CUDA.

COMPUTE UNIFIED DEVICE ARCHITECTURE

Hands-On GPU Programming with Python and CUDA

Explore high-performance parallel computing with CUDA

Packt

www.packt.com

Dr. Brian Tuomanen

Chapman & Hall/CRC
Computational Science Series

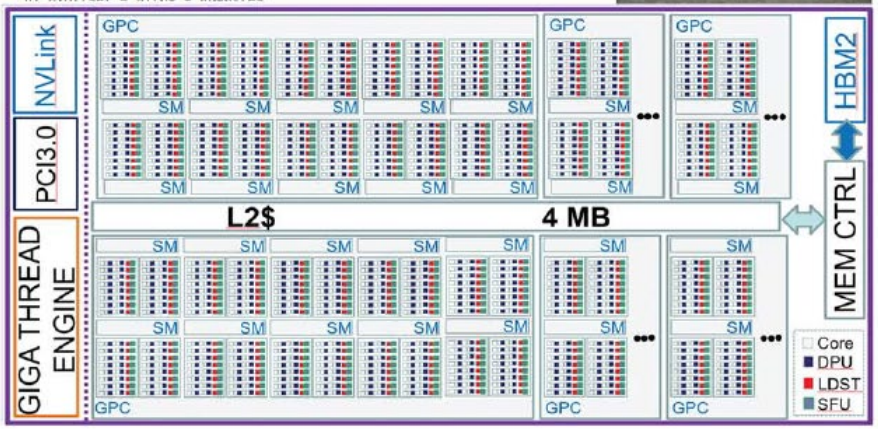
GPU Parallel Program Development Using CUDA

Tolga Soyata

CODE 9.2: imflipGCM.cu Hflip2() {...}
Hflip2() avoids computing BlkPerRow and RowBytes repeatedly.

```
// Improved Hflip() kernel that flips the given image horizontally
// BlkPerRow, RowBytes variables are passed, rather than calculated
__global__
void Hflip2(uch *ImgDst, uch *ImgSrc, ui Hpixels, ui BlkPerRow, ui RowBytes)
{
   ui ThrPerBlk = blockDim.x;
   ui MYbid = blockIdx.x;
   ui MYtid = threadIdx.x;
   ui MYgtid = ThrPerBlk * MYbid + MYtid;

   //ui BlkPerRow = CEIL(Hpixels,ThrPerBlk);
   //ui RowBytes = (Hpixels * 3 + 3) & (~3);
   ui MYrow = MYbid / BlkPerRow;
   ui MYcol = MYgtid - MYrow*BlkPerRow*ThrPerBlk;
   if (MYcol >= Hpixels) return;    // col out of range
   ui MYmirrorcol = Hpixels - 1 - MYcol;
```



CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Python Parallel Programming Cookbook

Second Edition

Over 70 recipes to solve challenges in multithreading and distributed system with Python 3

Packt>

www.packt.com

Giancarlo Zaccone

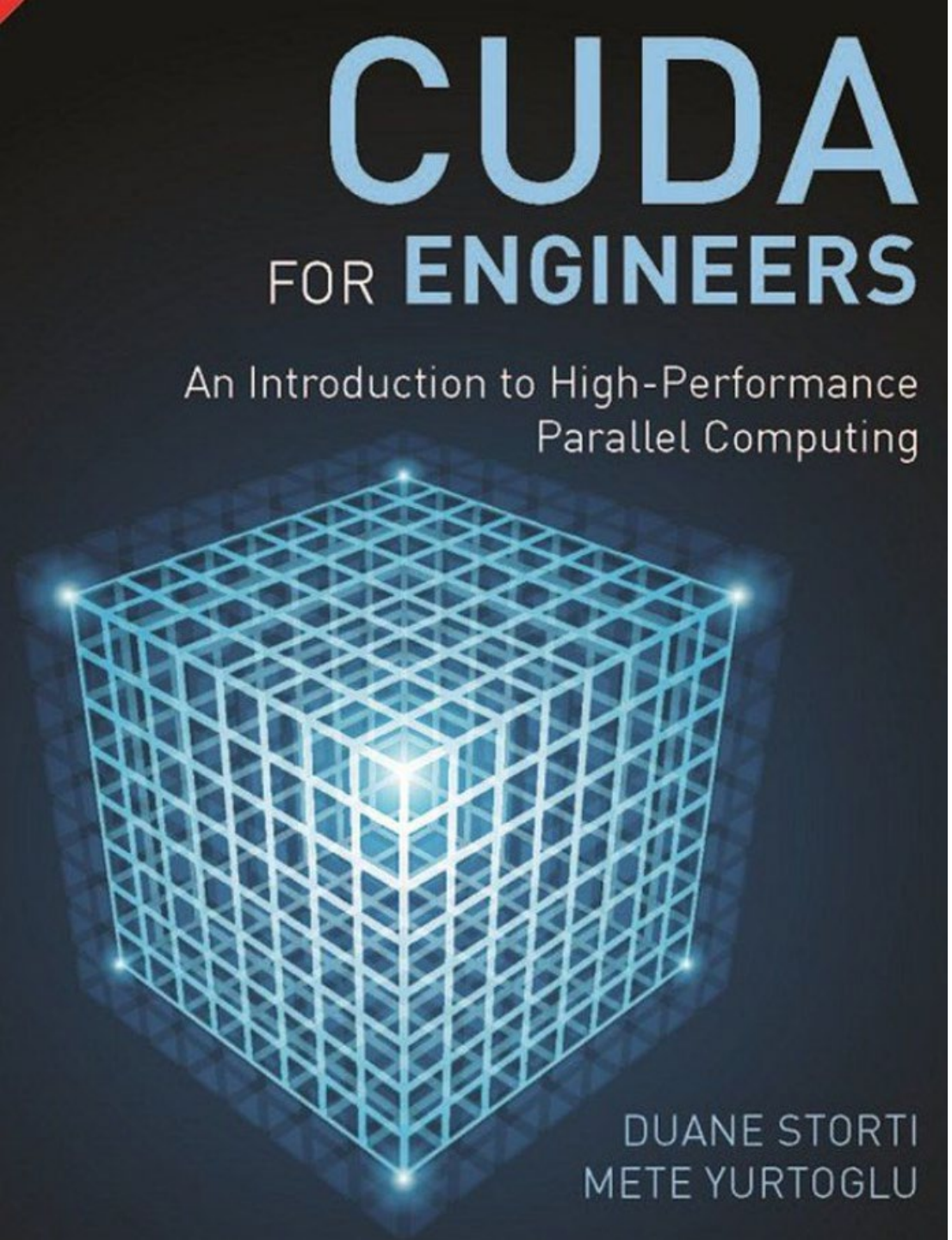
Learn CUDA Programming

A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++

Packt>

www.packt.com

Jaegeun Han and Bharatkumar Sharma



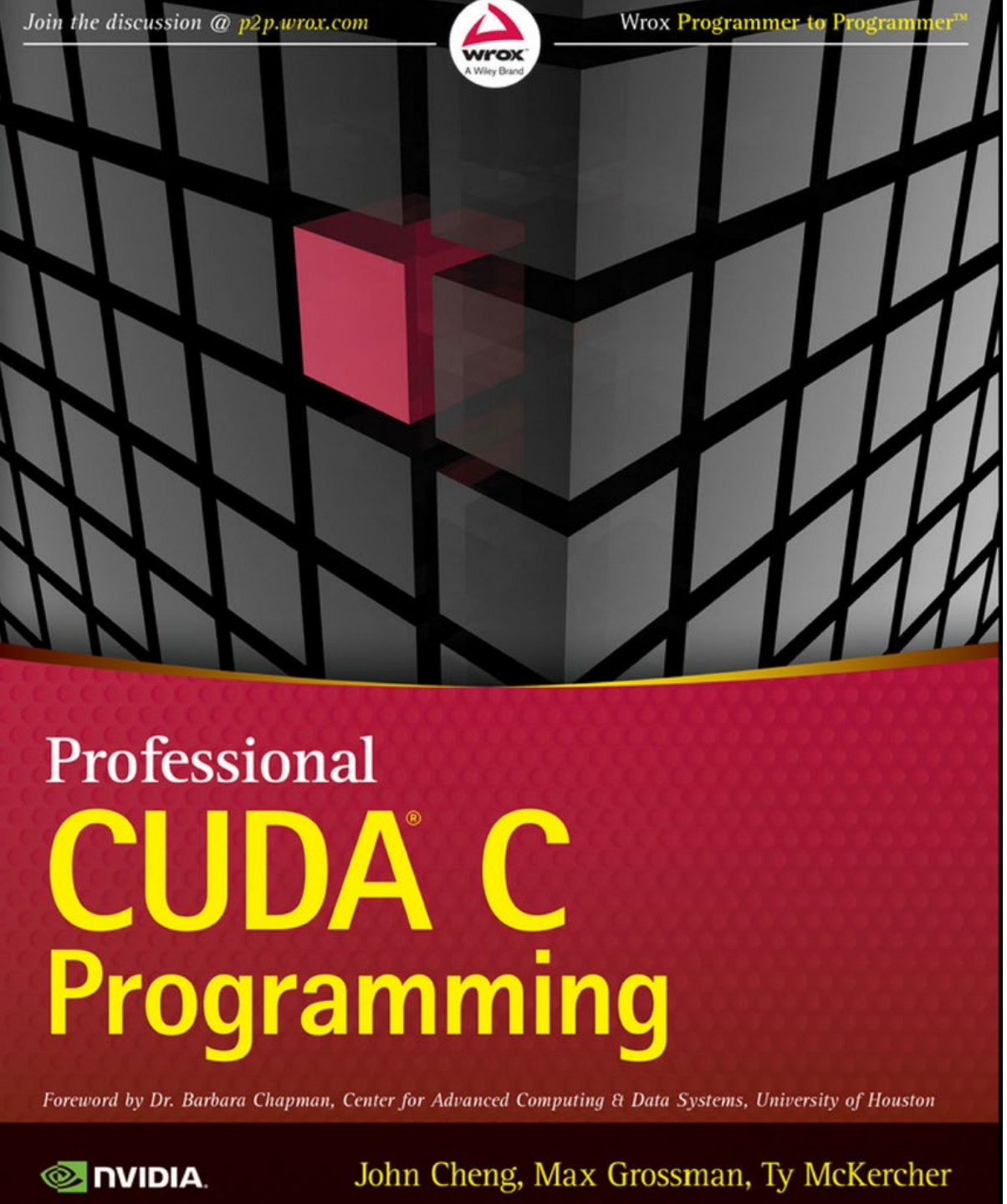
Programming on Parallel Machines

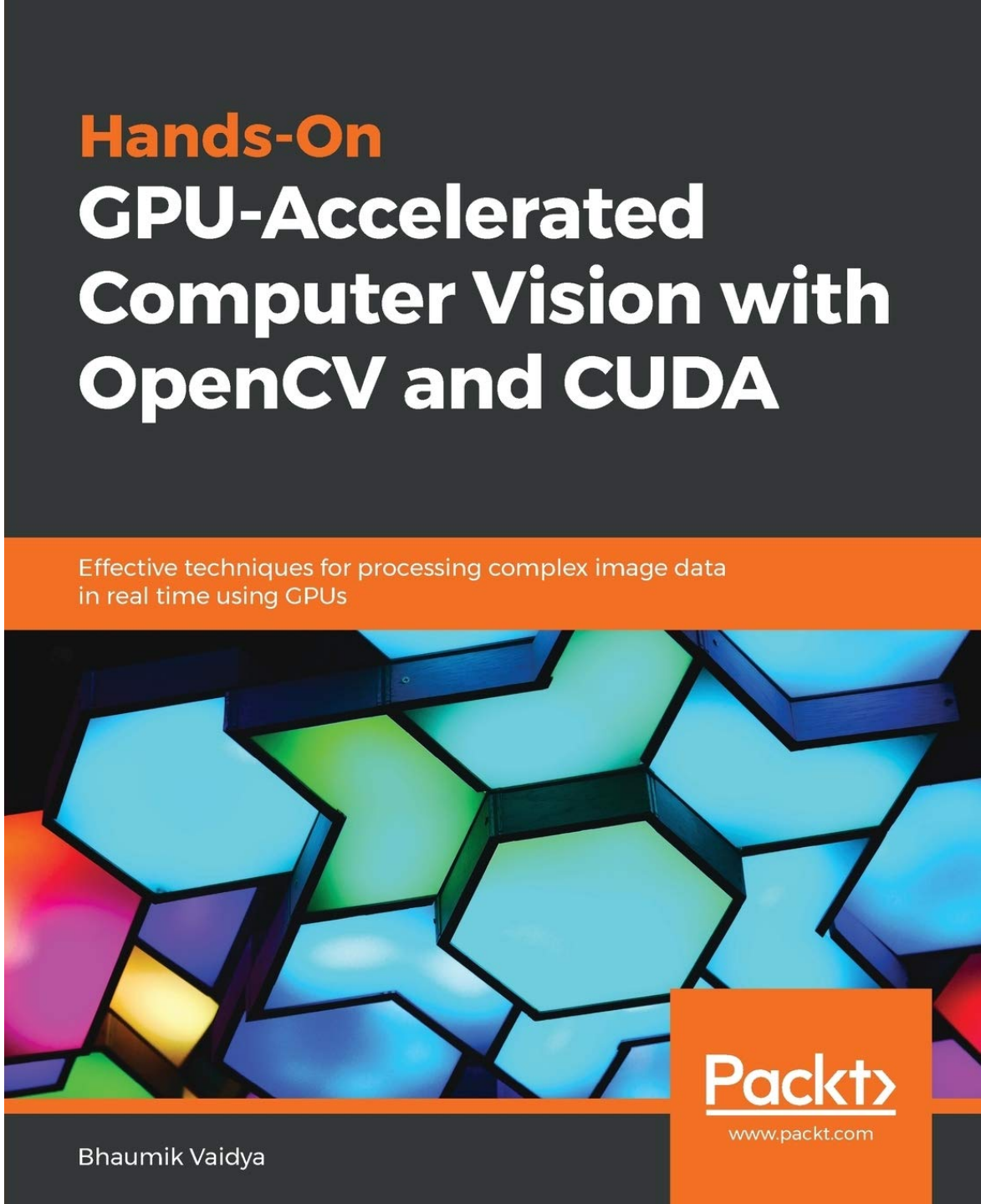
Norm Matloff

University of California, Davis

GPU, Multicore, Clusters and More

This book is often revised and updated, latest edition available at http://heather.cs.ucdavis.edu/ matloff/158/PLN/ParProcBook.pdf

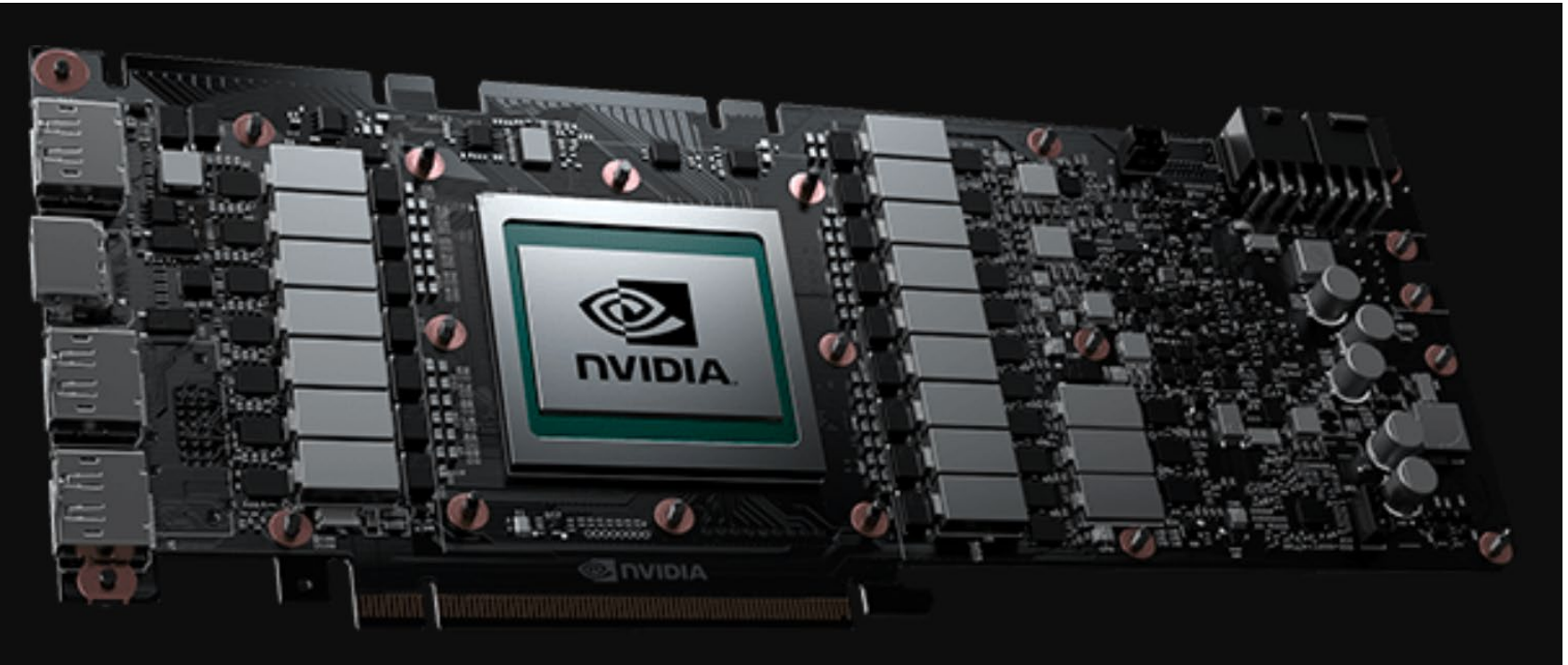

Join the discussion @ p2p.wrox.com
wrox
A Wiley Brand
Wrox Programmer to Programmer™
Professional
CUDA® C
Programming
Foreword by Dr. Barbara Chapman, Center for Advanced Computing & Data Systems, University of Houston
NVIDIA
John Cheng, Max Grossman, Ty McKercher


Hands-On
GPU-Accelerated
Computer Vision with
OpenCV and CUDA
Effective techniques for processing complex image data
in real time using GPUs
Packt
www.packt.com
Bhaumik Vaidya

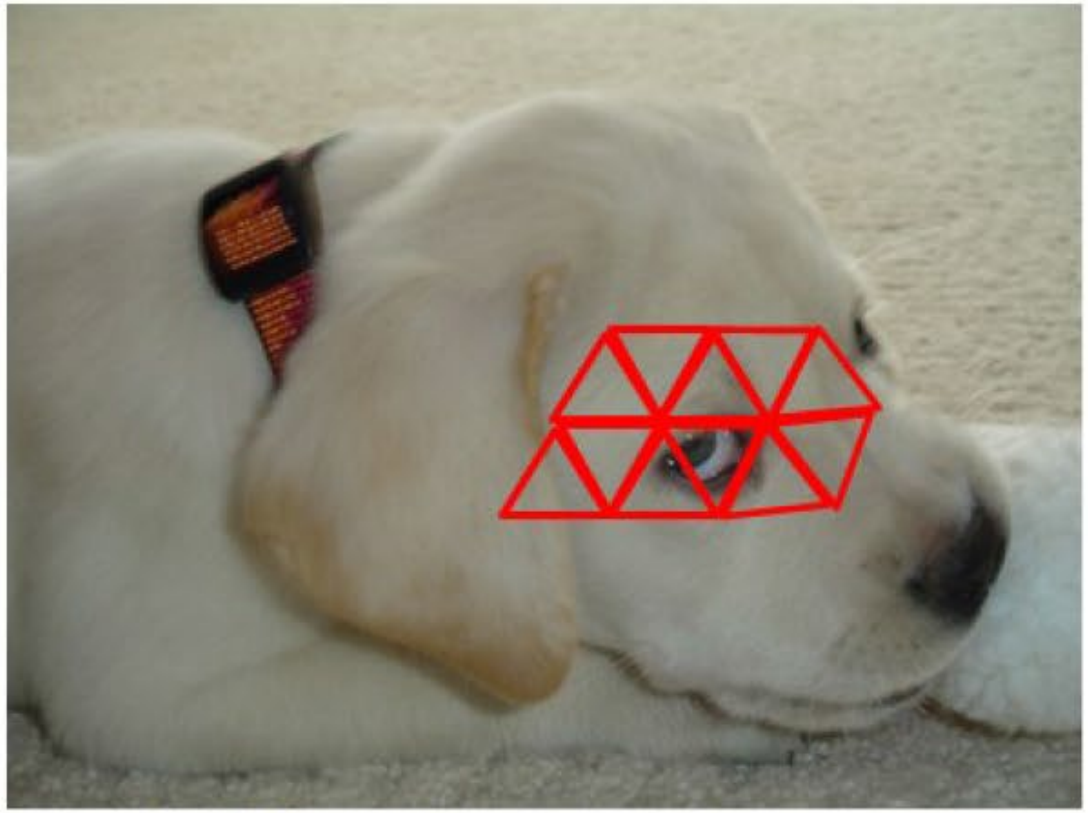
- En 1997 Intel introduce una unidad de procesamiento vectorial llamada MMX.

