More on GPU

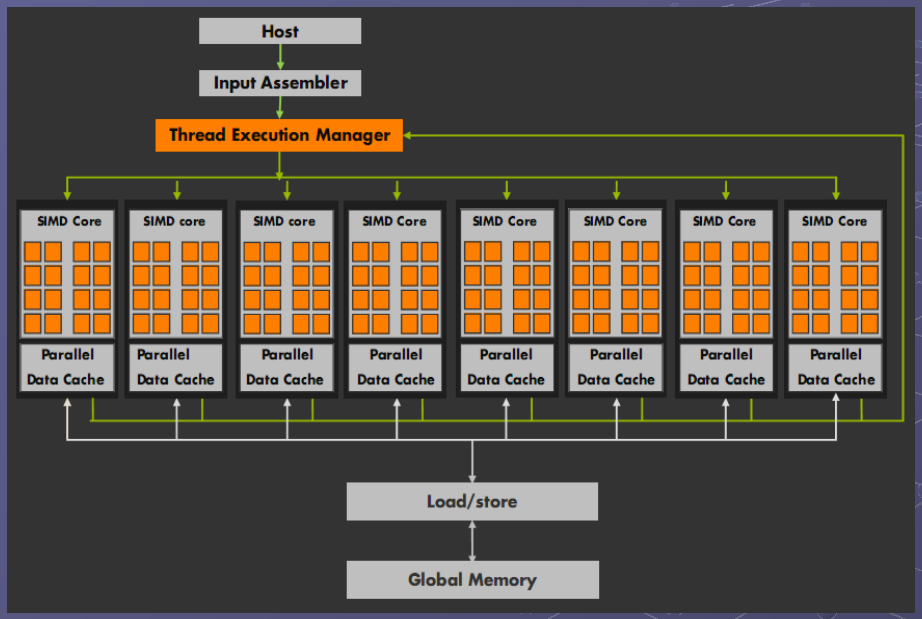
Graphics Processors (GPU) as General Purpose Supercomputers (GPGPU)

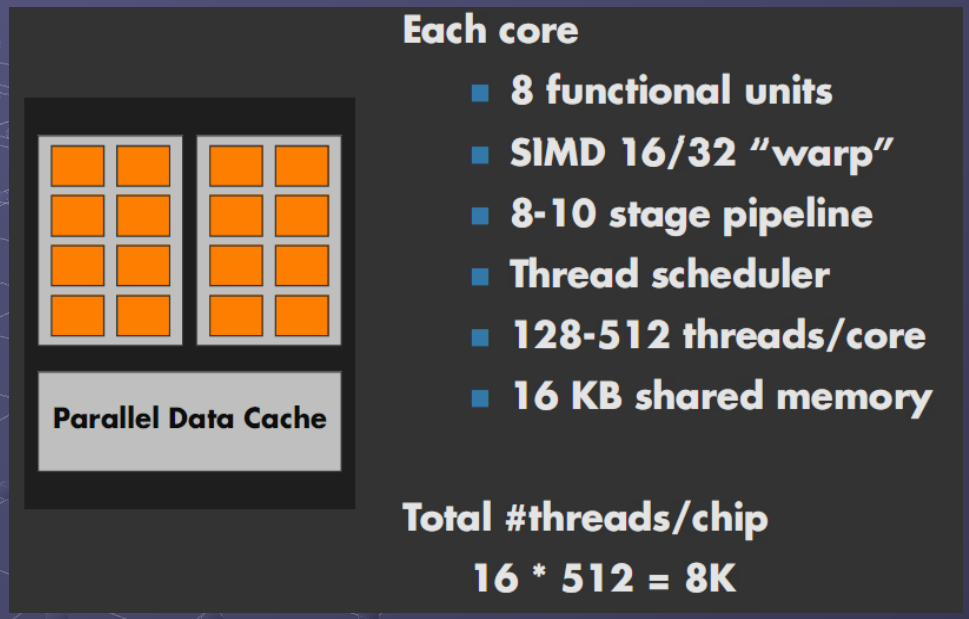
2019:
kepler wn14
RTX 2080 Ti

2008…
GeForce 9800 GTX, 128 Stream Proc., 512 MB
GeForce 9800 GX2, 256 Stream Proc., 1 GB
GeForce 9800 GT, 64 Stream Proc., 512 MB
[...]
2009: Tesla ~200 Proc., 4GB
2010: Fermi ~400 Proc., 4GB
2013: Kepler K20, ~2500 Procs., 6GB
2016: Kepler K80, ~5000 Procs.
2017/18: Pascal, Volta, Ampere > 5000 Procs., 40 GB

