Australian National University

Using GPUs to Accelerated Computational Performance

Dr Eric McCreath

Research School of Computer Science

The Australian National University

Overview

- GPU Architecture
- SIMT
- Kernels
- Memory
- Intermediate representations and runtimes
- "Hello World" - OpenCL
- "Hello World" - Cuda
- Lab Activity

Progress?

What has changed in the last 20 years in computing?

Me - ~1998

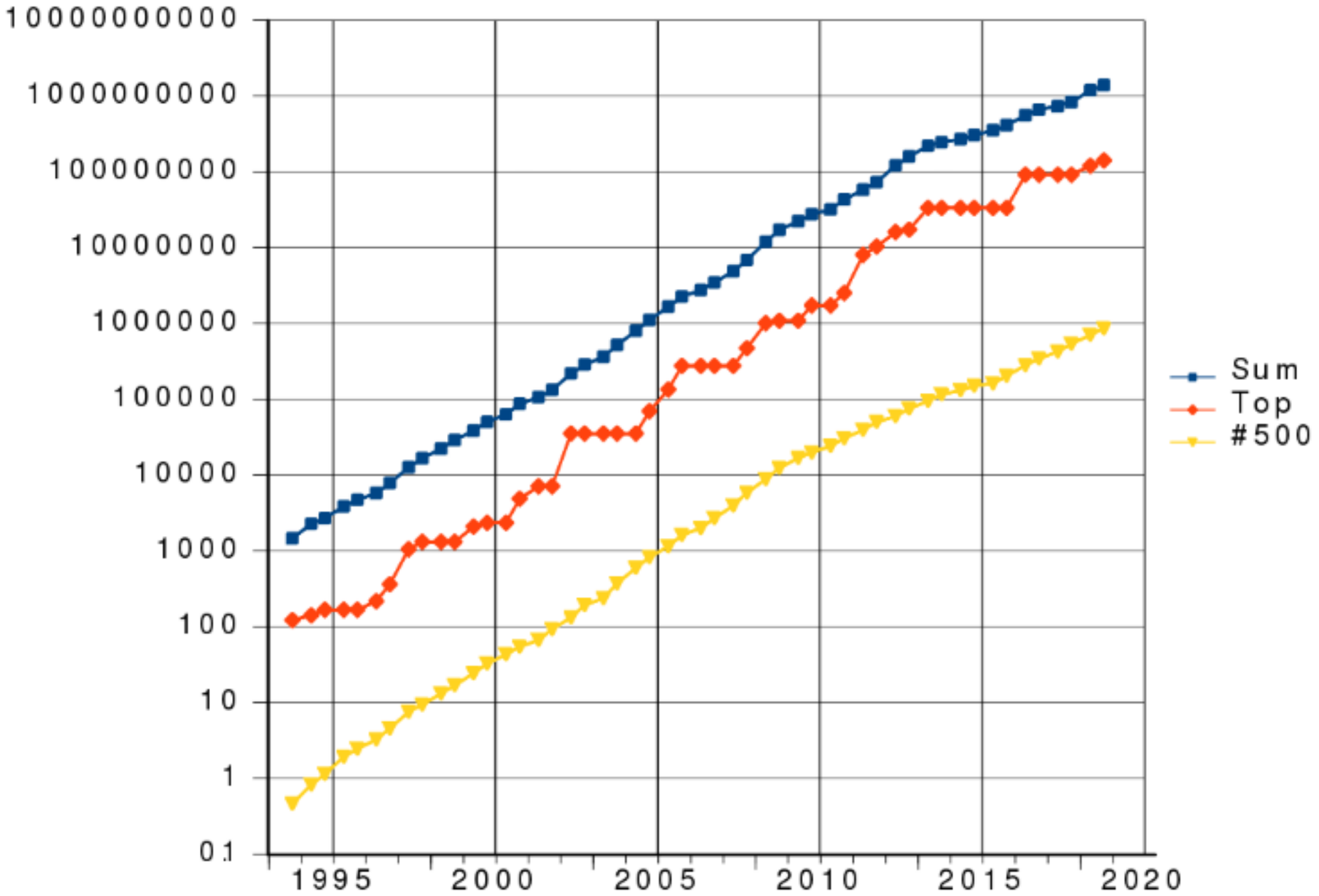
Me - more recently

Australian National University

GEForce

Super Computer Performance

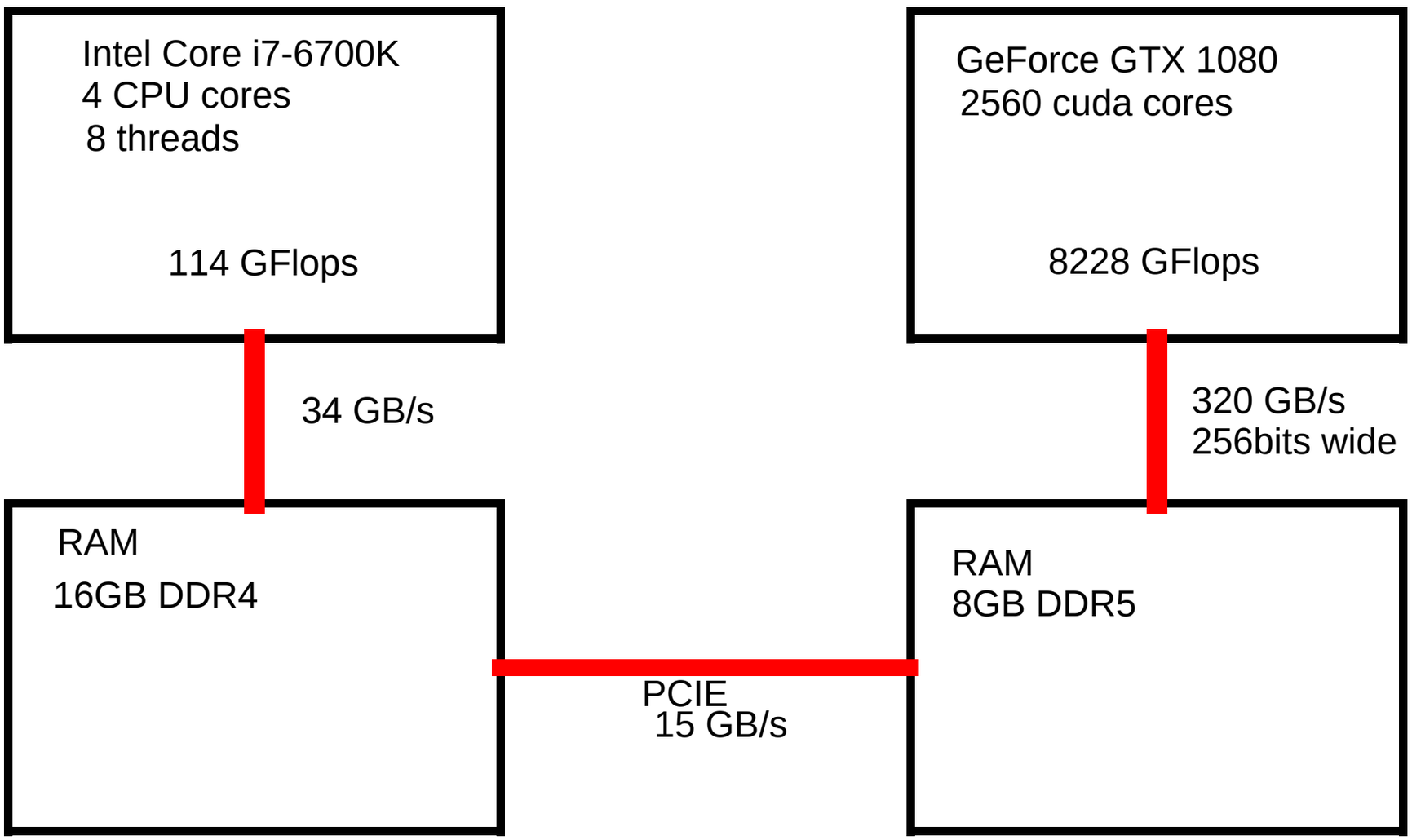
Rapid growth of supercomputer performance, based on data from top500.org site. The logarithmic y-axis shows performance in GFLOPS.



