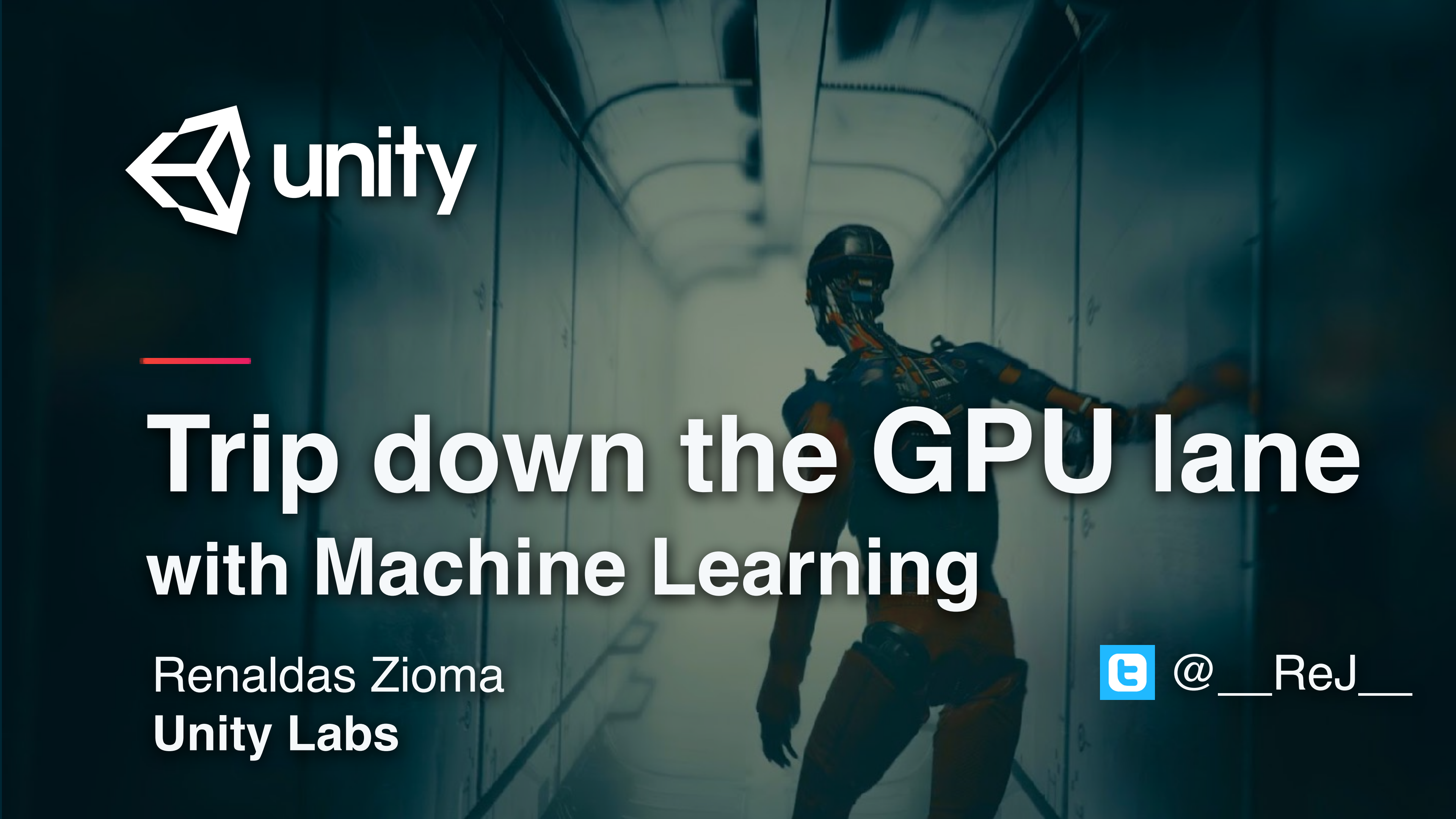


Trip down the GPU lane with Machine Learning

Renaldas Zioma
Unity Labs

@__ReJ__

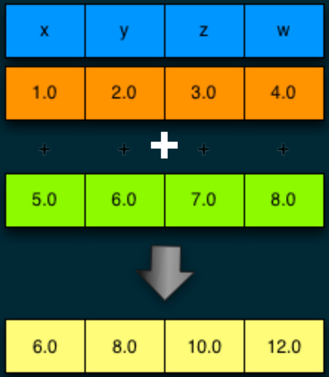
Despite architectural differences between CPU & GPU

what dominates the speed of training Convolutional Neural Net is the raw TFLOPs of a given chip!

CPU -vs- GPU

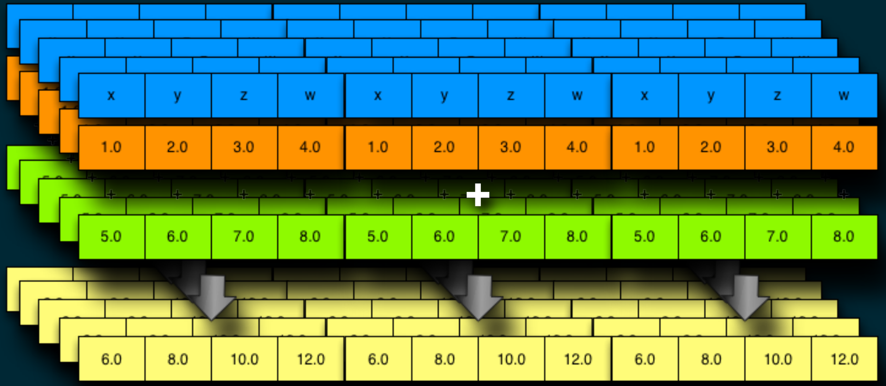
SIMD

Single **I**nstruction **M**ultiple **D**ata



SIMT

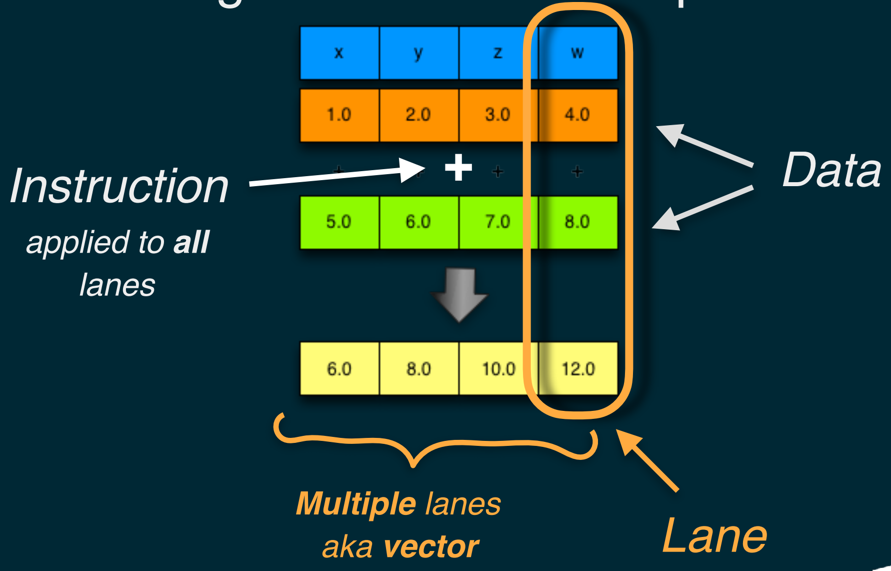
Single **I**nstruction **M**ultiple **T**hreads



unity

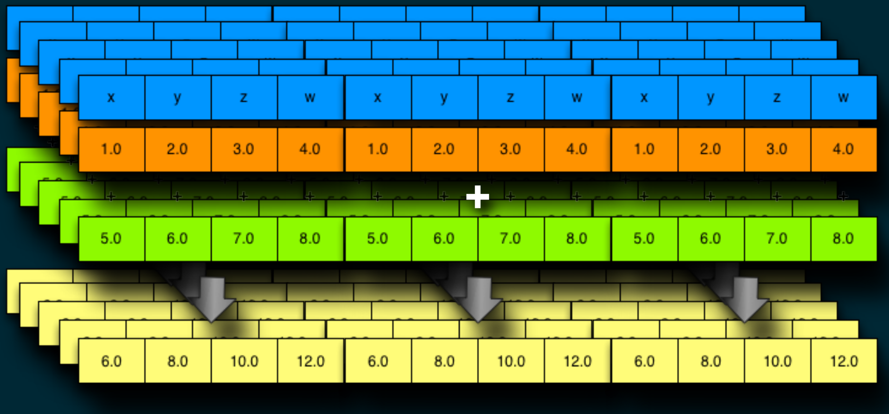
SIMD

Single Instruction Multiple Data



SIMT

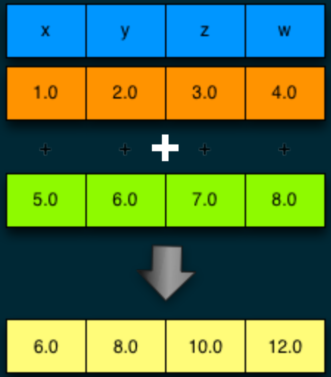
Single Instruction Multiple Threads



SIMD

4 lanes (SSE)

8 lanes (AVX)

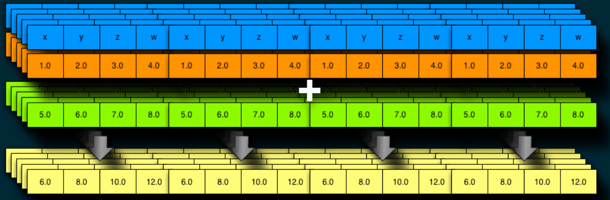


SIMT

32 lanes

16 lanes (Mobile GPU)

64 lanes (AMD)



SIMT is almost the same as SIM**D**, but much wider

unity

Out Of Order

Executes “any” instruction that has **all source operands ready**

Warp or Wavefront

Warp is 32 threads working in-sync
Threads must share the same program!

1 core* can keep 64** warps in flight
Executes warp that has **all source operands ready**

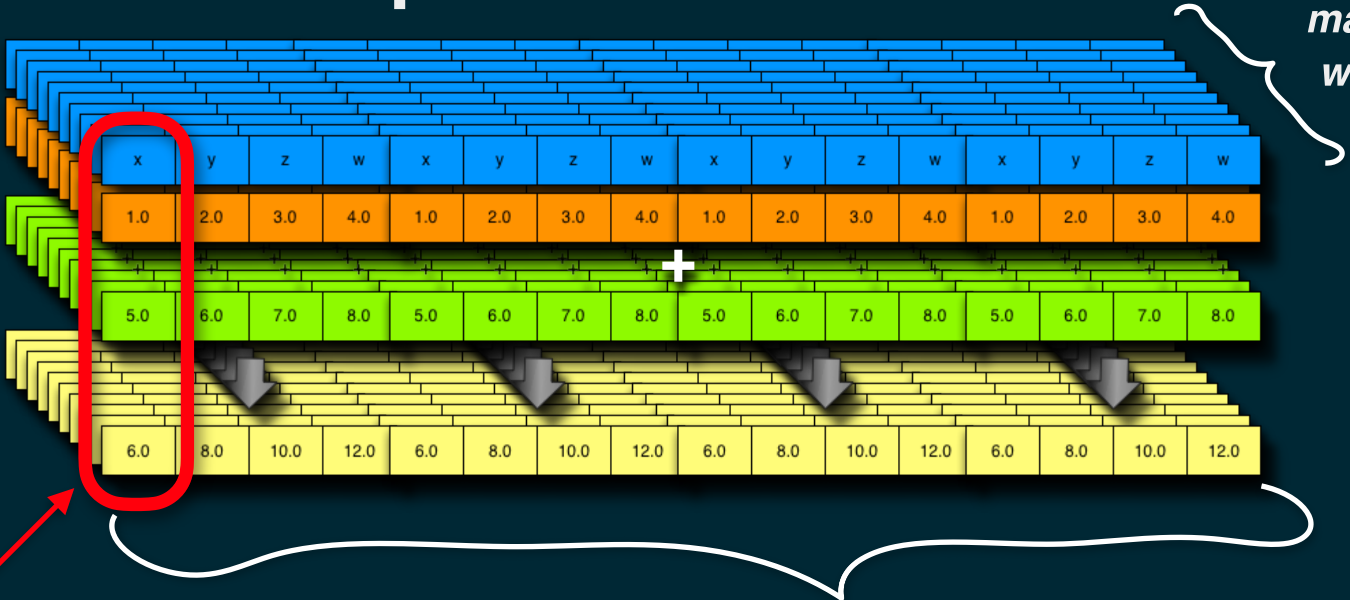
Very lightweight switch between warps

Aim is to hide memory latency

*) By Core here I mean Streaming Multiprocessor (SM) in Nvidia or Compute Unit (CU) in AMD GPU, but not *“CUDA core”.* While SM and CU are comparable to CPU core, “*CUDA core*” is more of a marketing term than a full-fledged core.
**) Depends actually - 40 on AMD, 128 on GP100…

unity

SIMT in one picture



Each core maintains many warps in flight

Thread = Lane

1 thread is mapped to 1 hardware lane

Warp** of **threads

unity

Warp

Warp is 32* threads working in a **lockstep**

All threads execute the **same** program

Each **thread** is mapped to a single **SIMT lane**

Processor will “juggle” warps to **hide** memory latency

*) The number of threads per warp is actually hardware dependent and ranges from 16 on Mobile to 64 on AMD

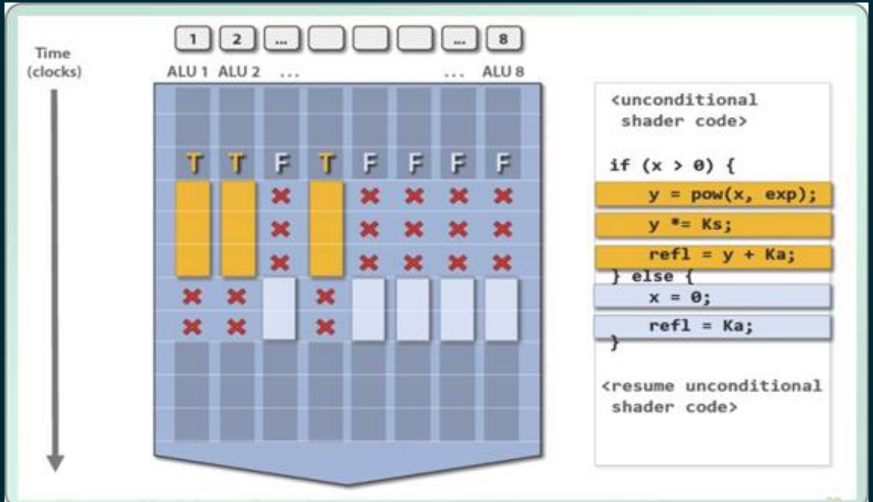
unity

Take Away

Need lots of parallel work - in **1000s* of threads**

IF-ELSE block executes **both** IF and ELSE**

If SIMT lane is not doing work, it is wasted!