2010: Tesla C1070
Laohu 北京

Hardware around 2006



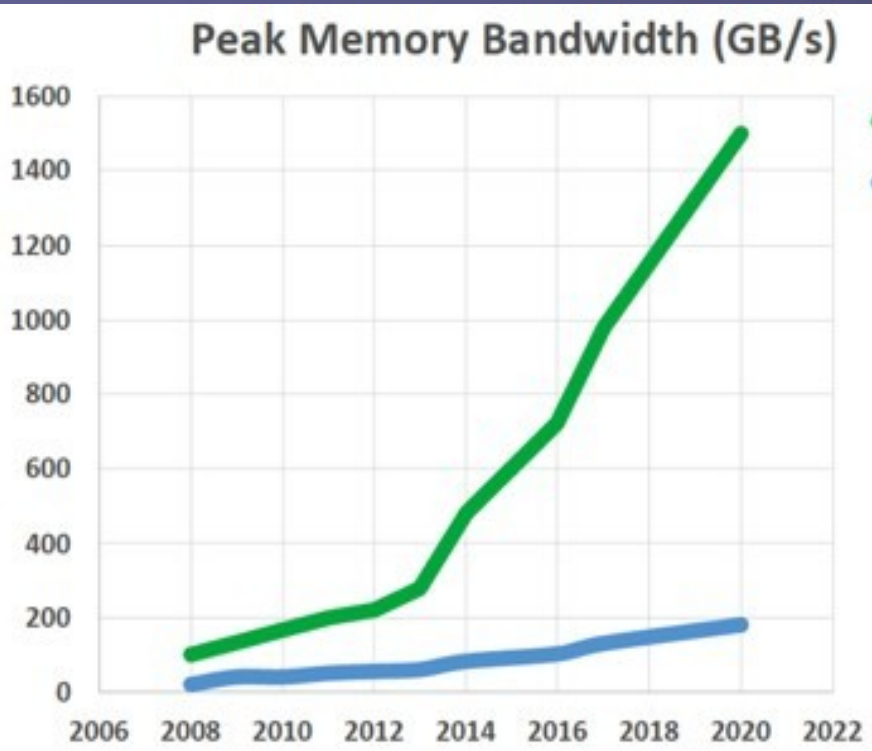


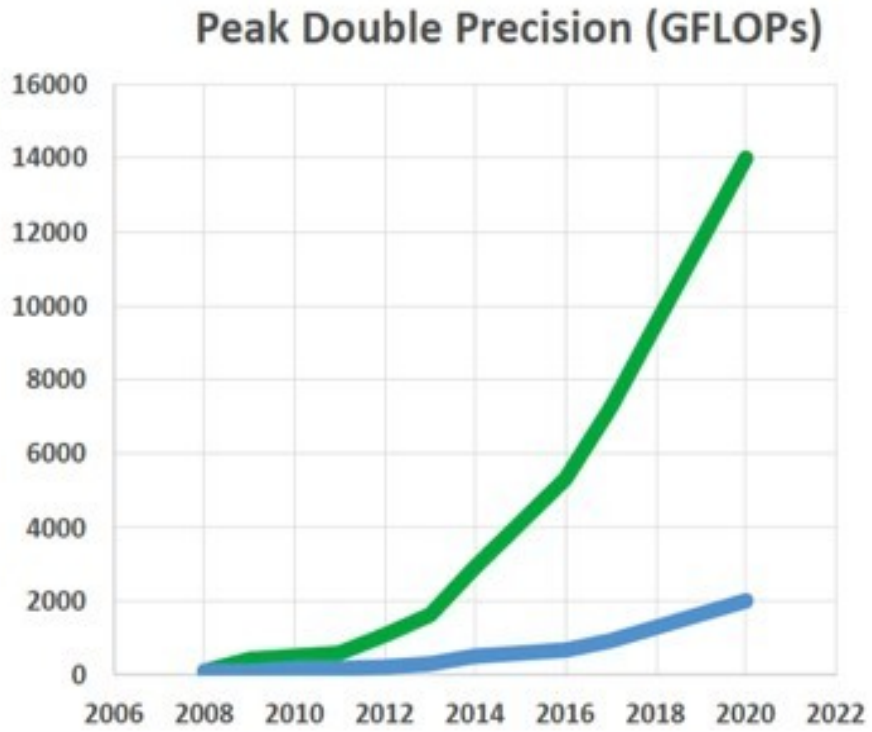
These are the physical parameters! The software ("runtime system") sees a "virtual GPU" which is MUCH larger!!