- Luego introduce SSE4, que es un conjunto de instrucciones SIMD.

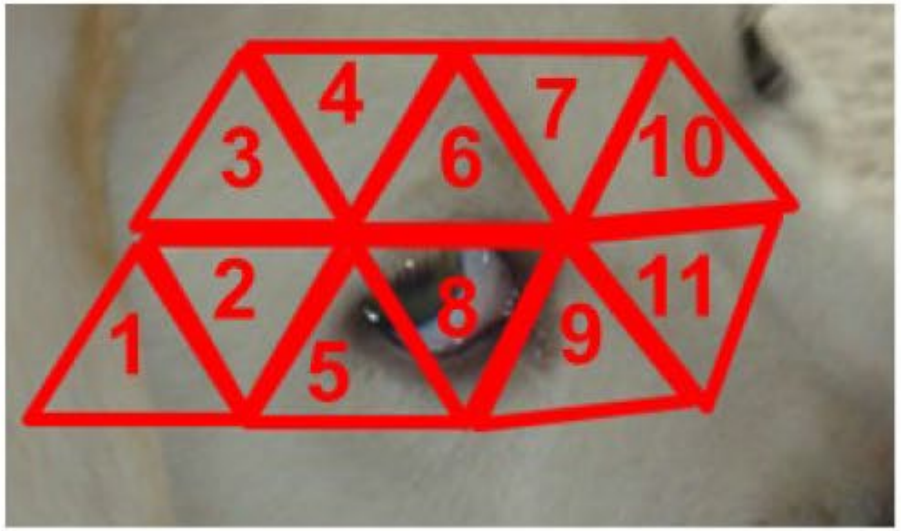
- GPU → Graphics Processing Unit



Early GPU Architectures

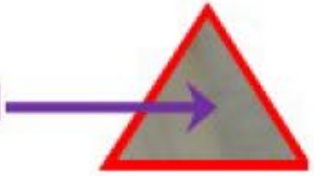


3D Object → Triangles

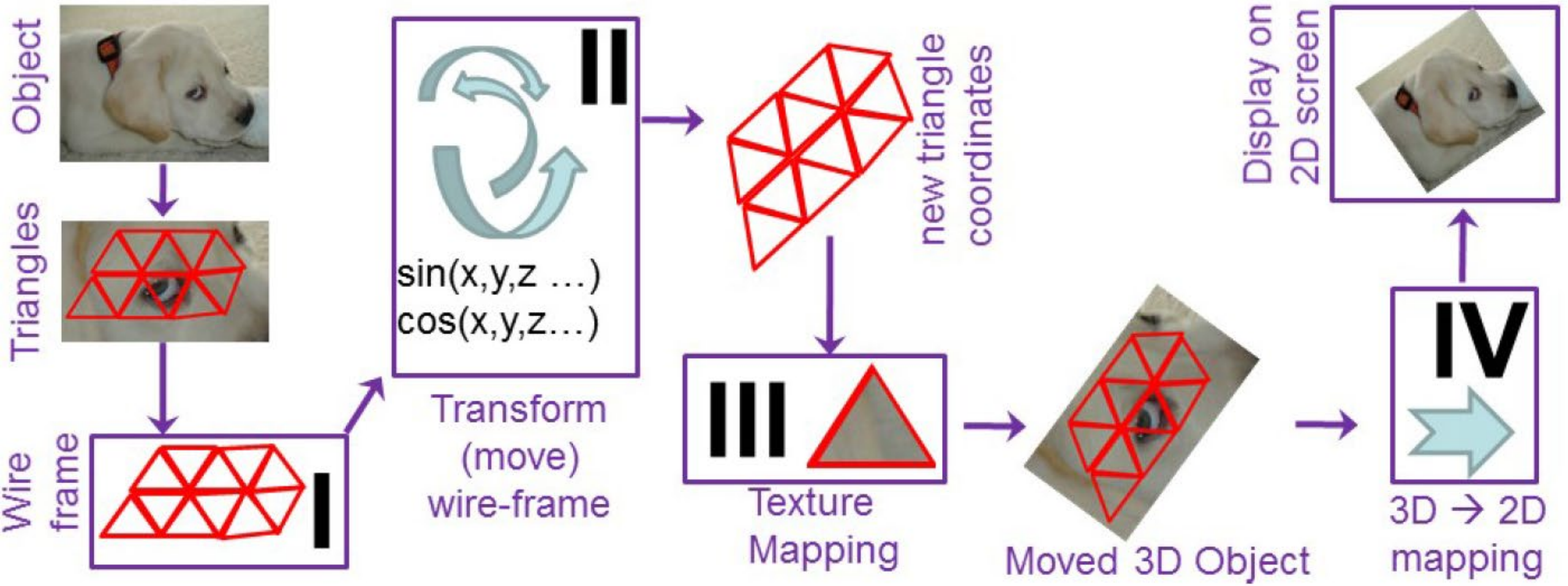


Triangle Coordinates

Tesselation

Texture Mapping 

- GPGPU → General Purpose GPU
- Intel, AMD y Nvidia.



Applications

Libraries	Compiler Directives	Programming Languages
Easy to use	Easy to Start	Most Performance
Most Performance	Portable Code	Most Flexibility
	OpenACC	CUDA

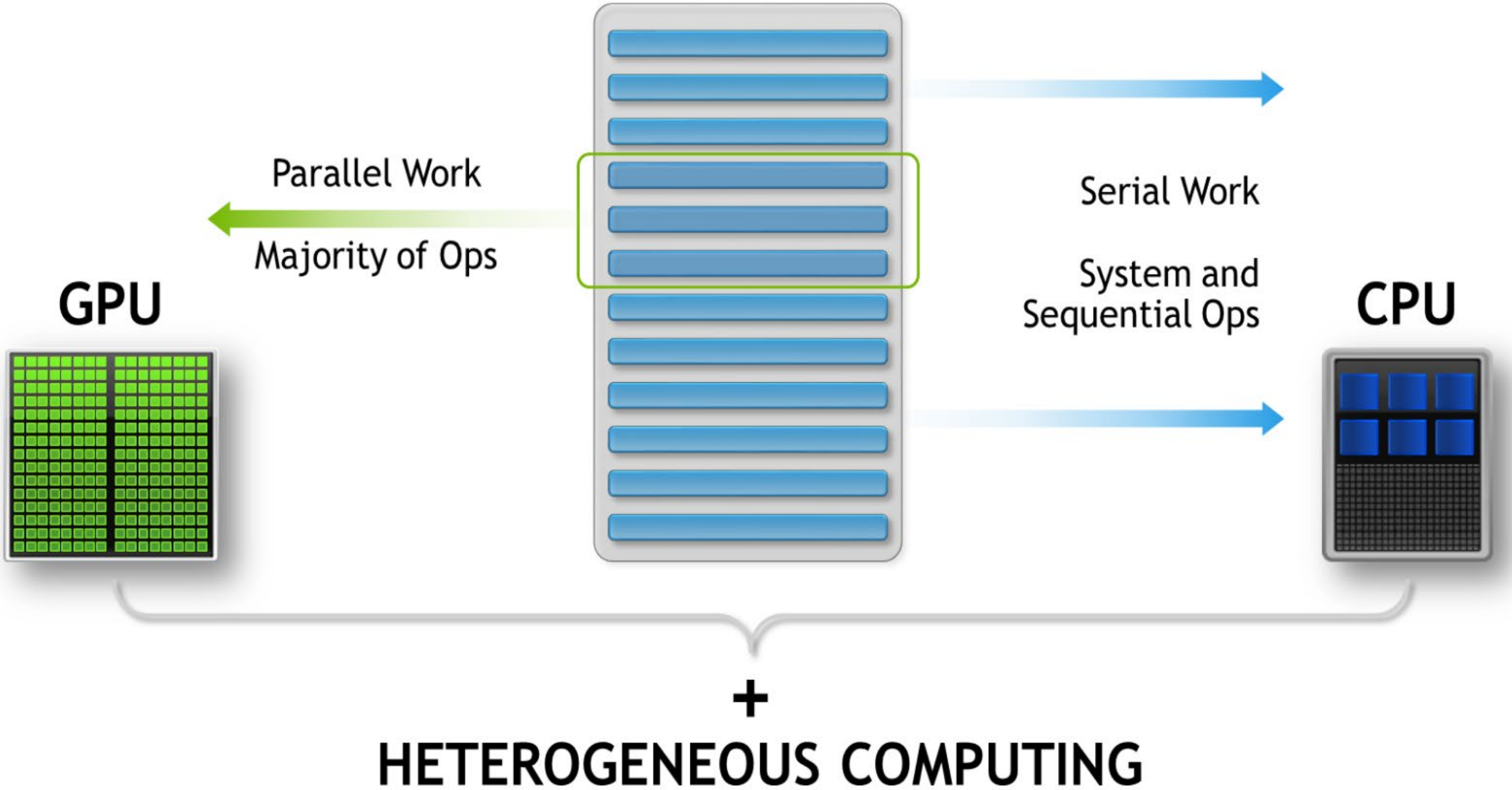
CUDA

- Nvidia introduced language Computer-Unifed Device Architecture (CUDA) in 2007.

- There are two predominant desktop GPU languages: OpenCL and CUDA.

- CUDA include host side code and a device side code (GPU code).

- *There is no such thing as GPU Programming ...*
- *GPU always interfaces to the CPU through certain APIs ...*
- *So, there is always CPU+GPU programming ...*

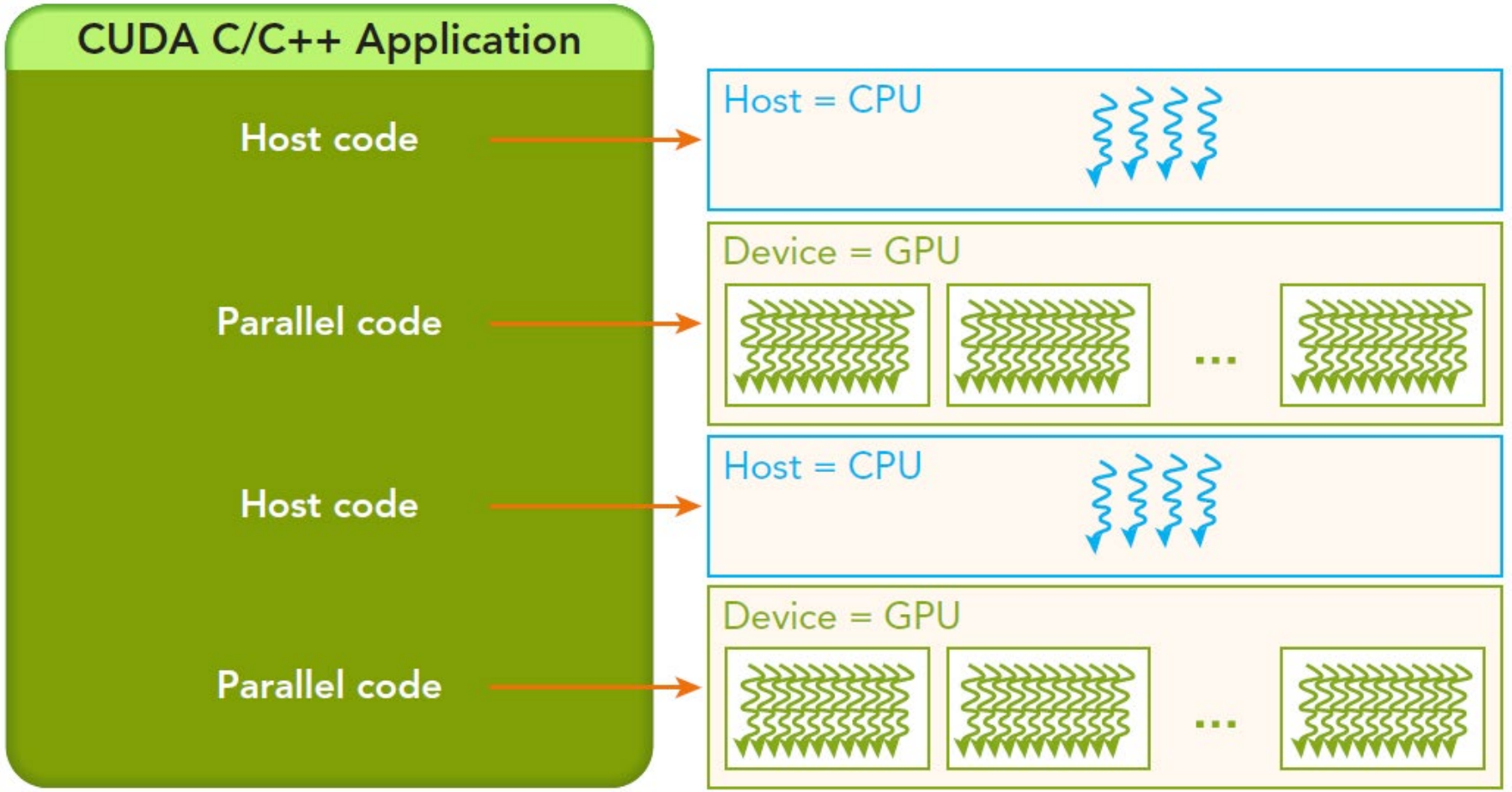


+

HETEROGENEOUS COMPUTING

Typical CUDA program

- The host code is written in ANSI C, and the device code is written using CUDA C.

