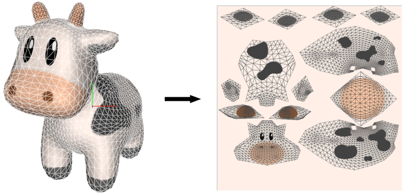
High-performance Data Analysis with GPUs

KSETA Topical Courses March 2021

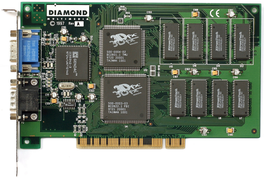
Markus Götz

Karlsruhe Institute of Technology (KIT) / 2021-03-11

Graphical Processing Unit (GPU)
A Brief History Primer



Source: Warren Moore, "Textures and Samples in Metal", 2014.

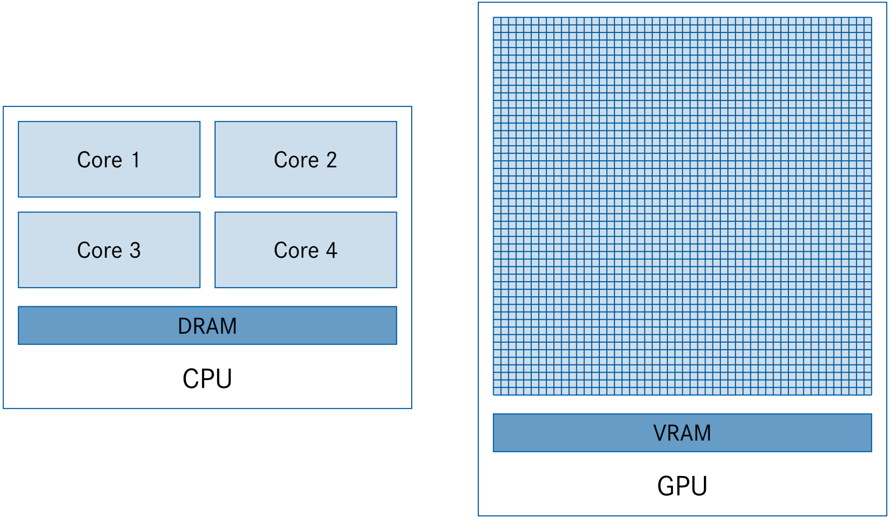


Source: Konstantin Lanzet, Wikipedia, "3dfx Voodoo Graphics"

- Early 1990s **real-time rendering**
  - Highly computational intensive
  - CPU did all the work
- Introduction of **dedicated hardware**
  - NVidia coined the term **GPU**
  - Short and repeated algorithms
  - Massively parallel vector operations
- Repurpose for “regular” problems
  - Express problem as an image
  - Flexible programming advancements, e.g. non-fixed function pipelines

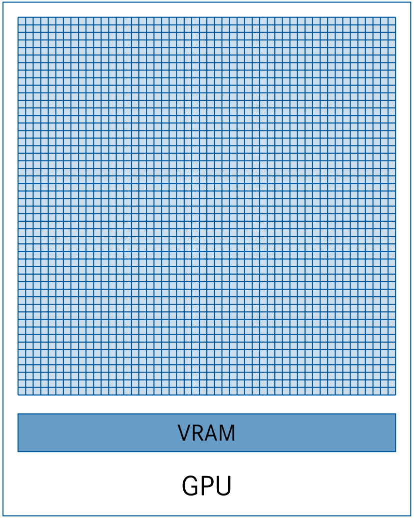
Graphical Processing Unit (GPU)

CPUs versus GPUs



Graphical Processing Unit (GPU)

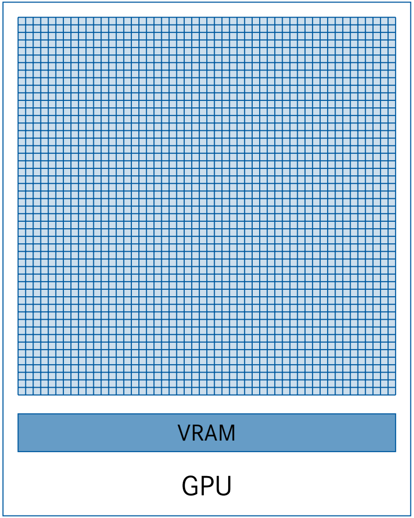
CPUs versus GPUs



- **CPU**—Central Processing Unit
  - Low number of strong cores
  - Independent and vector computation
  - Low to medium parallelization
- **GPU**—Graphical Processing Unit
  - High number of weak cores
  - Designed for vector operations
  - Lock-step mode
  - High parallelization

Graphical Processing Unit (GPU)