GeForce 8800 GTX:

575 MHz * 128 processors * 2 flop/inst * 2 inst/clock = 333 Gflops

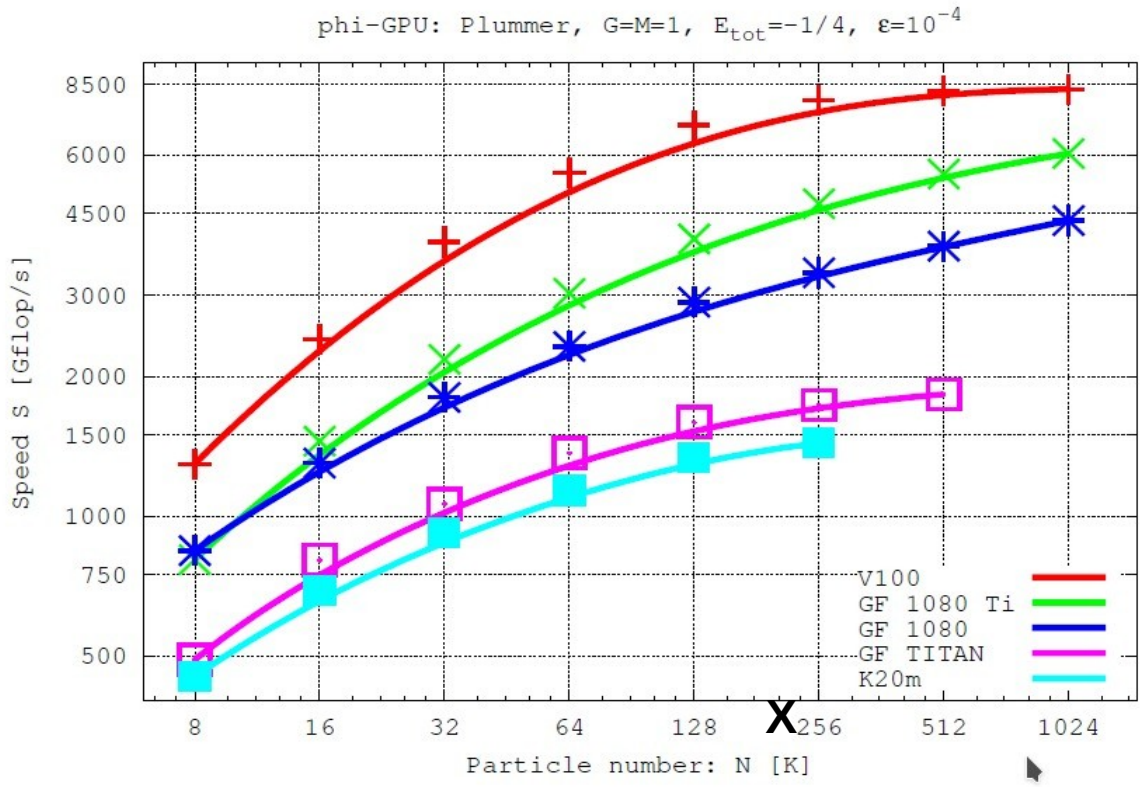
Peak Floating Point Operations per Second And Peak Memory Bandwidth for CPU and GPU





Chip to chip comparison of peak memory bandwidth in GB/s and peak double precision gigaflops for GPUs and CPUs since 2008. Data for Nvidia “Volta” V100 and Intel “Cascade Lake” Xeon SP are used for 2019 and projected into 2020. From:
https://www.nextplatform.com/2019/07/10/a-decade-of-accelerated-computing-augurs-well-for-gpus/

Kepler, Pascal, Volta, Scaling, it works...