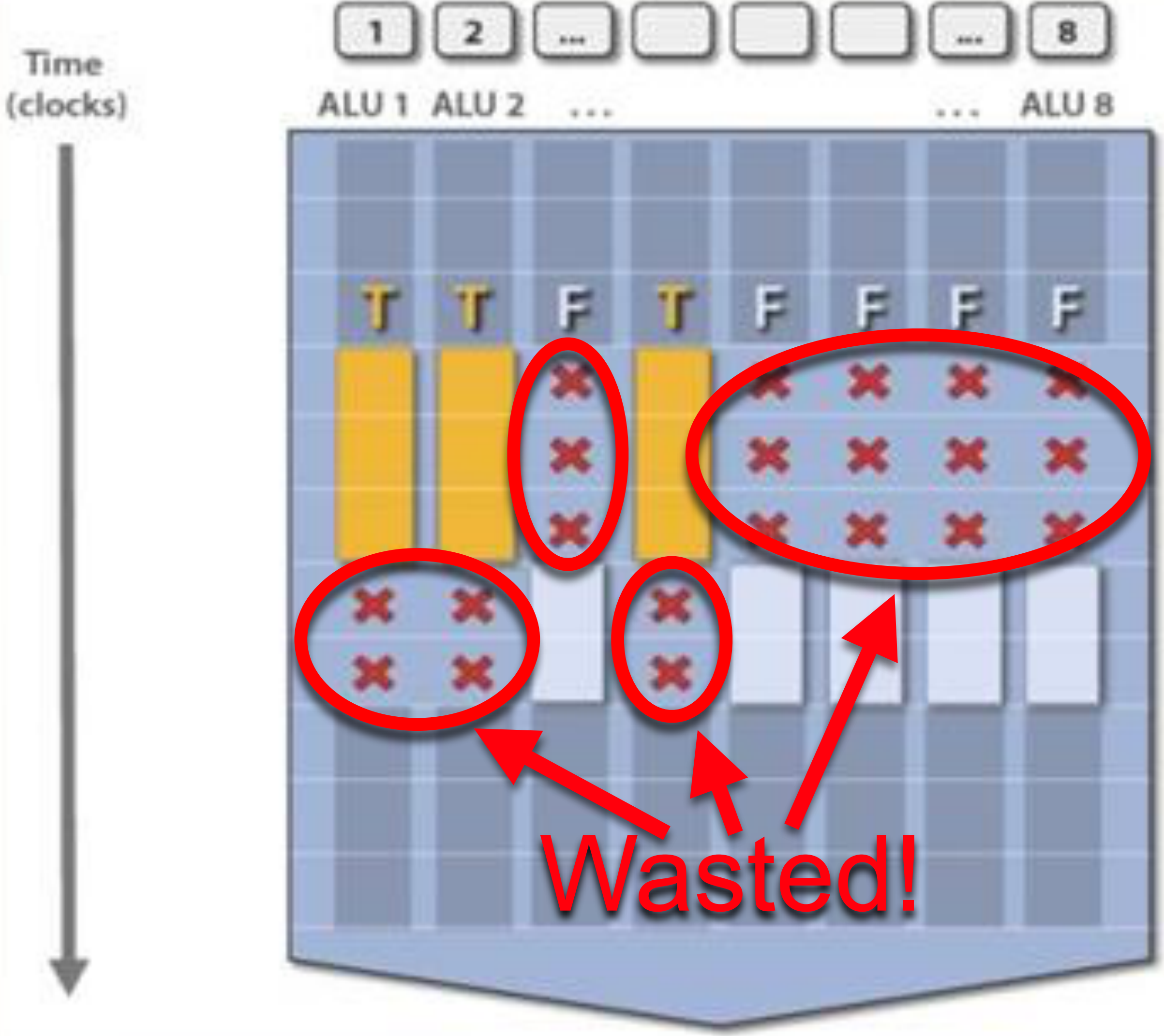


```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

*)Number of cores (SM/CU) × number of threads in warp × number of warps in flight

For example optimised matrix multiplication kernel would need at least **10240** threads to saturate GTX 1080 = 20 SMs × 32 warps × 16

**) Unless of course all threads in the warp agree on the result of conditional statement. In such case only one path needs to be executed, either IF or ELSE



```
<unconditional
 shader code>

if (x > 0) {
        y = pow(x, exp);
        y *= Ks;
        refl = y + Ka;
} else {
        x = 0;
        refl = Ka;
}

