Large Scale Data on GPGPUs

- General Purpose Graphical Processing Units (GPGPUs) focus on data-parallel computations rather than task-parallelism
- Scalable array of multithreaded Streaming Multiprocessors (SMs)

Large Scale Data on GPGPUs

Types of GPU

- **Integrated**: power is shared between GPU and CPU. Graphics card is built directly into the computer's processor.
  - Best for web browsing, social media, resource-light work such as spreadsheets, editing, light-resource demanding games etc.
  - Example: AMD Ryzen
- **Dedicated**: completely separated processor from the main CPU, has its main dedicated memory and a cooling system.
  - When buying a dedicated GPU, CPU processor needs to be a good match as well as the power supply. $\rightarrow$ Baseline processor: 8th gen Intel Core i7.
  - Main uses: AAA games and neural network-based machine learning models.
  - Example: Nvidia GTX, RTX, Quadro etc.

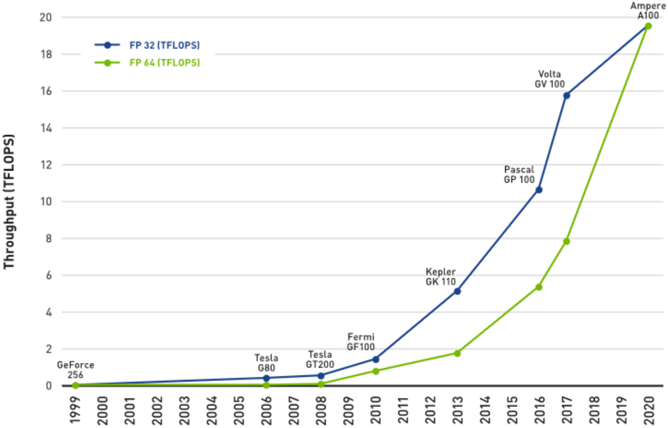
GPU Suppliers

PC	IP	SoC
AMD	Arm	Apple
Bolt	DMP	Qualcomm
Innosilicon	IMG	
Intel	Think Silicon	
Jingia	Verisilicon	
MetaX	Xi-Silicon	
Moore Threads		
Nvidia		
SiArt		
Xiangdixian		
Zhaoxin		

GPU suppliers

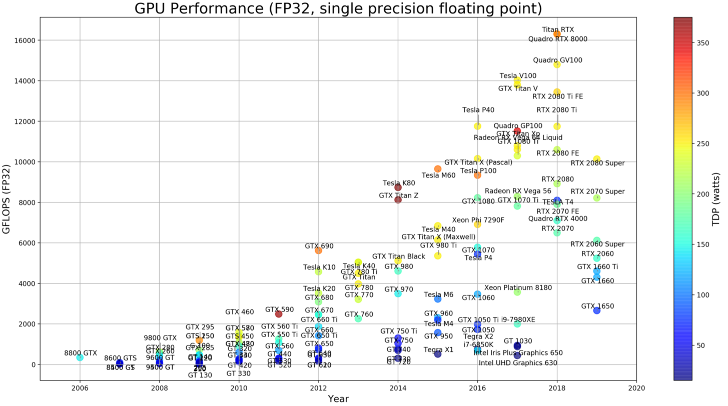
https://blog.siggraph.org/2023/01/2022-was-the-rise-of-gpu-suppliers.html/

GPU Architecture



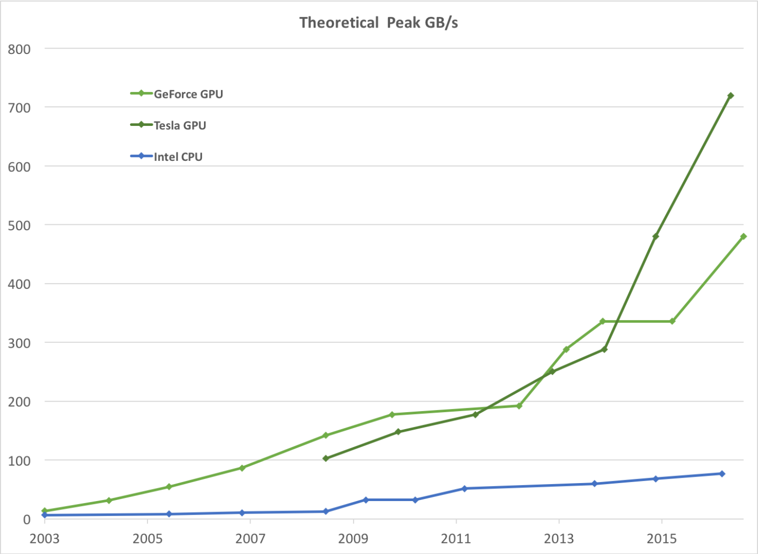
`https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9623445`