Volta V100

Pascal GF1080

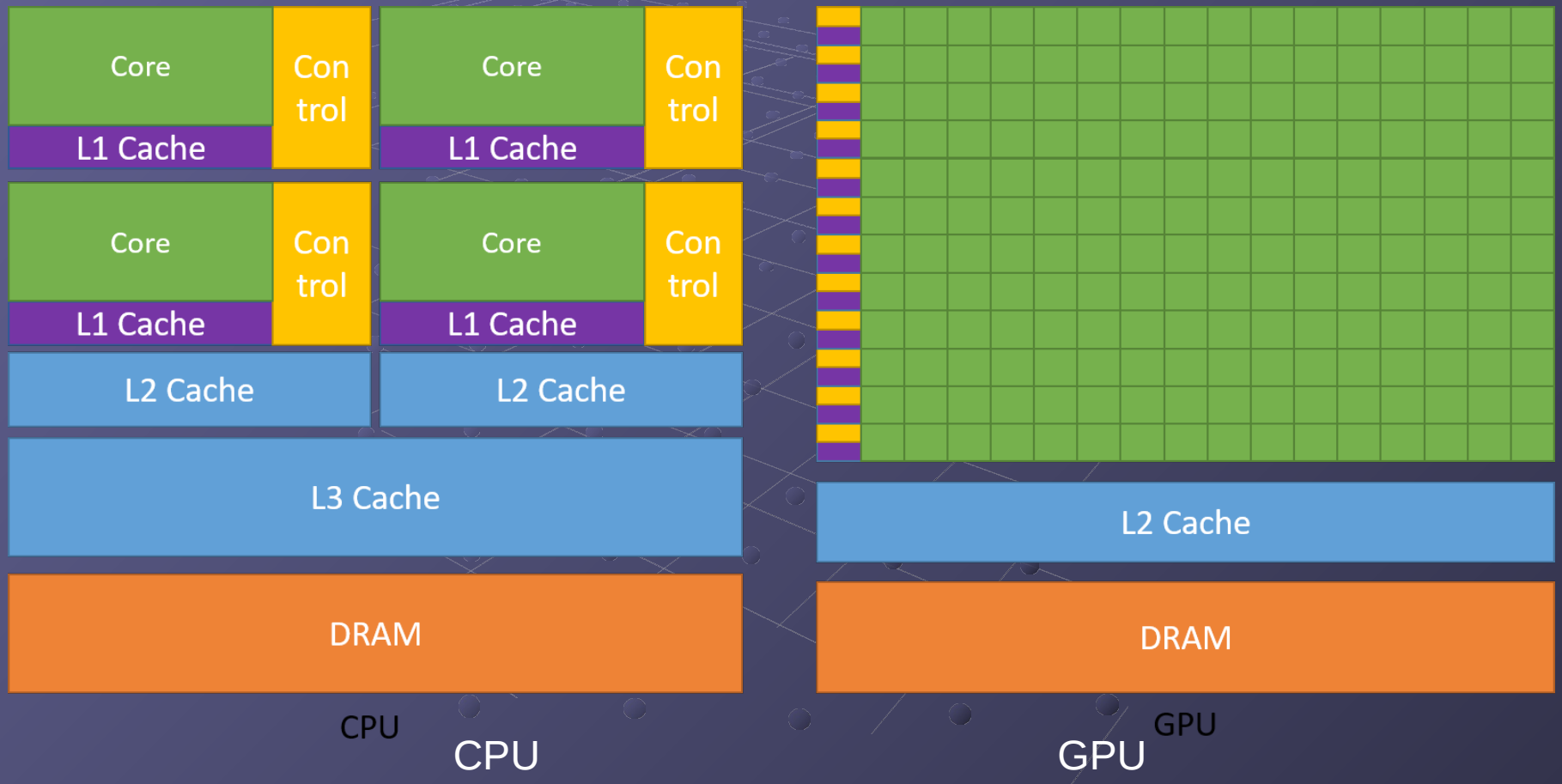
Kepler K20m

Spurzem, Berczik, et al., 2013, LNCS Supercomputing, 2013, pp. 13-25, Springer. (updated unpublished)

Fig. 4. Here we report a preliminary result from a benchmark test of our code on one Kepler K20 card; we compare with the performance on Fermi C2050 (used in the Mole-8.5 cluster), and the oldest Tesla C1060 GPU (used in the laohu cluster of 2009) - the latter is used as a normalization reference. We plot the speed ratio of our usual benchmarking simulation used in the previous figures, as a function of particle number. From this we see the sustained performance of a Kepler K20 would be about 1.4 - 1.5 Tflop/s.

