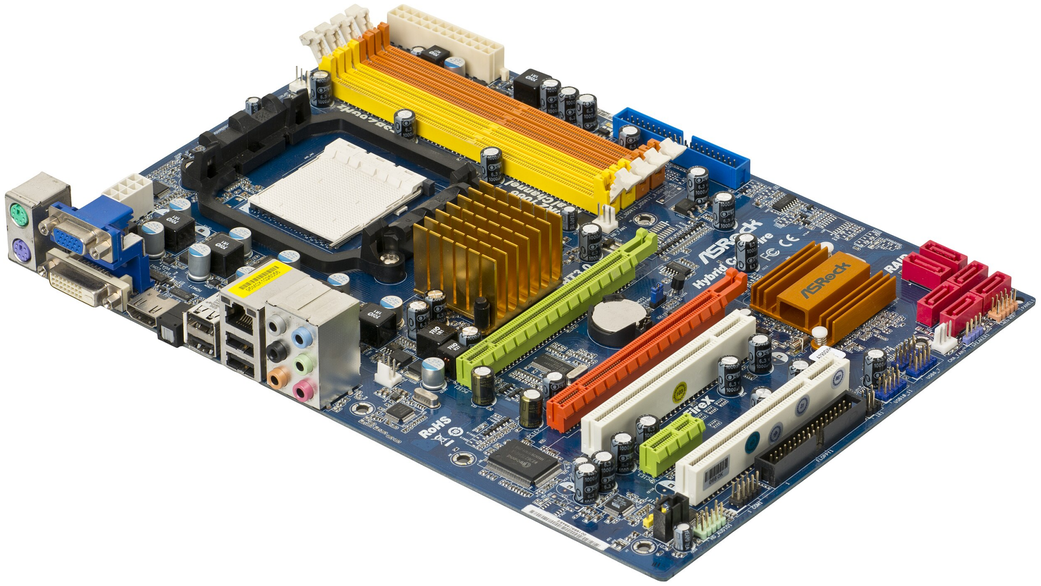
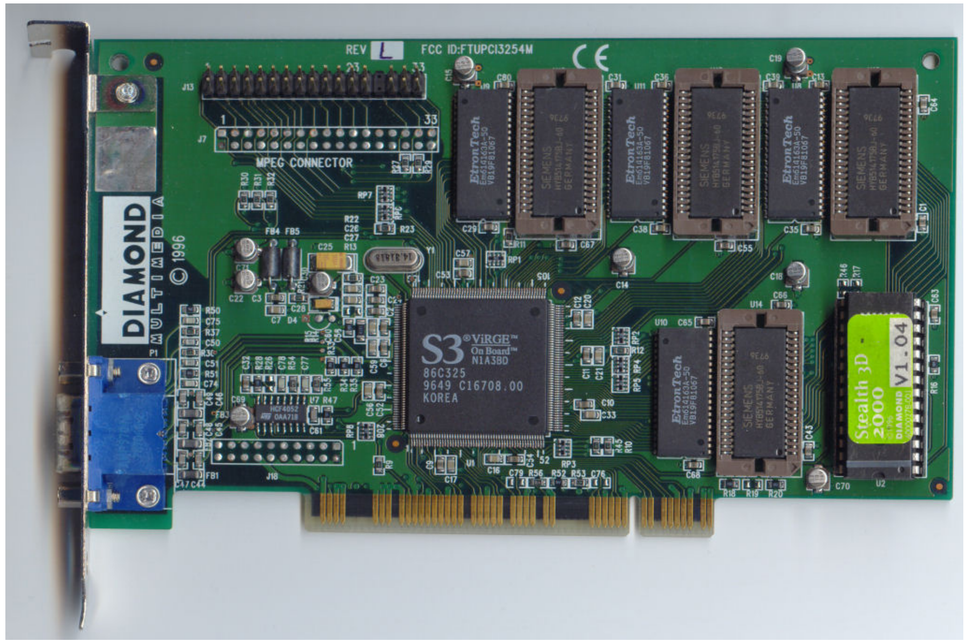
A (Brief) Overview of GPUs

HW3 due Monday at midnight;
Colloquium at Mudd today!



ASRock Motherboard
(integrated graphics)



S3 ViRGE

Image credits: https://en.wikipedia.org/wiki/Graphics_processing_unit

Outline

- Brief overview of data-level parallelism
- From vector processors to GPUs
- GPU programming

The (Way too Brief) Origin of GPUs

Westinghouse

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Can we get more compute by adding more ALUs to the system, where each ALU has distinct data to work on?