By AI.Graphic - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=33540287

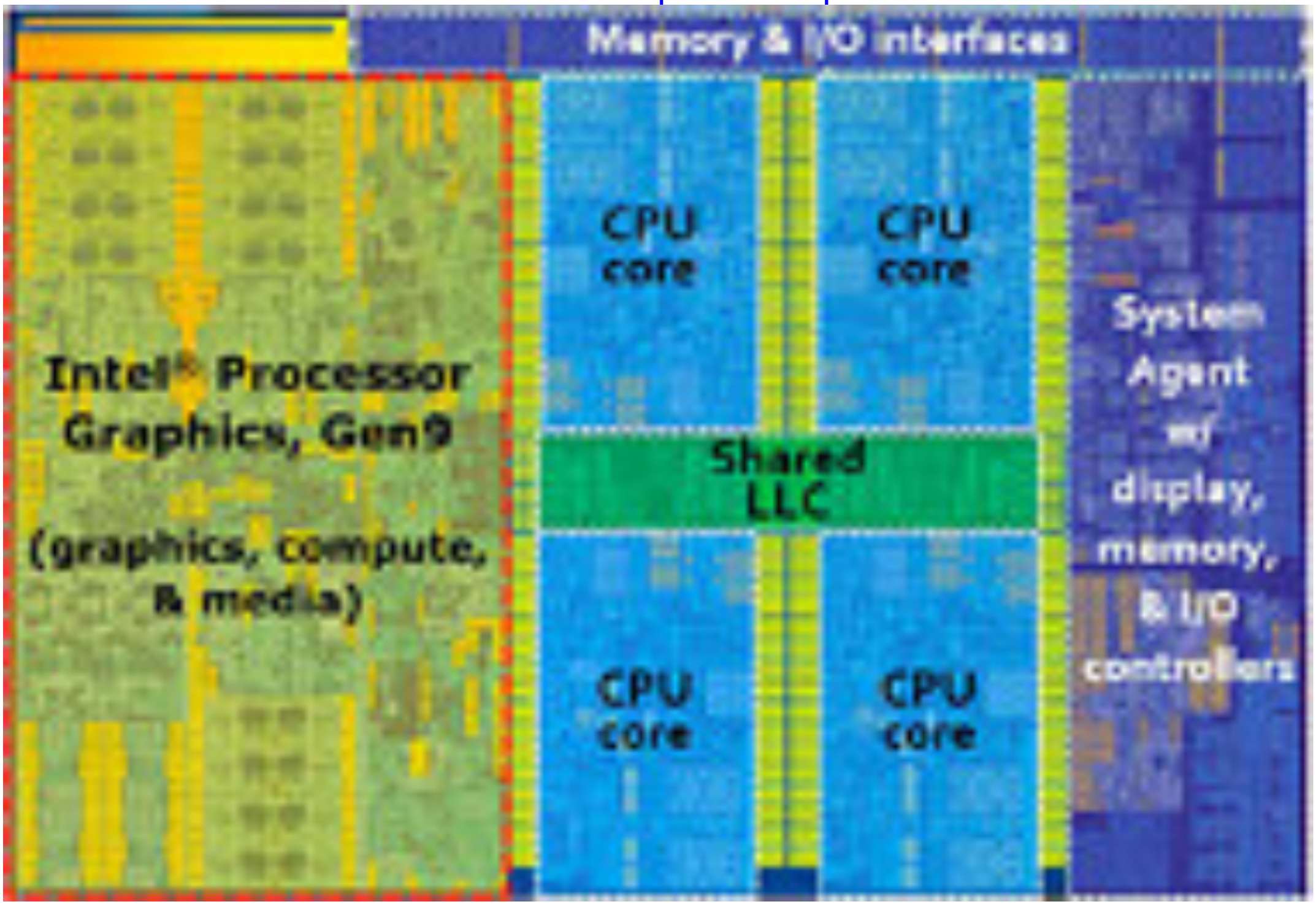
GPU vs CPU

- Just looking at the specs of a basic desktop computer we can see great potential in GPU computing.



