided specifications:

- **CUDA Cores**: 3752, 6080, 6344, 6344 (2008)
- **Tensor Cores**: 16, 32, 32, 64 (2008)
- **Boost Clock**: 1860 MHz, 1900 MHz, 1900 MHz, 1900 MHz (2008)
- **Memory Clock**: 12580 MHz (2008), 14580 MHz (2008)
- **Memory Bus Width**: 384-bit, 512-bit, 384-bit, 384-bit (2008)
- **ECC**: Single Parity, Double Parity, Double Parity, Double Parity (2008)
- **Tensor Performance**: 390 TFLOPS, 712 TFLOPS, 712 TFLOPS, 712 TFLOPS (2008)
- **TOF**: 0.53, 0.53, 0.53, 0.53 (2008)
- **Power**: 149W, 149W, 149W, 149W (2008)
- **Cooling**: Air, Air, Water, Water (2008)
- **Netflix**: Yes, Yes, Yes, Yes (2008)
- **GPU**: GA402, GA402, GA304, GA304 (2008)
- **Architecture**: Ampere, Ampere, Ampere, Ampere (2008)
- **Manufacturing Process**: 7nm, 7nm, 6nm, 6nm (2008)
- **Launch Price**: $499, $699, $999, $999 (2008)