GPU Architecture

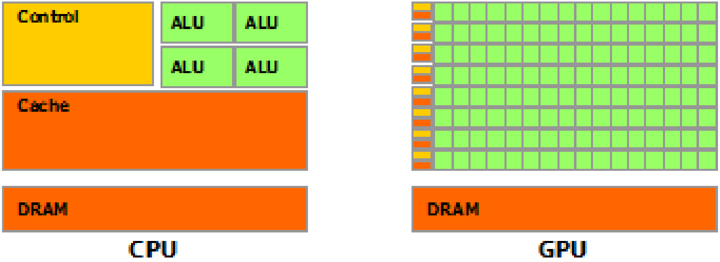


https://miro.medium.com/max/6058/1*Uyx0bONUqvbZLu8z1cYQ1g.png

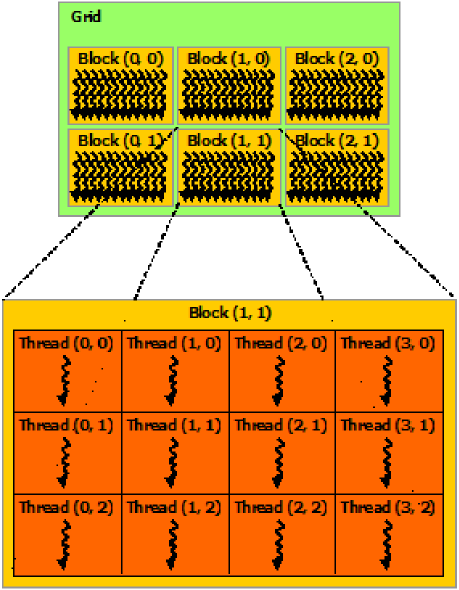
GPU Architecture: memory bandwidth



GPU Architecture



GPU Architecture



Grids, blocks and threads

- Usually, a grid is organized as a 2D array of blocks
- A block is organized as a 3D array of threads
- Both grids and blocks use the dim3 type with three unsigned integer fields
- Unused fields are initialized to 1 and ignored.