<resume unconditional
 shader code>
```

Cache

L3 + L2 + L1

Cache + “Scratchpad”

L2 + L1

Constant + Texture cache

LDS (Local Data Store)
explicitly programmable “cache”

Lots of registers!

LDS - Local Data Share

Piece of small, but ultra fast memory - up to **8TB/s**!

Explicitly programmable “cache”

Fast data exchange between threads

NVIDIA calls it “Shared Memory”*

* But “Shared Memory” is such a generic term, lets be more specific and call it Local Data Share for the rest of the slides.

One dimensional

Memory and execution is sequential

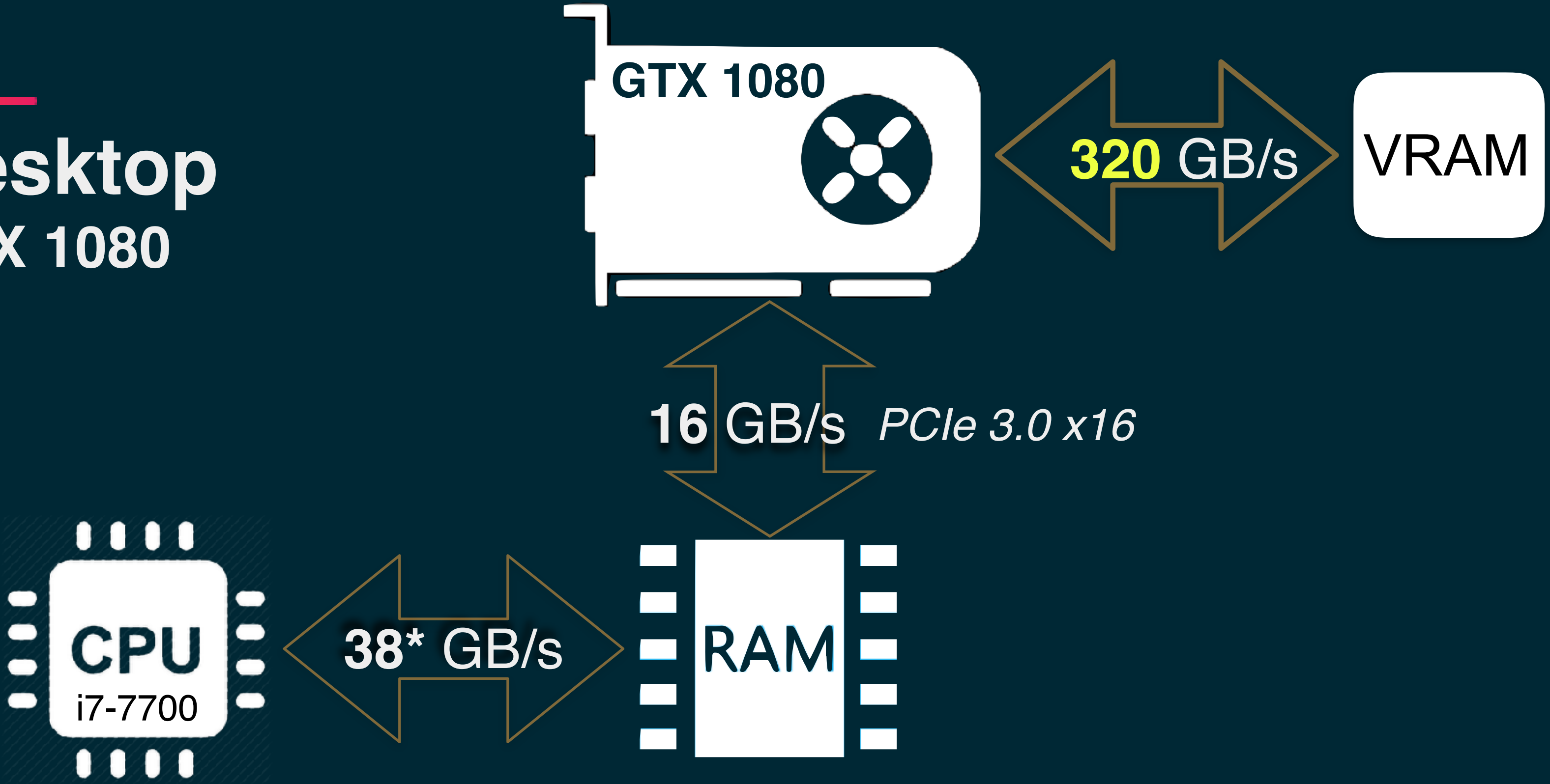
Multidimensional view

Some instructions see memory as 2D / 3D blocks

Work scheduled in groups

Memory Bandwidth

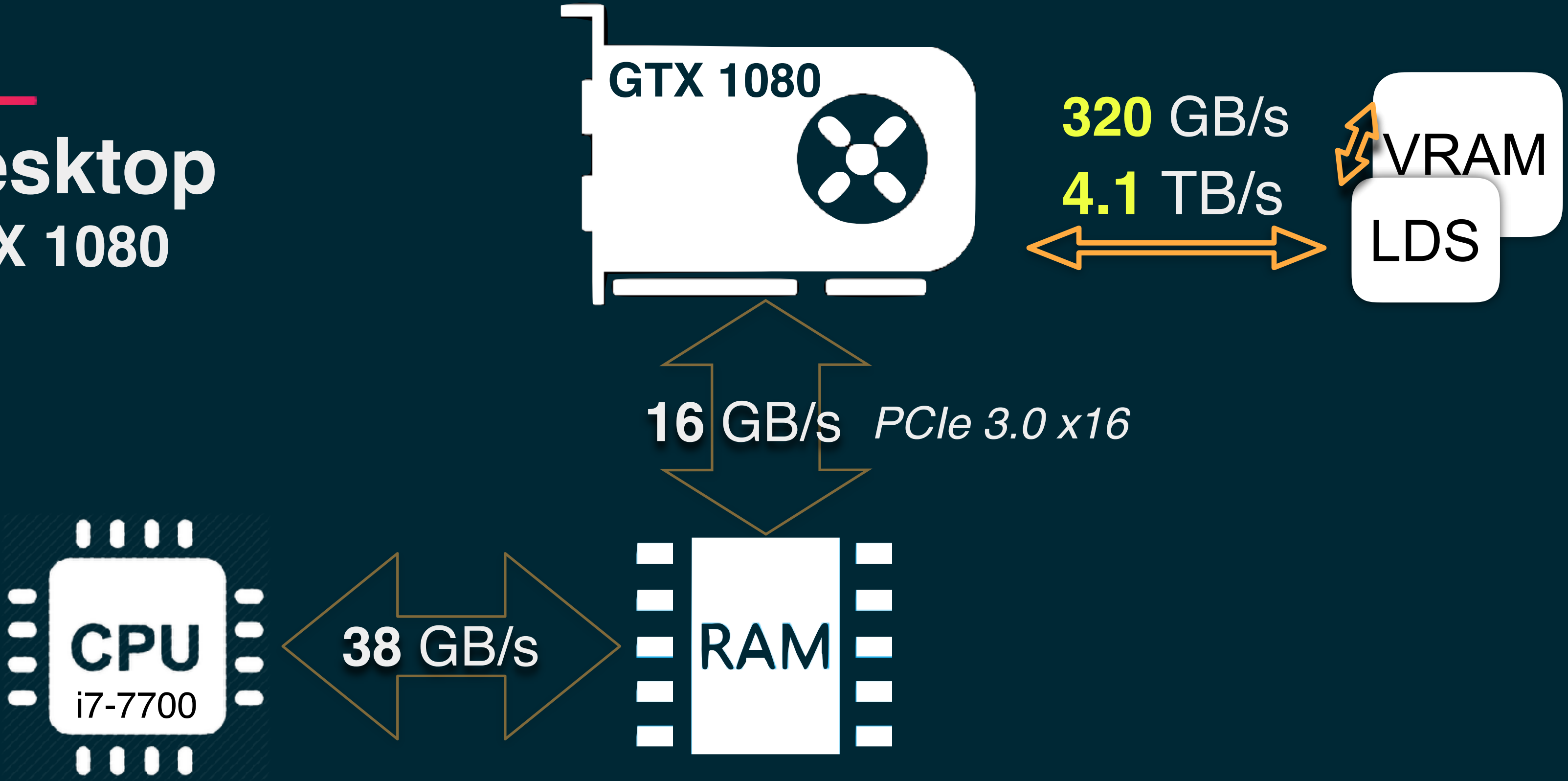
Desktop
GTX 1080



*) Numbers provided here and onward are for **peak** bandwidth. Practical achievable bandwidth is usually 75% of peak and can be even lower for CPU.

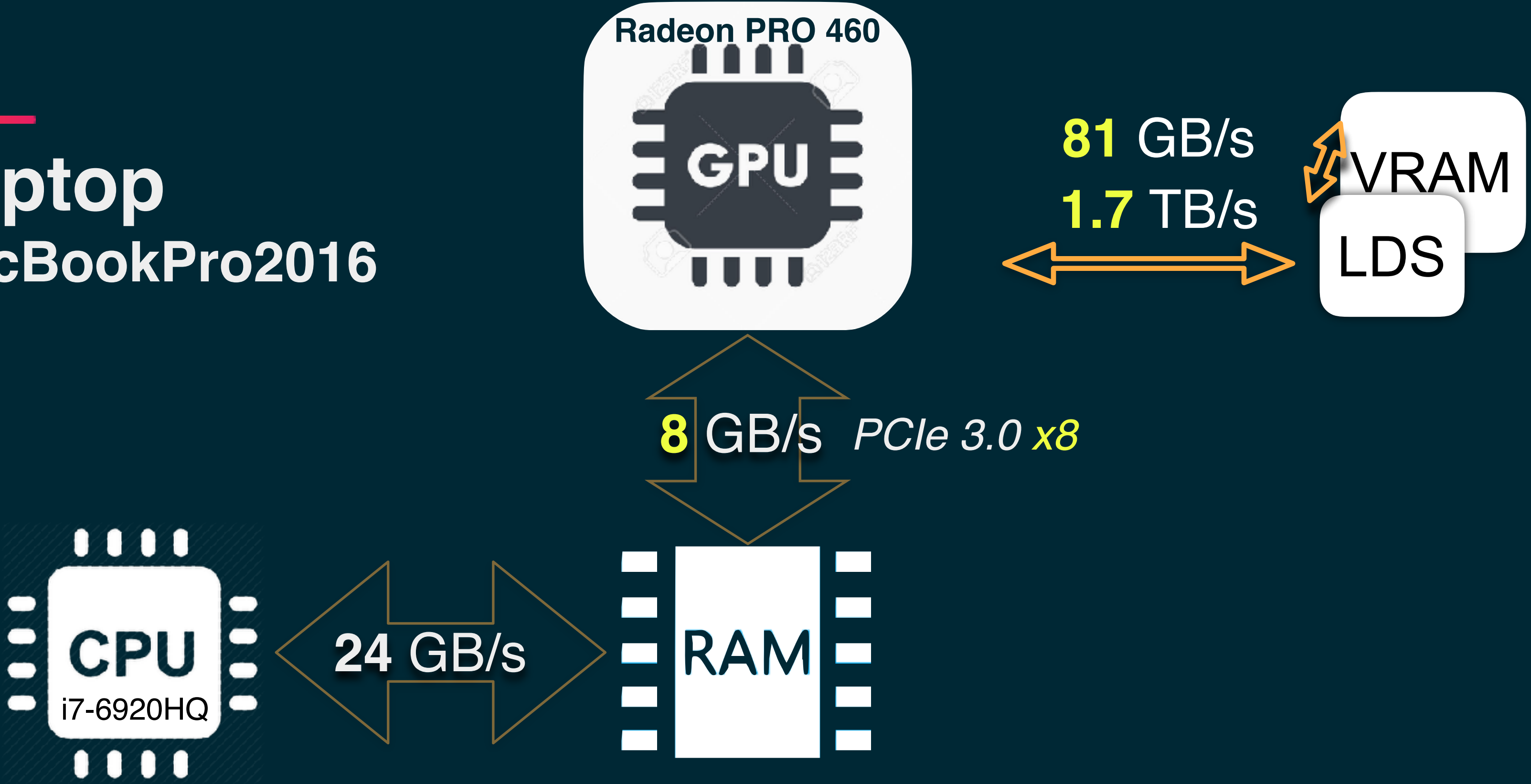
unity

Desktop
GTX 1080



Laptop

MacBookPro2016



unity

Take Away

MYTH: *PCIe is very slow*

CPU⇔GPU (PCIe) speed is not too terrible comparing with common CPU⇔memory ~ *1:3 ratio*

PCIe speed is mostly an issue when training with multi-GPU setup

Take Away

Need to access the same data several times on GPU to make worth the transfer

FLOPs per byte metric

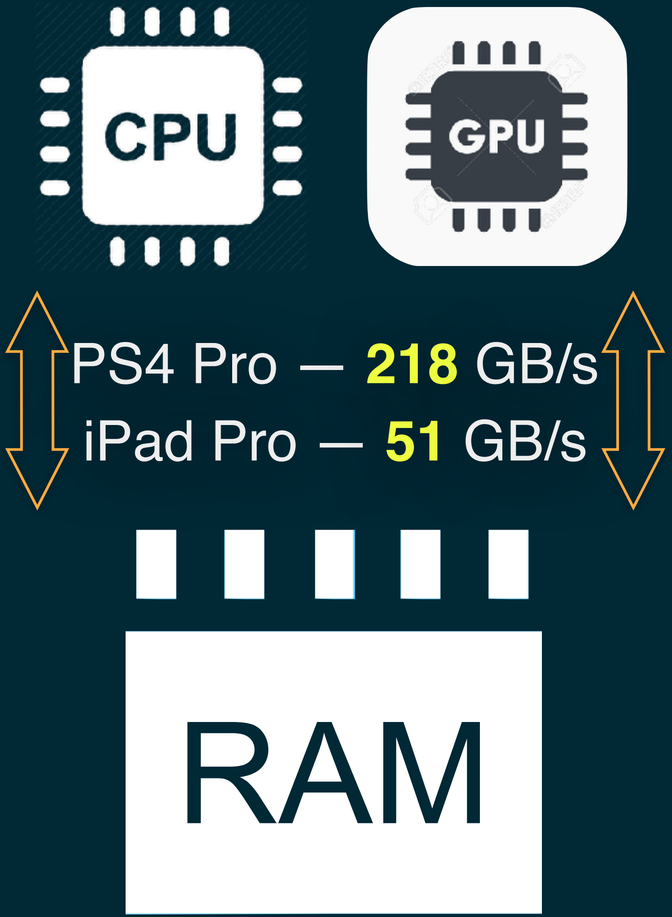
Take Away

Getting results **back** from GPU can be *slow*,
but NOT because of PCIe speed

Rather due to latency - CPU & GPU work async!

Mobile/Console

Unified Memory Architecture
CPU & GPU share same memory



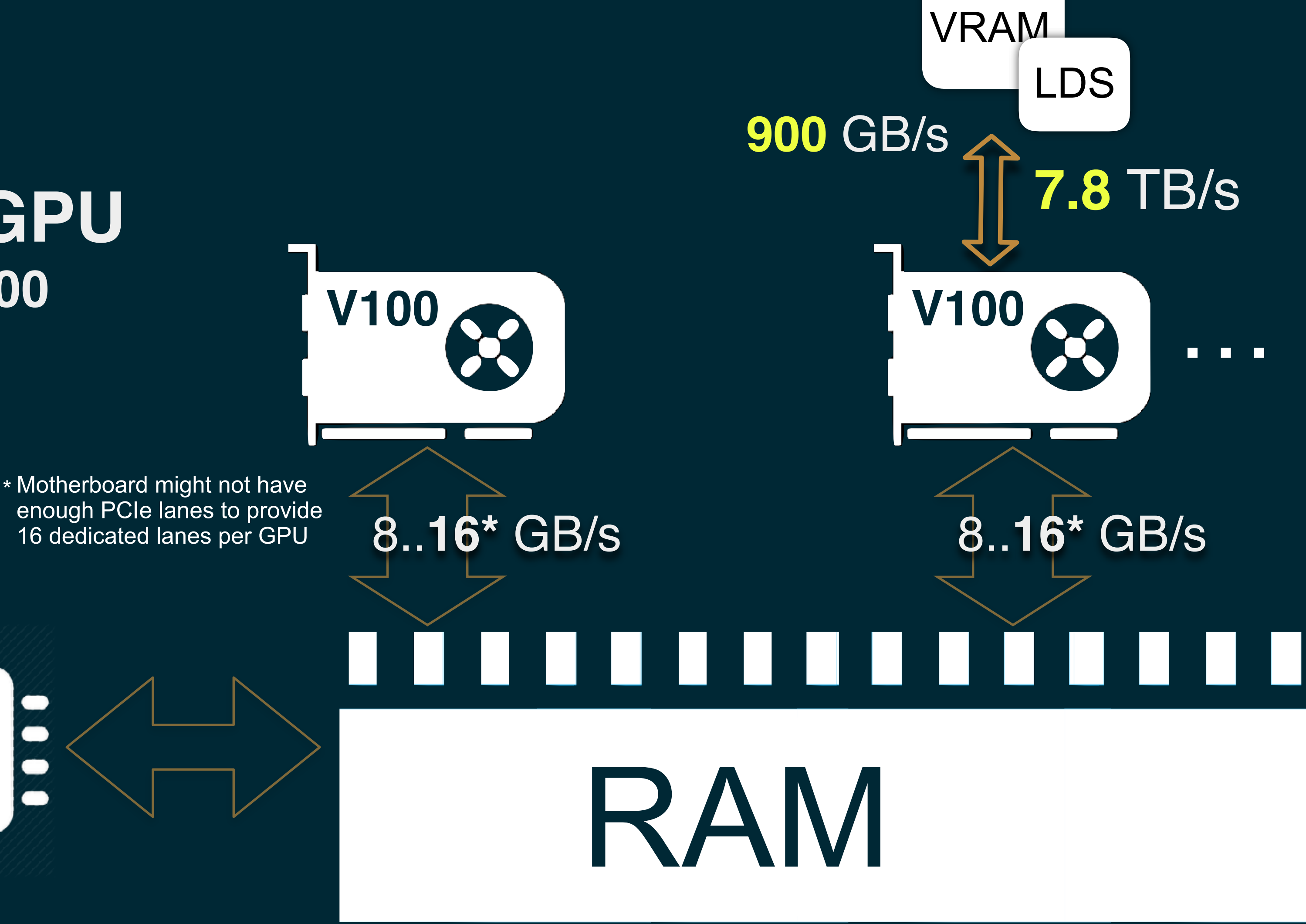
Take Away

CPU & GPU exchange data much faster
But overall bandwidth is limited

unity

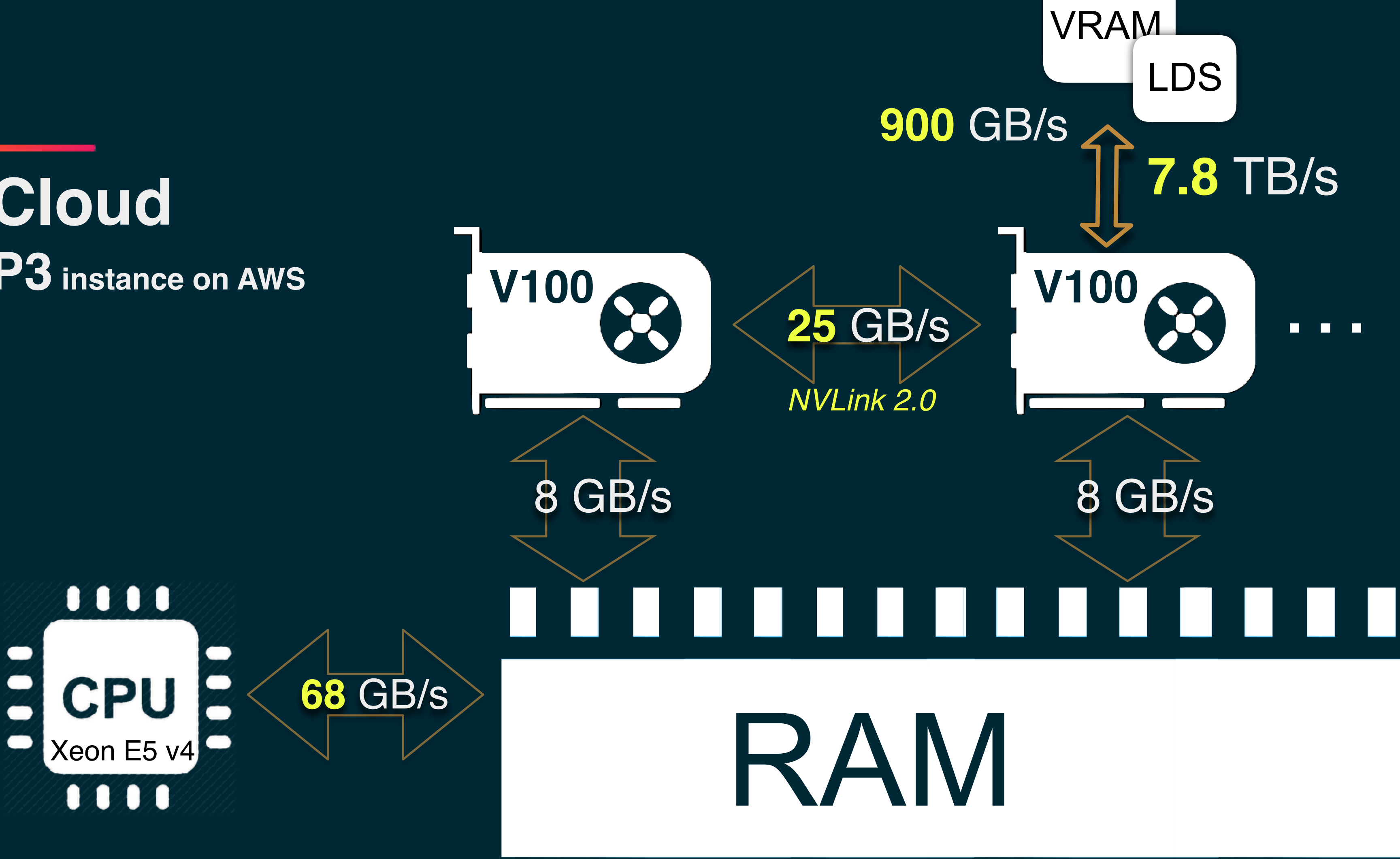
Multi-GPU

Tesla V100


VRAM
LDS
900 GB/s
7.8 TB/s
V100
V100
. . .
* Motherboard might not have enough PCIe lanes to provide 16 dedicated lanes per GPU
8..16* GB/s
8..16* GB/s
CPU
RAM

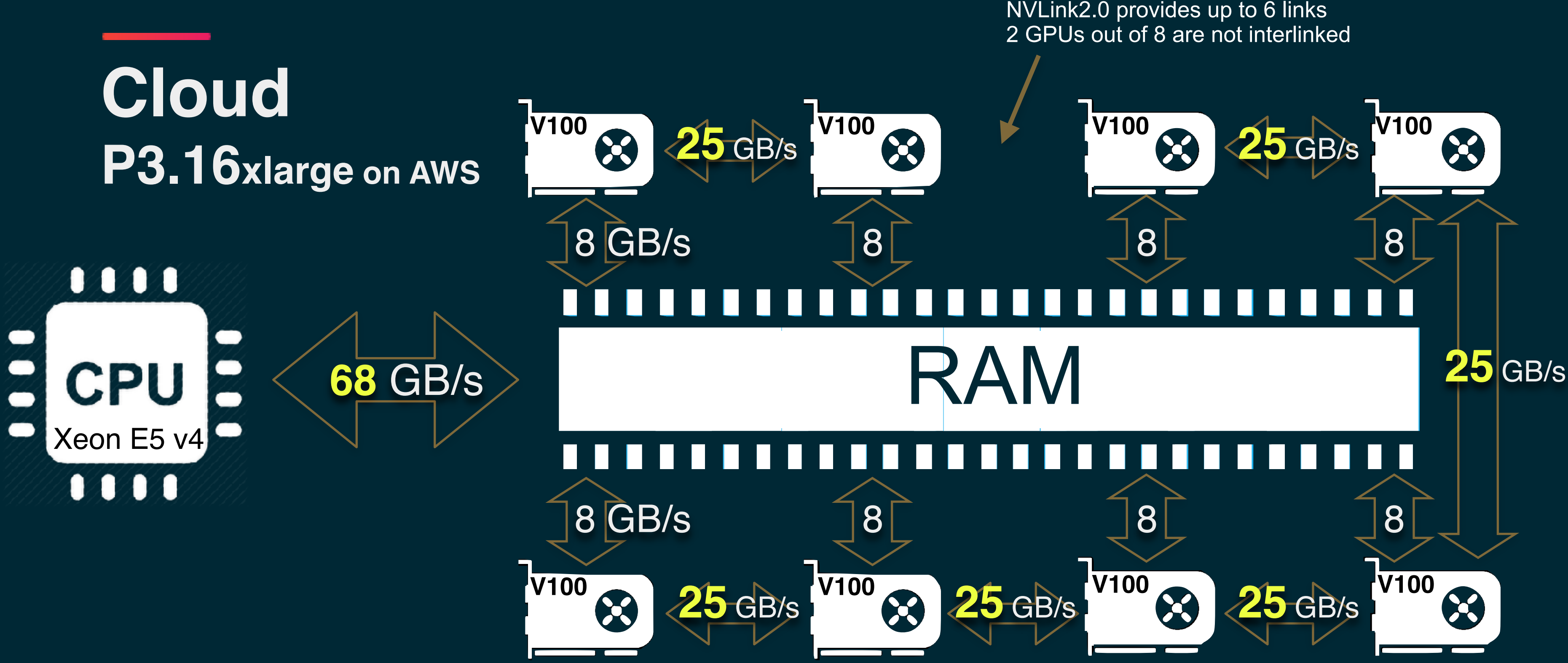
Cloud

P3 instance on AWS



Cloud

P3.16xlarge on AWS



unity

Take Away

PCIe speed is the bottleneck, if you need to synchronise multiple GPU

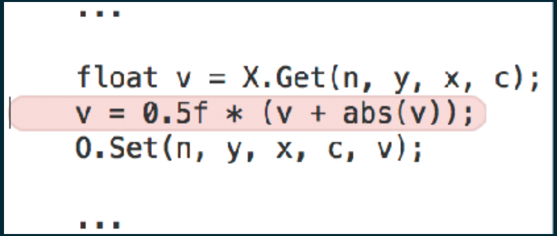
NVLink is essential for fast multi-GPU setup

Programming Model

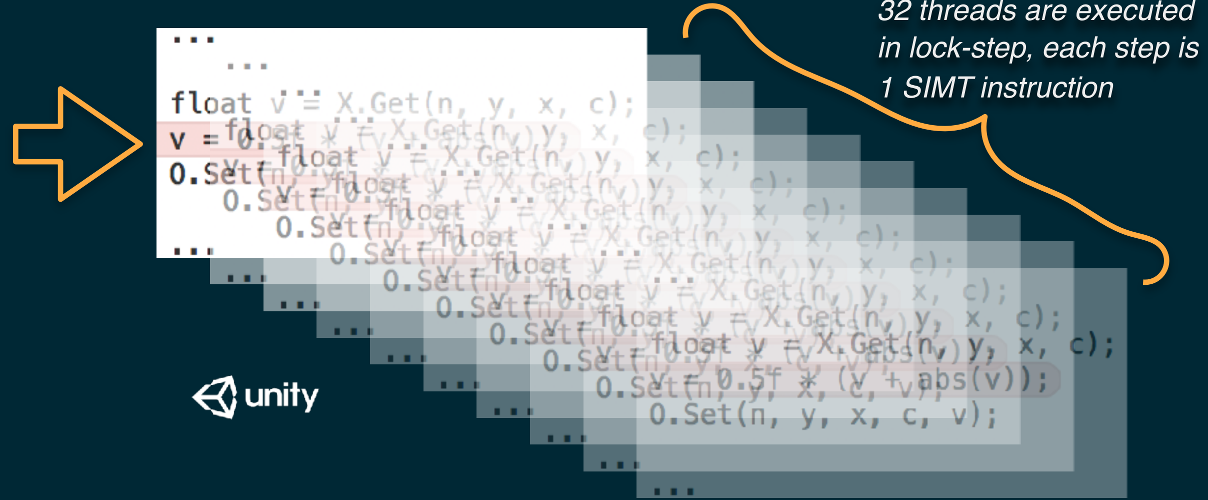
Programming model is scalar!

Compiler maps 32 **scalar** "threads" to 1 SIMT instruction

Memory access across 32 "threads" are grouped as well…