X = first GPU of laohu 2010

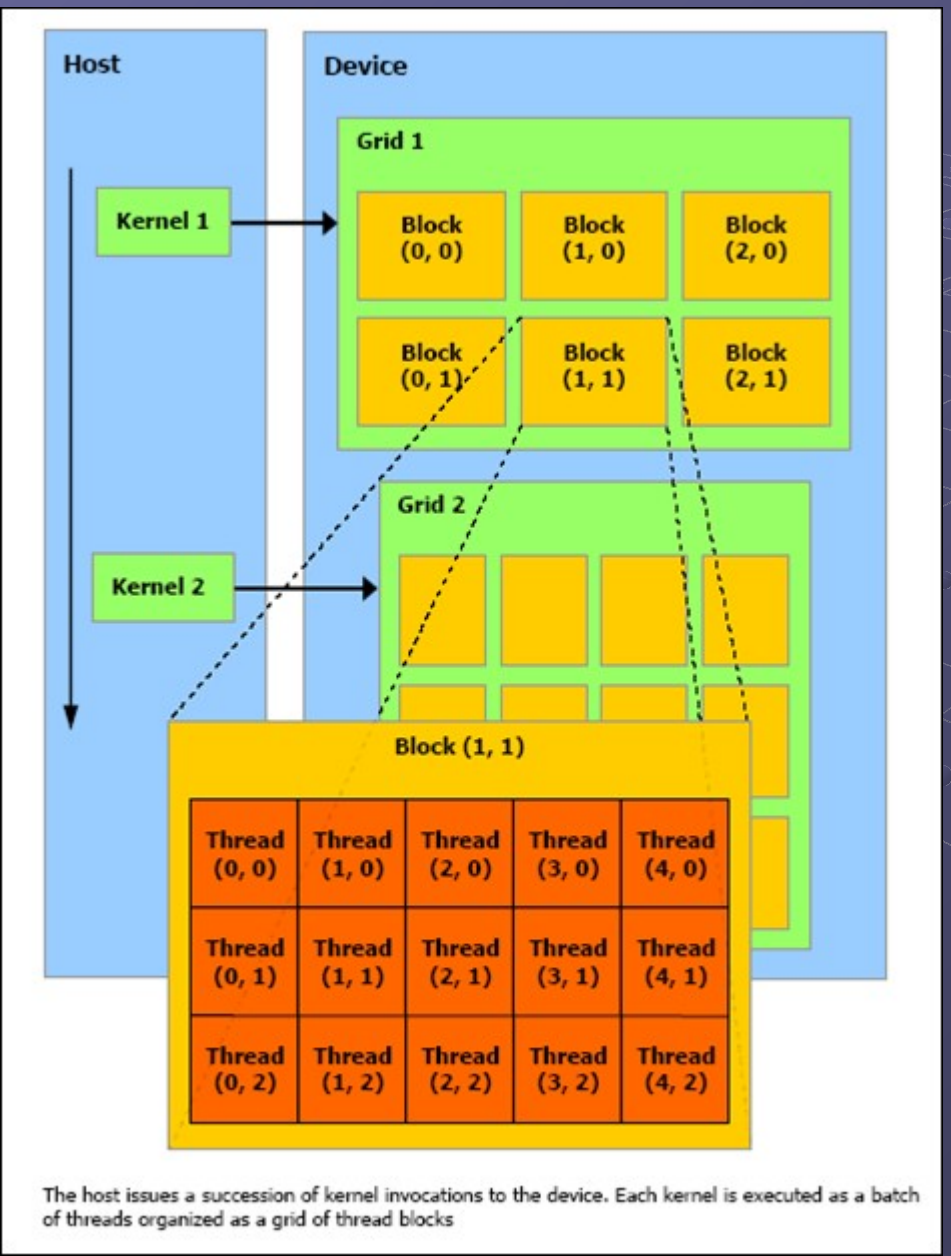
CPU and GPU; from CUDA NVIDIA Developer Zone at
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html