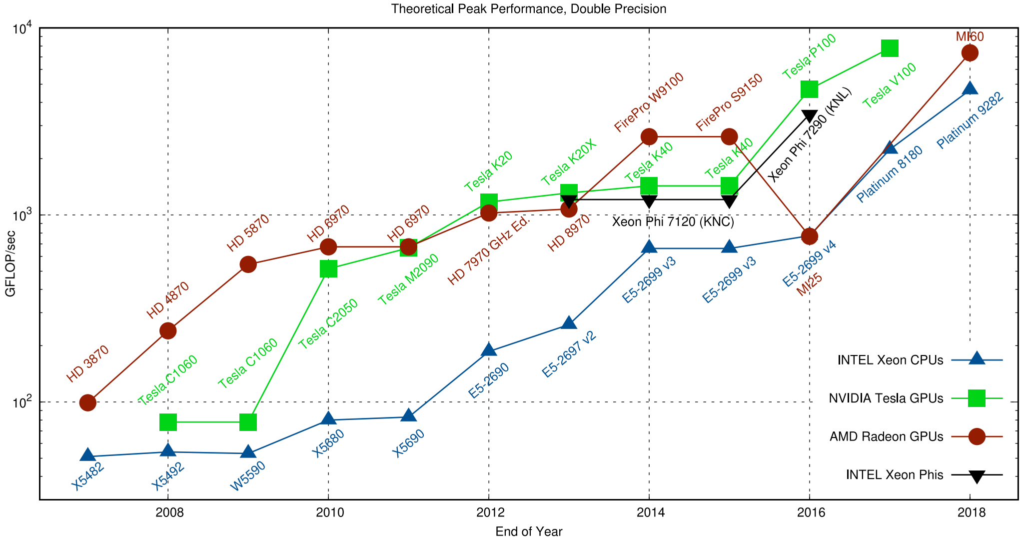
CPUs versus GPUs



- **CPU**—Central Processing Unit
  - Low number of strong cores
  - Independent and vector computation
  - Low to medium parallelization
- **GPU**—Graphical Processing Unit
  - High number of weak cores
  - Designed for vector operations
  - Lock-step mode
  - High parallelization
- $\rightarrow$ *race car* vs *transport train*

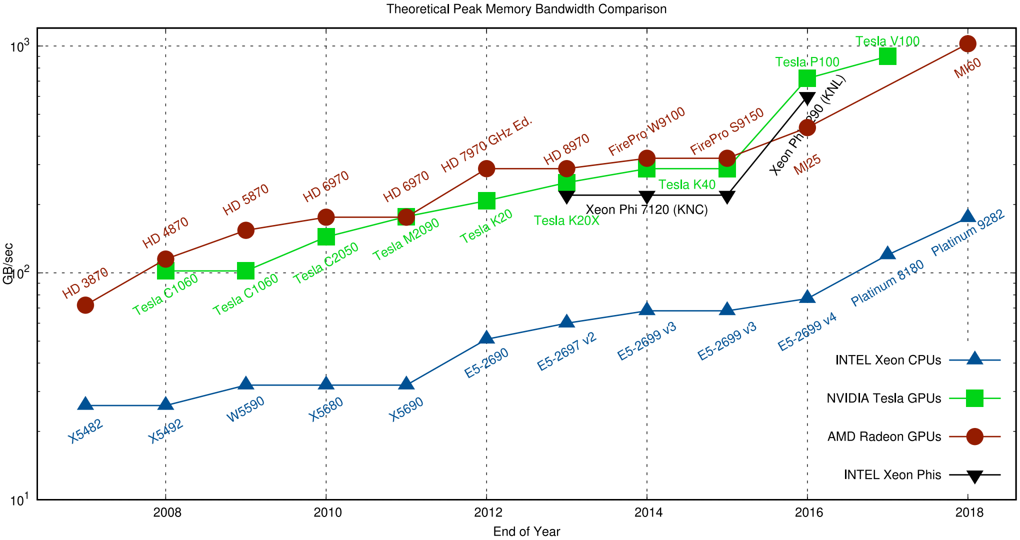
Graphical Processing Unit (GPU)

Theoretical Peak Performance, Double Precision



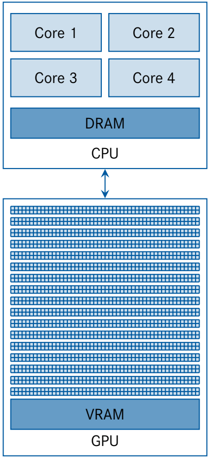
Source: Karl Rupp, CPU/GPU Performance Comparison, https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/.