GPU Execution

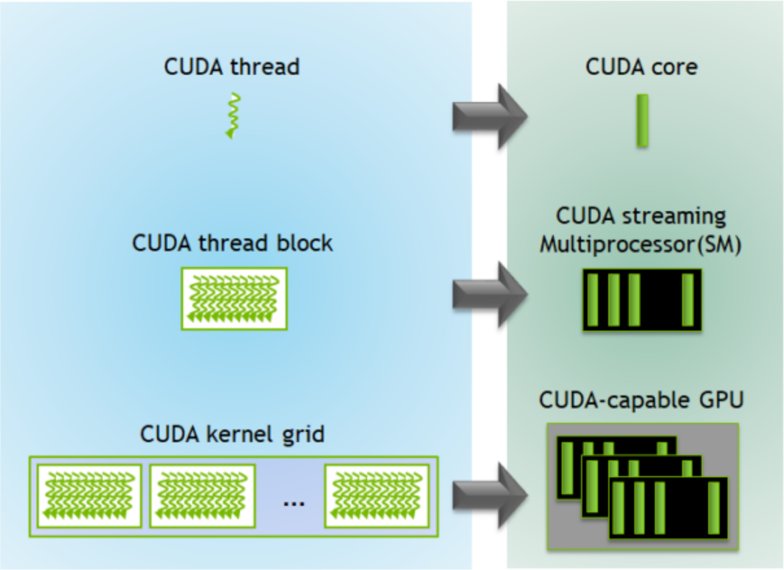
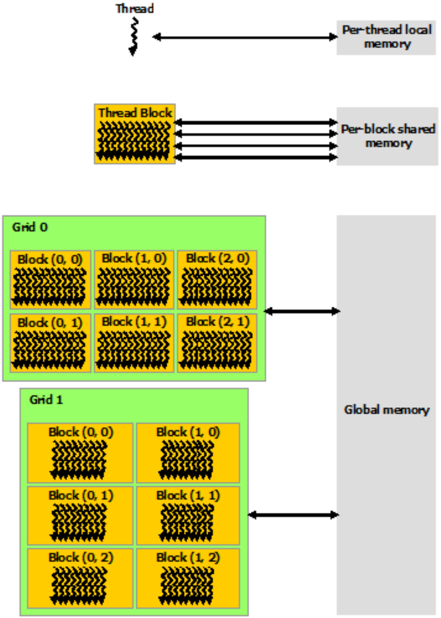
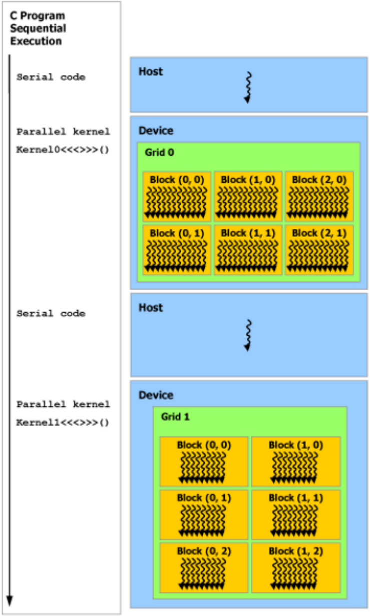


Figure 3. Kernel execution on GPU.

GPU Architecture



Heterogeneous programming



Data Partitioning

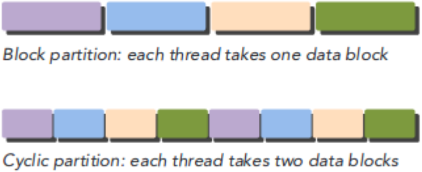
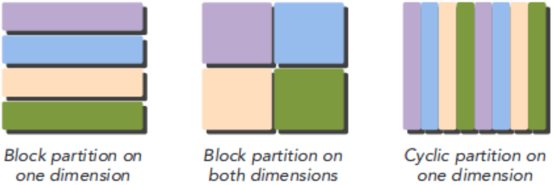


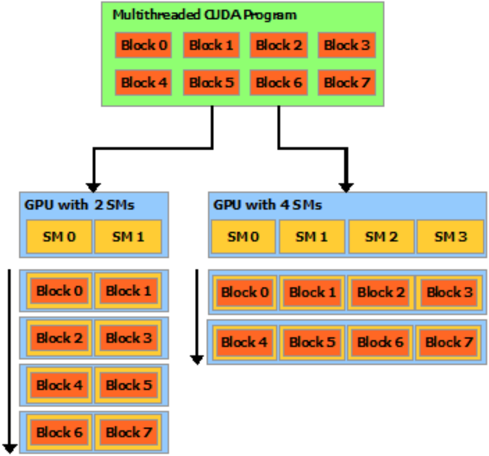
FIGURE 1-4



(from `http://www.hds.bme.hu/~fhegedus/C++/Professional%20CUDA%20C%20Programming.pdf`)

Auto Scaling

(Tesla V100 uses 80 SMs!)

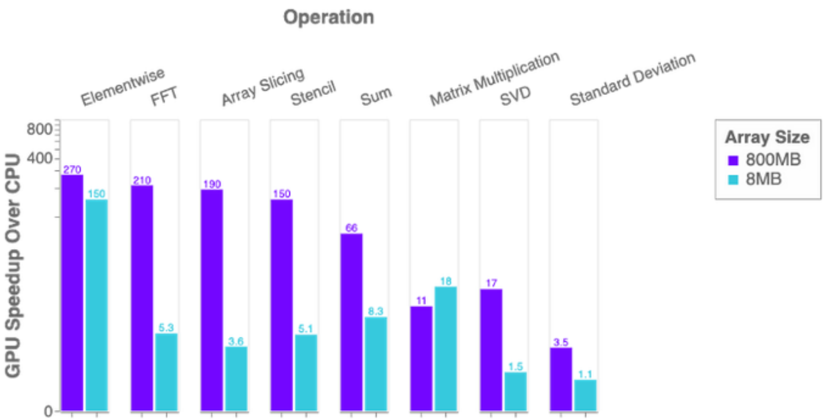


(from `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model`)