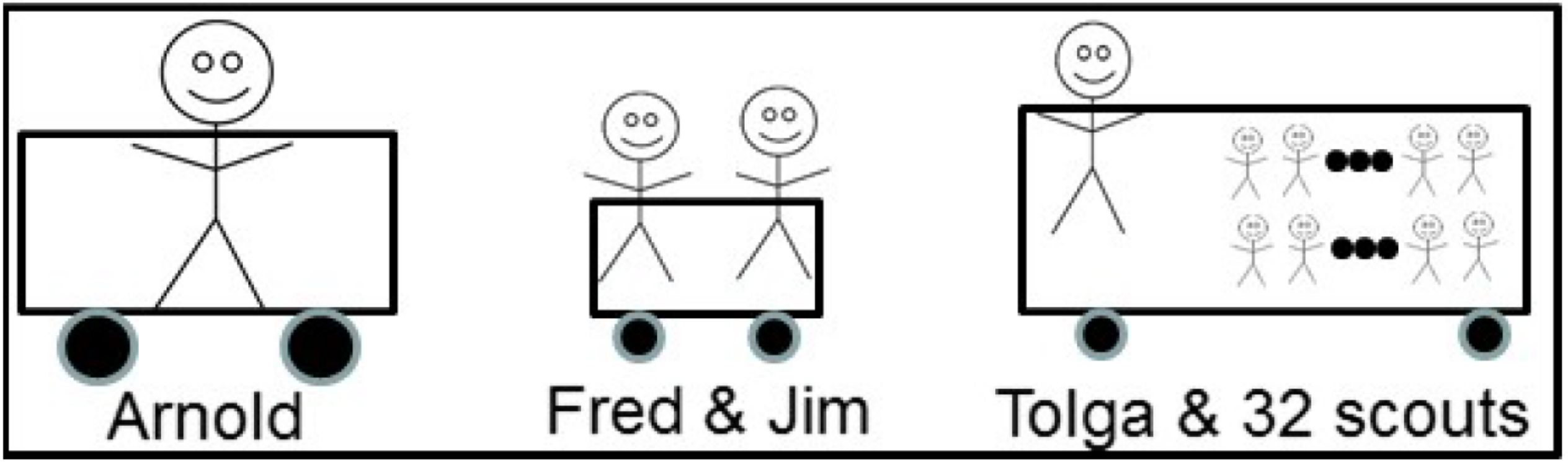


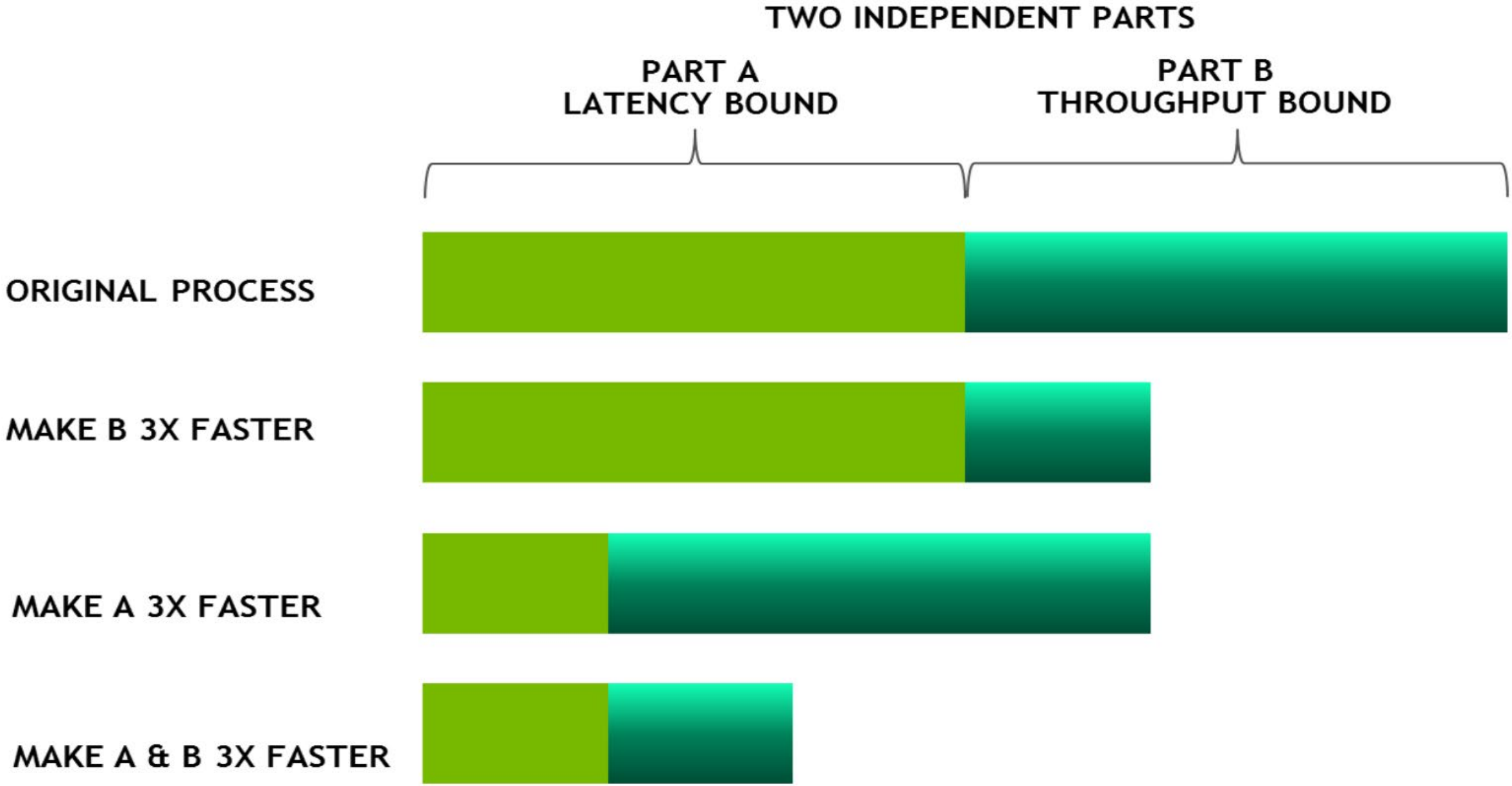
- GPU parallelism had to be exposed on the GPU side with a mechanism similar to the Pthreads.

- Nvidia designed its nvcc compiler that is capable of compiling CPU and GPU code simultaneously.

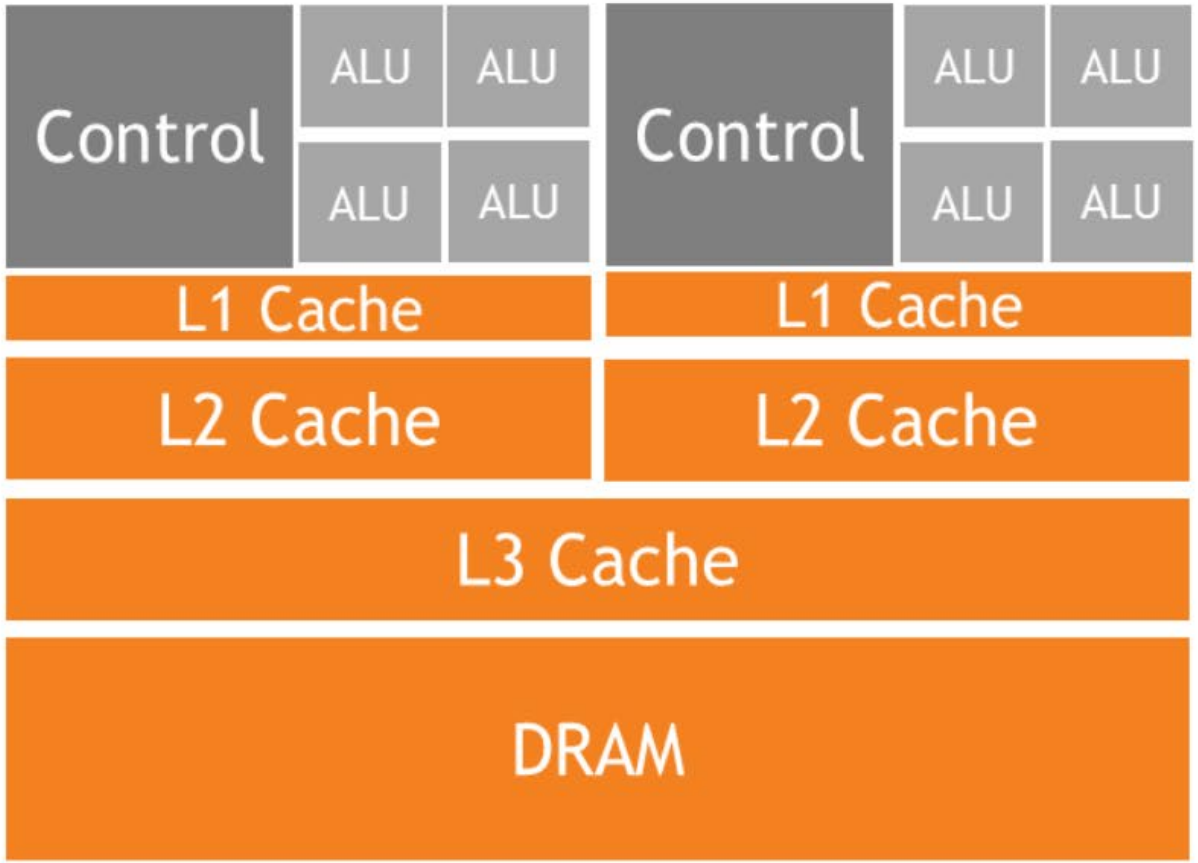
- The latest CUDA version is 10.2.

CPU vs GPU




TWO INDEPENDENT PARTS
PART A
LATENCY BOUND
PART B
THROUGHPUT BOUND
ORIGINAL PROCESS
MAKE B 3X FASTER
MAKE A 3X FASTER
MAKE A & B 3X FASTER

AMDAHL'S LAW


Control
ALU
ALU
ALU
ALU
Control
ALU
ALU
ALU
ALU
L1 Cache
L1 Cache
L2 Cache
L2 Cache
L3 Cache
DRAM

CPU


DRAM

GPU

- **Throughput** refers to the number of computations that can be performed simultaneously.

- **Latency** refers to the beginning-to-end duration of performing a single computation.

- **Amdahl's Law**

$$Speedup = \frac{1}{(1-p) + p/N}$$

How Does the GPU Achieve High Performance?

- The power that is consumed by the GPU is not proportional to the frequency of each core; instead, the dependence is something like quadratic.

- Each GPU core is in-order, work at lower frequencies, and their L1$ memories are not coherent.

- An OoO CPU executes them in the order of operand availability (Core i3, i5, and i7, as well as every Xeon).

- 32 threads glued together are called a “warp.“
- The minimum unit for data chunks is half warp or 16 elements.
- The GPU DRAM memory is made from GDDR5 (graphics double data rate type five synchronous dynamic random-access memory), which is parallel memory.
- Nvidia officially announced the first consumer graphics cards using GDDR6, the Turing-based GeForce RTX 2080 Ti, RTX 2080 & RTX 2070 on August 20, 2018,[14] RTX 2060 on January 6, 2019[15] and GTX 1660 Ti on February 22, 2019.

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

  $\rightarrow$ complement each other
- CPU: largely restricted to control tasks ($\sim$1000/sec)
  - Scripting fast enough
- Python + CUDA = **PyCUDA**
- Python + OpenCL = **PyOpenCL**

kepler versus pascal versus volta

NVIDIA Tesla K20 “Kepler” C-class Accelerator

- 2,496 CUDA cores, 2,496 = 13 SM $\times$ 192 cores/SM
- 5GB Memory at 208 GB/sec peak bandwidth
- peak performance: 1.17 TFLOPS double precision

NVIDIA Tesla P100 16GB “Pascal” Accelerator

- 3,586 CUDA cores, 3,586 = 56 SM $\times$ 64 cores/SM
- 16GB Memory at 720GB/sec peak bandwidth
- peak performance: 5.3 TFLOPS double precision

NVIDIA Tesla V100 32GB “Volta” Accelerator

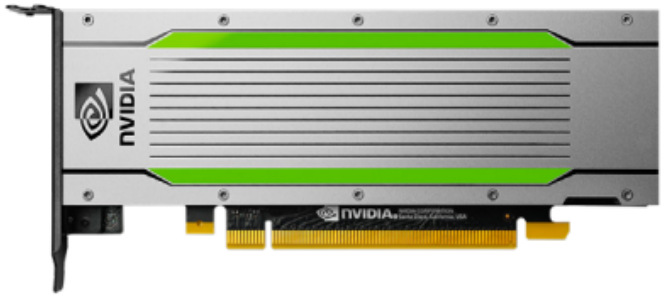
- 5,120 CUDA cores, 640 Tensor cores
- 32GB Memory at 870GB/sec peak bandwidth
- peak performance: 7.9 TFLOPS double precision

GEFORCE RTX 3090

GPU Engine Specs:	NVIDIA CUDA® Cores	10496
	Boost Clock (GHz)	1.70
	Base Clock (GHz)	1.40
Memory Specs:	Standard Memory Config	24 GB GDDR6X
	Memory Interface Width	384-bit
Technology Support:	Ray Tracing Cores	2nd Generation
	Tensor Cores	3rd Generation
	NVIDIA Architecture	Ampere
	Microsoft DirectX® 12 Ultimate	Yes
	NVIDIA DLSS	Yes
	PCI Express Gen 4	Yes
	NVIDIA Encoder	7th Generation
	NVIDIA Decoder	5th Generation
	VR Ready	Yes
Display Support:	Maximum Digital Resolution [1]	7680x4320
	Standard Display Connectors	HDMI 2.1, 3x DisplayPort 1.4a
	Multi Monitor	4
	HDCP	2.3
Founders Edition Card Dimensions:	Length	12.3" (313 mm)
	Width	5.4" (138 mm)
	Slot	3-Slot
Founders Edition Thermal Power Specs:	Maximum GPU Temperature (in C)	93
	Graphics Card Power (W)	350
	Required System Power (W) [2]	750

Tesla T4 Datacenter GPU

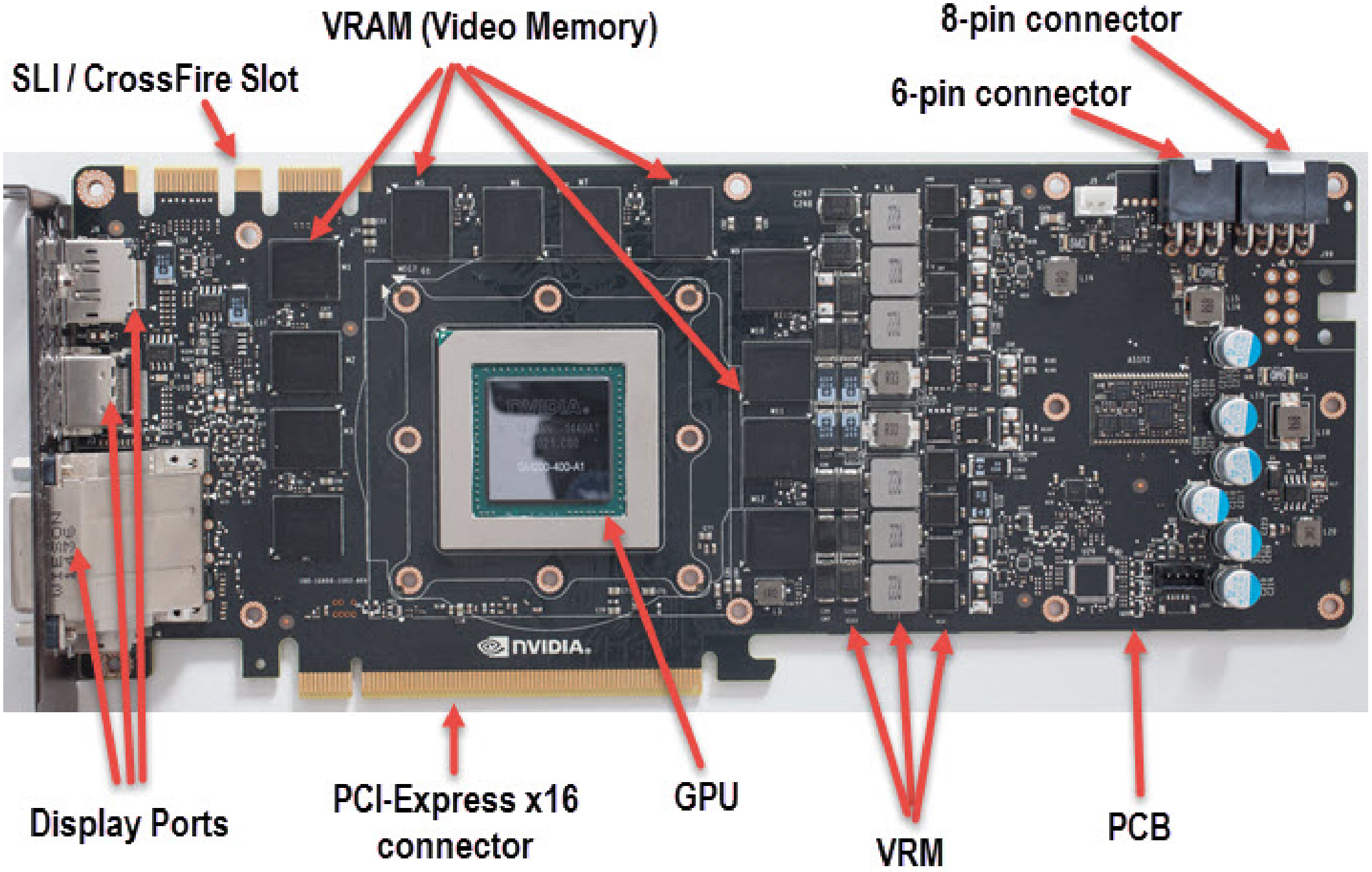
GPU clocks	Base	585 MHz
	Maximum Boost	1590 MHz



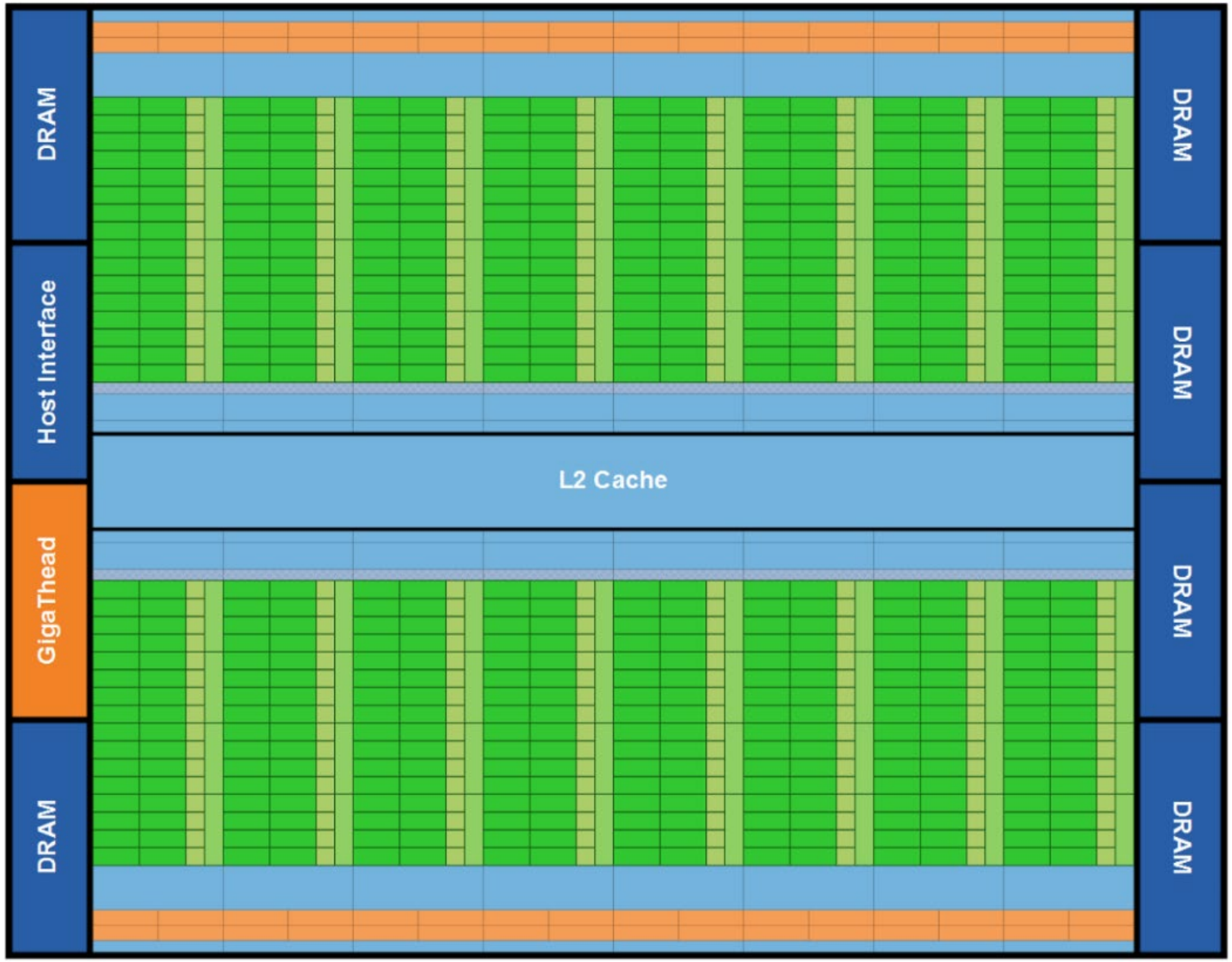
SPECIFICATIONS

GPU Architecture	**NVIDIA Turing**
NVIDIA Turing Tensor Cores	**320**
NVIDIA CUDA® Cores	**2,560**
Single-Precision	**8.1 TFLOPS**
Mixed-Precision (FP16/FP32)	**65 TFLOPS**
INT8	**130 TOPS**
INT4	**260 TOPS**
GPU Memory	**16 GB GDDR6 300 GB/sec**
ECC	**Yes**
Interconnect Bandwidth	**32 GB/sec**
System Interface	**x16 PCIe Gen3**
Form Factor	**Low-Profile PCIe**
Thermal Solution	**Passive**
Compute APIs	**CUDA, NVIDIA TensorRT™, ONNX**