```
...

float v = X.Get(n, y, x, c);
v = 0.5f * (v + abs(v));
O.Set(n, y, x, c, v);

...
```



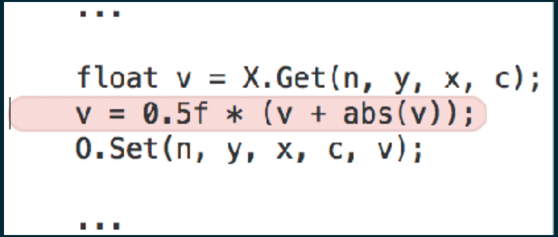
32 threads are executed in lock-step, each step is 1 SIMT instruction

```
...

float v = X.Get(n, y, x, c);
v = 0.5f * (v + abs(v));
O.Set(n, y, x, c, v);

...
```

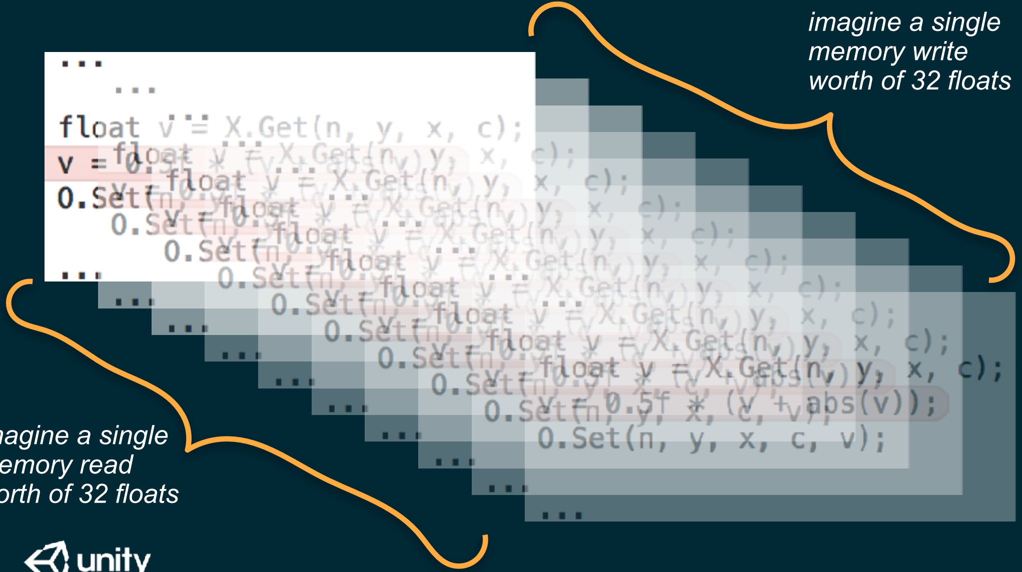
Memory accesses are grouped… wait, what?

Accesses are grouped into **large** transactions to **max memory bandwidth**

Single memory transaction 256 bit



```
...

float v = X.Get(n, y, x, c);
v = 0.5f * (v + abs(v));
O.Set(n, y, x, c, v);

...
```

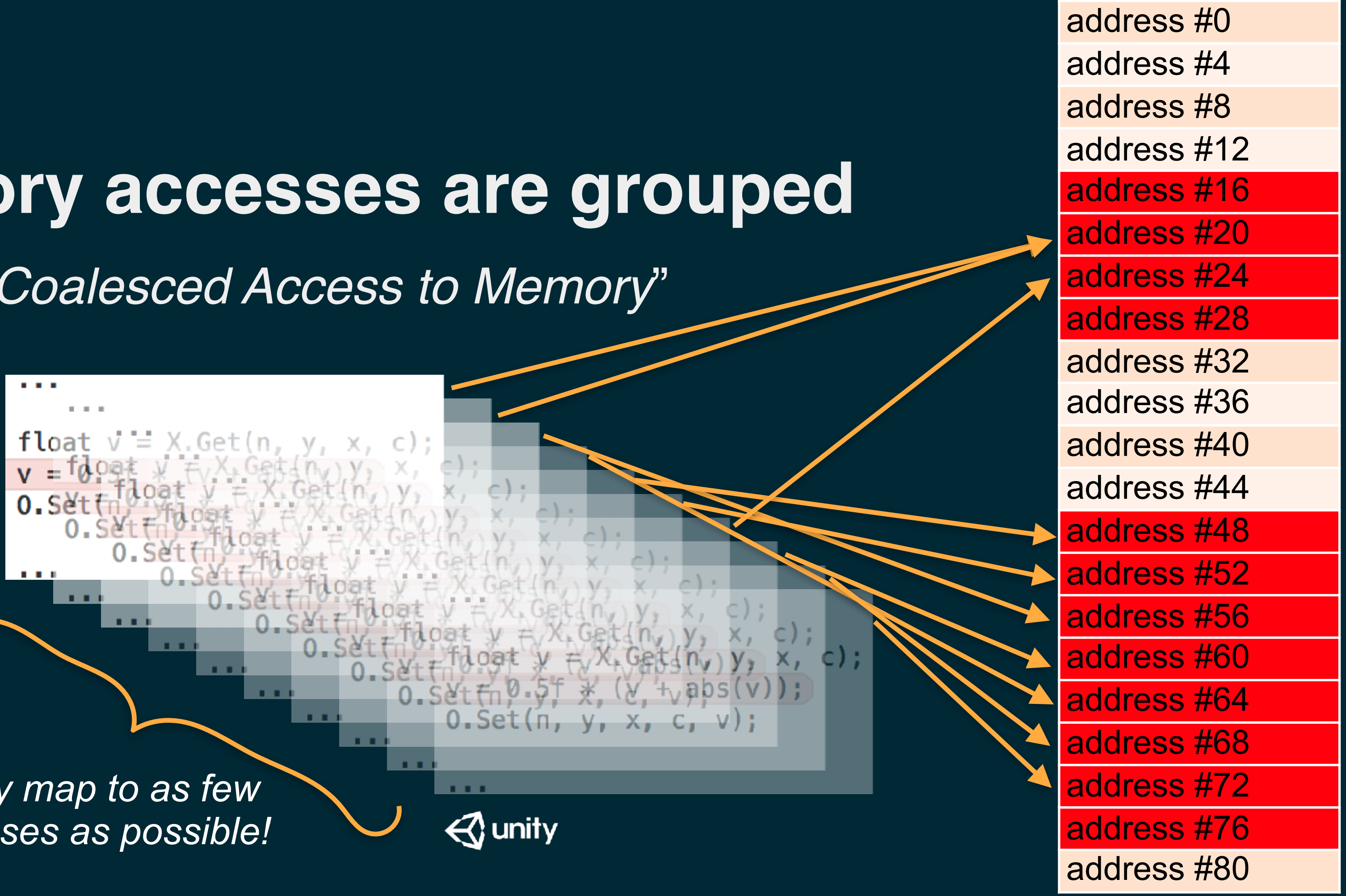

imagine a single memory write worth of 32 floats
imagine a single memory read worth of 32 floats