Python alternatives for GPUs

- Numpy: CuPy or Jax
- Pandas: RAPIDS cuDF
- scikit-learn: RAPIDS cuML
- DNN: cuDNN

CPU x GPU



pybench
Performance comparison

CUDA: Computer Unified Device Architecture

- $C/C++$ extension to prepare code to run in GPGPUs
- Compiler: nvcc
- CUDA program: kernel (functions that will run in the GPU)
- Defining a kernel in CUDA:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

CUDA: Computer Unified Device Architecture

- A kernel is defined using the __global__ declaration specifier and the number of threads that will execute that kernel
- Each thread that executes the kernel is given a **unique** thread ID
- thread ID accessed in the kernel function via built-in variables

CUDA: Computer Unified Device Architecture

- CUDA built-in variables:
  - `gridDim`: dimension of grid (type `dim3`)
  - `blockDim`: dimension of block (type `dim3`)
  - `blockIdx`: block index within a grid (type `uint3`)
  - `threadIdx`: thread index within a block (type `uint3`)
  - `warpSize`: warp size in threads (type `int`, usually 32)

CUDA: Computer Unified Device Architecture

- `threadIdx` is a 3-component vector (vector, matrix or volume)
  → for a 1D block, thread index and thread ID are the same
  → for a 2D block of size (Dx,Dy), thread ID of a thread of index (x,y) is (x+yDx)
  → for a 3D block of size (Dx,Dy,Dz), thread ID of a thread of index (x,y,z) is (x +yDx + zDxDy)
- each block can have at most 1024 threads (this number depends on the GPU model. Some new GPU models can take up to 2048 threads in a block)
- threads in the same thread block run on the same stream processor (SM) and communicate via shared memory, barrier synchronization or other synchronization primitives
- all blocks in the same grid contain the same number of threads

Alternatives for python

- PyCUDA or PyOpenCL
  (slides from `https://www.slideshare.net/GIUSEPPEDIBERNARDO/pycon9-dibernado-94735367`)
- Numba
  (slides from `https://devblogs.nvidia.com/numba-python-cuda-acceleration/`)

PyCUDA: workflow

PyCUDA Workflow: "Edit-Run-Repeat"

A two-fold aim:

1. usage of *existing* `CUDA C`
2. *on top* of the first layer, PyCUDA ⇒ *abstractions*

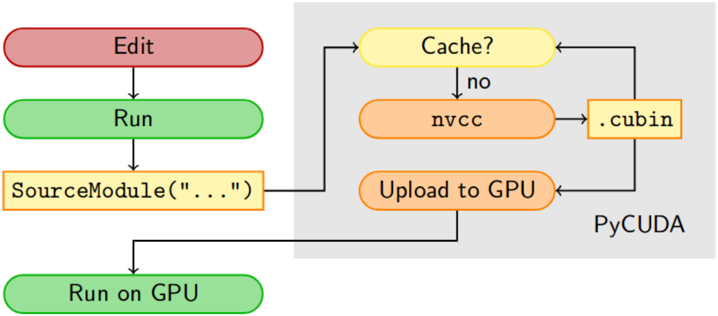


Figure: A. Klöckner et al. 2013, https://arxiv.org/abs/1304.5553

PyCUDA: hello world! (1)

The "Hello World" of PyCUDA: the Kernel, Part I

```
import numpy as np
import pycuda.driver as drv # import PyCUDA
import pycuda.autoinit # initialize PyCUDA
from pycuda.compiler import SourceModule

mod = SourceModule("""
  __global__ void add_them(float *dest, float *a, float *b)
{
int idx = threadIdx.x; // unique thread ID within a block
dest[idx] = a[idx] + b[idx];
}
""")

add_them = mod.get_function("add_them")
a = np.random.randn(400).astype(np.float32)
b = np.random.randn(400).astype(np.float32)
dest = np.zeros_like(a) # automatic allocated space on device
add_them(drv.Out(dest), # immediate invocation style
         drv.In(a), drv.In(b),
         block=(400,1,1), grid=(1,1)) # explicit memory copies
print(dest - a+b)
```