Inside a CPU

The Core i7-6700K quad-core processor

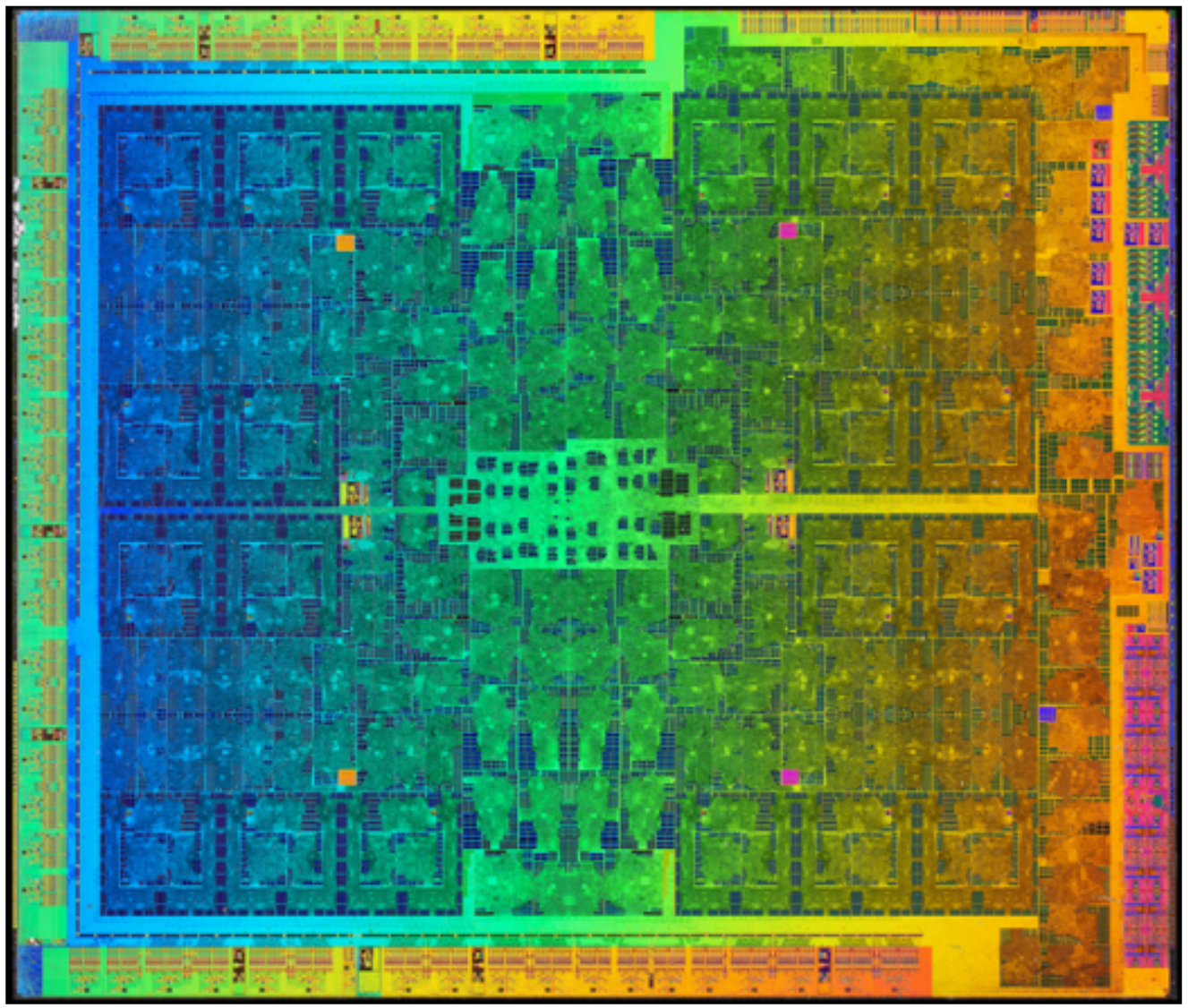


From https://www.techpowerup.com/215333/intel-skylake-die-layout-detailed

Inside the GPU

If we take a closer look inside a GPU we see some similarity with the CPU, although more repetition that comes with the many more cores.

GTX1070 - GP104 - Pascal



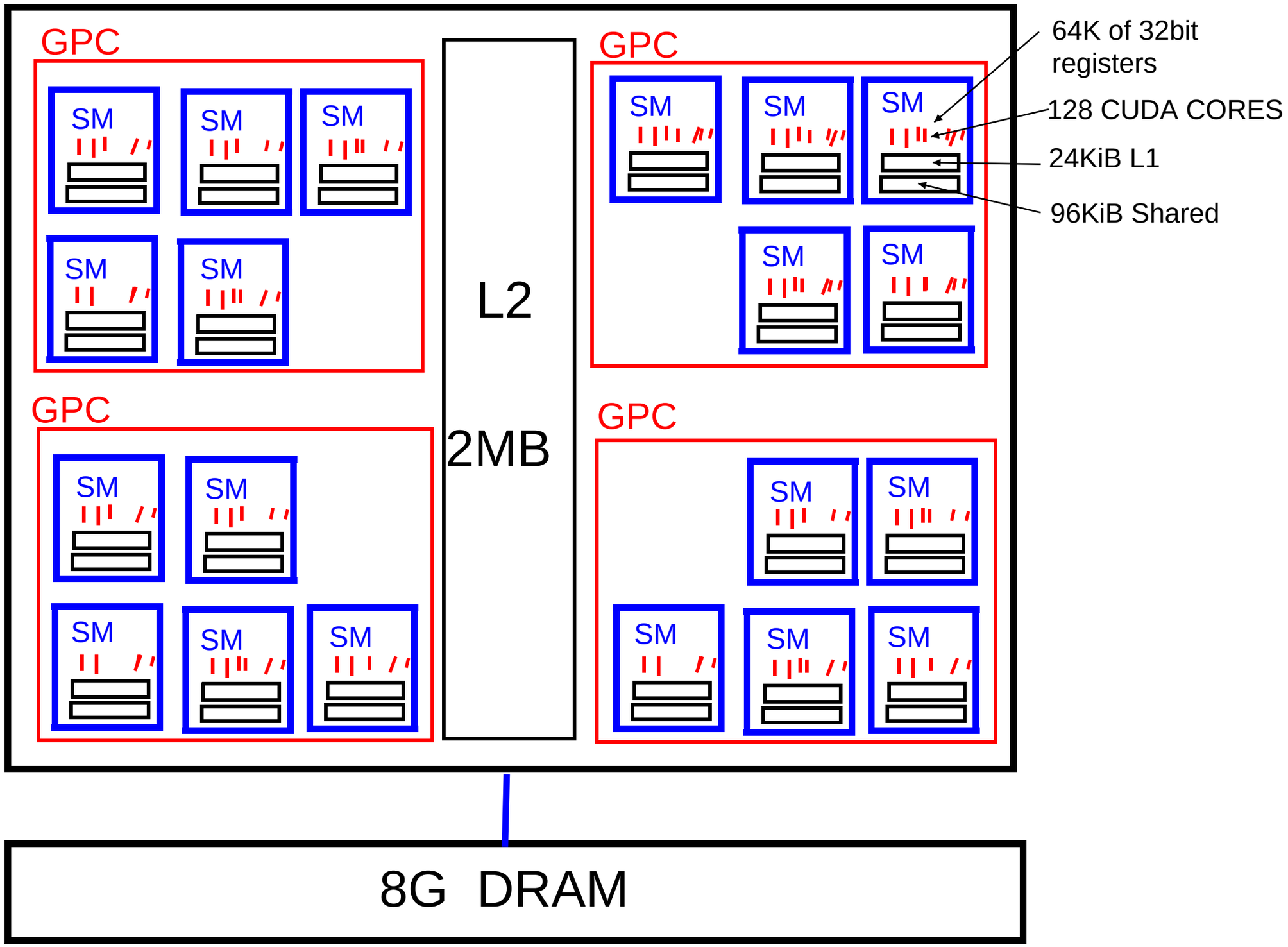
From https://www.flickr.com/photos/130561288@N04/36230799276 By Fritzchens Fritz Public Domain