```
...
float v = X.Get(n, y, x, c);
v = 0.5f * (v + abs(v));
O.Set(n, y, x, c, v);
...
```

unity

Memory accesses are grouped

Called "*Coalesced Access to Memory*"

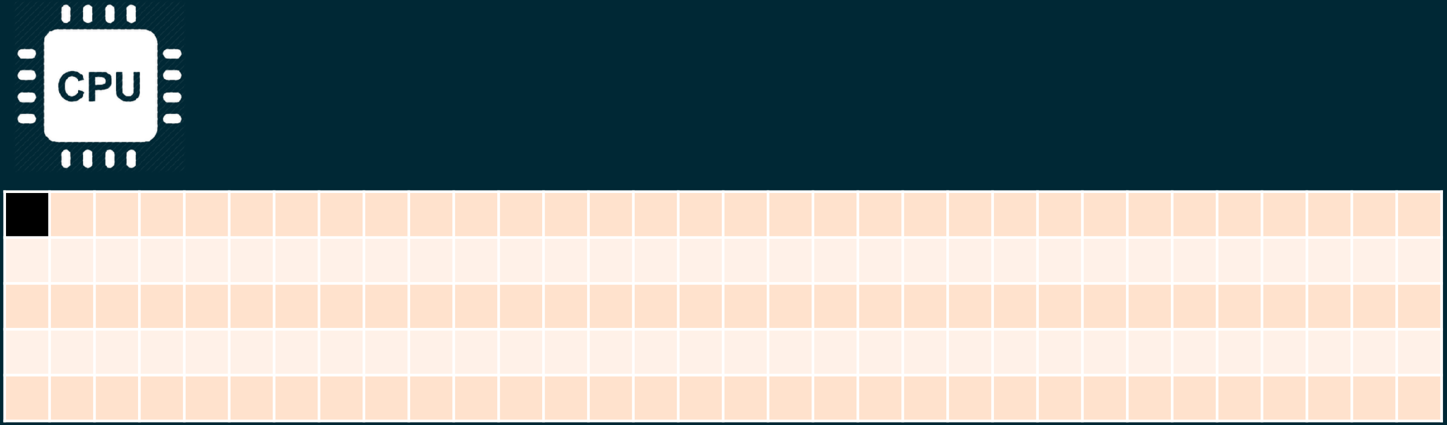


```
...
    ...
float v = X.Get(n, y, x, c);
v = 0.5f * (v + abs(v));
0.Set(n, y, x, c, v);
...
```

address #0
address #4
address #8
address #12
address #16
address #20
address #24
address #28
address #32
address #36
address #40
address #44
address #48
address #52
address #56
address #60
address #64
address #68
address #72
address #76
address #80

Will automatically map to as few cache line accesses as possible!

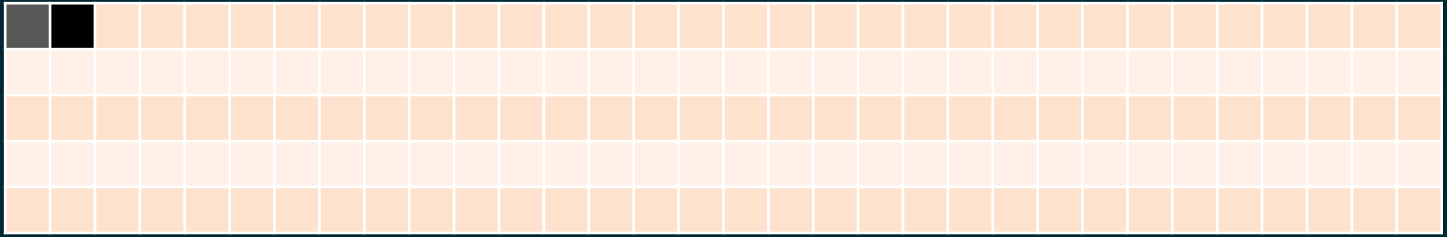
Naive “sequential” memory access



```
for (int i = 0; i < length; i++)
{
    memory[i] = 0;
}
```

Naive “sequential” memory access

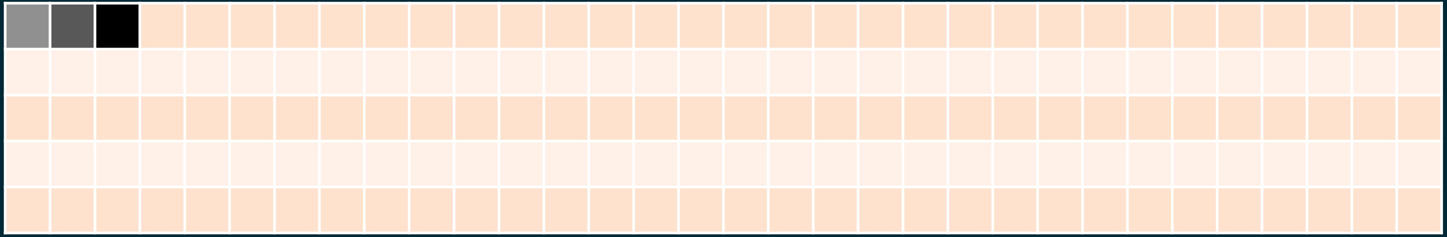
Good access pattern on CPU



```
for (int i = 0; i < length; i++)
{
    memory[i] = 0;
}
```

Naive “sequential” memory access

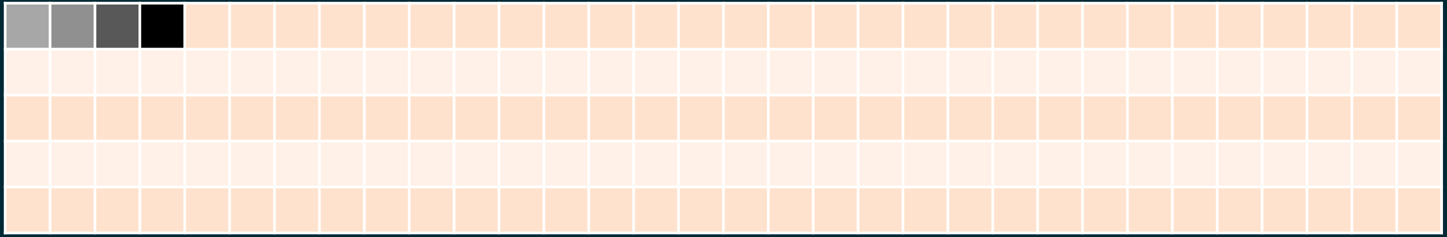
Good access pattern on CPU



```
for (int i = 0; i < length; i++)
{
    memory[i] = 0;
}
```

Naive “sequential” memory access

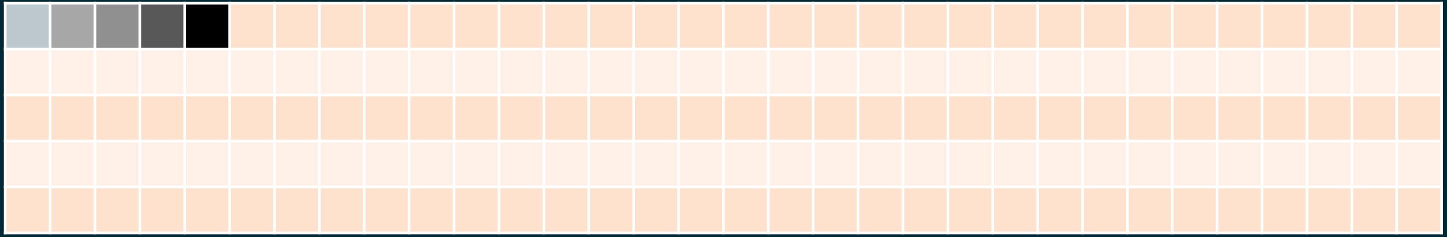
Good access pattern on CPU

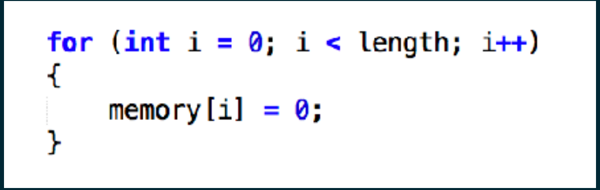


```
for (int i = 0; i < length; i++)
{
    memory[i] = 0;
}
```

Naive “sequential” memory access

Good access pattern on CPU

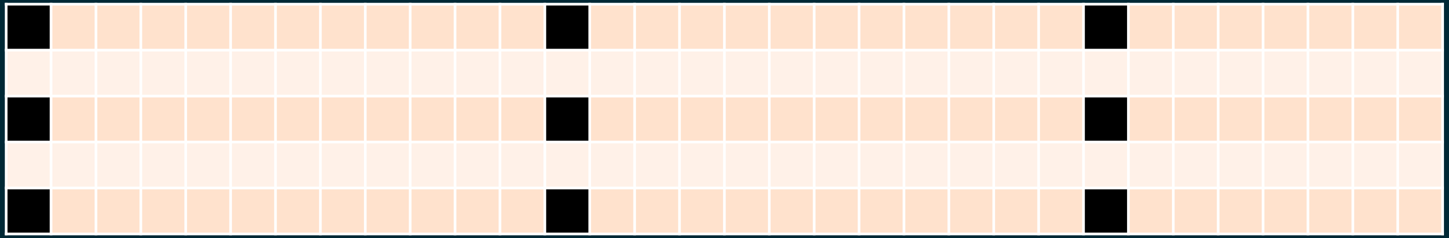




```
for (int i = 0; i < length; i++)
{
    memory[i] = 0;
}
```

Naive “sequential” memory access

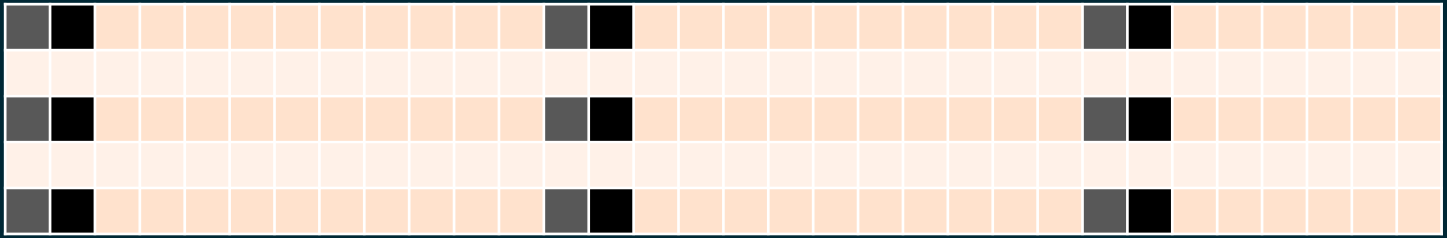
Turns BAD on GPU!