https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-product-brief.pdf

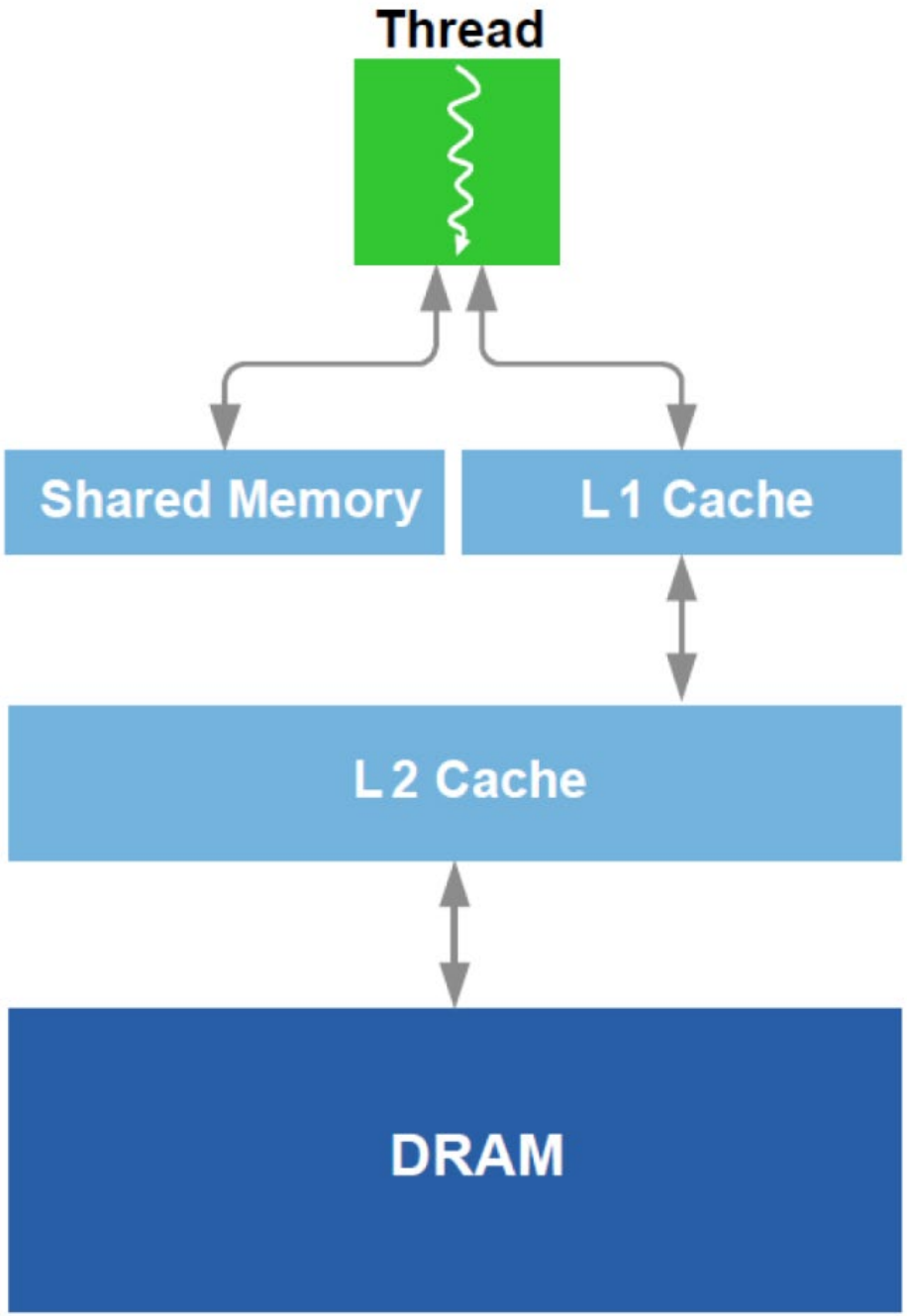


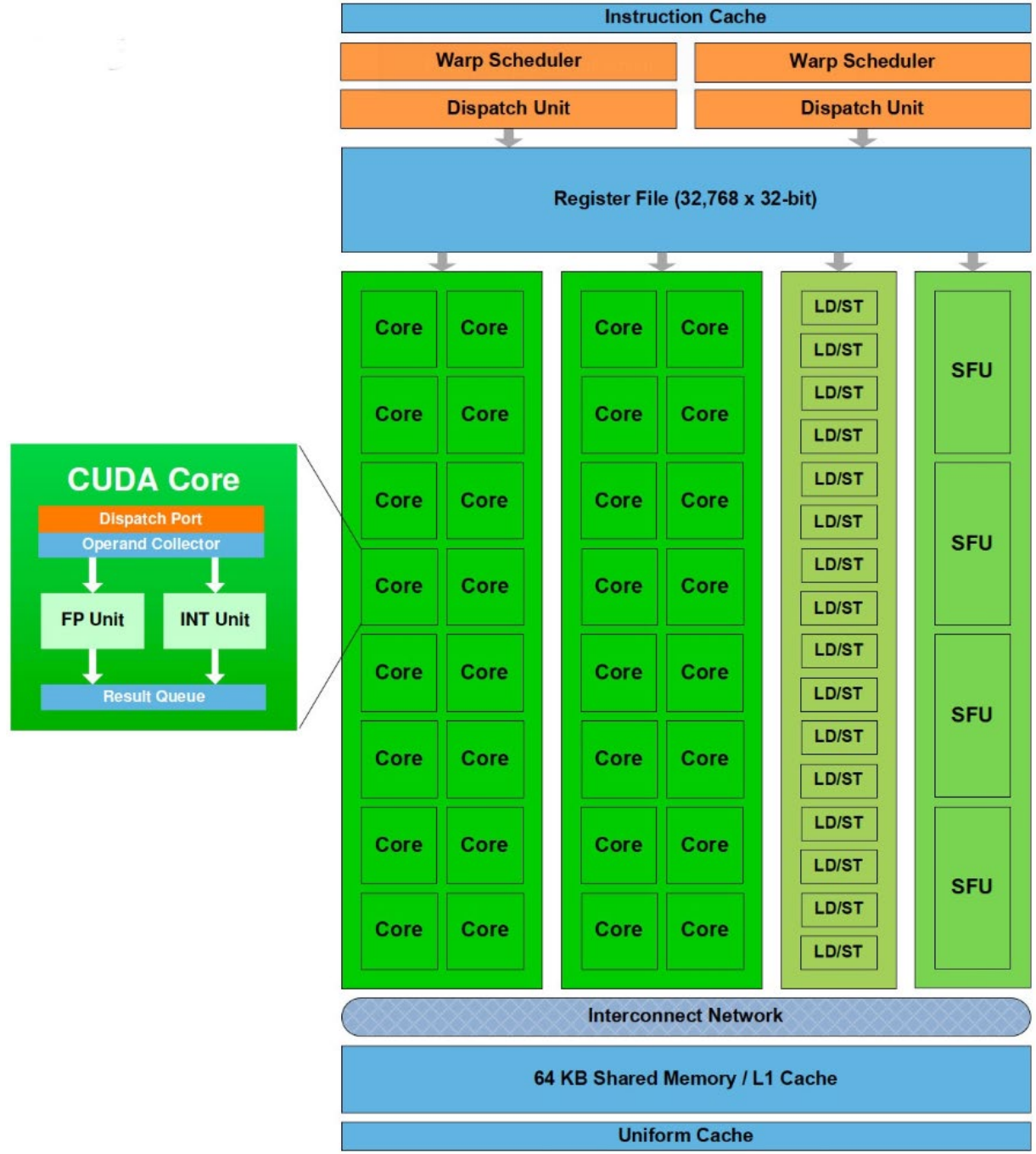
https://graphicscardhub.com/wp-content/uploads/2017/03/graphics-card-components.jpg



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

Fermi Memory Hierarchy





Fermi Streaming Multiprocessor (SM)

NVIDIA Titan Xp GPU specifications

Get specs HTML code

Name / Brand / Architecture	
Manufacturer:	NVIDIA
Model:	Titan Xp
Target market segment:	Desktop
Die name:	GP102
Architecture:	Pascal
Fabrication process:	16 nm
Transistors:	12.0 billion
Bus interface:	PCI-E 3.0 x 16
Launch date:	April 2017
Price at launch:	$1199

Frequency	
Base clock:	1405 MHz
GPU boost:	3.0
Boost clock:	1582 MHz

Memory specifications	
Memory size:	12 GB
Memory type:	GDDR5X
Memory clock:	1426 MHz
Memory clock (effective):	11408 MHz
Memory interface width:	384-bit
Memory bandwidth:	547.58 GB/s

Cores / Texture	
CUDA:	6.1
CUDA cores:	3840
ROPs:	96
Texture units:	240

SLI / Crossfire	
Maximum SLI options:	4-way

Electric characteristics	
Maximum power draw:	250 W

Video features	
Multi-monitor:	Up to 6 displays
Maximum digital resolution:	7680 x 4320 @60 Hz
Maximum DP resolution:	7680 x 4320 @60 Hz

Invest in an extremely high quality (and high eciency) power supply.

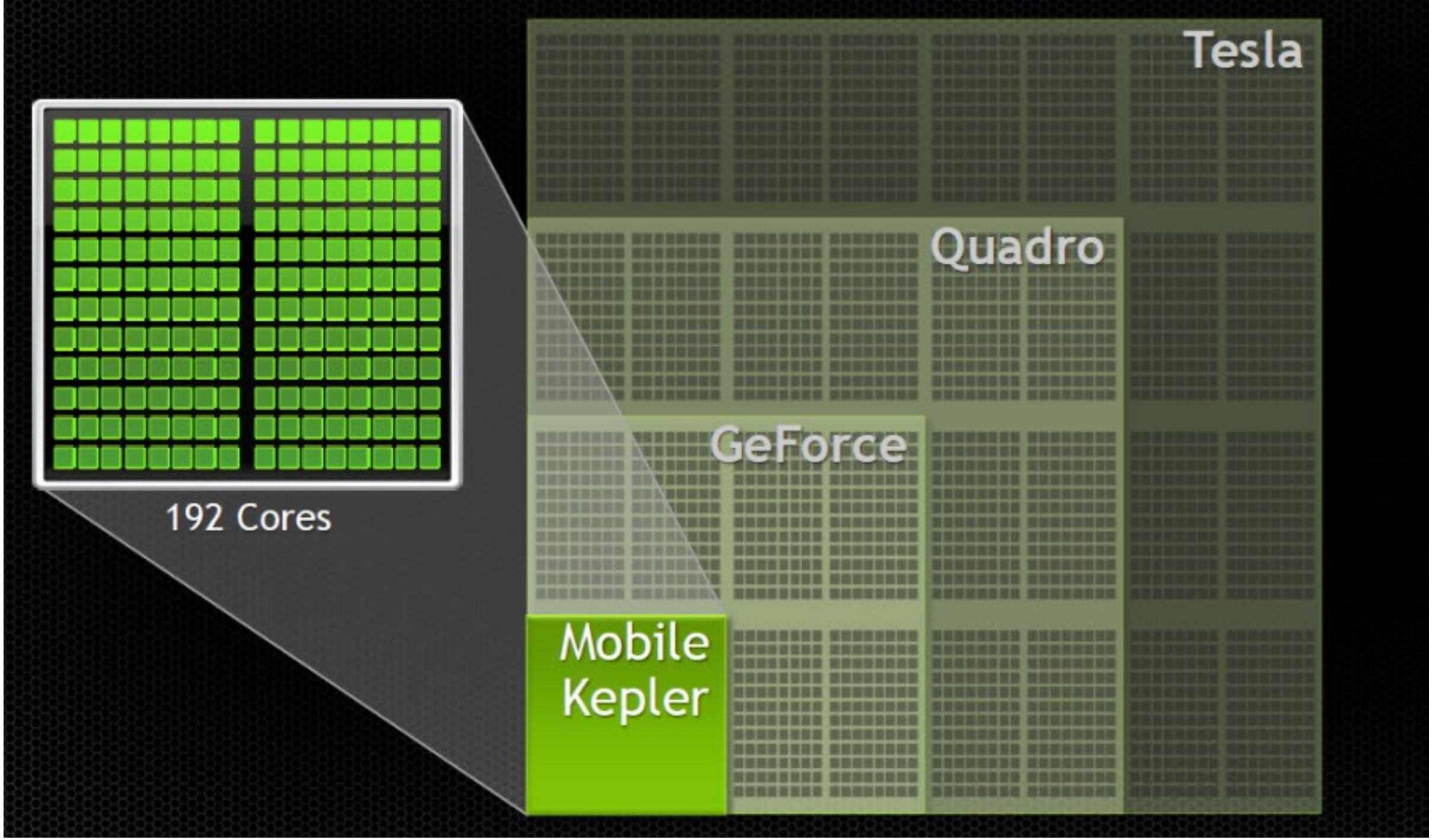
Specify your power supply very generously.
Ideally, your peak power consumption should be 50{60% of the peak.

Invest in a liquid-cooled block for the GPU to exhaust heat quickly.

Invest in an extremely high-quality computer chassis with lots of cooling.
They should have multiple 120mm or 140mm fans.

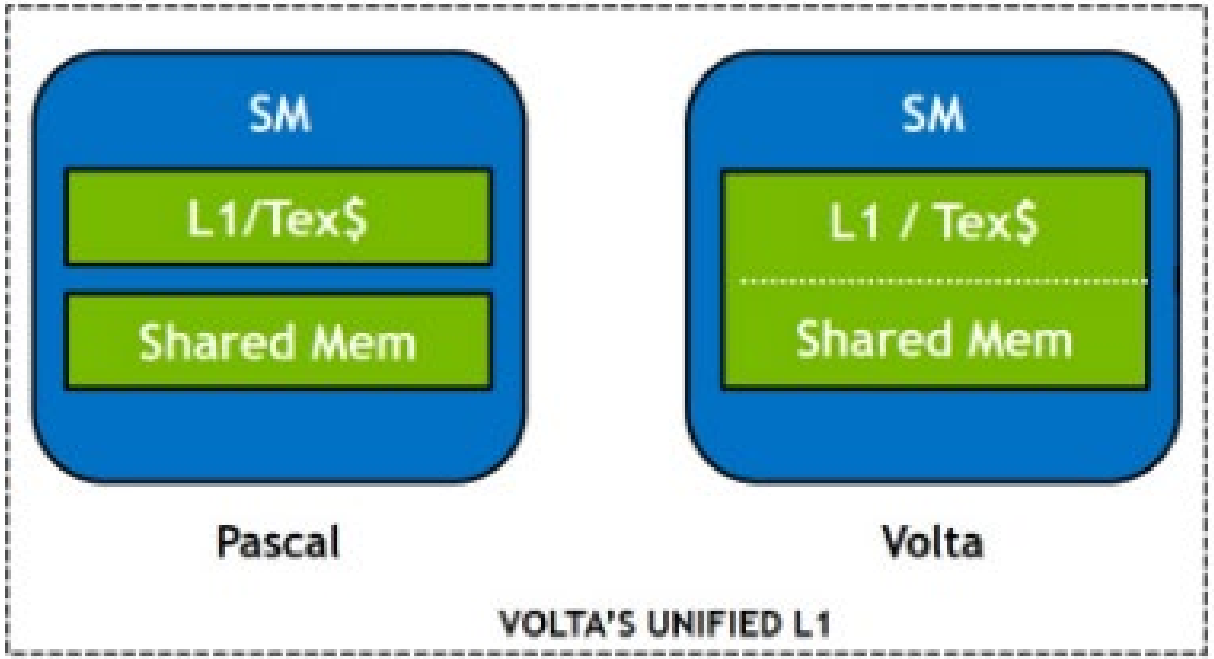
Get an Extreme Intel CPU or a Xeon

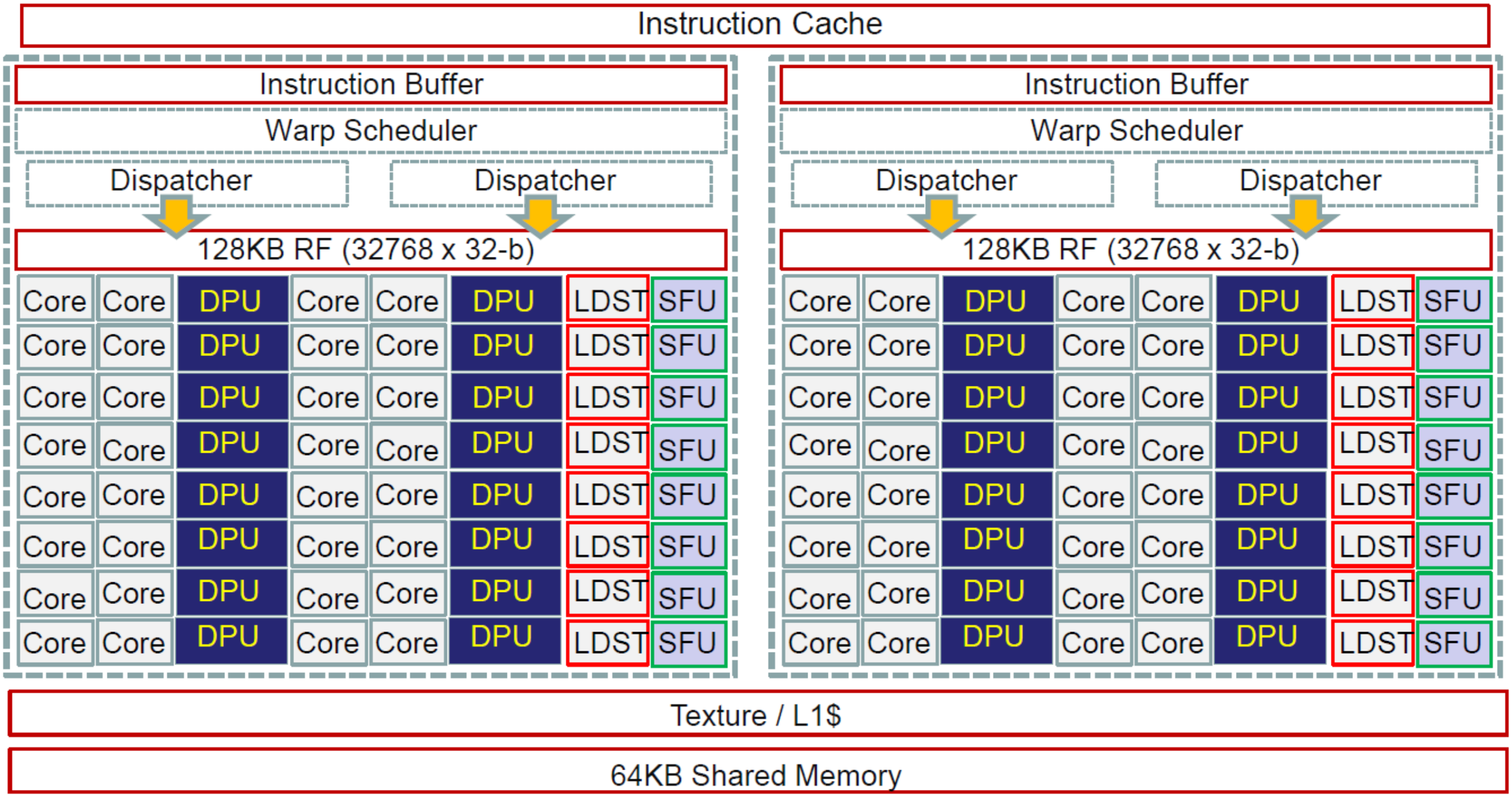
Invest in a Liquid CPU Cooler.