Key Parts Within a GPU

- Nvidia GPUs chips are partitioned into Graphics Processor Clusters (GPCs). So on the GP104 there is 4 GPCs.
- Each GPC is again partitioned into Streaming Multiprocessors (SMs). On the GP104 there is 5 SMPs per GPC.
- Each SM has "CUDA" cores which are basically ALU units which can execute SIMD instructions. On the GP104 there is 128 CUDA cores per SMs.
- On the GP104 each SMP has 24KiB of Unified L1 cache/texture cache and 96K of "shared memory".
- The GP104 chip has 2048KiB of L2 cache.

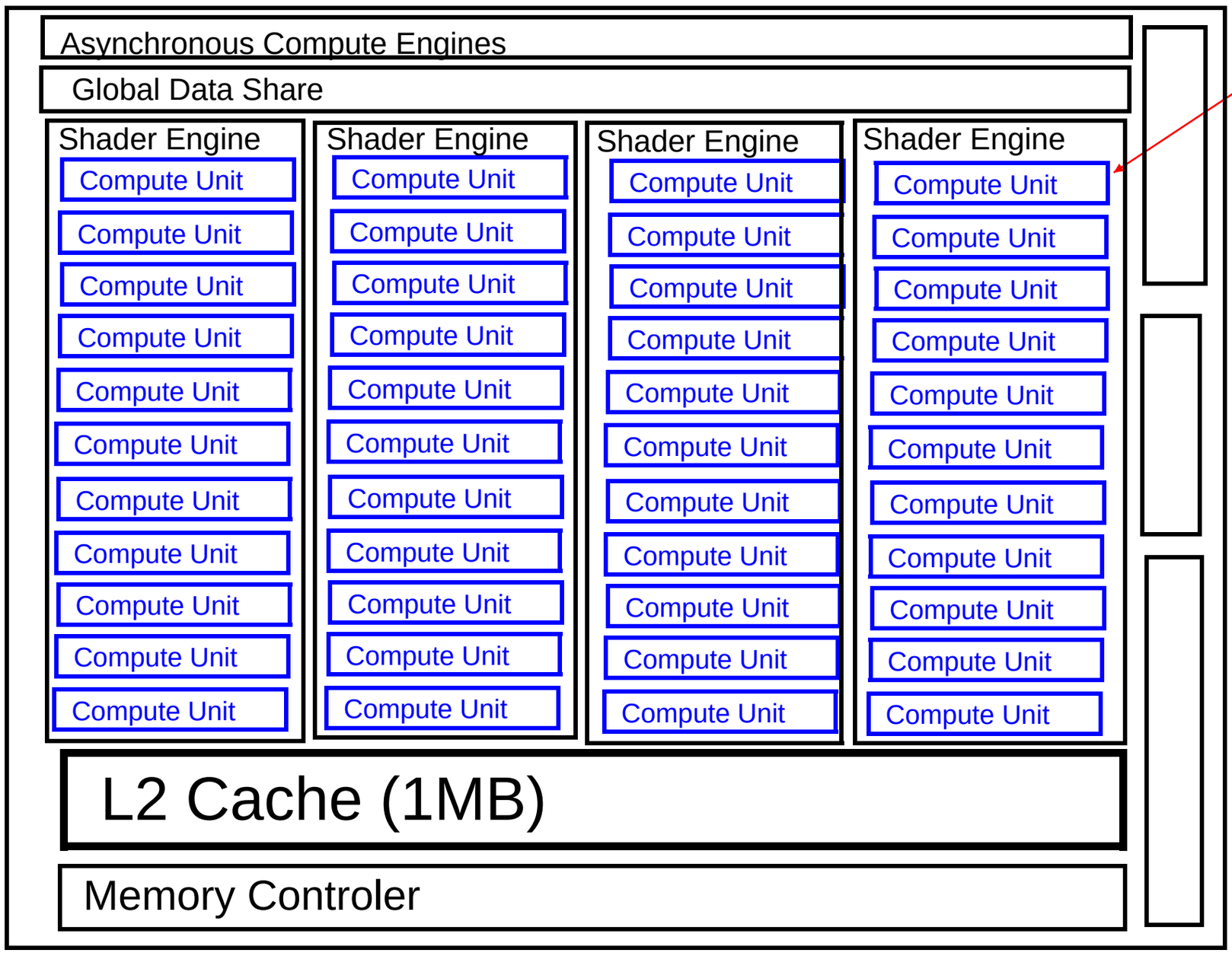
I think we need a diagram!!