```
for (int i = 0; i < length; i++)
{
    memory[i] = 0;
}
```

Naive “sequential” memory access

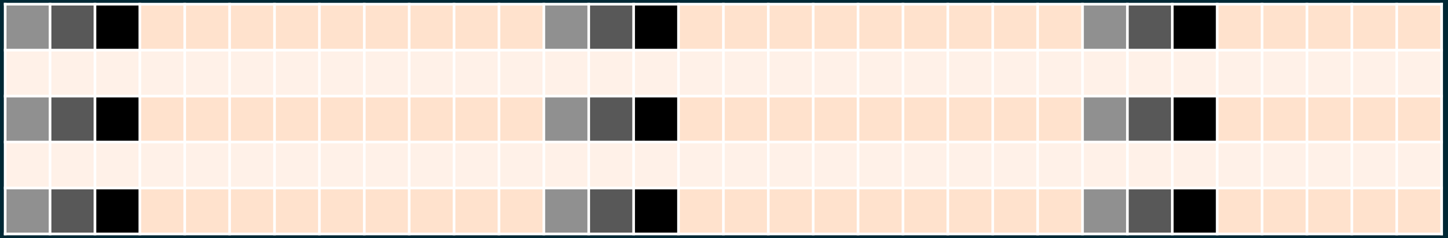
GPU runs many threads in parallel…



```
for (int i = 0; i < length; i++)
{
    memory[i] = 0;
}
```

Naive “sequential” memory access

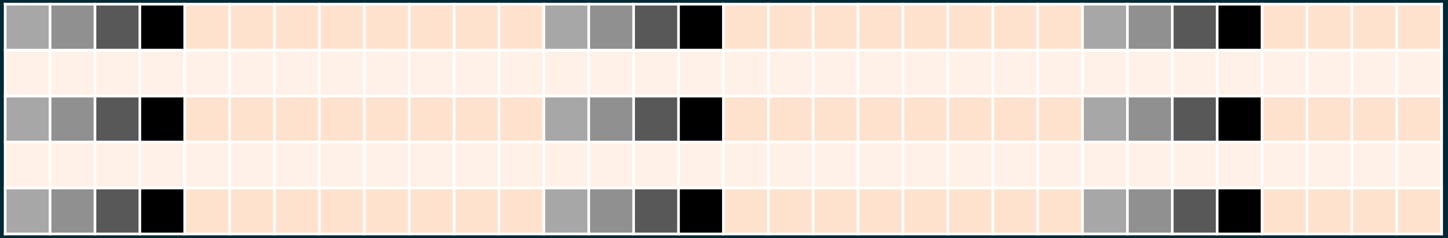
Each thread access different cache line



```
for (int i = 0; i < length; i++)
{
    memory[i] = 0;
}
```

Naive "sequential" memory access

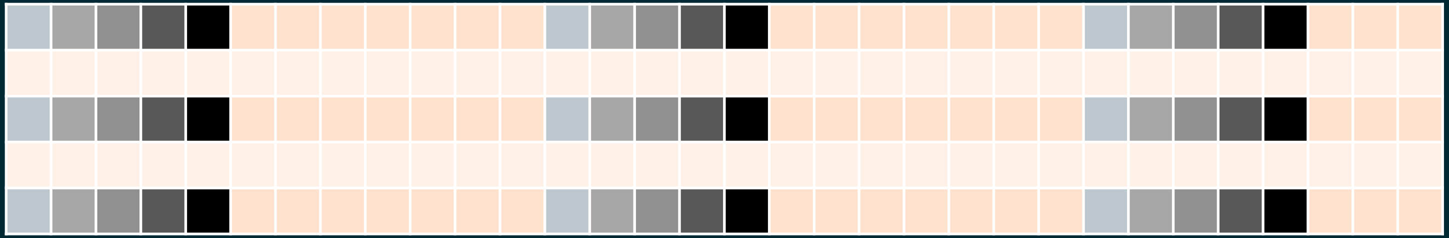
Each thread access different cache line

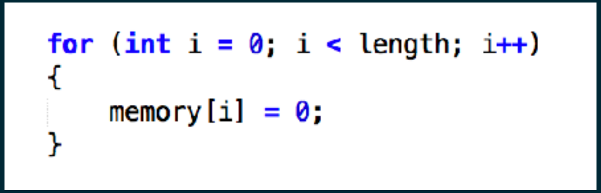


```
for (int i = 0; i < length; i++)
{
    memory[i] = 0;
}
```

Naive "sequential" memory access

Each thread access different cache line

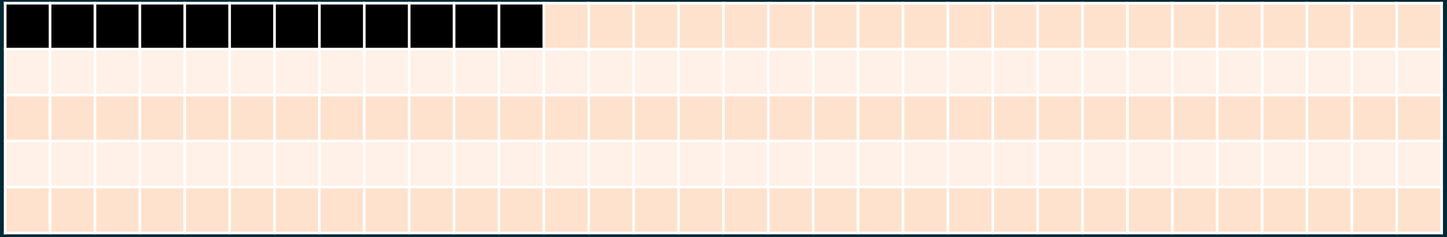


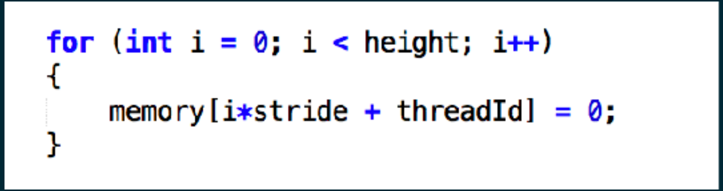


```
for (int i = 0; i < length; i++)
{
    memory[i] = 0;
}
```

Warp-aware memory access

Good!

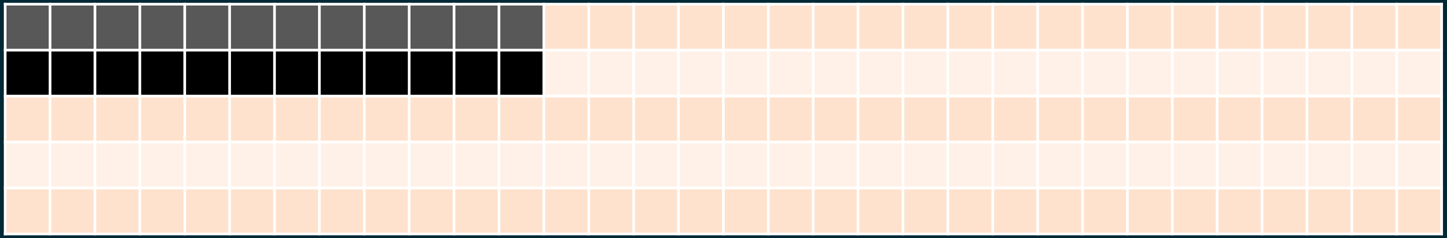




```
for (int i = 0; i < height; i++)
{
    memory[i*stride + threadId] = 0;
}
```

Warp-aware memory access

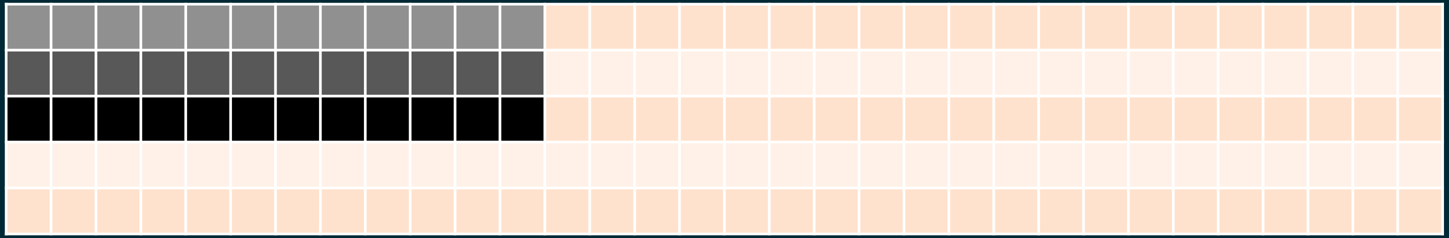
Good! Now nearby threads share cache lines



```
for (int i = 0; i < height; i++)
{
    memory[i*stride + threadId] = 0;
}
```

Warp-aware memory access

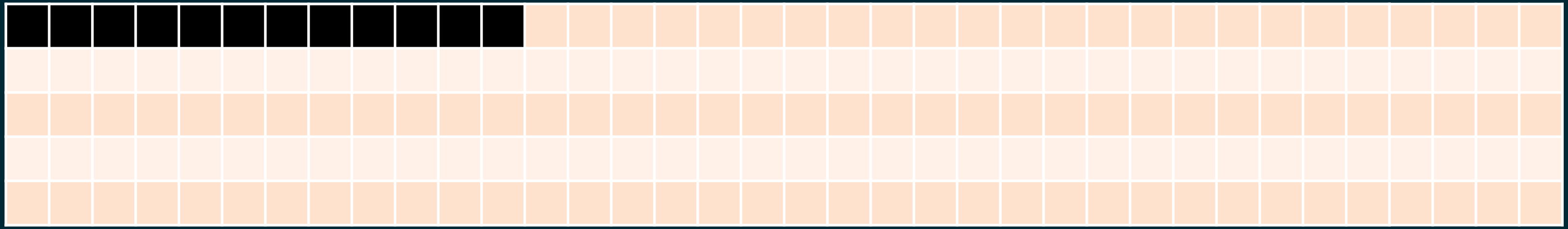
Good! Now nearby threads share cache lines



```
for (int i = 0; i < height; i++)
{
    memory[i*stride + threadId] = 0;
}
```

Warp-aware memory access

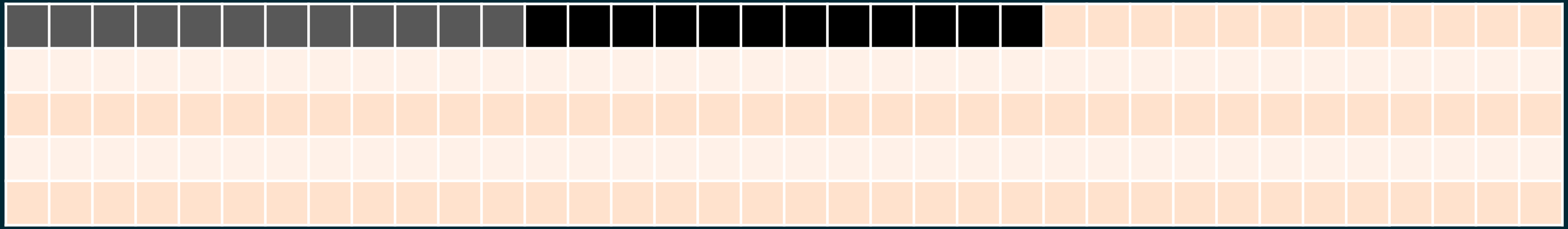
Another good!



```
for (int i = 0; i < length / THREADS_IN_WARP; i++)
{
    memory[i*THREADS_IN_WARP + threadId] = 0;
}
```

Warp-aware memory access

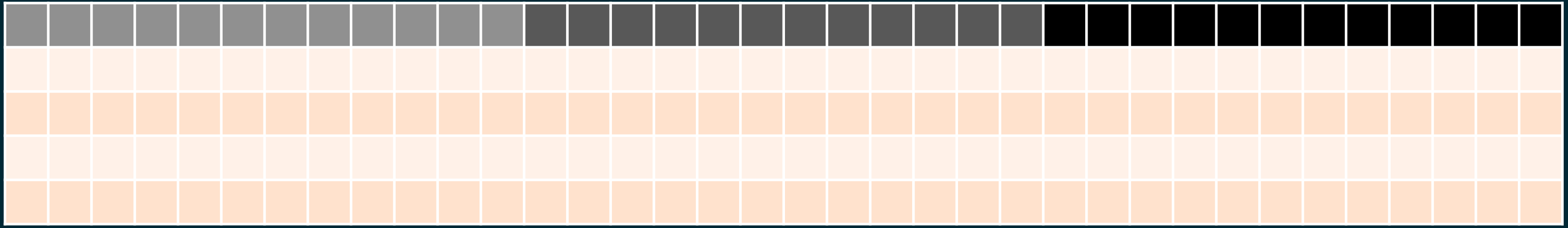
Another good!



```
for (int i = 0; i < length / THREADS_IN_WARP; i++)
{
    memory[i*THREADS_IN_WARP + threadId] = 0;
}
```

Warp-aware memory access

Another good!