Graphics require extensive simultaneous parallel data operations! Extend the ISA with single-instruction multiple data (SIMD) instructions

Graphics accelerators become mainstream extensions to commodity devices (attached via I/O); often uses SIMT (thread) instruction extensions

1960s

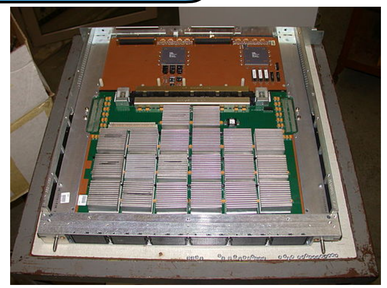


Image credit: https://en.wikipedia.org/wiki/Vector_processor

1996

Image credit: https://en.wikipedia.org/wiki/MMX_(instruction_set)

2000s

From our textbook: “GPUs and CPUs do not go back in computer architecture genealogy to a common ancestor; there is no “missing link” that explains both”

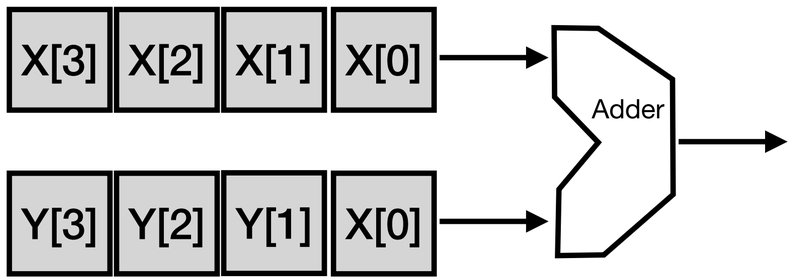
The Key Idea of Vector Processing

- Any application that requires linear algebra-like routines is going to exhibit “DAXPY”-like behavior
- If this is the case, then there are no data dependences between operations ➡️ the operation is *vectorizable*
- Each element of the vectorizable instruction can be computed individually and then the result can be aggregated ➡️ this algorithmic technique is called *scatter-gather*

DAXPY

$Y = a * X + Y$

```
for (int i = 0; i < n; i++) {
    Y[i] = a * X[i] + Y[i];
}
```



Chat with your neighbor(s)!

Suppose we wanted to implement an instruction for “vector add” that takes two *vectors* `A` and `B` as inputs and produces a *vector* as output. How would the instruction be constructed? Where would `A` and `B` be stored? How would the compute units be utilized?

Just like before, we can encode vector instructions as an instruction in the ISA with its own opcode and inputs

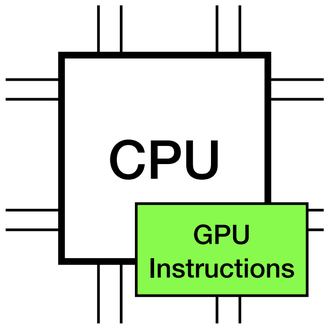
If we want to store an entire vector in a register, then we will need a special type of vector register in which multiple data can be stored!

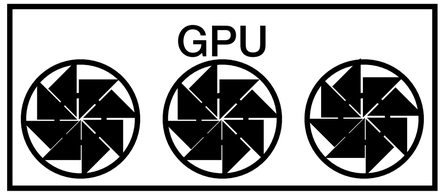
Once data is fetched from the special “vector register”, the contents are split (wires redirected) to the various compute units!

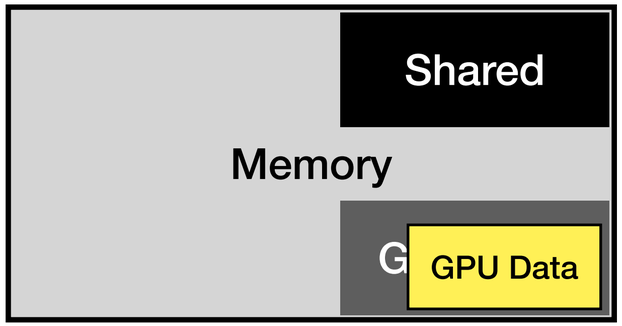
Data Level Parallelism

- The notion of encoding several variables (e.g., elements within a vector) to be processed in parallel is referred to as *data level parallelism*
- In general, the most common type of data-level parallelism is *single-instruction multiple-data* (SIMD) ➡️ the common example of this is to add the elements of two vectors together
- To construct a vector register, data needs to be *contiguous* in memory so that a small number of loads can retrieve all of the data to a single register ➡️ vector registers are often larger than a data word!
- Because fetching data to a large vector register is larger than a word, it often requires several fetches to memory to load into a single register ➡️ memory bandwidth has been and remains a bottleneck!

The GPU Programming Model







In NVIDIA's CUDA language, GPU code written like C code that is distributed across *threads*

The inputs to these C functions are passed explicitly as inputs and outputs to the *driver*, which is implemented in the operating system

The data (and thread spawning) is organized into *blocks* and *grids*

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply(float *dest, float *a, float *b) {
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest  = numpy.zeros_like(a)
mod.get_function("multiply")(drv.Out(dest), drv.In(a), drv.In(b), block=(400, 1, 1), grid=(1, 1))

print(dest)
```

The Link Between GPUs and Neural Networks

- Neural networks can be encoded as matrices ➡️ each layer of the network as a dimension of the matrix, and the data at that index represents the weight of the connection
- If we have large models with lots of connections, then we need to perform a large number of matrix multiplication operations
- In general, if the data is encoded as a floating point value, then we can measure the overall performance of the GPU as the number of FLOPS (float point operations per second)

Chat with your neighbor(s)!

What are some of the performance bottlenecks for processing on a GPU? Think about the construction and organization of the data, how data is communicated around the system, etc…

Takeaways

- GPUs depend on large degrees of parallelizability and vectorizability in the development of an application
- To communicate data between a processor and GPU, there needs to be a special instruction set for instructions to be compiled down to and a location for these instructions to be read from
- Bottlenecks in GPU computation come from data communication and lack of alignment in data layouts
- Parallelizability in the GPU leverages the lack of dependencies between computations in deriving a larger goal!