Key Parts Within A GPU



AMD

If we had a look at an AMD GPU we would see something similar.
So the Radeon R9 290 series block diagram is:



Each compute unit has:
64 stream processors
4*64KB vector registers
64KB local shared data
16KB L1 Cache
texture and scheduler components

Some Terminology

- CUDA (Compute Unified Device Architecture) is Nvidia's programming model and parallel programming platform developed by Nvidia for there GPU devices. It comes with its own terminology.
- The stream multiprocessor (SM) is a key computational grouping within a GPU, although "stream multiprocessor" is Nvidia's terminology. AMD would call them "compute units".
- Also "CUDA cores" would be called "shader units" or "stream processors" by AMD.

Kernels

Kernels are the small pieces of code that execute in a thread (or work-item) on the GPU. They are written in *c* . For a single kernel one would normally launch many threads. Each thread is given the task of working on a different data item (data parallelisim).

In CUDA kernels have the "__global__" compiler directive before them, they don't return anything (type void), parameters can be basic types, structs, or pointers. Below is a simple kernel that adds one to each element of an array.

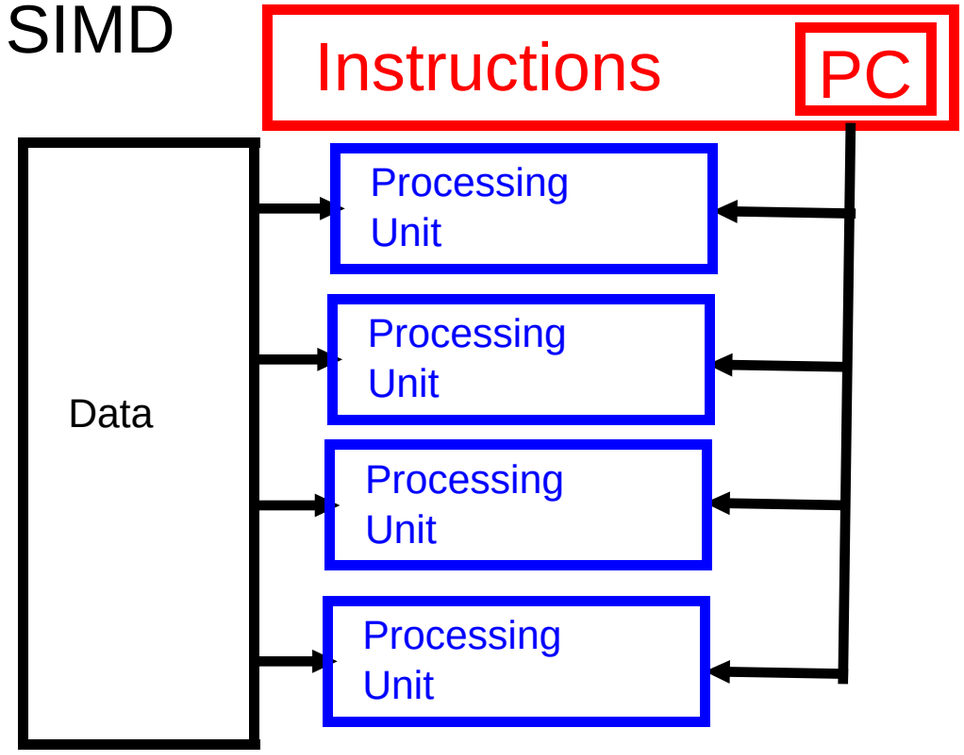
```
__global__ void addone(int n, int *data) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) data[idx] = data[idx] + 1;
}
```