COMPUTE KERNEL

PyCUDA: hello world! (2)

The "Hello World" of PyCUDA: the Kernel, Part II

```
import numpy as np
import pycuda.driver as drv # import PyCUDA
import pycuda.autoinit # initialize PyCUDA
from pycuda.compiler import SourceModule

a = np.random.randn(4,4).astype(np.float32) # host memory
a_gpu = drv.mem_alloc(a.nbytes) # allocate device memory
drv.memcpy_htod(a_gpu, a) # host-to-device

mod = SourceModule("""
  __global__ void multiply_by_two(float *a)
{
int idx = threadIdx.x + threadIdx.y*4;
a[idx] *= 2;
}
""")

func = mod.get_function("multiply_by_two")
func(a_gpu, block=(4,4,1)) # launching the kernel
a_doubled = np.empty_like(a)
drv.memcpy_dtoh(a_doubled, a_gpu) # fetching the data
print(a_doubled)
```

COMPUTE KERNEL

Using abstraction: GPUArrays

```
import numpy as np
import pycuda.autoinit
import pycuda.gpuarray as gpuarray

a_gpu = gpuarray.to_gpu(np.random.randn(4,4).astype(np.float32))
a_doubled = (2*a_gpu).get()
print(a_doubled)
print(a_gpu)
```

GPUArrays: computational linear algebra

- element-wise algebraic operations (+, -, *, /, sin, cos, exp)
- tight integration with numpy
  - gpuarray.to_gpu(numpy_array)
  - numpy_array = gpuarray.get()
- mixed data types (int32 + float32 = float64)

PyCUDA: device properties

How to Query Device Properties

Querying Device Properties with PyCUDA

```
import pycuda.driver as drv
import pycuda.autoinit
print("PyCUDA version:pycuda.VERSION_TEXT)
print("%d device(s) found." % drv.Device.count())
for ordinal in range(drv.Device.count()):
  dev = drv.Device(ordinal)
  print("Device Number:  %d Device Name:  %s" % (ordinal, dev.name()))
  print(" Compute Capability:  %d.%d" % dev.compute_capability())
  print(" Max Thread per Block:  %d" % dev.max_threads_per_block)
  print(" Max Block dim Z: %d" % dev.max_block_dim_z)
  print(" Total Memory:  %s KB" % (dev.total_memory()//(1024)))
  print(" Memory Clock Rate (KHz):  %d" % dev.clock_rate)
  print(" Memory Bus Width (bits):  %d" % dev.global_memory_bus_width)
  print(" Peak Memory Bandwidth (GB/s):  %f" %
2.0*dev.clock_rate*(dev.global_memory_bus_width/8)/1.0e6)
```

Output

```
PyCUDA version:  2017.1.1
2 device(s) found.
Device Number:  0 Device Name:  GeForce GTX 980
 Compute Capability:  5.2
 Max Thread per Block:  1024
 Max Block dim Z: 64
 Total Memory:  4135040 KB
 Memory Clock Rate (KHz):  1215500
 Memory Bus Width (bits):  256
 Peak Memory Bandwidth (GB/s):  77.792
```

Numba

- Python compiler from Anaconda
- Compile Python code for execution on CUDA-capable GPUs or multicore CPUs
- Numba team implemented pyculib that provides a Python interface to CUDA libraries:
  - cuBLAS (dense linear algebra)
  - cuFFT (Fast Fourier Transform)
  - cuRAND (random number generation)

Numba example (1)

```
import numpy as np
from numba import vectorize

@vectorize(['float32(float32, float32)'], target='cuda')
def Add(a, b):
  return a + b

# Initialize arrays
N = 100000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on GPU
C = Add(A, B)
```

Numba example (2)

```
import numpy as np
from pyculib import rand as curand

prng = curand.PRNG(rndtype=curand.PRNG.XORWOW)
rand = np.empty(100000)
prng.uniform(rand)
print rand[:10]
```

Numba example: Mandelbrot

https://github.com/harrism/numba_examples/blob/master/mandelbrot_numba.ipynb