Tesla
Quadro
GeForce
Mobile
Kepler
192 Cores

GPU Memory evolution

Memory type	Properties	Volta V100	Pascal P100	Maxwell M60	Kepler K80
Register	Size per SM	256 KB	256 KB	256 KB	256 KB
L1	Size	32...128 KiB	24 KiB	24 KiB	16...48 KiB
	Line size	32	32 B	32 B	128 B
L2	Size	6144 KiB	4,096 KiB	2,048 KiB	1,536 Kib
	Line size	64 B	32B	32B	32B
Shared memory	Size per SMX	Up to 96 KiB	64 KiB	64 KiB	48 KiB
	Size per GPU	up to 7,689 KiB	3,584 KiB	1,536 KiB	624 KiB
	Theoretical bandwidth	13,800 GiB/s	9,519 GiB/s	2,410 GiB/s	2,912 GiB/s
Global memory	Memory bus	HBM2	HBM2	GDDR5	GDDR5
	Size	32,152 MiB	16,276 MiB	8,155 MiB	12,237 MiB
	Theoretical bandwidth	900 GiB/s	732 GiB/s	160 GiB/s	240 GiB/s

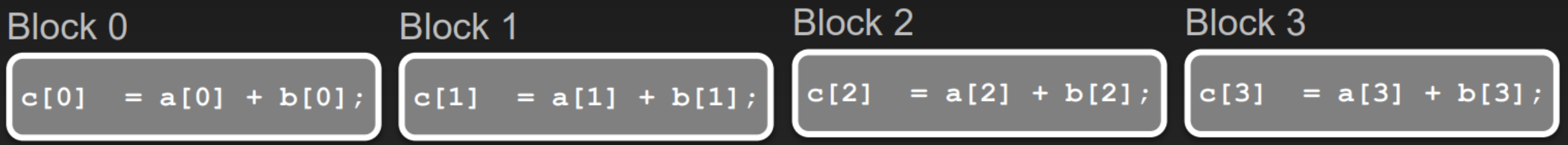



Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatcher
Dispatcher
128KB RF (32768 x 32-b)
Core
DPU
LDST
SFU
Instruction Buffer
Warp Scheduler
Dispatcher
Dispatcher
128KB RF (32768 x 32-b)
Texture / L1$
64KB Shared Memory

Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:



https://obj.umiacs.umd.edu/gpusummit/slides/04_maryland_cuda_hands-on.pdf

May 2, 2020

A HISTORY OF NVIDIA STREAM MULTIPROCESSOR

I spent last week-end getting accustomed to CUDA and SIMT programming. It was a prolific time ending up with a Business Card Raytracer running close to 700x faster[1], from 101s to 150ms.

This pleasant experience was a good pretext to spend more time on the topic and learn about the evolution of Nvidia architecture. Thanks to the abundant documentation published over the years by the green team, I was able to go back in time and fast forward though the fascinating evolution of their stream multiprocessors.

Visited in this article:

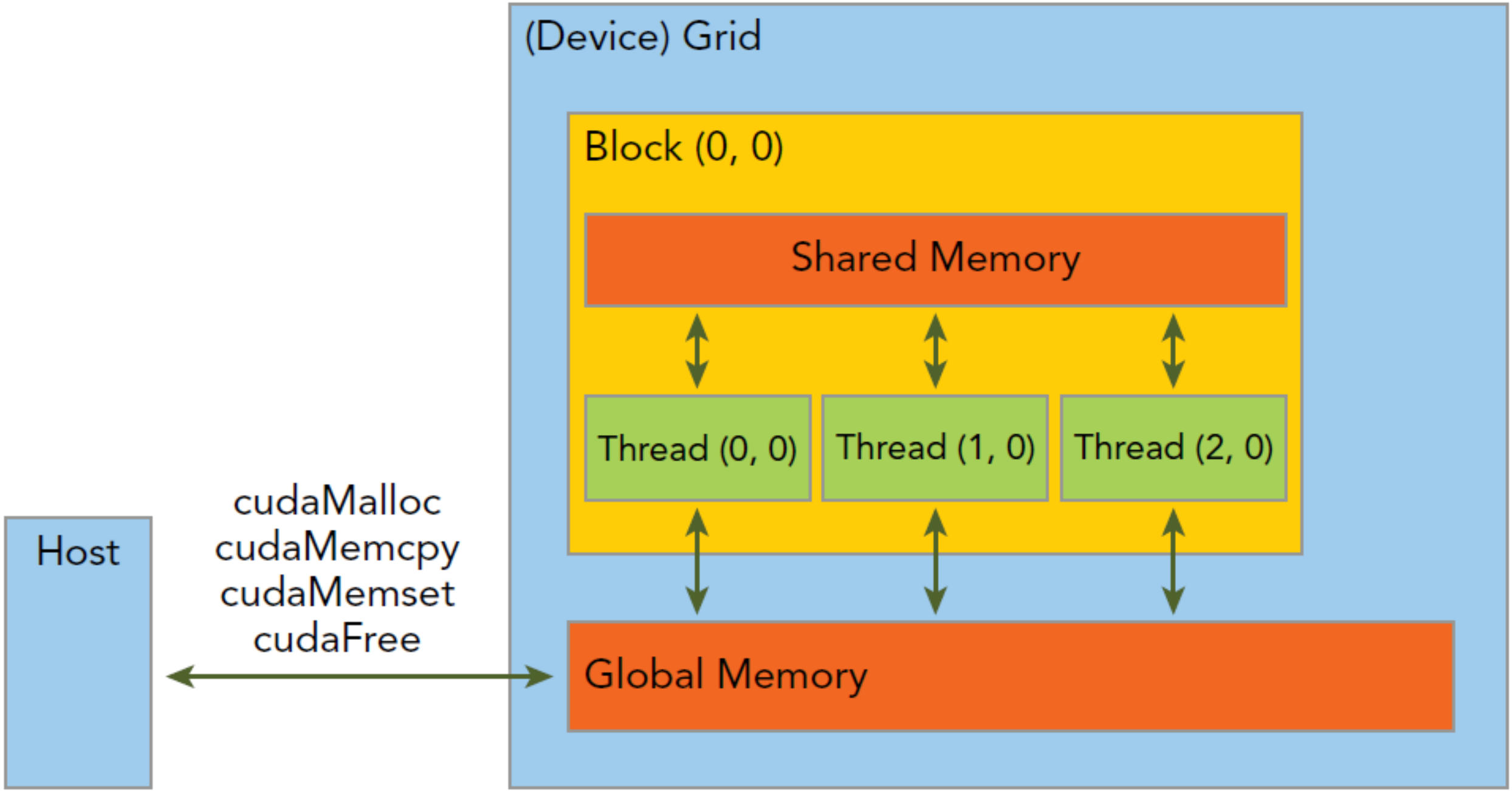
Year	Arch	Series	Die	Process	Enthusiast Card
2006	Tesla	GeForce 8	G80	90 nm	8800 GTX
2010	Fermi	GeForce 400	GF100	40 nm	GTX 480
2012	Kepler	GeForce 600	GK104	28 nm	GTX 680
2014	Maxwell	GeForce 900	GM204	28 nm	GTX 980 Ti
2016	Pascal	GeForce 10	GP102	16 nm	GTX 1080 Ti
2018	Turing	GeForce 20	TU102	12 nm	RTX 2080 Ti

Host and Device Memory Functions

STANDARD C FUNCTIONS	CUDA C FUNCTIONS
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

```
cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count,
    cudaMemcpyKind kind )
```

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice


(Device) Grid
Block (0, 0)
Shared Memory
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Global Memory
Host
cudaMalloc
cudaMemcpy
cudaMemset
cudaFree

CUDA: Installation Ubuntu 18.04

- https://medium.com/leadkaro/setting-up-pycuda-on-ubuntu-18-04-for-gpu-programming-with-python-830e03fc4b81
- sudo apt-get install python3-venv
- sudo python3 -m venv v_pycuda
- source v_pycuda/bin/activate
- pip3 install --upgrade pip
- pip3 install pycuda

```
(v_pycuda) r@r:~/v_pycuda$ sudo gedit pyvenv.cfg
```

Open

*pyvenv.cfg
/home/r/v_pycuda

```
1 home = /usr/bin
2 include-system-site-packages = true
3 version = 3.8.5
```

Google colab

- 34 GB of disk
- 12 GB RAM
- **Colab (Free)** — Intel(R) Xeon(R) CPU @ 2.20GHz

- **Colab (Free)** — Tesla K80

https://colab.research.google.com

https://towardsdatascience.com/colab-pro-is-it-worth-the-money-32a1744f42a8

Google colab

In the free version of Colab, notebooks can run for at most 12 hours, and idle timeouts are much stricter than in Colab Pro.

How can I get the most out of Colab Pro?

Resources in Colab Pro are prioritized for subscribers who have recently used less resources, in order to prevent the monopolization of limited resources by a small number of users. To get the most out of Colab Pro, consider closing your Colab tabs when you are done with your work, and avoid opting for GPUs or extra memory when it is not needed for your work. This will make it less likely that you will run into usage limits within Colab Pro. For more information, see Making the Most of your Colab Subscription

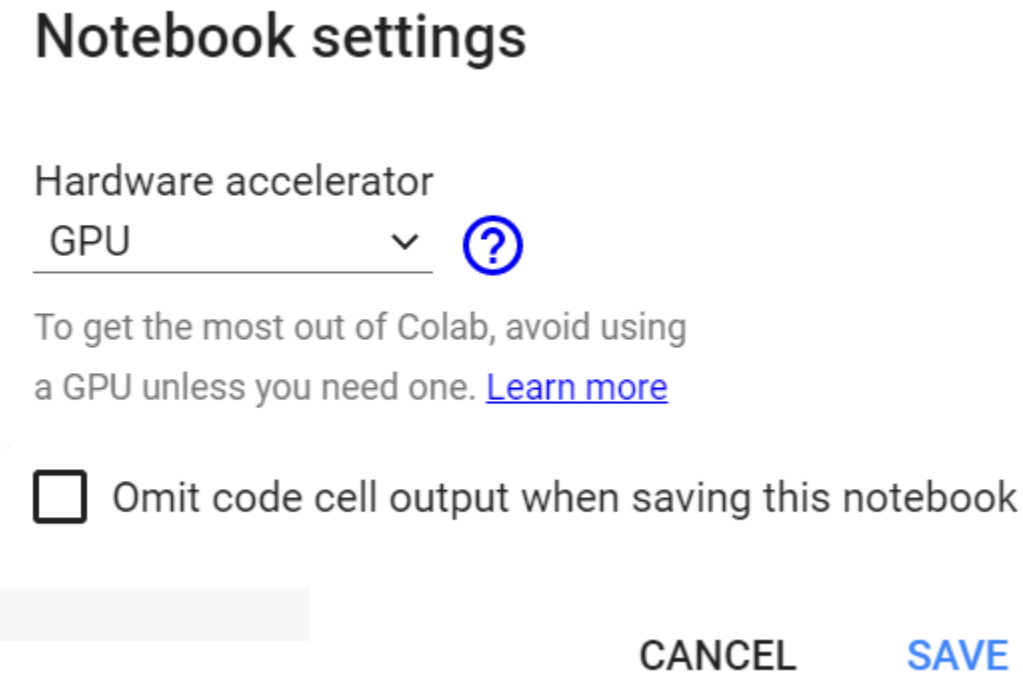
What kinds of GPUs are available in Colab Pro?

With Colab Pro you get priority access to our fastest GPUs. For example, you may get access to T4 and P100 GPUs at times when non-subscribers get K80s. You also get priority access to TPUs. There are still usage limits in Colab Pro, though, and the types of GPUs and TPUs available in Colab Pro may vary over time.

In the free version of Colab there is very limited access to faster GPUs and to TPUs, and usage limits are much lower than they are in Colab Pro.

https://colab.research.google.com/signup

- **Runtime disconnection** — the runtime will disconnect if the notebook is sitting idle for some time
- **Memory** — it is possible to run out of memory
- **Dependence** — there's no guarantee that GPU or TPU will be available at the time you need it, so keep that in mind




```
pip install pycuda
```

```
Collecting pycuda
  Downloading https://files.pythonhosted.org/packages/5a/56/4682a5118a234d15aa1c8768a528aac4858c7b04d2674e18d586d3dfda04/pycuda-2021.1.tar.gz (1.7MB)
     |████████████████████████████████| 1.7MB 24.9MB/s
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
    Preparing wheel metadata ... done
Collecting mako
  Downloading https://files.pythonhosted.org/packages/f3/54/dbc07fbb20865d3b78fdb7cf7fa713e2cba4f87f71100074ef2dc9f9d1f7/Mako-1.1.4-py2.py3-none-any.whl (75kB)
     |████████████████████████████████| 81kB 11.3MB/s
Requirement already satisfied: appdirs>=1.4.0 in /usr/local/lib/python3.7/dist-packages (from pycuda) (1.4.4)
Collecting pytools>=2011.2
  Downloading https://files.pythonhosted.org/packages/49/5b/136e5688da9bbd915ee8190bfd6a007fc0b19d71f26d5a2ab4b737b2eeb4/pytools-2021.2.6.tar.gz (63kB)
     |████████████████████████████████| 71kB 10.9MB/s
Requirement already satisfied: MarkupSafe>=0.9.2 in /usr/local/lib/python3.7/dist-packages (from mako->pycuda) (1.1.1)
Requirement already satisfied: numpy>=1.6.0 in /usr/local/lib/python3.7/dist-packages (from pytools>=2011.2->pycuda) (1.19.5)
Building wheels for collected packages: pycuda
  Building wheel for pycuda (PEP 517) ... done
  Created wheel for pycuda: filename=pycuda-2021.1-cp37-cp37m-linux_x86_64.whl size=627880 sha256=723a3400c9fbabf83a6052197e623be15ad257af87f63ed63ae3847bdf4fb419
  Stored in directory: /root/.cache/pip/wheels/d5/55/64/fd4dddcc5f1c25eebd90b5291c3769101dc978c70165685512
Successfully built pycuda
Building wheels for collected packages: pytools
  Building wheel for pytools (setup.py) ... done
  Created wheel for pytools: filename=pytools-2021.2.6-py2.py3-none-any.whl size=60643 sha256=3ed1befe875278349be59668293cda556ecc659ba5e026476dcec7ab272e42b7
  Stored in directory: /root/.cache/pip/wheels/8c/a6/65/447b9b4fd1d9bde84ad2fea2431a38f69f3fb573476a98ae03
Successfully built pytools
Installing collected packages: mako, pytools, pycuda
Successfully installed mako-1.1.4 pycuda-2021.1 pytools-2021.2.6
```

https://towardsdatascience.com/google-colab-how-does-it-compare-to-a-gpu-enabled-laptop-851c1e0a2ca9

PyCUDA exposes *all* of CUDA.