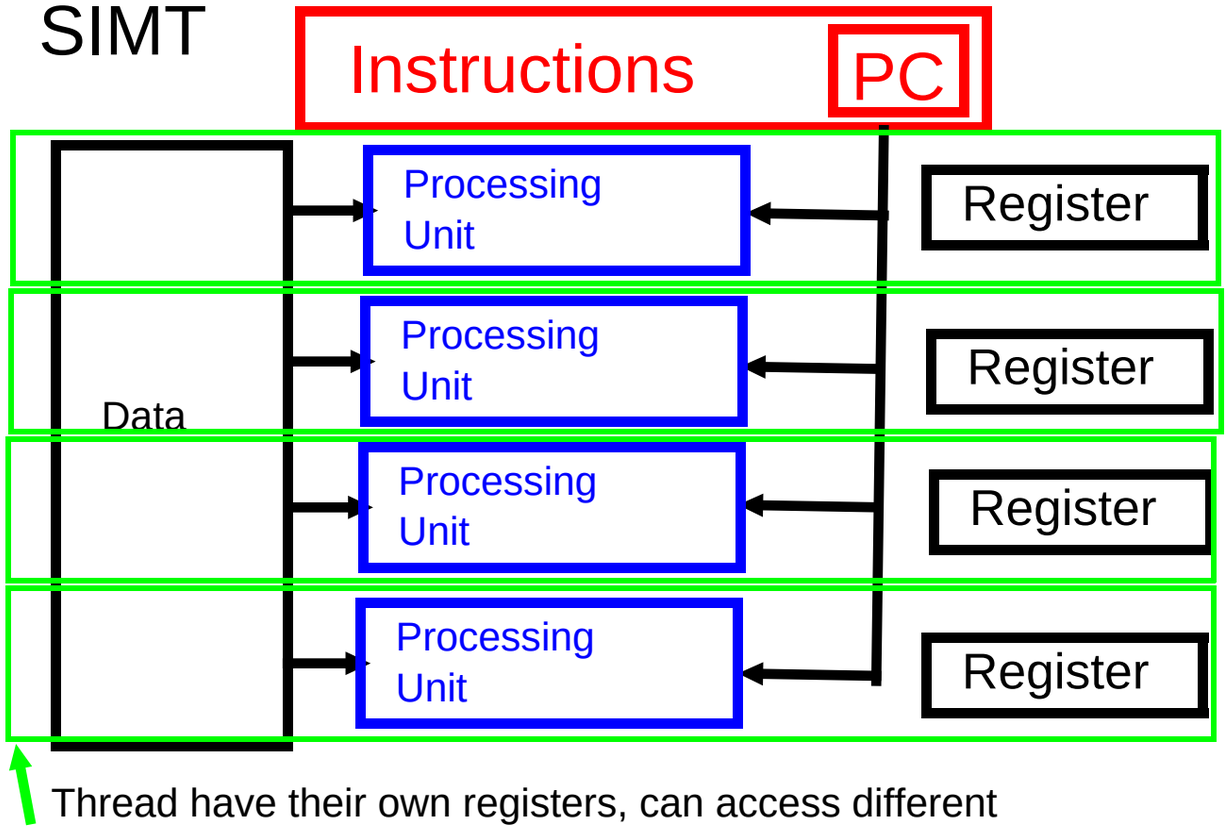
To launch this kernel with 10 blocks with 256 thread per block you would:

```
addone<<<10,256>>>(n, data);  // "n" is the number of items in the array "data"
```

SIMT

Single Instruction Multiple Data (SIMD), describle by Flynn in 1966, and typically has single instructions operate on a vectors of data items. This saves on duplicating the instruction execution hardware and the memory has good spatial locality. GPUs have an extension on this called Single Instruction Multiple Thread (SIMT), this provides more context for each of these 'threads'.





Thread have their own registers, can access different addresses, and can follow divergent paths in the code.