```
for (int i = 0; i < length / THREADS_IN_WARP; i++)
{
    memory[i*THREADS_IN_WARP + threadId] = 0;
}
```

Programming Model part 2

CUDA pipeline

CUDA C ⇨ PTX ⇨ cubin / SASS

CUDA C — that is what you usually write

cubin / SASS — that is what actually runs on GPU

CUDA

CUDA C ⇨ PTX ⇨ cubin / SASS

PTX — bytecode for GPU

Intermediate step

NVIDIA **only**, but architecture* independent

cubin / SASS — binary for GPU

NVIDIA **only** and architecture* specific

* By saying 'architecture' here, I mean: Volta, Pascal, Maxwell, Kepler, etc

CUDA

CUDA C ⇨ PTX

nvcc — nvidia **open** source LLVM based compiler

PTX ⇨ cubin / SASS

ptxas — nvidia **closed** source assembler

OpenCL

Doesn’t integrate well with the cross-platform engine

DirectX + OpenCL = ?

PS4 + OpenCL = ?

Mobile + OpenCL = ?

Code is compiled by OpenCL run-time (“driver”)

Result might be hit-or-miss in terms of performance

Performance is not portable

Platform specific Zoo

DirectCompute — Windows, XBOX

Metal Compute — iOS, MacOS

GLSL Compute — *PS4, Linux, Android ES3.1*

Vulkan Compute is cross-platform, but not widely implemented *yet!*

All Integrate well with the rendering engine

Asyncronous compute - graphics and compute workloads simultaneously

Unity Compute ☛ bit.ly/unity-compute-docs

Cross compiles to platform specific Compute:

- **Direct**Compute — Windows, XBOX
- **Metal** Compute — iOS, MacOS
- **GLSL** Compute — *PS4, Linux, Android ES3.1*
- **Vulkan** Compute — Android, …

Integrated well with the rendering engine

Performance is not portable

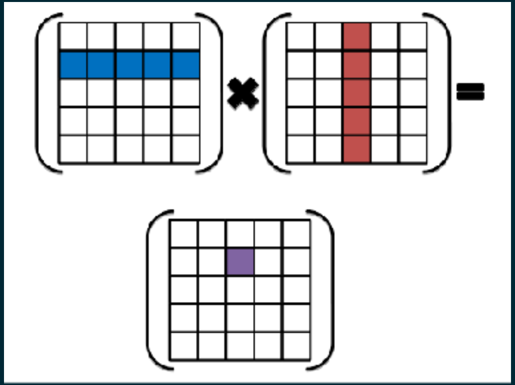
Practical Performance

Large Matrix Multiplication

Workhorse of Machine Learning

- **Fully Connected** layer is matrix multiplication
- **Convolutional** layer has a lot in common /w matrix multiplication

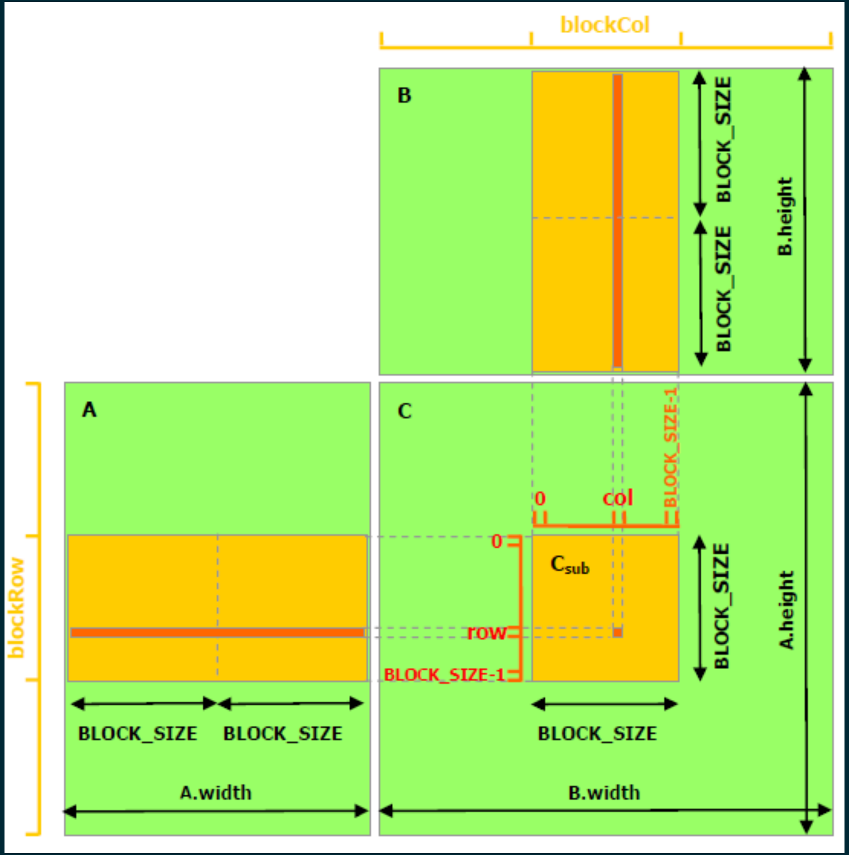
SGEMM in BLAS



Matrix multiplication

Math ops = $O(N^3)$

Memory ops = $O(N^2+N^2)$



unity

Matrix multiplication

Math ops = $O(N^3)$

Memory ops = $O(N^2+N^2)$

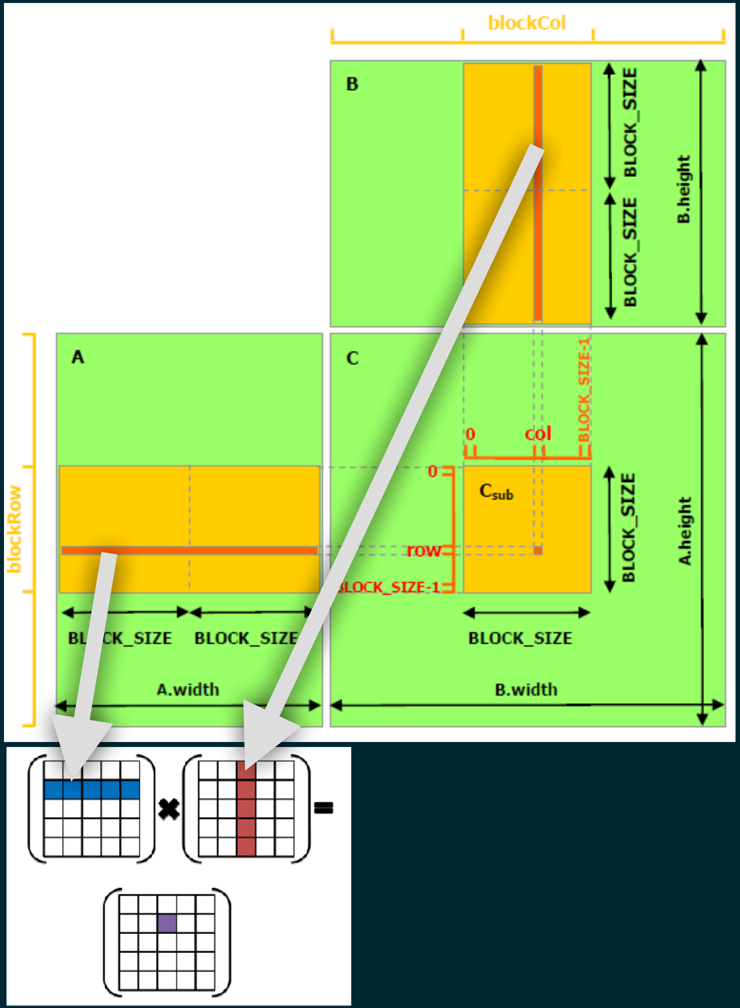
Classical solution

Work in *blocks* aka *tiles*!

- Load source **tiles** from the memory to the **cache** (LDS)

Accumulate multiplication result in the cache

Store accumulator **tile** back to the memory



Matrix multiplication

Math ops = $O(N^3)$

Memory ops = $O(N^2+N^2)$

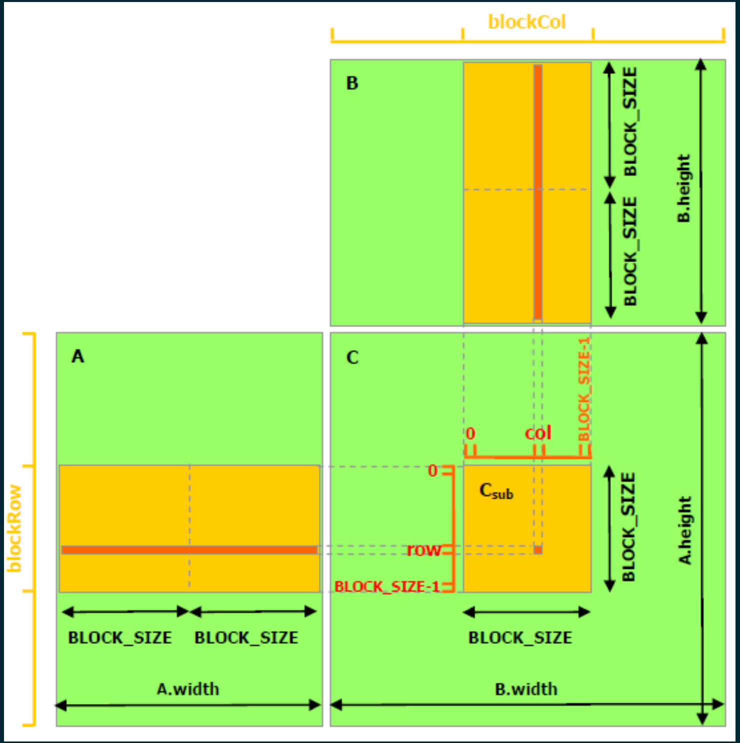
Classical solution

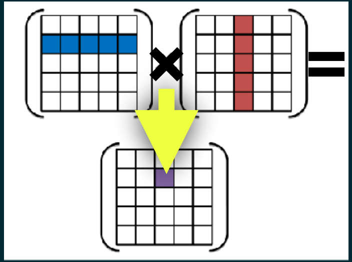
Work in *blocks* aka *tiles*!

Load source **tiles** from the memory to the **cache** (LDS)

► Accumulate multiplication result in the cache

Store accumulator **tile** back to the memory





Matrix multiplication

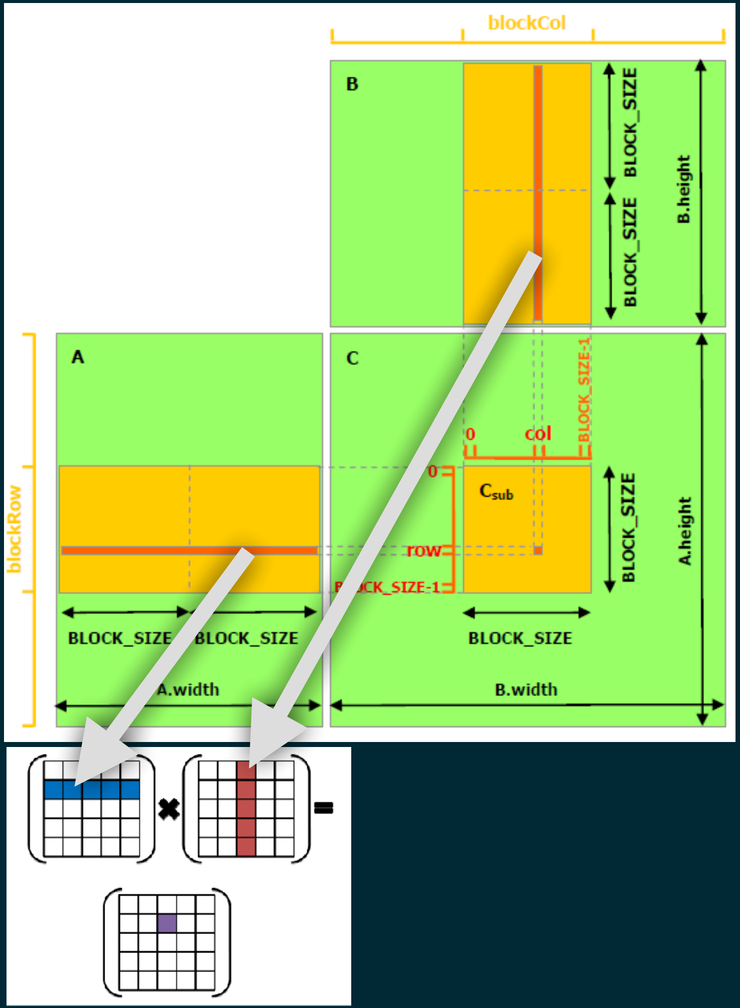
Math ops = $O(N^3)$

Memory ops = $O(N^2+N^2)$

Classical solution

Work in *blocks* aka *tiles*!