For example:

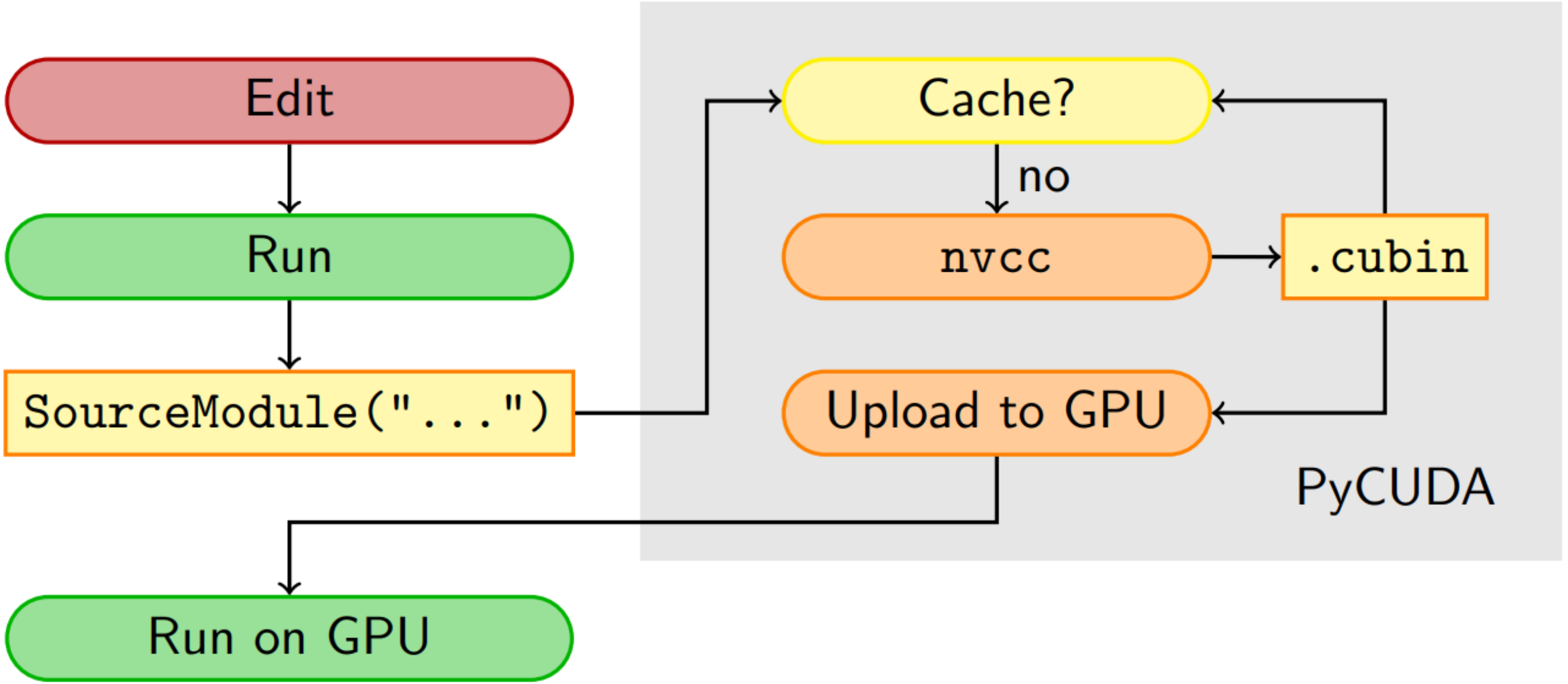
- Arrays and Textures
- Pagelocked host memory
- Memory transfers (asynchronous, structured)
- Streams and Events
- Device queries
- GL Interop

http://lorenabarba.com/gpuatbu/Program_files/kloeckner-slides.pdf

PyCUDA: Vital Information

- `http://mathema.tician.de/software/pycuda`
- Complete documentation
- MIT License
  (no warranty, free for all use)
- Requires: numpy, Python 2.4+
  (Win/OS X/Linux)
- Support via mailing list

https://documen.tician.de/pycuda/

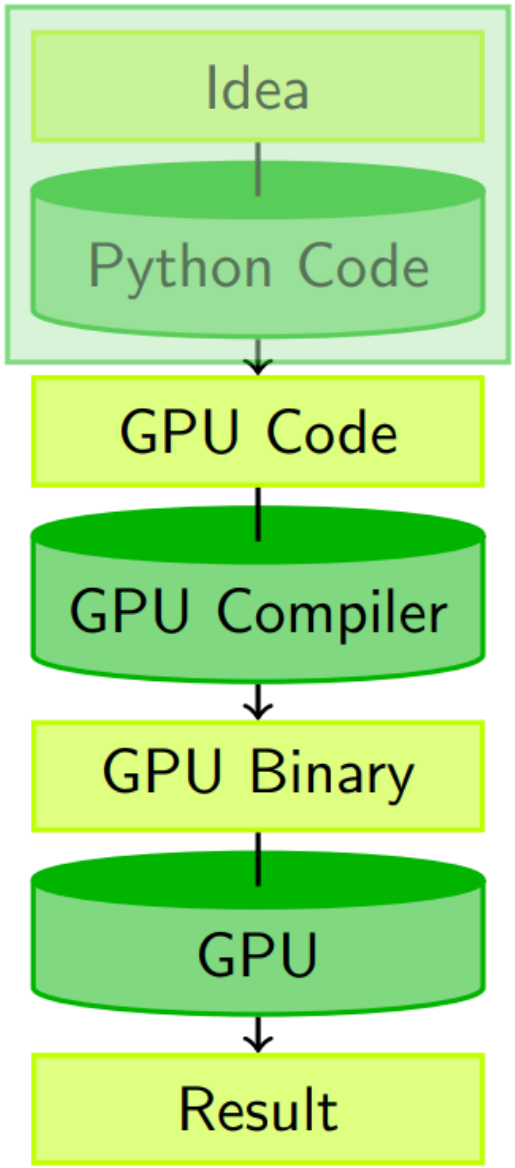


https://andreask.cs.illinois.edu/dl/main.pdf

`pycuda.gpuarray`:

- Meant to look and feel just like `numpy`.
  - `gpuarray.to_gpu(numpy_array)`
  - `numpy_array = gpuarray.get()`
- +, -, *, /, fill, sin, exp, rand,
  basic indexing, norm, inner product, ...
- Mixed types (int32 + float32 = float64)
- `print gpuarray` for debugging.
- Allows access to raw bits
  - Use as kernel arguments, textures, etc.

Metaprogramming



In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

PyCUDA advantages

- Abstractions like **pycuda.compiler.SourceModule** and **pycuda.gpuarray.GPUArray** make CUDA programming convenient

- Automatic Error Checking. All CUDA errors are automatically translated into Python exceptions.

- you can just as well keep your data on the card between kernel invocations–no need to copy data all the time.

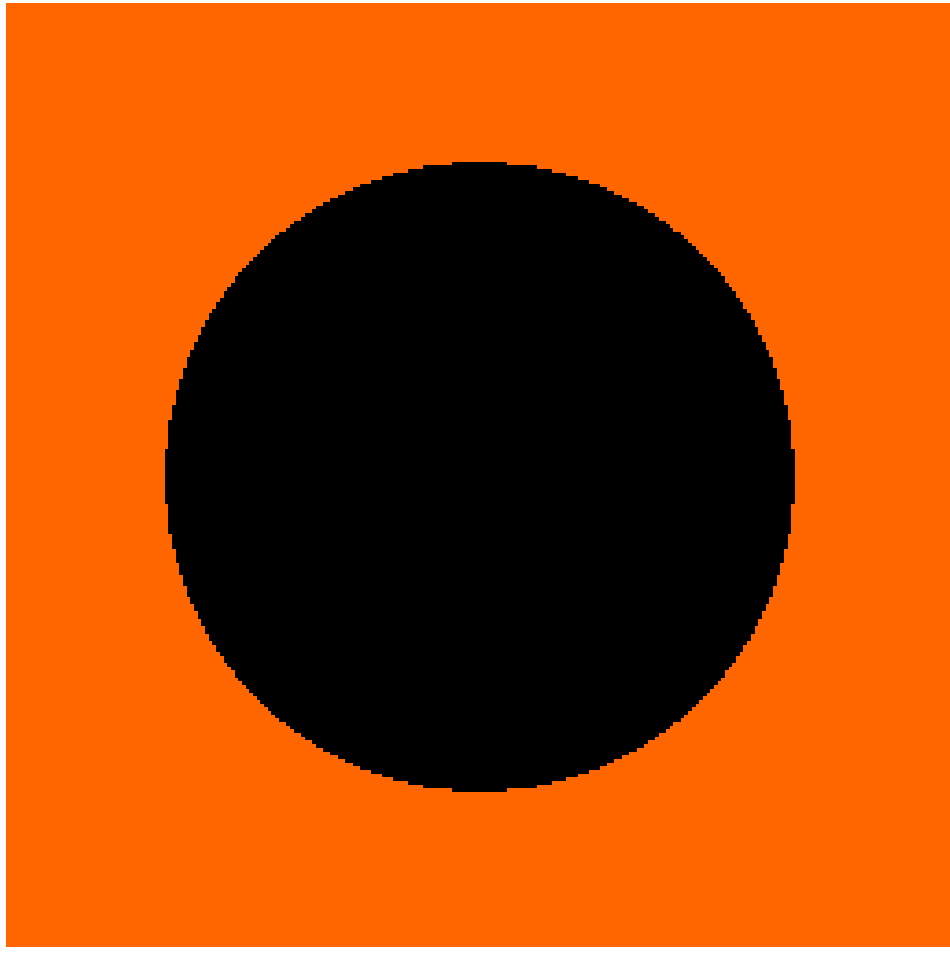
- PyCUDA will automatically infer what cleanup is necessary and do it for you.

Mandelbrot Set

- Is the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from z=0.

https://www.geogebra.org/m/BUVhcRSv#material/Npd3kBKn

$$Speedup = \frac{1}{.01 + .99/640} \approx 86.6$$



Profiling your code

- Premature optimization is the root of all evil.
- Our goal here is to find the bottlenecks and hotspots of a program.
- We should find candidate portions of the code to offload onto the GPU before we even think about rewriting the code to run on the GPU.
- A **profiler** will conveniently allow us to see where our program is taking the most time, and allow us to optimize accordingly.

cProfile module

- We can run the profiler from the command line with -m cProfile, and specify that we want to organize the results by the cumulative time spent on each function with -s cumtime, and then redirect the output into a text file with the > operator.
- python3 -m cProfile -s cumtime mandelbrotset.py

```
rogergo@pdw10:~/Dropbox/git/Python$ python3 -m cProfile -s cumtime mandelbrotset.py
It took 14.690768718719482 seconds to calculate the Mandelbrot graph.
It took 0.1142573356628418 seconds to dump the image.
         419348 function calls (410259 primitive calls) in 15.136 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    624/1    0.011    0.000   15.138   15.138 {built-in method builtins.exec}
        1    0.000    0.000   15.138   15.138 mandelbrotset.py:1(<module>)
        1   14.690   14.690   14.691   14.691 mandelbrotset.py:10(simple_mandelbrot)
    342/2    0.002    0.000    0.333    0.167 <frozen importlib._bootstrap>:966(_find_and_load)
    342/2    0.001    0.000    0.333    0.166 <frozen importlib._bootstrap>:936(_find_and_load_unlocked)
    333/2    0.002    0.000    0.333    0.166 <frozen importlib._bootstrap>:651(_load_unlocked)
```

```
Administrator: Anaconda Prompt (anaconda3)

(base) C:\Dropbox\Javeriana\Docencia\ParallelProgramming\code\cuda>python -m cProfile -s cumtime mandelbrot_set_cpu.py
```

```
python -m cProfile -s cumtime mandelbrot_set_cpu.py > MandelbrotCPU_Time.txt
```

```
1 It took 31.264559268951416 seconds to calculate the Mandelbrot graph.
2 It took 0.36343979835510254 seconds to dump the image.
3          620456 function calls (610474 primitive calls) in 32.475 seconds
4
5    Ordered by: cumulative time
6
7    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
8     628/1    0.008    0.000   32.477   32.477 {built-in method builtins.exec}
9         1    0.000    0.000   32.477   32.477 mandelbrot_set_cpu.py:1(<module>)
10        1   31.263   31.263   31.265   31.265 mandelbrot_set_cpu.py:11(simple_mandelbrot)
11       25    0.003    0.000    1.450    0.058 __init__.py:1(<module>)
12    338/6    0.004    0.000    0.859    0.143 <frozen importlib._bootstrap>:986(_find_and_load)
13    335/6    0.003    0.000    0.859    0.143 <frozen importlib._bootstrap>:956(_find_and_load_unlocked)
14    471/6    0.001    0.000    0.858    0.143 <frozen importlib._bootstrap>:211(_call_with_frames_removed)
15    320/6    0.003    0.000    0.858    0.143 <frozen importlib._bootstrap>:650(_load_unlocked)
16    269/6    0.002    0.000    0.857    0.143 <frozen importlib._bootstrap_external>:777(exec_module)
17   387/17    0.002    0.000    0.777    0.046 {built-in method builtins.__import__}
18   550/42    0.002    0.000    0.777    0.018 <frozen importlib._bootstrap>:1017(_handle_fromlist)
19        1    0.001    0.001    0.433    0.433 pyplot.py:4(<module>)
20        1    0.000    0.000    0.304    0.304 pyplot.py:963(savefig)
```

pycuda.autoinit

- pycuda.autoinit picks a GPU to run on, based on availability and the number.

- It also creates a GPU context for subsequent code to run in. Both the chosen device and the created context are available from pycuda.autoinit as importable symbols, if needed.

- The use of pycuda.autoinit is not compulsory — if needed, users can construct their own device-choice and context-creation methods using the facilities available in pycuda.driver.

Consultando la GPU

- GTX 1080
  - 2560 Cuda cores

```
cuda_cores_per_mp = { 5.0 : 128, 5.1 : 128, 5.2 : 128, 6.0 : 64, 6.1 : 128, 6.2 : 128}[compute_capability]
```

- We treat (global) device memory on the GPU as we do dynamically allocated heap memory in C (with the malloc and free functions) or C++ (as with the new and delete operators).

- PyCUDA covers all of the overhead of memory allocation, deallocation, and data transfers with the gpuarray class.

GeForce and TITAN Products

GPU	Compute Capability
NVIDIA TITAN RTX	7.5
Geforce RTX 2080 Ti	7.5
Geforce RTX 2080	7.5
Geforce RTX 2070	7.5
Geforce RTX 2060	7.5
NVIDIA TITAN V	7.0
NVIDIA TITAN Xp	6.1
NVIDIA TITAN X	6.1
GeForce GTX 1080 Ti	6.1
GeForce GTX 1080	6.1
GeForce GTX 1070	6.1
GeForce GTX 1060	6.1
GeForce GTX 1050	6.1
GeForce GTX TITAN X	5.2
GeForce GTX TITAN Z	3.5

GeForce Notebook Products

GPU	Compute Capability
Geforce RTX 2080	7.5
Geforce RTX 2070	7.5
Geforce RTX 2060	7.5
GeForce GTX 1080	6.1
GeForce GTX 1070	6.1
GeForce GTX 1060	6.1
GeForce GTX 980	5.2
GeForce GTX 980M	5.2
GeForce GTX 970M	5.2
GeForce GTX 965M	5.2
GeForce GTX 960M	5.0
GeForce GTX 950M	5.0
GeForce 940M	5.0
GeForce 930M	5.0
GeForce 920M	3.5

```python
import pycuda.driver as drv
drv.init()

print ('Se detecta dispositivo con capacidad CUDA \n{}' .format(drv.Device.count()))
i=0
gpu_device = drv.Device(i)
print ('Dispositivo{}: {}'.format(i,gpu_device.name()))

#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
compute_capability = float('%d.%d' % gpu_device.compute_capability())
print ('\t Compute Capability: {}'.format(compute_capability))
#imprimiendo la memoria de la GPU
print ('\t Memory: {} MB'.format(gpu_device.total_memory()))

#vamos a solicitar otros atributos de la GPU y lo guardamos en un diccionario de Pyton
Atributos_GPU = gpu_device.get_attributes().items()

atributos_dispositivo = {}

for k,v in Atributos_GPU:
    atributos_dispositivo[str(k)] = v

#Aqui se muestran los atributos leyendo la variable

num_mp = atributos_dispositivo['MULTIPROCESSOR_COUNT']

# Cores per multiprocessor is not reported by the GPU!
# We must use a lookup table based on compute capability.
# See the following:
# http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

cuda_cores_per_mp = { 5.0 : 128, 5.1 : 128, 5.2 : 128, 6.0 : 64, 6.1 : 128, 6.2 : 128}[compute_capability]

print ('\t ({}) Multiprocessors, ({}) CUDA Cores / Multiprocessor: {} CUDA Cores'.format(num_mp, cuda_cores_per_mp, num_mp*cuda_cores_per_mp))

atributos_dispositivo.pop('MULTIPROCESSOR_COUNT')

for k in atributos_dispositivo.keys():
    print ('\t {}: {}'.format(k, atributos_dispositivo[k]))
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

```
Se detecta dispositivo con capacidad CUDA
1
Dispositivo0: GeForce GTX 1080 with Max-Q Design
     Compute Capability: 6.1
     Memory: 8589934592 MB
     (20) Multiprocessors, (128) CUDA Cores / Multiprocessor: 2560 CUDA Cores
     ASYNC_ENGINE_COUNT: 1
     CAN_MAP_HOST_MEMORY: 1
     CLOCK_RATE: 1366000
     COMPUTE_CAPABILITY_MAJOR: 6
     COMPUTE_CAPABILITY_MINOR: 1
     COMPUTE_MODE: DEFAULT
     CONCURRENT_KERNELS: 1
     ECC_ENABLED: 0
     GLOBAL_L1_CACHE_SUPPORTED: 1
     GLOBAL_MEMORY_BUS_WIDTH: 256
     GPU_OVERLAP: 1
     INTEGRATED: 0
     KERNEL_EXEC_TIMEOUT: 1
     L2_CACHE_SIZE: 2097152
     LOCAL_L1_CACHE_SUPPORTED: 1
     MANAGED_MEMORY: 1
     MAXIMUM_SURFACE1D_LAYERED_LAYERS: 2048
     MAXIMUM_SURFACE1D_LAYERED_WIDTH: 32768
     MAXIMUM_SURFACE1D_WIDTH: 32768
     MAXIMUM_SURFACE2D_HEIGHT: 65536
     MAXIMUM_SURFACE2D_LAYERED_HEIGHT: 32768
     MAXIMUM_SURFACE2D_LAYERED_LAYERS: 2048
     MAXIMUM_SURFACE2D_LAYERED_WIDTH: 32768
     MAXIMUM_SURFACE2D_WIDTH: 131072
     MAXIMUM_SURFACE3D_DEPTH: 16384
     MAXIMUM_SURFACE3D_HEIGHT: 16384
     MAXIMUM_SURFACE3D_WIDTH: 16384
     MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS: 2046
     MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH: 32768
     MAXIMUM_SURFACECUBEMAP_WIDTH: 32768
     MAXIMUM_TEXTURE1D_LAYERED_LAYERS: 2048
```

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import  SourceModule
cuda.init()
```

[4]
```
import numpy as np
```

```
print ('Se detecta dispositivo con capacidad CUDA \n{}' .format(cuda.Device.count()))
i=0
gpu_device = cuda.Device(i)
print ('Dispositivo{}: {}'.format(i,gpu_device.name()))
```

```
Se detecta dispositivo con capacidad CUDA
1
Dispositivo0: Tesla T4
```

```python
#vamos a solicitar otros atributos de la GPU y lo guardamos en un diccionario de Pyton
Atributos_GPU = gpu_device.get_attributes().items()
atributos_dispositivo = {}
for k,v in Atributos_GPU:
    atributos_dispositivo[str(k)] = v
#Aqui se muestran los atributos leyendo la variable
num_mp = atributos_dispositivo['MULTIPROCESSOR_COUNT']
# Cores per multiprocessor is not reported by the GPU!
# We must use a lookup table based on compute capability.
# See the following:
# http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities
cuda_cores_per_mp = { 5.0 : 128, 5.1 : 128, 5.2 : 128, 6.0 : 64, 6.1 : 128, 6.2 : 128, 7.5:64}[compute_capability]
print ('\t ({}) Multiprocessors, ({}) CUDA Cores / Multiprocessor: {} CUDA Cores'.format(num_mp, cuda_cores_per_mp, num_mp*cuda_cores_per_mp))
atributos_dispositivo.pop('MULTIPROCESSOR_COUNT')
for k in atributos_dispositivo.keys():
    print ('\t {}: {}'.format(k, atributos_dispositivo[k]))
```

```
        (40) Multiprocessors, (64) CUDA Cores / Multiprocessor: 2560 CUDA Cores
        ASYNC_ENGINE_COUNT: 3
        CAN_MAP_HOST_MEMORY: 1
        CLOCK_RATE: 1590000
        COMPUTE_CAPABILITY_MAJOR: 7
        COMPUTE_CAPABILITY_MINOR: 5
        COMPUTE_MODE: DEFAULT
        CONCURRENT_KERNELS: 1
        ECC_ENABLED: 1
        GLOBAL_L1_CACHE_SUPPORTED: 1
        GLOBAL_MEMORY_BUS_WIDTH: 256
        GPU_OVERLAP: 1
        INTEGRATED: 0
        KERNEL_EXEC_TIMEOUT: 0
        L2_CACHE_SIZE: 4194304
        LOCAL_L1_CACHE_SUPPORTED: 1
        MANAGED_MEMORY: 1
        MAXIMUM_SURFACE1D_LAYERED_LAYERS: 2048
```