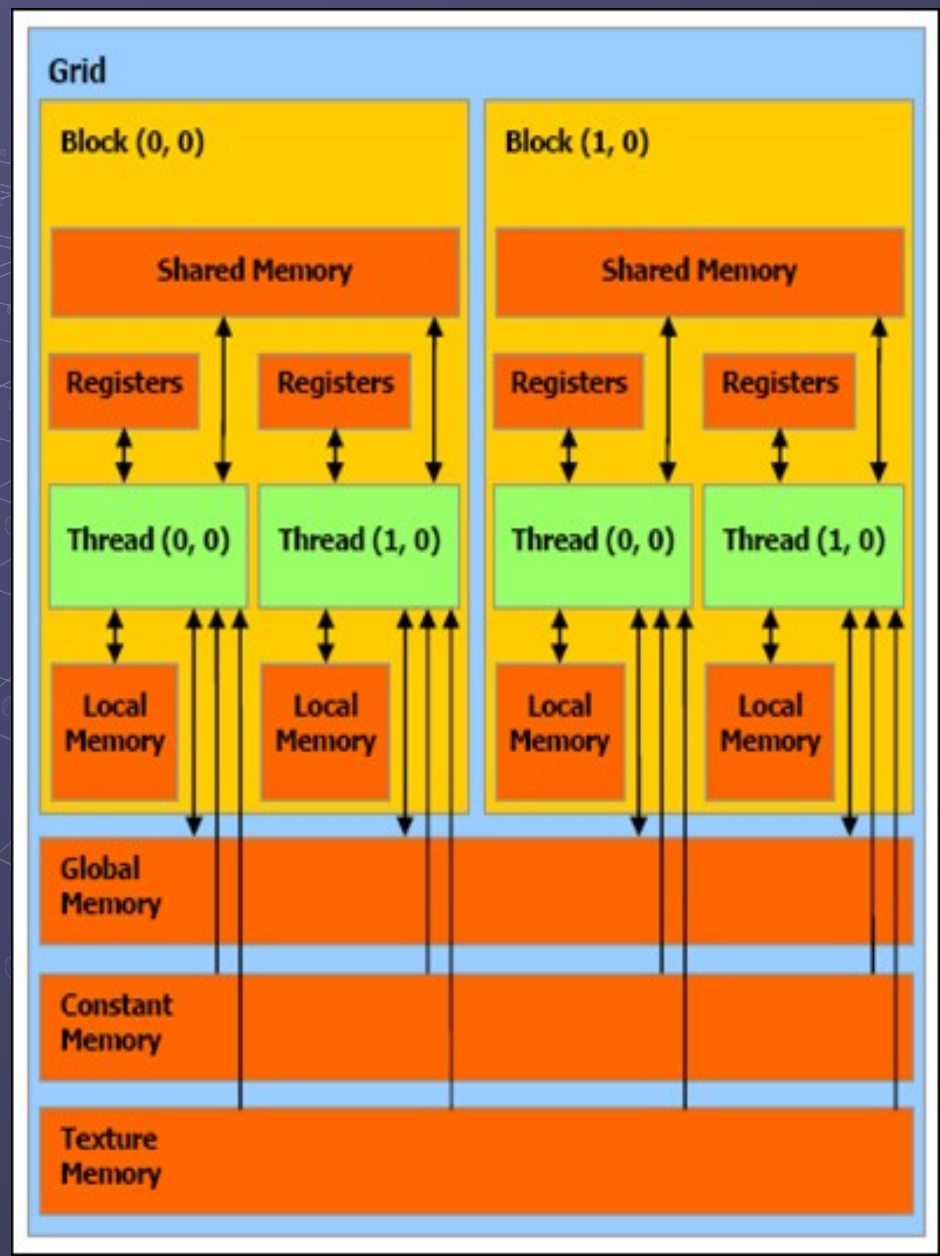
“The GPU devotes more transistors to computing”
“favours data parallel operations”