Graphical Processing Unit (GPU)
Theoretical Memory Bandwidth



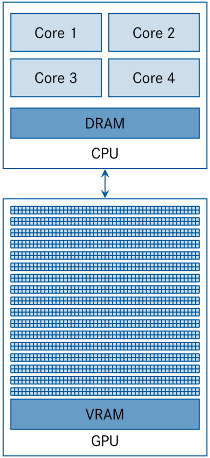
Source: Karl Rupp, CPU/GPU Performance Comparison, https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/.

Programming GPUs
Processing Flow

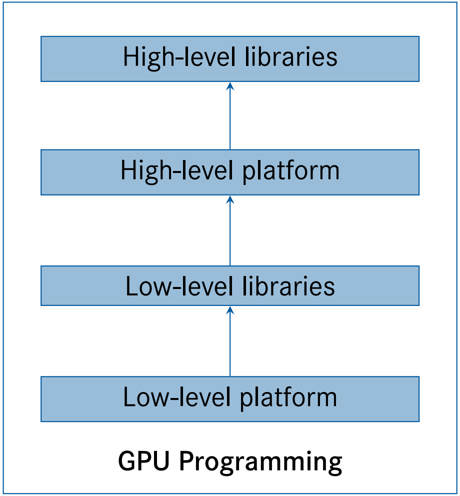


- GPU is accelerator card
  - **Separate entity**, connected via bus
  - Computations controlled by CPU
- **Template processing workflow**
  1. Transfer data from CPU to GPU
  2. Load GPU program and execute
  3. Transfer results from GPU to CPU
- CPU is independent
  - Concurrent other computations
  - Explicitly synchronization

Programming GPUs
Caveats



- **Lock-step** computation
  - Processors are technically grouped
  - Diverging processing, e.g. `if-else`-clause, causes waiting
- **Memory hierarchy**
  - Actually three memory levels
  - Unified address space at performance cost
- **Limits to parallelization**
  - Unsuitable computation problem
  - Computation-communication-hiding



- Scripting languages, algorithms
- Examples: RAPIDS, HeAT,

- Bindings for device primitives
- Examples: CuPy, Numba, JuliaGPU

- Low-level language, algorithms
- Examples: Thrust, CuBLAS, Gingko

- Direct API to device primitives
- Examples: CUDA, HIP, OpenCL

Programming GPUs – Low-level Platform

Compute Unified Device Architecture (CUDA)

Source: https://nvidia.com

- **Proprietary NVidia API**
  - General purpose GPU programming
  - Introduced first in 2007
  - Current version: 11.0
- Available for C/C++
  - Modification of base languages
  - Requires custom `nvcc` compiler
- Most widely used API, ≈**25% share** on Top-500 **supercomputers** (2019)

Low-level Platforms

CUDA Example

```
__global__ void add(int n, float *x, float *y) {
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}

int main(void) {
    int N = 1<<20;
    float *x, *y;

    // Allocate Unified Memory - accessible from CPU or GPU
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));
    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }
    // Run kernel on 1M elements on the GPU
    add<<<1, 256>>>(N, x, y);

    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();
    // Free memory
    cudaFree(x);
    cudaFree(y);

    return 0;
}
```

Source: NVidia©, modified example from: https://developer.nvidia.com/blog/even-easier-introduction-cuda/

High-level Platforms
Tensor Computation Frameworks

- Access GPU from scripting languages
  - Predominantly in Python
  - Wrappers around low-level calls
- Abstraction: **multi-dimensional tensors**
- Application areas
  - **Deep learning**
  - Simulations
  - . . .
- Different computational backends
  - All: CPU and GPU
  - Additionally: TPU, FPGA, IPU

Source: https://pytorch.org

Source: https://tensorflow.org

Source: https://mxnet.apache.org

PyTorch
Overview

- Why PyTorch?
  - Currently **fastest** implementation
  - Largest **community** in users
  - Wide adoption in research
- Specifics computational features
  - **Eager computation**
  - Tight integration with NumPy
  - Strong sparse tensor support
- Tooling and libraries around PyTorch
  - Debuggers and performance profiling
  - Bayesian approaches, e.g. `pyro`

PyTorch
Tensor Initialization

```
>>> import torch
>>> vector = torch.arange(10)
>>> vector
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

PyTorch
Tensor Initialization

```
>>> import torch
>>> vector = torch.arange(10)
>>> vector
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> import torch
>>> matrix = torch.tensor([
        [1, 2, 3, 4],
        [5, 6, 7, 8]
    ])
>>> matrix.shape
torch.Size([2, 4])
```

PyTorch
Tensor Initialization

```
>>> import torch
>>> vector = torch.arange(10)
>>> vector
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> import torch
>>> matrix = torch.tensor([
        [1, 2, 3, 4],
        [5, 6, 7, 8]
    ])
>>> matrix.shape
torch.Size([2, 4])
```

- Various other initializers: `full()`, `one()`, `zeros()`, `rand()`, `...`

PyTorch
Meta Information

- Information about the PyTorch tensor object itself