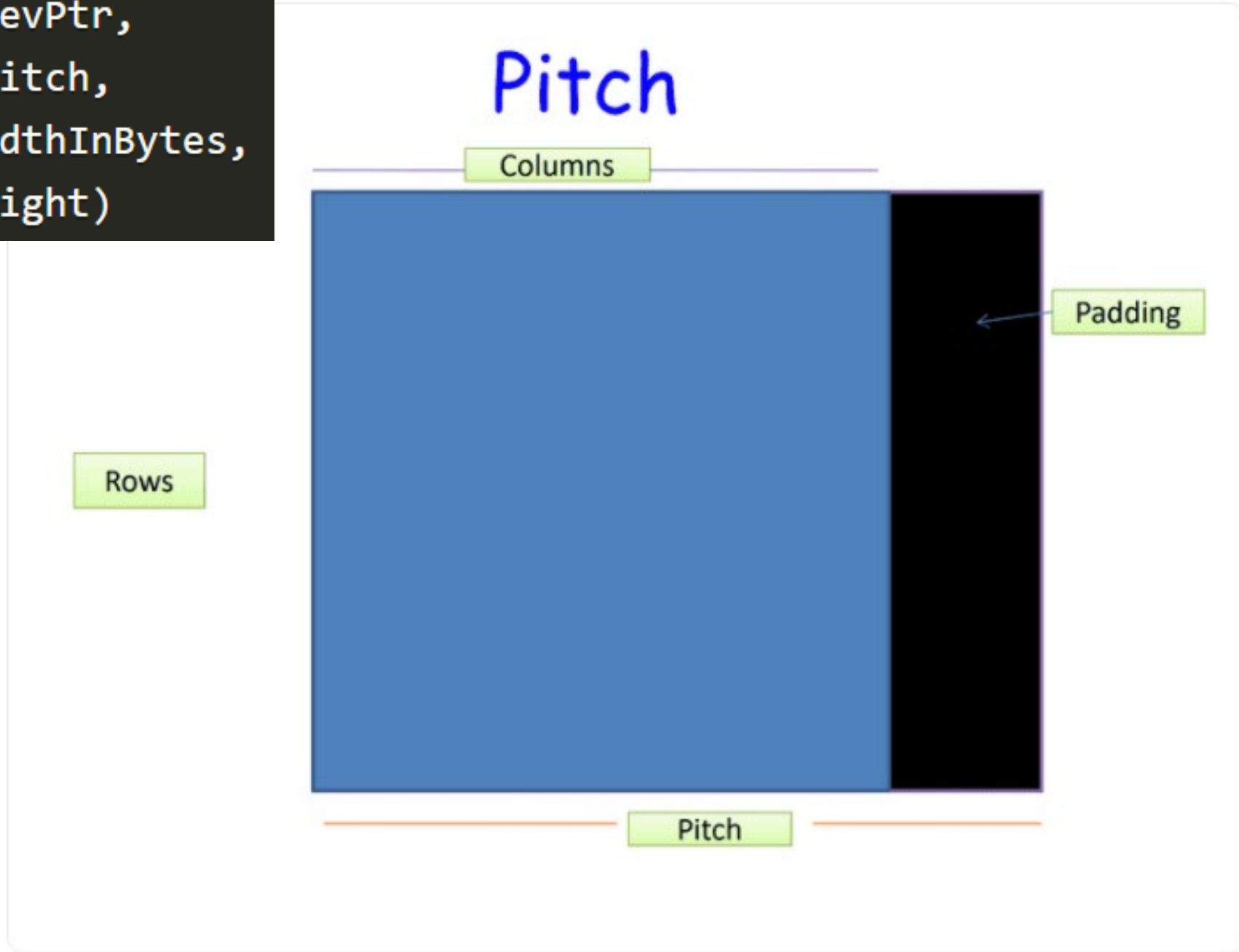
gpuarray Class

- We must contain our host data in some form of NumPy array (let's call it host_data), and then use the gpuarray.to_gpu(host_data) command to transfer this over to the GPU and create a new GPU array.

- We will initialize PyCUDA with import pycuda.autoinit

- These pointwise operations performed on the GPU are in parallel since the computation of one element is not dependent on the computation of any other element.

```
cudaMallocPitch( void** devPtr,
                 size_t* pitch,
                 size_t widthInBytes,
                 size_t height)
```



https://nichijou.co/cudaRandom-memAlign/

Basic tasks for CUDA parallelism

- Launching a kernel with specified grid dimensions (numbers of blocks and threads)
- Specifying that functions should be compiled to run on the device (GPU), the host (CPU), or both
- Accessing and utilizing block and thread index values for a computational thread
- Allocating memory and transferring data

- Let's now perform a simple computation within the GPU (pointwise multiplication by a constant on the GPU), and then retrieve the GPU data into a new with the **gpuarray.get** function.

Numpy type	C type	Description
np.int8	`int8_t`	Byte (-128 to 127)
np.int16	`int16_t`	Integer (-32768 to 32767)
np.int32	`int32_t`	Integer (-2147483648 to 2147483647)
np.int64	`int64_t`	Integer (-9223372036854775808 to 9223372036854775807)
np.uint8	`uint8_t`	Unsigned integer (0 to 255)
np.uint16	`uint16_t`	Unsigned integer (0 to 65535)
np.uint32	`uint32_t`	Unsigned integer (0 to 4294967295)
np.uint64	`uint64_t`	Unsigned integer (0 to 18446744073709551615)
np.intp	`intptr_t`	Integer used for indexing, typically the same as `ssize_t`
np.uintp	`uintptr_t`	Integer large enough to hold a pointer
np.float32	`float`	
np.float64 / *np.float_*	`double`	Note that this matches the precision of the builtin python *float*.

https://numpy.org/doc/stable/user/basics.types.html

Speed test

- To generate an array of 50 million random 32-bit floating point values, and then we will time how long it takes to scalar multiply the array by two on both devices.

- Prun→ Run a statement through the python code profiler.
  - -s sort profile by given key

```
In [1]: runfile('C:/Users/PUJC/anaconda3/test1.py', wdir='C:/
Users/PUJC/anaconda3')
Tiempo total de computo en CPU: 0.055897
El tiempo tomado por la GPU es: 3.309280

In [2]: runfile('C:/Users/PUJC/anaconda3/test1.py', wdir='C:/
Users/PUJC/anaconda3')
Tiempo total de computo en CPU: 0.057367
El tiempo tomado por la GPU es: 0.008958
```

Valid Arg	Meaning
“calls”	call count
“cumulative”	cumulative time
“file”	file name
“module”	file name
“pcalls”	primitive call count
“line”	line number
“name”	function name
“nfl”	name/file/line
“stdname”	standard name
“time”	internal time

```
In [3]: %prun -s cumulative exec(time_code)
Tiempo total de computo en CPU: 0.809000
El tiempo tomado por la GPU es: 14.699000
         275975 function calls (272590 primitive calls) in 20.030 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   14.699   14.699 gpuarray.py:485(__mul__)
        1    0.000    0.000   14.621   14.621 gpuarray.py:351(_axpbz)
        1    0.000    0.000   14.618   14.618 decorator.pyc:decorator-gen-133>:1(get_axpbz_kernel)
        1    0.000    0.000   14.618   14.618 tools.py:416(context_dependent_memoize)
        1    0.000    0.000   14.618   14.618 elementwise.py:417(get_axpbz_kernel)
        1    0.000    0.000   14.618   14.618 elementwise.py:155(get_elwise_kernel)
        1    0.000    0.000   14.618   14.618 elementwise.py:126(get_elwise_kernel_and_types)
        1    0.001    0.001   14.617   14.617 elementwise.py:41(get_elwise_module)
        1    0.000    0.000   14.614   14.614 compiler.py:285(__init__)
        1    0.002    0.002   14.614   14.614 compiler.py:190(compile)
        1    0.001    0.001   13.990   13.990 compiler.py:69(compile_plain)
        2    0.000    0.000   13.979    6.990 prefork.py:226(call_capture_output)
        2    0.000    0.000   13.979    6.990 prefork.py:44(call_capture_output)
        1    0.000    0.000   13.892   13.892 compiler.py:36(preprocess_source)
        2    0.000    0.000   13.887    6.944 subprocess.py:452(communicate)
        2    0.000    0.000   13.887    6.944 subprocess.py:702(_communicate)
        6    0.000    0.000   13.886    2.314 threading.py:309(wait)
       32   13.886    0.434   13.886    0.434 {method 'acquire' of 'thread.lock' objects}
        4    0.000    0.000   13.885    3.471 threading.py:902(join)
        1    4.809    4.809    4.809    4.809 {method 'random_sample' of 'mtrand.RandomState' objects}
        1    0.002    0.002    0.561    0.561 compiler.py:178(_find_pycuda_include_path)
        1    0.007    0.007    0.557    0.557 __init__.py:16(<module>)
        2    0.000    0.000    0.458    0.229 __init__.py:3233(_call_aside)
        1    0.000    0.000    0.458    0.458 __init__.py:3250(_initialize_master_working_set)
```

```
In [3]: %prun -s cumulative exec(time_profile)
Tiempo total de computo en CPU: 0.069925
El tiempo tomado por la GPU es: 1.788083
         390085 function calls (387113 primitive calls) in 2.571 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     76/1    0.000    0.000    2.777    2.777 {built-in method builtins.exec}
        1    0.000    0.000    1.788    1.788 gpuarray.py:485(__mul__)
        1    0.000    0.000    1.774    1.774 gpuarray.py:351(_axpbz)
        1    0.000    0.000    1.773    1.773 <decorator-gen-127>:1(get_axpbz_kernel)
        1    0.000    0.000    1.773    1.773 tools.py:416(context_dependent_memoize)
        1    0.000    0.000    1.773    1.773 elementwise.py:417(get_axpbz_kernel)
        1    0.000    0.000    1.773    1.773 elementwise.py:155(get_elwise_kernel)
        1    0.000    0.000    1.773    1.773 elementwise.py:126(get_elwise_kernel_and_types)
        1    0.000    0.000    1.772    1.772 elementwise.py:41(get_elwise_module)
        1    0.000    0.000    1.770    1.770 compiler.py:285(__init__)
        1    0.000    0.000    1.770    1.770 compiler.py:190(compile)
        1    0.000    0.000    1.340    1.340 compiler.py:69(compile_plain)
        2    0.000    0.000    1.334    0.667 prefork.py:226(call_capture_output)
        2    0.000    0.000    1.334    0.667 prefork.py:44(call_capture_output)
        2    0.000    0.000    1.328    0.664 subprocess.py:920(communicate)
        2    0.000    0.000    1.328    0.664 subprocess.py:1271(_communicate)
       28    1.327    0.047    1.327    0.047 {method 'acquire' of '_thread.lock' objects}
        4    0.000    0.000    1.326    0.332 threading.py:1012(join)
       10    0.000    0.000    1.326    0.133 threading.py:1050(_wait_for_tstate_lock)
        1    0.000    0.000    1.309    1.309 compiler.py:36(preprocess_source)
        1    0.614    0.614    0.614    0.614 {method 'random' of 'numpy.random.mtrand.RandomState' objects}
     47/8    0.000    0.000    0.532    0.067 <frozen importlib._bootstrap>:978(_find_and_load)
     47/8    0.000    0.000    0.532    0.067 <frozen importlib._bootstrap>:948(_find_and_load_unlocked)
     48/9    0.000    0.000    0.529    0.059 <frozen importlib._bootstrap>:663(_load_unlocked)
    60/10    0.000    0.000    0.528    0.053 <frozen importlib._bootstrap>:211(_call_with_frames_removed)
     38/7    0.000    0.000    0.526    0.075 <frozen importlib._bootstrap_external>:722(exec_module)
        1    0.000    0.000    0.417    0.417 compiler.py:178(_find_pycuda_include_path)
        1    0.000    0.000    0.415    0.415 __init__.py:15(<module>)
```

```
In [4]: %prun -s cumulative exec(time_profile)
Tiempo total de computo en CPU: 0.086902
El tiempo tomado por la GPU es: 0.019006
         149 function calls (148 primitive calls) in 0.657 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      2/1    0.000    0.000    0.904    0.904 {built-in method builtins.exec}
        1    0.000    0.000    0.904    0.904 <string>:1(<module>)
        1    0.571    0.571    0.571    0.571 {method 'random' of 'numpy.random.mtrand.RandomState' objects}
        1    0.000    0.000    0.072    0.072 gpuarray.py:1047(to_gpu)
        1    0.000    0.000    0.058    0.058 gpuarray.py:230(set)
        1    0.057    0.057    0.057    0.057 gpuarray.py:1231(_memcpy_discontig)
        2    0.028    0.014    0.028    0.014 gpuarray.py:162(__init__)
        1    0.000    0.000    0.013    0.013 gpuarray.py:485(__mul__)
        1    0.000    0.000    0.013    0.013 gpuarray.py:412(_new_like_me)
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        4    0.000    0.000    0.000    0.000 iostream.py:386(write)
        5    0.000    0.000    0.000    0.000 iostream.py:197(schedule)
        5    0.000    0.000    0.000    0.000 socket.py:357(send)
        1    0.000    0.000    0.000    0.000 gpuarray.py:351(_axpbz)
        1    0.000    0.000    0.000    0.000 driver.py:547(function_prepared_async_call)
        5    0.000    0.000    0.000    0.000 threading.py:1092(is_alive)
        1    0.000    0.000    0.000    0.000 gpuarray.py:19(_get_common_dtype)
        1    0.000    0.000    0.000    0.000 stride_tricks.py:37(as_strided)
        2    0.000    0.000    0.000    0.000 gpuarray.py:109(splay)
        2    0.000    0.000    0.000    0.000 __init__.py:671(wrapper)
```

ElementWise Kernel

- In CUDA context, a kernel is a function that is launched directly onto the GPU by CUDA.

- We will have to allocate some empty memory on the GPU that is pointed to by the out variable.

```
device_data = gpuarray.to_gpu(host_data)
# allocate memory for output
device_data_2x = gpuarray.empty_like(device_data)
```

```
total time to compute on CPU: 0.771056
total time to compute on GPU: 1.321093
```

ElementWise multiplication

- Speedcomparison()

```
In [19]: runfile('C:/Dropbox/Javeriana/Docencia/ParallelProgramming/
code/cuda/Create_Kernel_PyCUDA.py', wdir='C:/Dropbox/Javeriana/
Docencia/ParallelProgramming/code/cuda')
total time to compute on CPU: 0.771056
total time to compute on GPU: 1.321093

In [20]: speedcomparison()
total time to compute on CPU: 0.886317
total time to compute on GPU: 0.000000
```

Map and Reduce operations

- We use reduce operations with *associative binary operations*; this means that, no matter the order we perform our operation between sequential elements of the list, will always invariably give the same result.

- A basic function in PyCUDA that reproduces the functionality of reduce—InclusiveScanKernel.

```
reduce(lambda x, y : x + y, [1,2,3,4])
```

PyCUDA reduce functionality

```
In [1]: run simple_scankernel0.py
[ 1  3  6 10]
[ 1  3  6 10]
```

```
(base) C:\WINDOWS\system32>pip install mako
DEPRECATION: Python 2.7 will reach the end of its life on January 1st, 2020. Please upgrade your Python as Python 2.
n't be maintained after that date. A future version of pip will drop support for Python 2.7. More details about Pyth
 support in pip, can be found at https://pip.pypa.io/en/latest/development/release-process/#python-2-support
Collecting mako
  Downloading https://files.pythonhosted.org/packages/50/78/f6ade1e18aebda570eed33b7c534378d9659351cadce2fcbc7b31be5
/Mako-1.1.2-py2.py3-none-any.whl (75kB)
     |UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU| 81kB 1.0MB/s
Requirement already satisfied: MarkupSafe>=0.9.2 in c:\users\deeplearning_puj\anaconda2\lib\site-packages (from mako
.1.1)
Installing collected packages: mako
Successfully installed mako-1.1.2
```

Find the maximum value in a float32 array

- The main change we made is to replace the a + b string with a > b ? a : b

FORMAT:

conditional Expression ? expression1 :
expression2;

** if the conditional Expression is **true**, expression1 executes, otherwise if the conditional Expression is **false**, expression 2 executes.

- We finally display the last value of the resulting element in the output array, which will be exactly the last element.

https://mathbits.com/MathBits/CompSci/looping/operator2.htm

Kernel

```
1  // Perform an element-wise addition of A and B and store in C.
2  // There are N elements per array.
3  void vecadd(int *C, int* A, int *B, int N)
4  {
5      for(int i = 0; i < N; ++i)
6      {
7          C[i] = A[i] + B[i];
8      }
9  }
```

```
1  // Perform an element-wise addition of A and B and store in C.
2  // There are N elements per array and NP CPU cores.
3  void vecadd(int *C, int* A, int *B, int N, int NP, int tid)
4  {
5      int ept = N/NP; // elements per thread
6      for(int i = tid*ept; i < (tid+1)*ept; ++i)
7      {
8          C[i] = A[i] + B[i];
9      }
10
11 }
```

```
1  // Perform an element-wise addition of A and B and store in C
2  // N work-items will be created to execute this kernel.
3  __kernel
4  void vecadd(__global int *C, __global int* A, __global int *B)
5  {
6      int tid = get_global_id(0); // OpenCL intrinsic function
7      C[tid] = A[tid] + B[tid];
8  }
```

The MapReduce operation with PyCUDA

PyCUDA provides a functionality to perform reduction operations on the GPU. This is possible with the `pycuda.reduction.ReductionKernel` method:

```
ReductionKernel(dtype_out, arguments, map_expr ,reduce_expr,
                name,optional_parameters)
```

Here, we note that:

- `dtype_out`: This is the output's data type. It must be specified by the `numpy.dtype` data type.
- `arguments`: This is a C argument list of all the parameters involved in the reduction's operation.
- `map_expr`: This is a string that represents the mapping operation. Each vector in this expression must be referenced with the variable `i`.
- `reduce_expr`: This is a string that represents the reduction operation. The operands in this expression are indicated by lowercase letters, such as `a, b, c, ..., z`.
- `name`: This is the name associated with `ReductionKernel`, with which the kernel is compiled.
- `optional_parameters`: These are not important in this recipe as they are the compiler's directives.

The method executes a kernel on vector arguments (at least one), performs `map_expr` on each entry of the vector argument, and then performs `reduce_expr` on its outcome.