GPU Structure

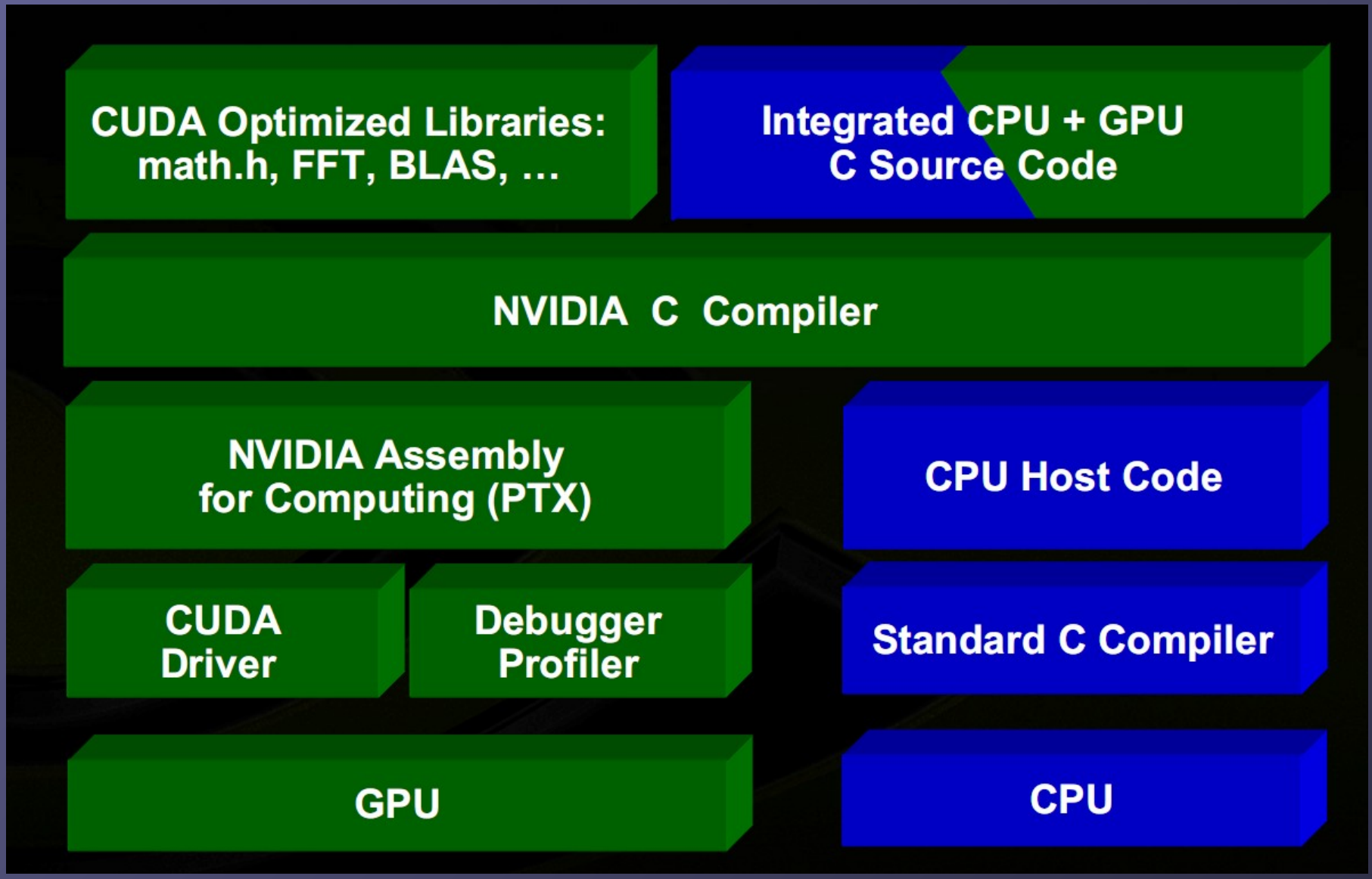
https://docs.nvidia.com/cuda/parallel-thread-execution/index.html



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks



CUDA


CUDA Optimized Libraries:
math.h, FFT, BLAS, …
Integrated CPU + GPU
C Source Code
NVIDIA C Compiler
NVIDIA Assembly
for Computing (PTX)
CPU Host Code
CUDA
Driver
Debugger
Profiler
Standard C Compiler
GPU
CPU

GPU Computing Applications

Libraries and Middleware						
cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