```
>>> import torch
>>> volume = torch.randn((3, 4, 5,))
>>> volume.shape
torch.Size([3, 4, 5])

>>> volume.dtype
torch.float32

>>> volume.device
torch.device(type='cpu')
```

Hands-on Part 1

PyTorch
Element-wise Operations

- PyTorch can operate on **multi-dimensional tensors** as **operands**
- Avoids explicit looping, code resembles equations

```
>>> import torch
>>> a = torch.arange(10)
>>> a
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> b = torch.ones(10, dtype=a.dtype)
>>> b
tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

>>> a + b
tensor([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

PyTorch
Broadcasting

- PyTorch attempts to **broadcast** non-matching shapes
- Dimensions equal to 1 are repeated based on other operand

```
>>> import torch
>>> vector = torch.arange(10)
>>> vector + 3
tensor([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

- **Slicing** a torch tensor allows to obtain partial tensor content

```
>>> import torch
>>> matrix = torch.randn(size=(10, 3,))
>>> matrix[2]
tensor([1.2646, 0.4840, 0.4133])
```

Reduction Operations

- **Reductions** combine all elements of tensors (or slices) to a singular output
- Typical examples are: `max()`, `min()`, `sum()`, ...

```
>>> import torch
>>> a = torch.arange(10)
>>> a
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> a.sum()
tensor(45)
```

Hands-on Part 2

PyTorch
Using the GPU

- PyTorch provides the `cuda` submodule to interact with CUDA
- Contains mostly the meta-information about the GPUs

```
>>> import torch
>>> torch.cuda.is_available()
True

>>> torch.cuda.device_count()
1

>>> torch.cuda.get_device_name()
'A100-SXM4-40GB'
```

- Data can be explicitly **moved** from **CPU** to **GPU** and vice versa
- Some calls, e.g. `print()`, will do this automatically behind the scenes

```
>>> import torch
>>> m = torch.arange(10)
>>> m.device
device(type='cpu')

>>> m_gpu = m.cuda()
>>> m_gpu.device
device(type='cuda:0')

>>> m_gpu.cpu().device
device(type='cpu')
```

PyTorch
Direct Data Allocation on GPU

- Data can be directly allocated and/or moved to GPU
- There are several approaches to do so

```
>>> import torch
>>> torch.arange(2, device='cuda')
tensor([0, 1], device='cuda:0')

>>> torch.cuda.FloatTensor([1.0, 2.0])
tensor([1., 2.], device='cuda:0')

>>> torch.set_default_tensor_type('torch.cuda.FloatTensor')
>>> torch.randn(5).device
torch.device(type='cuda:0')
```

PyTorch
Computing on the GPU

- PyTorch allows us to use the **same interface** for computation on GPU as on CPU

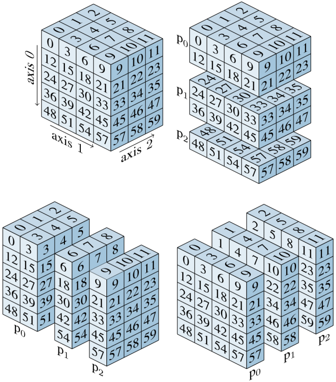
```
>>> import torch
>>> a = torch.arange(10, device='cuda:0')
>>> a
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], device='cuda:0')

>>> a.sum()
tensor(45, device='cuda:0')
```

Hands-on Part 3

High-level Libraries

HeAT—A Distributed Tensor Framework

- **Python** framework for data-intesive computing
  - n-dimensional numerical tensors
  - Low-level operations and full-blown algorithms
- Joint development: **KIT-SCC, FZJ** and **DLR**
- Transparent data-scaling via **MPI**
  - Start prototyping on laptop
  - Port without changes to HPC cluster
  - Data-parallelization strategy
- **Up to three orders of magnitude faster** than comparable frameworks (e.g. dask, `horovod`)



Conclusion

- **GPUs** are massively **parallel** vector **co-processors**
- Different approaches to programming
  - High-level scripting, easy, less flexible and performant
  - Low-level languages, difficult, maximal control and performance
- A large set of computational problems unsuitable for GPUs
- **Benchmark, profile, debug!**

Get Your Own Hands Dirty...

...once again

Install the requirements first...

```
pip install numpy torch
```

... and clone the **slides** and **Jupyter notebooks**

```
git clone https://github.com/Helmholtz-AI-Energy/kseta_2021.git
```

Available under the *BSD-3 license*.

Steinbuch Centre for Computing