Dot product of two vectors

- The dot product or scalar product is an algebraic operation that takes two equal length sequences of numbers and returns a single number that is the sum of the products.
- Is a typical MapReduce operation, where the Map operation is an index-by-index producto and the reduction operation is the sum of all products.

`pycuda.gpuarray.arange`(*start*, *stop*, *step*, *dtype=None*, *stream=None*)

Create a `GPUArray` filled with numbers spaced *step* apart, starting from *start* and ending at *stop*.

For floating point arguments, the length of the result is *ceil((stop - start)/step)*. This rule may result in the last element of the result being greater than *stop*.

dtype, if not specified, is taken as the largest common type of *start*, *stop* and *step*.

ReductionKernel

- Neutral serves as an initial value.

```
ReductionKernel()
```

```
1220(<genexpr>)
        11    0.000    0.
         1    0.000    0.
```

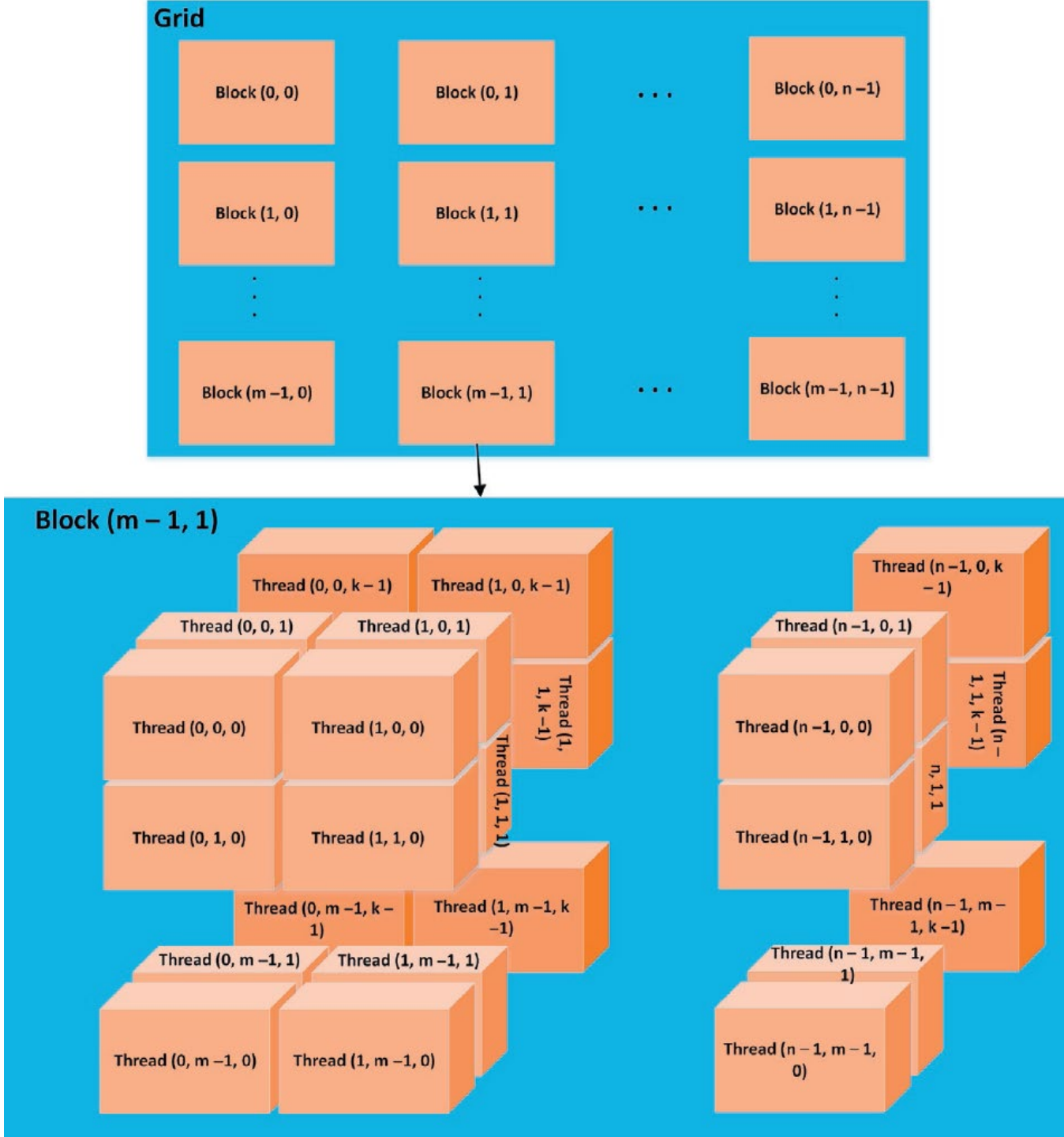
Arguments

```
ReductionKernel(dtype_out, neutral, reduce_expr,
                map_expr=None, arguments=None,
                name="reduce_kernel", keep=False,
                options=None, preamble="")
```

CUDA structure

- A thread block is a group of threads that cooperate via shared memory and synchronize their execution to coordinate their memory accesses.

- A grid consists of a group of thread blocks and a kernel is executed on a group of grids.



SourceModule

- To compile raw inline CUDA C code into usable kernels that we launch from Python.
- We'll have to "pull out" a reference to the kernel we want to use with PyCUDA's get_function, before we can actually launch it.
- Multiplicate a vector by a escalar, example.

Threads, blocks and grids

- CUDA C/C++ keyword __global__ indicates a function that:
  - Runs on the device.
  - Is called from host code.

- A thread is a sequence of instructions that is executed on a single core of the GPU

- There is an x at the end because there is also threadIdx.y and threadIdx.z.

- Blocks are further executed in abstract batches known as grids, which are best thought of as blocks of blocks.

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

- With M threads per block, a unique index for each thread is given by:

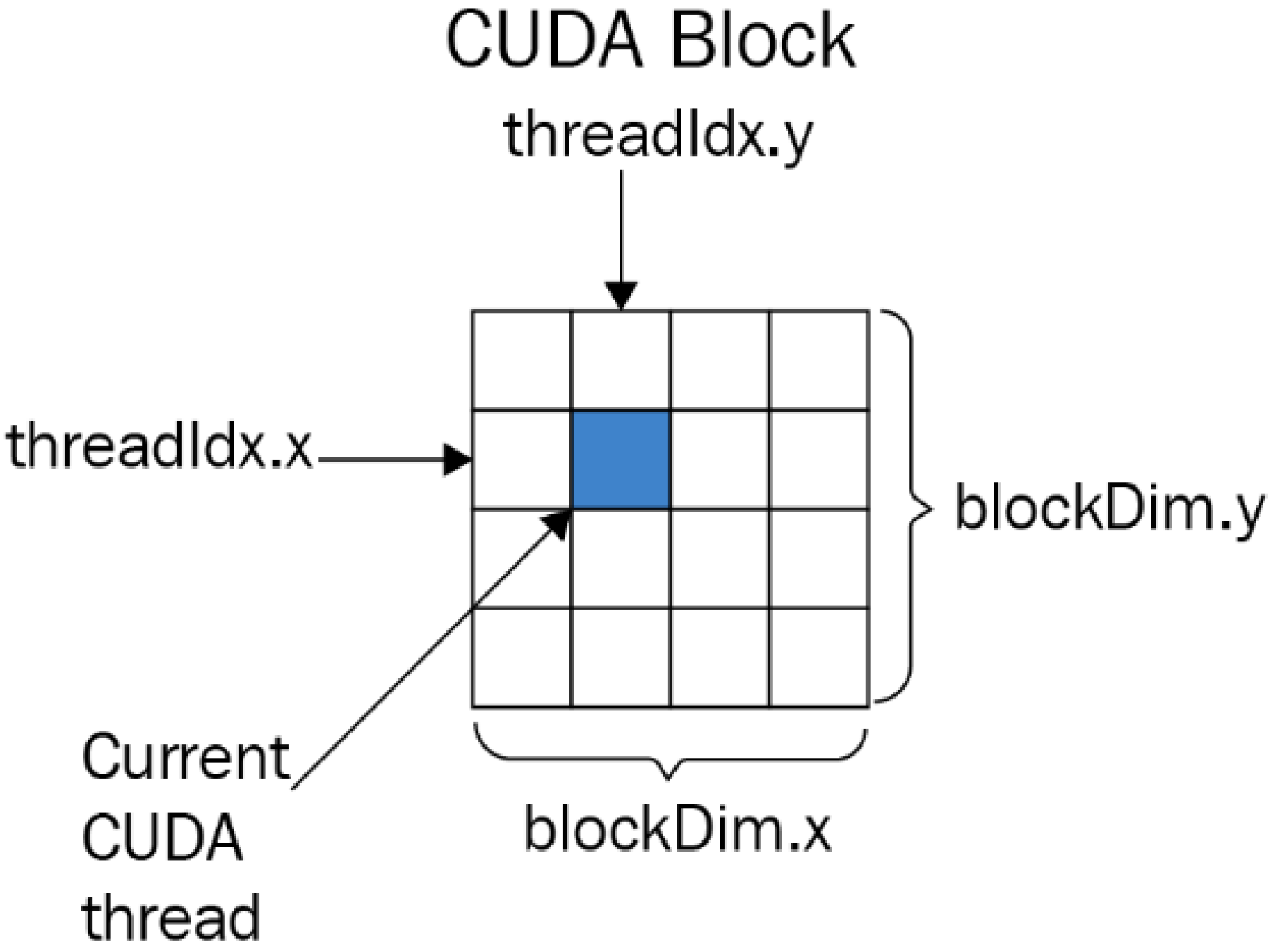
```
int index = threadIdx.x + blockIdx.x * M;
```

https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf

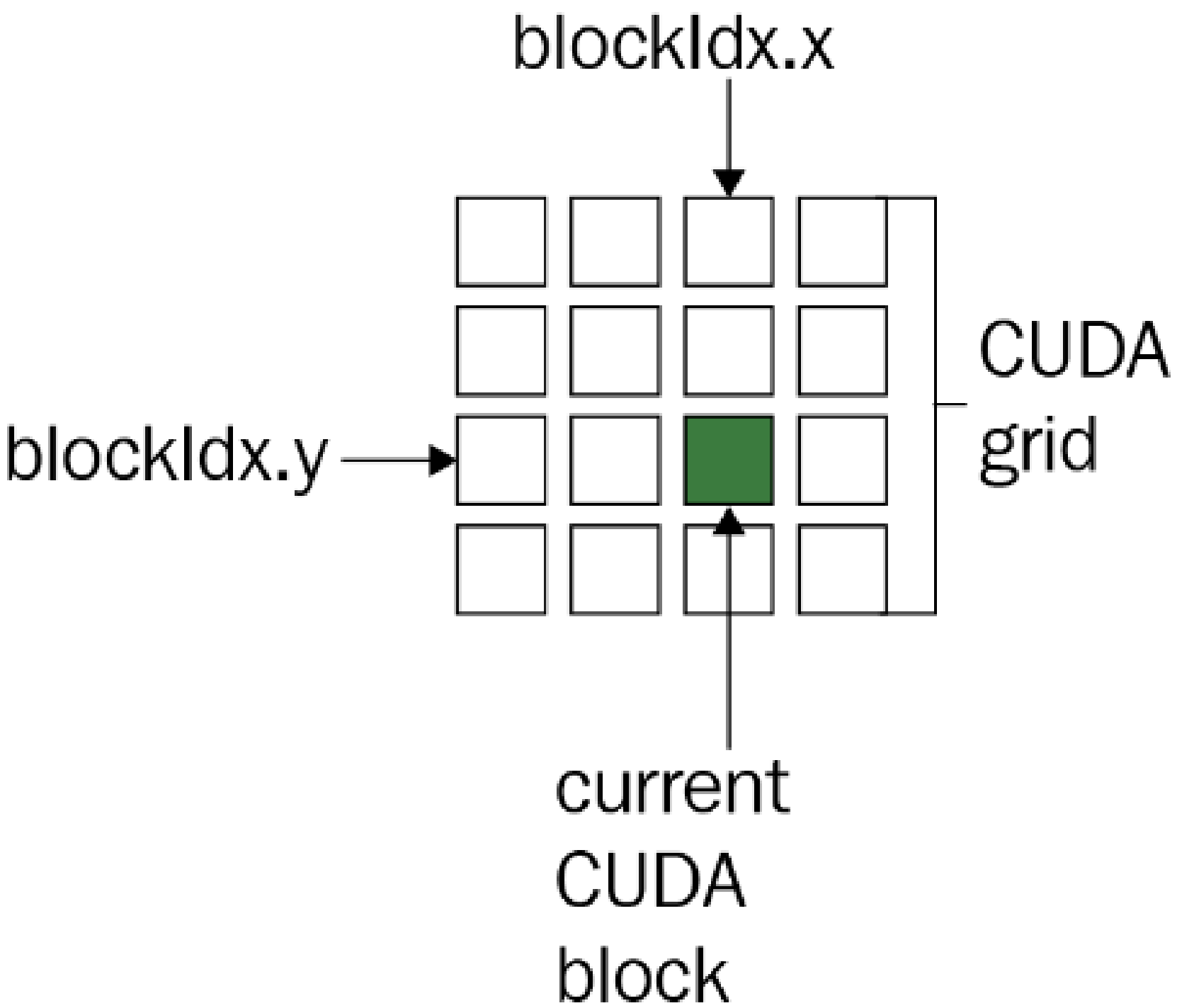
Indexing Arrays: Example

- Which thread will operate on the red element?


```
int index = threadIdx.x + blockIdx.x * M;
          =       5     +      2      * 8;
          = 21;
```


CUDA Block
threadIdx.y
threadIdx.x
blockDim.y
Current
CUDA
thread
blockDim.x

- we can determine which block we are in within a two-dimensional grid with blockIdx.x and blockIdx.y



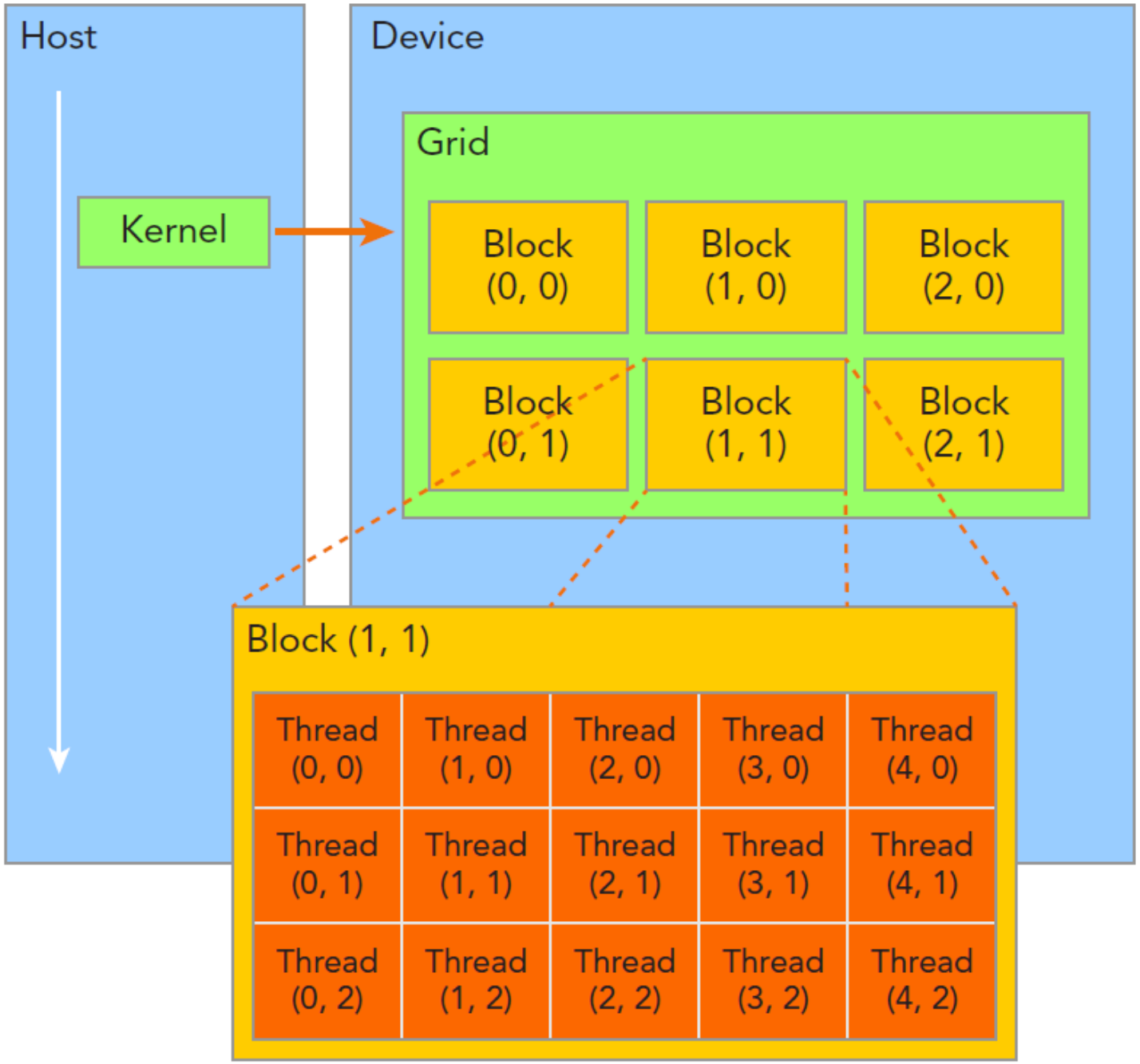
- Threads rely on the following two unique coordinates to distinguish themselves from each other:

➤ `blockIdx` (block index within a grid)

➤ `threadIdx` (thread index within a block)

- The dimensions of a grid and a block are specified by the following two built-in variables:
- ➤ `blockDim` (block dimension, measured in threads)
- ➤ `gridDim` (grid dimension, measured in blocks)

- Both grids and blocks use the `dim3` type with three unsigned integer fields. The unused fields will be initialized to 1 and ignored.



PyCUDA: Conway's game of life

- The Game of Life (often called LIFE for short) is a cellular automata simulation that was invented by the British mathematician John Conway back in 1970.

- Any live cell with fewer than two live neighbors dies
- Any live cell with two or three neighbors lives
- Any live cell with more than three neighbors dies
- Any dead cell with exactly three neighbors comes to life

- We will use matplotlib.animation module

CUDA events

- The time library from Python is used to measure CPU timings.
- A CUDA event is a GPU timestamp recorded at a specified point in a PyCUDA program.
- We can record two events, one at the start of our code and one at the end.
- The time is recorded before and after calling the kernel function using the record function, and the time difference is calculated to measure the timing of the kernel function.

PyCUDA: Matrix Multiplication