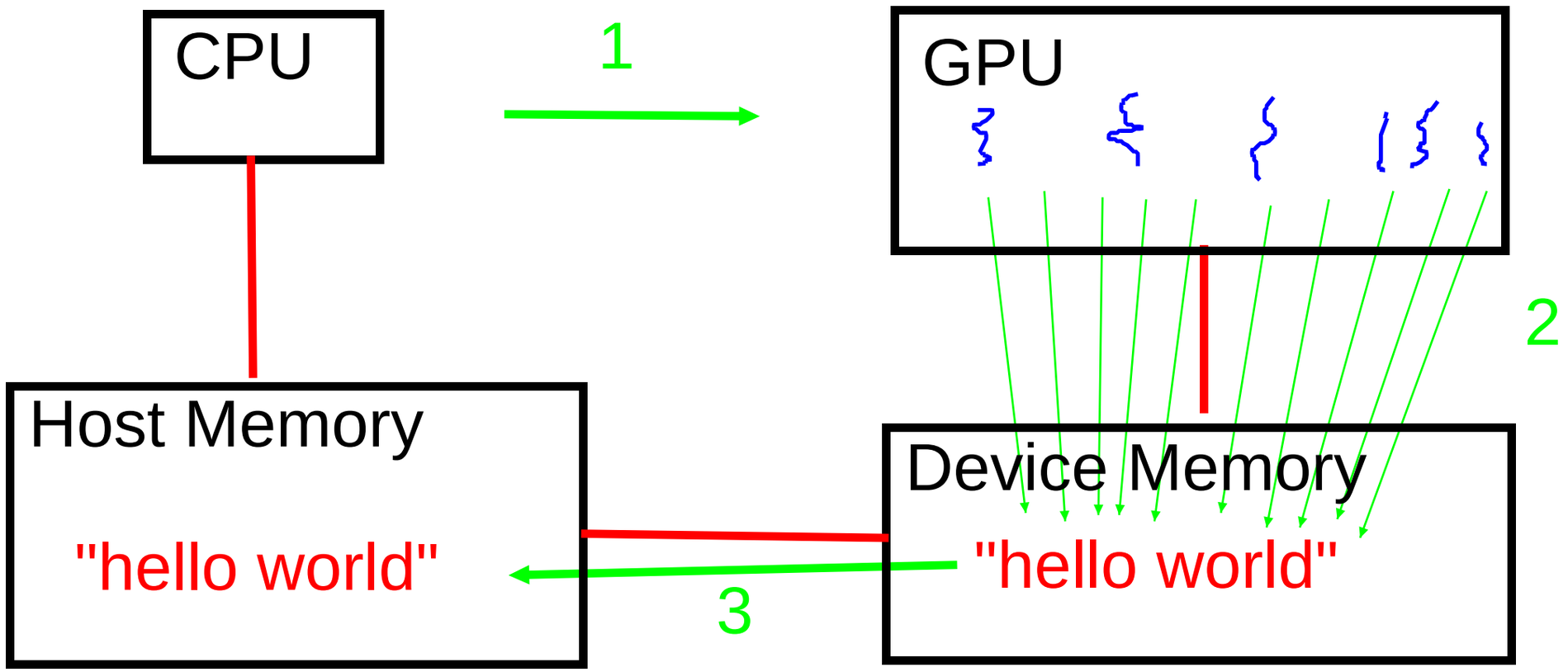
Memory

Memory bandwidth and latency can often significantly impact performance so one of the first performance considerations or questions when porting a program to the GPU is: Which memory to use and how to best use this memory. Memory is described by its scope from the threads perspective. The key memory types to consider are:

- registers - fast and local to threads.
- shared memory - fast memory that is shared within the block (local memory in OpenCL).
- global memory - this is main memory of the GPU, it is accessible to all threads in all blocks and persists over the execution of the program.
- constant memory - can't change over kernel execution, great if threads all want to access the same constant information.

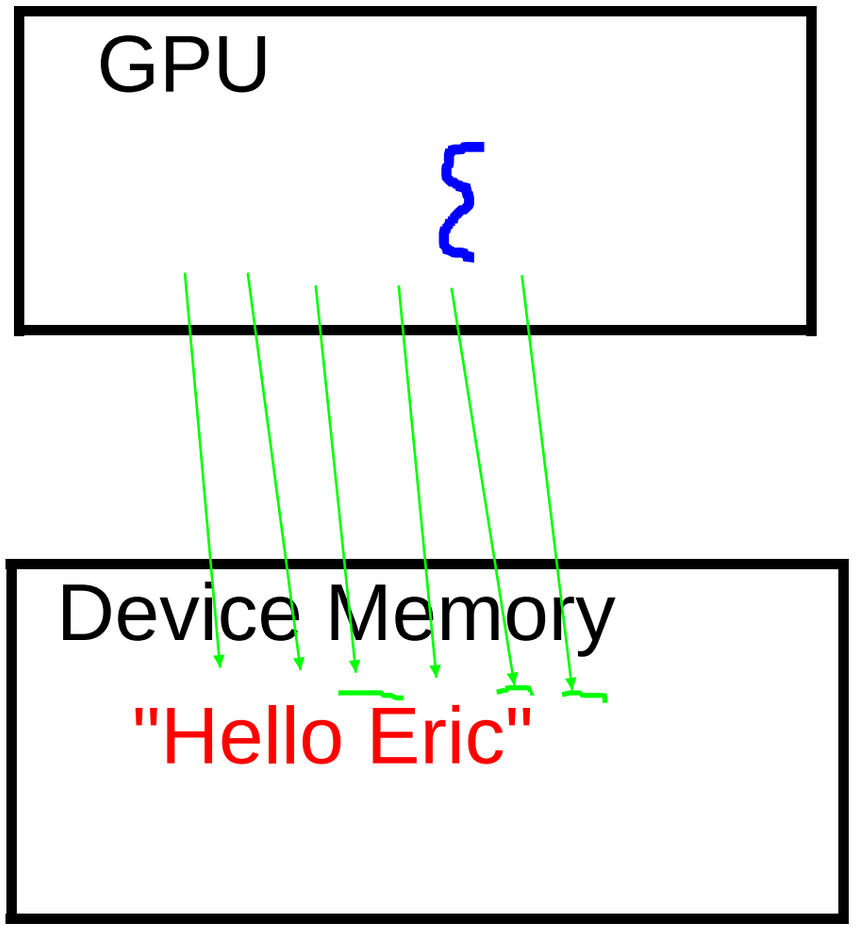
"Hello World" - OpenCL

- So in this implementation of "Hello World" we are getting the GPU to do the work of generating the string in parallel. So a single thread does the work of outputing a single character in the string we output.



Overview Of Lab Activity

Bascially in this first lab you will have a go compiling and run the code. And then make a small modification to the "hello world" programs. This involves make add your name to the "hello" and also making 1 thread be copy over 2 characters, rather, than just the one.



References

- Flynn's taxonomy https://en.wikipedia.org/wiki/Flynn's_taxonomy
- Using CUDA Warp-Level Primitives, Lin and Grover, https://devblogs.nvidia.com/using-cuda-warp-level-primitives/
- Cuda C Programming Guide, https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
- Benchmarking the cost of thread divergence in CUDA, Bialas and Strzelecki, https://arxiv.org/pdf/1504.01650.pdf