Programming Languages					
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

Ampere and Volta:
Tensor Cores/NVLink

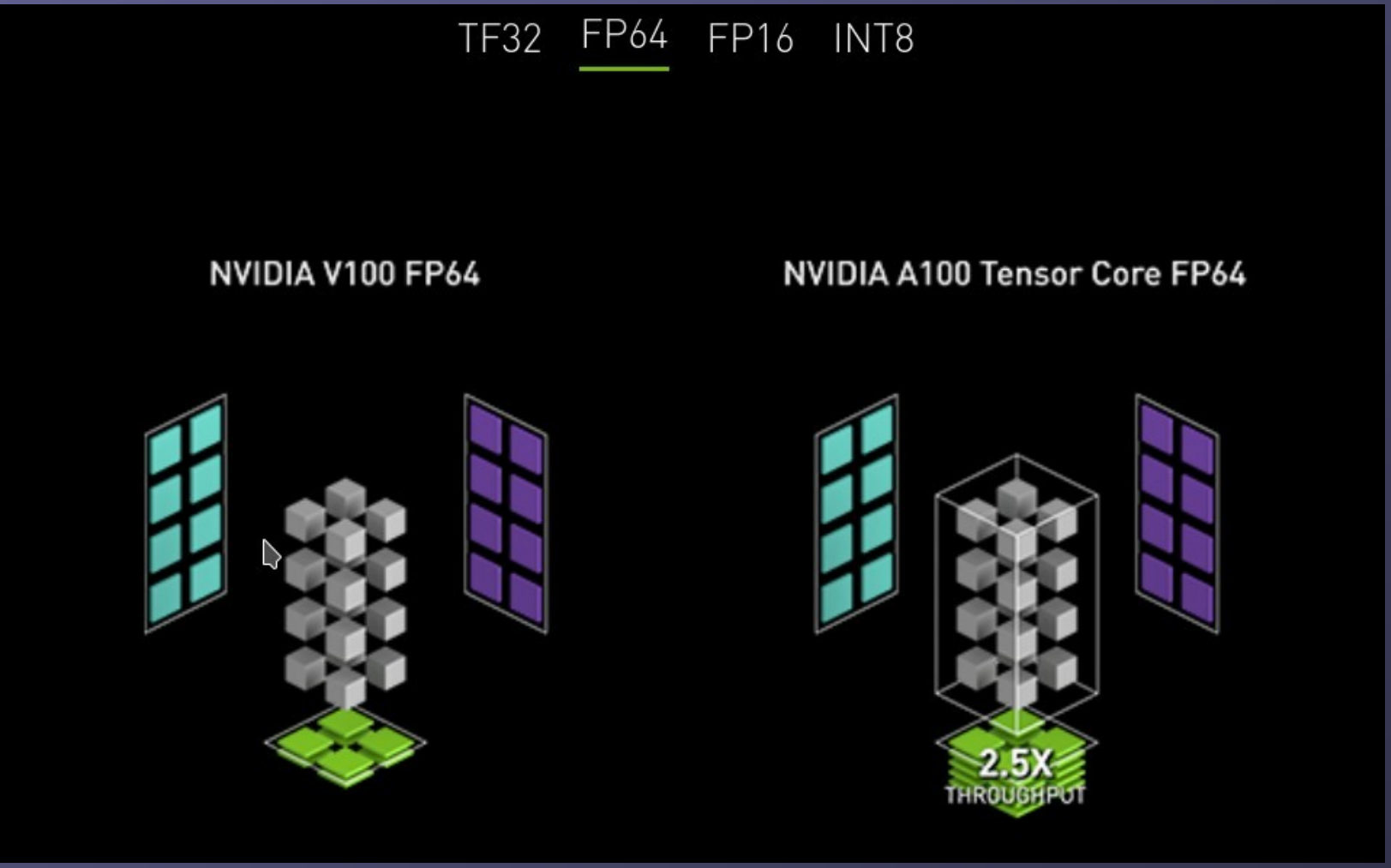
CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
Kepler (3.x)	Tegra K1	GeForce 700/800	Quadro K	Tesla K
				

New feature in Volta, Ampere, Turing: Tensor Cores

https://www.nvidia.com/en-us/data-center/tensor-cores/

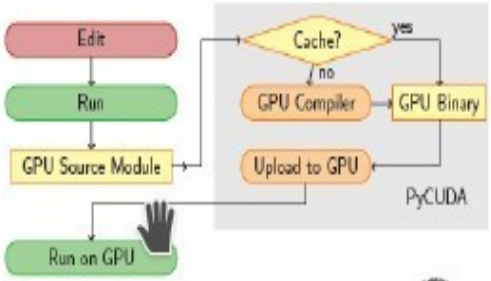
FP64 Tensor Cores: "A100 brings the power of Tensor Cores to HPC, providing the biggest milestone since the introduction of double-precision GPU computing for HPC. By enabling matrix operations in FP64 precision, a whole range of HPC applications that need double-precision math can now get a 2.5X boost in performance and efficiency compared to prior generations of GPUs." (Quote from NVIDIA webpages)



Python + CUDA = PyCUDA

- CUDA C Code = Strings
- Generate Code Easily
  - Automated Tuning
- Batteries included: GPU Arrays, RNG, ...
- Integration: `numpy` arrays, Plotting, Optimization, ...

- All of CUDA in a modern scripting language
- Full Documentation
- Free, open source (MIT)
- Also: PyOpenCL



BROWN

http://mathema.tician.de/software/pycuda

Simple CUDA example

CPU C program

```
void addMatrix(float *a, float *b,
               float *c, int N)
{
  int i, j, index;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      index = i + j * N;
      c[index]=a[index] + b[index];
    }
  }
}

void main()
{
  .....
  addMatrix(a, b, c, N);
}
```

CUDA C program

```
__global__ void addMatrix(float *a,float *b,
                          float *c, int N)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  int j=blockIdx.y*blockDim.y+threadIdx.y;
  int index = i + j * N;
  if ( i < N && j < N)
    c[index]= a[index] + b[index];
}

void main()
{
  ..... // allocate & transfer data to GPU
  dim3 dimBlk (blocksize, blocksize);
  dim3 dimGrd (N/dimBlk.x, N/dimBlk.y);
  addMatrix<<<dimGrd,dimBlk>>>(a, b, c,N);
}
```