- Load source **tiles** from the memory to the **cache** (LDS)
- Accumulate multiplication result in the cache
- Store accumulator **tile** back to the memory



unity

Matrix multiplication

Math ops = $O(N^3)$

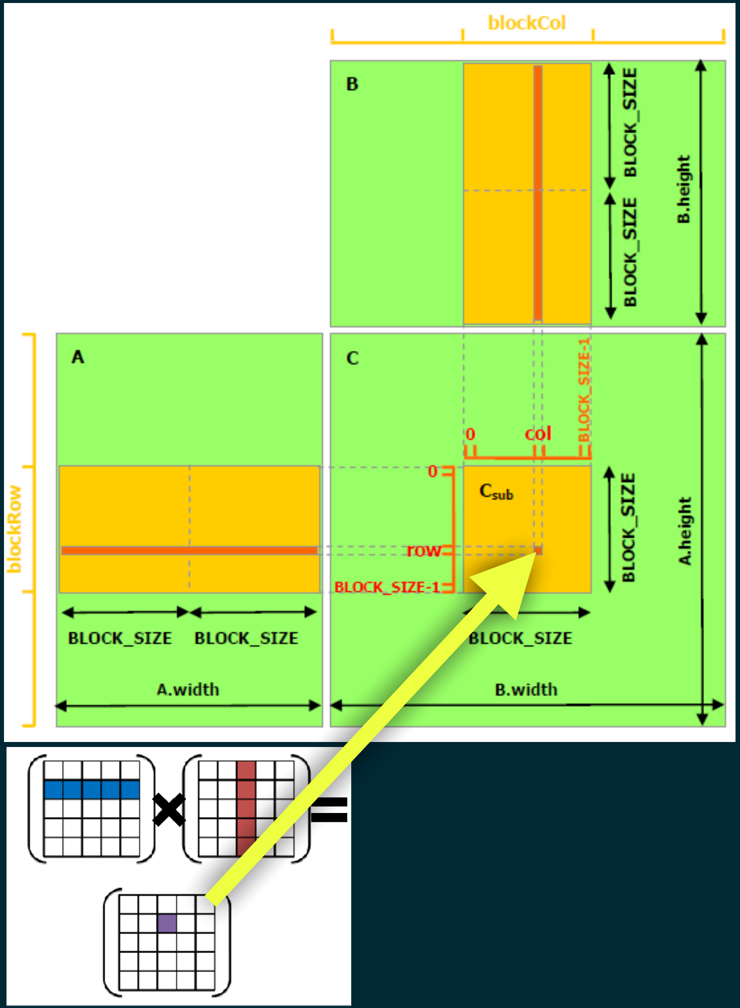
Memory ops = $O(N^2+N^2)$

Classical solution

Work in *blocks* aka *tiles*!

Load source **tiles** from the memory to the **cache** (LDS)

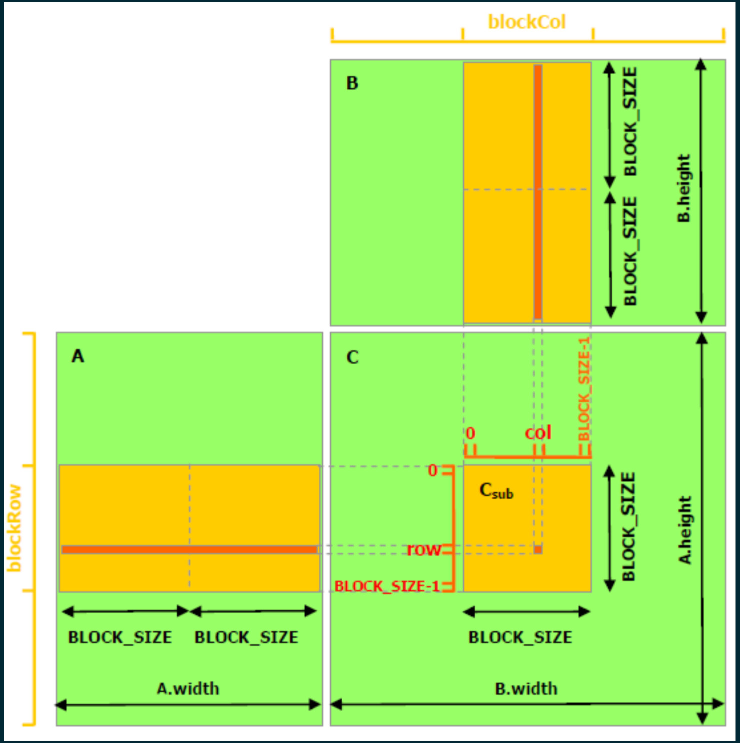
- Accumulate multiplication result in the cache
- Store accumulator **tile** back to the memory

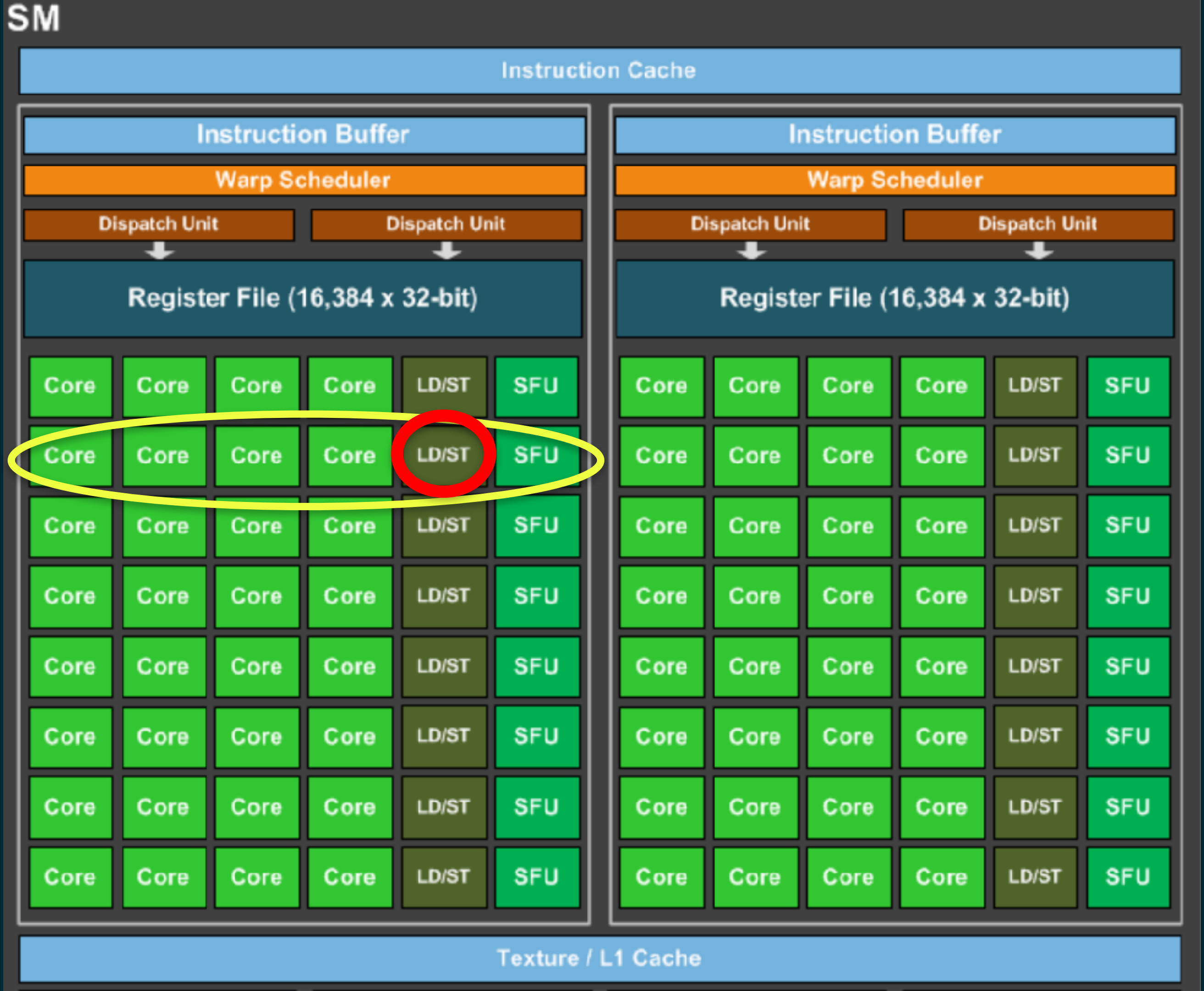


Not too fast!

Still **far** from the maximum TFLOPs

Why?



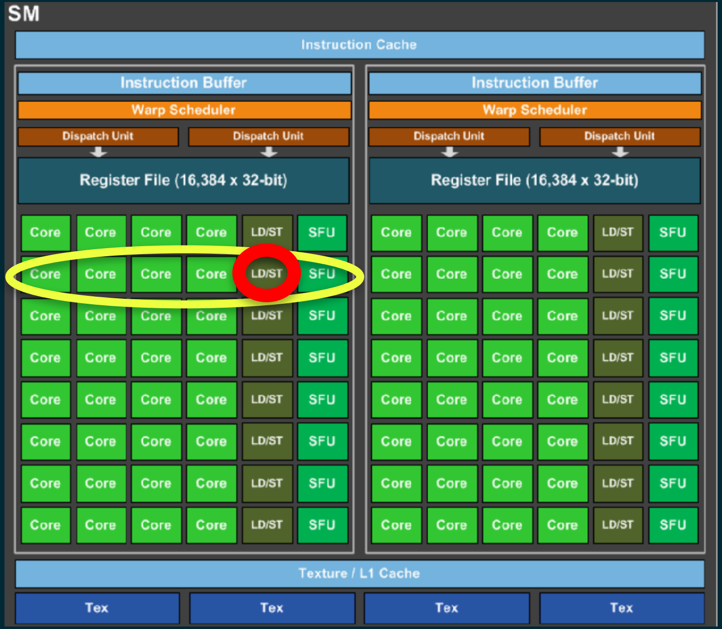

SM
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core
LD/ST
SFU
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core
LD/ST
SFU
Texture / L1 Cache

More ALU than LoaD/STore

GPU can issue **more MultiplyAdd** instructions than memory reads **per cycle**

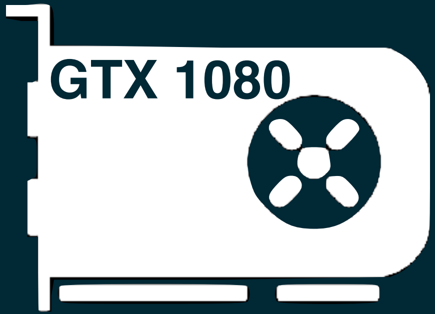
GPU packs more **arithmetic** (ALU) than **memory** access **units** (LD/ST)

4:1 is a common ratio



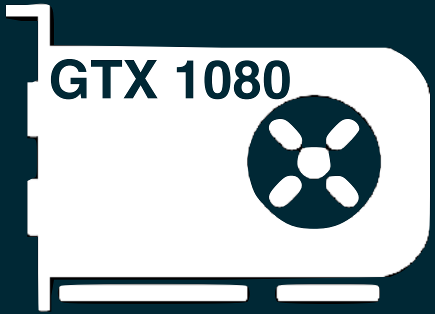
unity

GTX 1080



320 GB/s	— VRAM
4.1 TB/s	— LDS
30+ TB/s	— Registers

GTX 1080



320 GB/s	—	VRAM
4.1 TB/s	—	LDS
30+ TB/s	—	Registers

File of 16K scalar registers shared for up to 16 warps

Up to **256** scalar (FP32) registers per thread

unity

Matrix multiplication #2

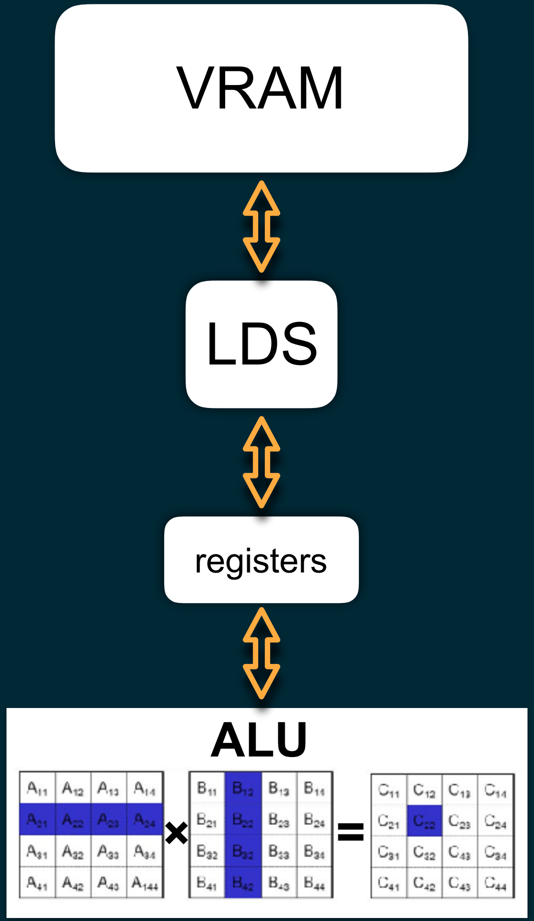
Cache big tiles in LDS

to minimize VRAM bandwidth

Cache small tiles (4x4) in registers

to minimize LD/ST issue

Arithmetic operations on 4x4 blocks



Take away

Reaching the best performance requires data dimensions to be multiple of tile size

Minibatch size is especially crucial for **Fully Connected** layers

Take away

Sweet spot for **Convolutional** layers is between 16 and 32

Preferably all dimensions divisible by 4 or 8

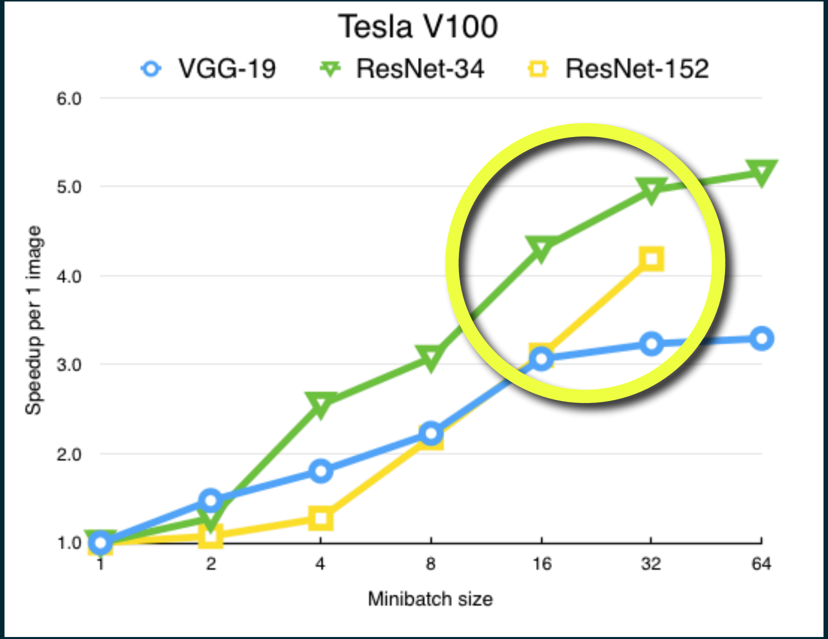
- small tile loaded into registers
- Volta TensorCore

Speed of Convolutional Nets due to minibatch size

Sweet spot:

VGG — 16+

ResNet — 32+



unity

Despite architectural differences between CPU & GPU

what dominates the speed of training Convolutional Neural Net

*is the raw TFLOPs of a given chip!**

*) Given that reasonably optimised code is used - like *cuDNN* lib for GPU and *Intel-MKL-DNN* for CPU

Xeon E5 v4 @ 2.2Ghz × 20 cores
0.7 TFLOPs**

i7-7700 @ 3.6Ghz × 4 cores
0.36 TFLOPs

iPad Pro, A9Xm @ 2.26Ghz × 2 cores
0.08 TFLOPs

Tesla V100 @ 1.4Ghz × 80 SM cores
14.9 TFLOPs***

GTX 1080 Ti @ 1.5Ghz × 28 SM cores
11.34 TFLOPs

iPad Pro, A9X-PVR-7XT @ 0.45Ghz × 12
0.35 TFLOPs

**) CPU numbers here are measured and do not completely agree with theoretical - some errors might have crept in ;)

***) Numbers for both CPU & GPU are specified at full FP32 precision

Hiring ML + Graphics experts